

מיקרו-מעבדים ושפת אסמלר

תרגול מס' 1 - מבוא

מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן



נהלי הקורס

- מרצה: ד"ר רן גלו
15:00-17:00
- יומן ראשון:
- מתרגם: רומן גולמן (xfory2@gmail.com)
 - יומן שני: 14:00-14:45
 - יומן חמישי: 14:00-14:45
- ציון: מבחן % 80%
- תרגילים: % 20
 - 8\8 תרגילי בית (תיאורטי + תכנות)
 - הגשה בזוגות
- אתר הקורס:
 - מודל
- פיאצה (שאלות בנוגע לתרגילים): <https://piazza.com/biu.ac.il/spring2020/83255/home>

בסיס 2 ובסיס 16

- בסיס 10 יש 10 ספרות: 0,1,2,3,4,5,6,7,8,9 שמהן מרכיבים את המספרים השונים
- בסיס 2 יש 2 ספרות: 0,1 שמהן מרכיבים את המספרים
- בסיס 16 יש 16 ספרות שמהן מרכיבים את המספרים:

Digit	Value
0	0
1	1
2	2
3	3
4	4
5	5

Digit	Value
6	6
7	7
8	8
9	9
A	10
B	11

Digit	Value
C	12
D	13
E	14
F	15

מעבר בסיסים

- כיצד מרכיב מספר בבסיס 10?

$$243_{10} = 2 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0$$

- כיצד מרכיב מספר בבסיס 2?

$$1001001_2 = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 73_{10}$$

- כיצד נראה מספר בבסיס 16?

$$12AB4_{16} = 1 \cdot 16^4 + 2 \cdot 16^3 + A \cdot 16^2 + B \cdot 16^1 + 4 \cdot 16^0 = 76468_{10}$$

(A=10, B=11)

מעבר בסיסים

- כיצד נעבור מבסיס 10 לבסיס 2?
- אלגוריתם: מחלקים ב-2 והשארית היא הביט הבא (מפסיקים כשהתוצאה היא 0).

$75 / 2 = 37$ with remainder 1
 $37 / 2 = 18$ with remainder 1
 $18 / 2 = 9$ with remainder 0
 $9 / 2 = 4$ with remainder 1
 $4 / 2 = 2$ with remainder 0
 $2 / 2 = 1$ with remainder 0
 $1 / 2 = \mathbf{0}$ with remainder 1



LSB

MSB

$$75_{10} = 1001011_2$$

- בבסיס 16 זה בדוק אותו אלגוריתם רק שמחקим ב-16 והשארית היא בין 0 ל 15.

מעבר בסיסים

- מעבר מהיר מבסיס 2 לבסיס 16:

1000100111011011101110110110110111

2 2 7 6 E E D B 7

חיבור בבסיס 2

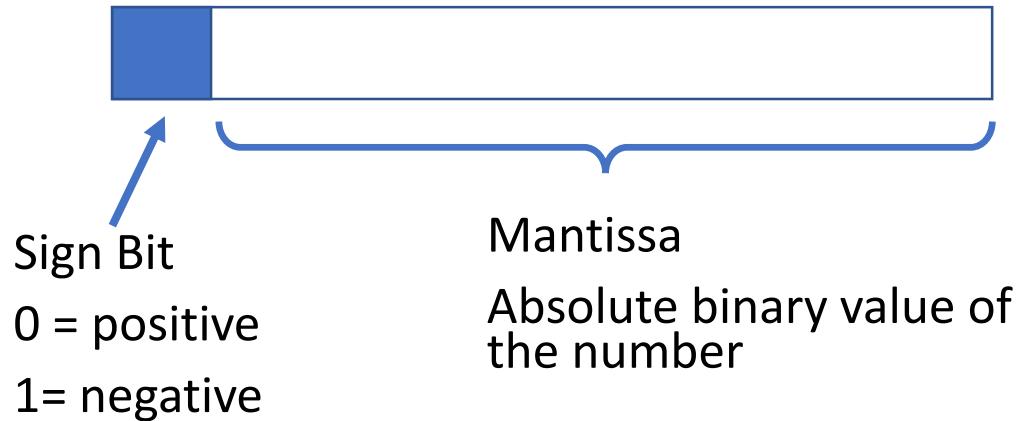
- חיבור:

$$\begin{array}{r} (1) \quad (1) \\ 11010101 \\ + \quad 11000100 \\ \hline 111011001 \end{array}$$

- בעולת החיבור נסופה סיבית חדשה, היא נקראת **Carry**
- אם מבצעים חיבור של שני מספרים בני 8 סיביות. כמה סיביות נדרשות ע"מ לשמר את התוצאה?

סימן ומנטיסה

- ייצוג המספרים במעבד הוא סדרה של ביטים, והיא צריכה ליזג גם מספרים שליליים
- פתרון נאייבי: להוסיף בית שמייצג סימן



- הבעיה בשיטה היא שמוסבר להשתמש בה כדי לחבר מספרים עם סימנים הפוכים

המשלים ל 1

- כאשר המספר חיובי המנטיסה היא הערך המוחלט של המספר
- כאשר המספר שלילי המנטיסה היא המספר המשלים לערך המוחלט
(כלומר הביטים הפוכים)



$$(9)_2 = 01001$$
$$(-9)_2 = 10110$$

• דוגמא:

- בשיטה זו החיבור הוא חיבור פשוט של המספרים הבינאריים וחיסור הוא חיבור של המספר הנגדי
- החיסרון של השיטה הוא ייצוג כפול של 0

המשלים ל 2

- כאשר המספר חיובי המנטיסה היא הערך המוחלט של המספר
- כאשר המספר שלילי המנטיסה היא המספר המשלים לערך המוחלט פלוס אחד
(הופכים את כל הביטים ומוסיפים אחד לתוצאה)



$$(9)_{10} = 01001_2$$

$$01001 \rightarrow 10110 \rightarrow 10110 + 00001 = 10111$$

$$(-9)_{10} = 10111$$

אם נפעיל את האלגוריתם על מספר שלילי, נקבל חזרה את המספר החיובי.

משלים ל-2

- ניתן לייצג מספרים בתחום: $[1 - 2^{n-1} : 2^{n-1}]$ לא סימטרי!
- חיבור: חיבור בינארי רגיל כאשר זורקים את ה carry של חיבור 2 הספרות האחרונות.
- חיסור: הופכים את המספר השני לשיליי (היפוך סיביות+1), ואז עושים חיבור.
- בשיטת משלים ל-2 אין אף כפול.

משלים ל₂

$$(-7)_{10} = 11001_2$$

$$(9)_{10} = 01001_2$$

$$\begin{array}{r}
 & (1) & (1) \\
 & 11001 \\
 + & 01001 \\
 \hline
 100010 & = (2)_2
 \end{array}$$

זורקים

- דוגמא:

- את אותו החישוב ניתן לרשום גם בבסיס 16, אך בשביל זה נצטרך להרחיב את המספרים ל8 ביט:

- כדי להרחיב את המספרים שמיוצגים במשלים ל₂, פשוט נשכפל את הספירה השמאלית

$$(-7)_{10} = 11001_2 = \underline{111}11001_2 = F9_{19}$$

$$(9)_{10} = 01001_2 = \underline{000}01001_2 = 09_{16}$$

$$\begin{array}{r}
 & (1) \\
 & F9 \\
 + & 09 \\
 \hline
 102
 \end{array}$$

$9 + 9 = 18_{10} = 12_{16}$

גָלִישָׁה (Overflow)

- שיטת משלים ל 2 מייצגת מספרים בתחום $[1 - 2^{n-1} : 2^{n-1} - 2^n]$ כאשר ח' זה מספר הסיביות במספר.
- אם תוצאה החיבור חורגת מטווח זה אנחנו בבעיה – זו היא גליישה.

מספר שלילי מינימלי ב 8 סיביות : $-128 = -2^{(8-1)}$

$$(-107) + (-60) = -167$$

התוצאה חורגת מהתחום

(1) (1)

• דוגמא:

$$\begin{array}{r} 10010101 \\ + 11000100 \\ \hline \textcolor{blue}{1}01011001 \end{array}$$

- לפי סימן שלילי של חיבור 2 מספרים חיוביים, או סימן חיובי של חיבור מספרים שליליים, המעבד יודע שהתרחשה גליישה ומדליק דגל overflow.

Carry vs overflow

- דוגמא לחיבור (4ビット) בו יש carry ואין overflow

$$\begin{array}{r} 1111 \\ + 1111 \\ \hline 11110 \end{array} = (-2)_{10}$$

$(-1)_{10} = 1111_2$

- דוגמא לחיבור (4ビット) בו יש overflow ואין carry

$$\begin{array}{r} 0111 \\ + 0111 \\ \hline 1110 \end{array} = (-2)_{10}$$

$7_{10} = 0111_2$

מספר שלילי

BYTE/WORD/DWORD

- כאשר רוצים להקצות זיכרון כדי לשמר מספר, הוא יכול להיות באחד מהגדלים הבאים:

- Byte = 8 bits



- WORD = 2 Bytes



- DWORD = 2 WORDS



(הקצת משטנה)

NUMBER DW 15B8h

Variable name

Define Word

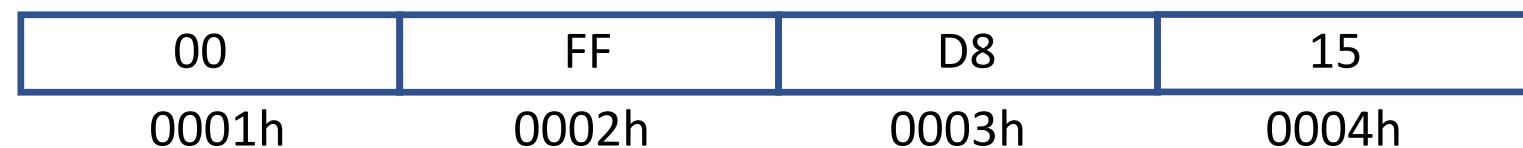
Variable value

• דוגמא:

big/littleEndian

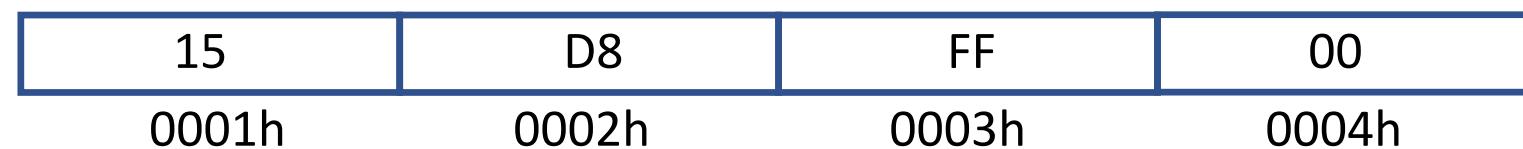
- נניח משתנה Word (DD) בזיכרון (4 תאים בגודל Byte). קיימות 2 דרכים לשמר את המידע בתאים:
 - **byte-Little endian**- שומרים את ה byte שהוא LSB's בכתובת הנמוכה בזיכרון ואת ה-byte שהוא MSB's אחרון בזיכרון:

NUMBER DD 15D8FF00h



- **byte-Big endian**- שומרים את ה byte שהוא MSB בכתובת ה"נמוכה" בזיכרון ואת byte שהוא LSB בכתובת ה"גבוהה" בזיכרון:

NUMBER DD 15D8FF00h



חשיבות לזכור!

- בקורס אנחנו לעבוד אר ורך עם **Little endian**
- טיפ נחמד כדי לזכור:
 - מספר 'נמוך' – לכתובת 'נמוכה'
 - מספר 'גבוה' – לכתובת 'גבוהה'

נספח

- חיסור בינארי: כמו חיסור רגיל בטור:

$$\begin{array}{r} & \overset{(2)}{0} \\ - & 11011001 \\ - & 11000100 \\ \hline & 00010101 \end{array}$$

- שיטה למעבר למשלים ל₂:

- עוברים מימין לשמאל, את האפסים שלפני ה1 הראשון משאים כמו שהם, גם את ה1 הראשון משאים. את הביטים שאחרי ה1 הראשון הופכים:

011001000

100111000

References

- http://www.cs.technion.ac.il/~tamer/Atam/2006_t01_2.pdf

מיקרו-מעבדים וشبת אסמלר

תרגול מס' 2 – DOSBOX

מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן



הקדמה - למה צריך DOSBOX?

- זהוי סביבת סימולציה.
אנחנו בעצם מודמים את מחשב שמכיל מעבד 8086
(= Disk Operating System) DOS ומריץ את מערכת הפעלה הישנה
- סביבת הסימולציה מאפשרת לדמות סביבת עבודה "אמיתית" Caino היה לנו כרטיס מחשב עם 8086, למשל:
 - פסיקות חומרה (מקלדת / משען חיוני Timer)
 - תכונות מודולרי
- בעצם, זו הדרך שלנו לדמות פעילות של 8086 דרך המחשב שלנו, שהמעבד שלו הוא לא 8086.

התקנה

- התקינו את האקסזוב (גרסת 0.74)
<https://www.dosbox.com/download.php?main=1>
- צרו תיקייה עם שם באנגלית שלא עולה על 8 תווים (לצורך הדוגמאות, נבחר 6086\C)
(עדיף שהתיקייה תהיה ישרה בכוון: C או בכוון אחר, המערכת לא תוכל לנתר לתיקייה שיש בשם שלה רווחים, כמו שאפשר היום במערכות אחרות)
- כעת, כניסה לפיאצה והורידו את Masm6.11 ('Resources → General Resources' תחת 'MasM6.11')
- חלצו את הקובץ לתוך 6086
- כעת, קבצי התוכנה הרלוונטיים מצויים בתיקייה 'Bin' (שאר הקבצים הם הסברים\דוגמאות, בתכלס לא תצטרכו אותם)
- נעביר את כל התוכן ל6086 (מי שרצה יכול להשאיר בחוין ולאחר מכן לנתר בהתאם לפי הצורך)

התקנה (המשר)

גודל	סוג	תאריך שינוי	שם
458 KB	קובץ STS	15/02/2017 11:42	CURRENTSTS
125 KB	ישום	09/02/2017 17:44	CV
45 KB	ישום	09/02/2017 17:45	CVPACK
385 KB	הרחבת ישום	09/02/2017 17:45	DMW0.DLL
104 KB	הרחבת ישום	09/02/2017 17:44	DOSXNT
68 KB	הרחבת ישום	09/02/2017 17:44	EED1CXX.DLL
108 KB	ישום	09/02/2017 17:45	EMD1D1.DLL
39 KB	ישום	09/02/2017 17:45	EMM386
19 KB	ישום	09/02/2017 17:45	EXEHDR
22 KB	קובץ ERR	09/02/2017 17:45	EXP
243 KB	ישום	09/02/2017 17:45	H2INC.ERR
55 KB	ישום	09/02/2017 17:45	H2INC
14 KB	קובץ מערכת	09/02/2017 17:45	HELPMAKE
42 KB	ישום	09/02/2017 17:45	HIMEM.SYS
59 KB	ישום	09/02/2017 17:45	IMPLIB
199 KB	ישום	09/02/2017 17:45	LIR
48 KB	ישום	09/02/2017 17:46	LINK
10 KB	קובץ ERR	09/02/2017 17:44	MASM
			ML.ERR

- שימוש לב שיש לכם 3 קבצים ספציפיים:
Masm, Link, CV

- עוד נדבר עליהם
ב המשר...

תכנית אסמלר

- נועזק באסמלר של 86x: (בפרט וויריאנט 8086)

```
.model small
.data
    INFO DW 1000h
.stack 100h
.code
START:
    MOV AX, 0
    MOV BX, AX
    MOV AH, 4CH
    INT 21h
END START
```

- **תכנית אסמלר לדוגמא:**

- בשיל להריצ את התכנית נצטרך סביבה שمدמה את
את מעבד 8086 – DOSBOX

איך מרכיבים תוכנית?

בגadol, המסלול שקדם עובר לפני ש'הופר' לתוכנית הוא:

- א. קומpileציה
- ב. קישור
- ג. הרצה

לשם השוואה, כשהעבדנו עד היום בסודם Visual Studio, המערכת הייתה מבצעת את על התהליך הנ"ל מאחוריו הקלעים, וכבר היינו מקבלים תוכנית 'מוגמרת'.

asm.exe הוא בעצם הקומPILEר שלנו, exe.link ישתמש ל קישור, ובסוף נקבל קובץ להרצה.

איך מריםים תוכנית? (המשך)

סיומת של קובץ הרצה תהיה ".exe" (בדרכן כלל) או ".com" (בקובץ שהוא *small tiny* – נגיעה לזה בהמשך).

הסיומת *.exe* אולי מוכרת לחלקכם, בעצם זהה הסיומת של הרבה מאוד תוכניות שאנו חנו מרים גם היום (קצתו התקינה זו הדוגמא הנפוצה ביותר)

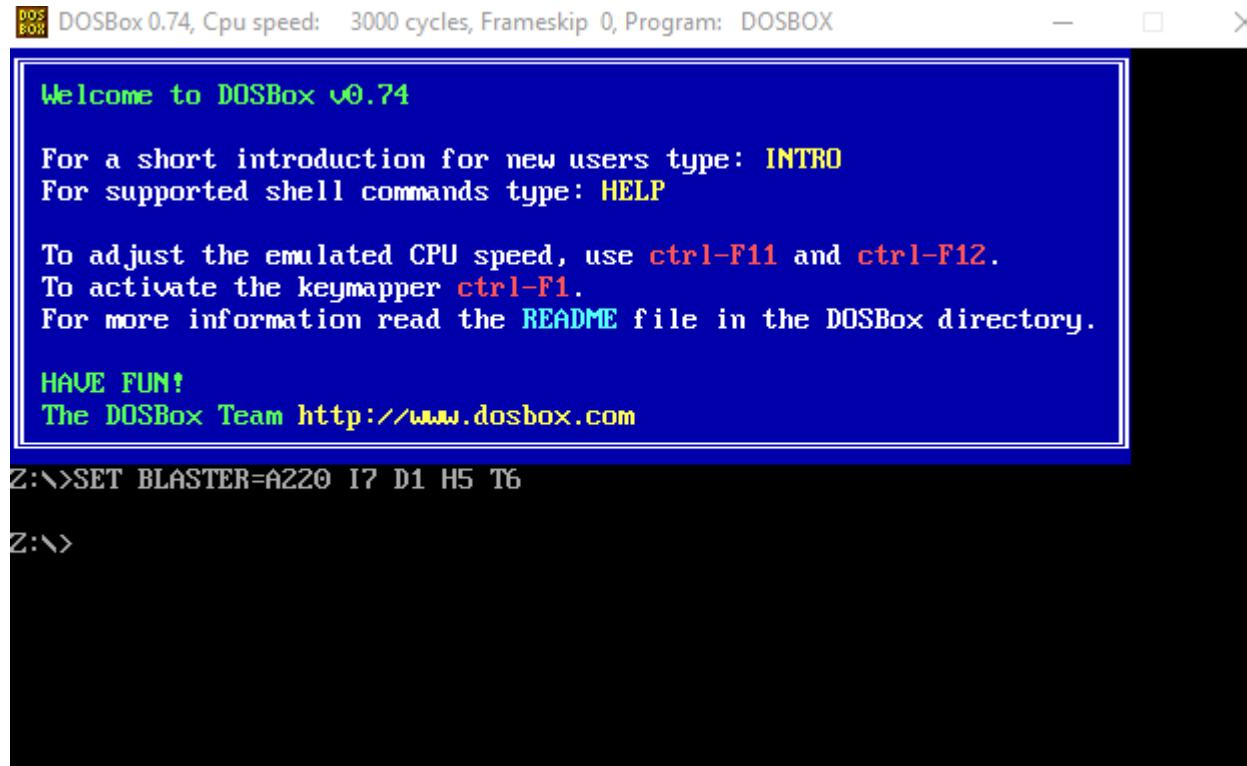
כמו כן, יהיה את קובץ הקישור (*.obj*) וקובץ האסמבלר (*.asm*).

בקובץ *hmasm* נכתב את הקוד. נלמד בהמשך על משמעות קובץ הקישור.

- כל קובץ הרצה יודע לקבל פרמטרים מסוימים.

בקובץ *masm.exe*, *LINK.exe* – הן בעצם תוכניות שמקבלות כפרמטר שם של קובץ.

سبיבת הDOSBox – פקודות בסיסיות



מן הסתם, נרצה לגשת לקבצים במחשב שלנו, כדי להריץ את קובץ הwsma שעבדנו עליו כל כך קשה.

לכן נרצה שהמערכת 'תכיר' את הנתיב בו הקובץ נמצא.

כשאנחנו בDOS, אנחנו בעצם 'מנוטים' בתיקיות המחשב. בכל פעם נרצה להגיע לקובץ הקוד שלנו ולקומpileר, כדי להריץ דרכו את הקוד.

למשל, בדוגמא כאן, ניתן לראות את הכתיבה <\z. זה אומר שכרגע אナンחנו בכוון שהמערכת 'מכירה' אותו ככוון z.

אם נרצה להגיע לכוון שהמערכת מכירה c"C" – פשוט נכתב בשורת הכתיבה "c"

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use **ctrl-F11** and **ctrl-F12**.
To activate the keymapper **ctrl-F1**.
For more information read the **README** file in the DOSBox directory.

HAVE FUN!
The DOSBox Team <http://www.dosbox.com>

```
Z:\>SET BLASTER=A220 I7 D1 H5 T6
Z:\>mount c c:\ 
Mounting c:\ is NOT recommended. Please mount a (sub)directory next time.
Drive C is mounted as local directory c:\ 
Z:\>c:
C:\>cd 8086
8086\
```

כדי שנוכל לגשת לקבצים של המחשב דרך DOS נצרך 'לקשר' בין ה'מעבד' שלנו למחשב.

פועלה זו תעשה ע"י פקודה Mount

זכורים את התיקייה שיצרנו?
از אנחנו רוצים שהמערכת תכיר בה.

למשל, הפקודה

mount c c:\8086

אומרת: "כשאני ארצה לגשת לכונן c ב-DOSBOX,
אני מתוכנן שתיגש לנットיב 6086\c:
במחשב שבו אני משתמש"

לאחר ביצוע הפקודה `mount c c:\8086`, **כל** כונן c של המחשב שלנו 'מופה' לכונן C של ה-X-BOX
וניתן לגשת דרך DOSBOX לכל תיקייה במחשב שלנו.

שיםו לב שהמערכת שלחה זההה שלא כדאי למפות את כל כונן c, אלא עדיף **לציין תיקייה ספציפית** בכונן.
ambilahim lma?

לאחר מכן, ניגש לכונן שמייפינו כ"c" ע"י:

מספר פקודות DOS שימושיות

הציג עץ תיקיות

```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX
TMP      OBJ          598 02-08-2017 9:56
TMP2     ASM          7,915 26-06-2017 22:48
TMP2     EXE          1,071 01-10-2017 18:09
TMP2     OBJ          897 02-10-2017 19:17
TMP2TIME EXE          959 26-06-2017 17:36
TMP3     ASM          5,470 26-06-2017 15:02
TMP3     EXE          951 20-06-2017 11:00
TMP3     OBJ          813 01-10-2017 18:08
TMP4     ASM          7,030 27-06-2017 11:56
TMP4     EXE          1,162 27-06-2017 16:29
TMP4     OBJ          1,294 01-10-2017 18:02
TMP4     TXT          3,553 25-06-2017 14:53
TMP42    ASM          716 25-06-2017 17:15
TMP42    OBJ          203 01-10-2017 18:09
TMP5     ASM          3,266 27-06-2017 10:24
TMP5     EXE          746 27-06-2017 10:25
TMP5     OBJ          534 01-10-2017 18:08
TMP6     ASM          141 15-10-2017 12:27
TMP6     EXE          526 15-10-2017 0:43
TMP6     LST          1,369 15-10-2017 12:13
TMP6     OBJ          134 15-10-2017 13:25
  57 File(s)           3,572,298 Bytes.
   7 Dir(s)            262,111,744 Bytes free.

C:\8086>
```

- לשם הציג תוכן התיקייה
שבה אנחנו נמצאים כרגע
במערכת, נכתב "dir".
- קיבל רשימה של כל
הקבצים בתיקייה עם
פרטים נוספים על כל קובץ

מעבר בין תיקיות

- **נשתמש בפקודה "cd" כדי לעבור לתיקייה כלשהי.**
- **ኖכל לעשות זאת ע"י כתיבת הנתיב המדויק של התיקייה.**
- **עם זאת, אם נכתב את שם התיקייה בלבד, ניכנס לתיקייה רק אם היא עצמה נמצאת במקום שאנו 'נעמוד' עליו באותו רגע.**
- **למשל, אם אנחנו בכון c (זכרים? מיפויו וכתבנו לאחר מכן "c:"), מספיק שנכתב "cd 8086" כדי להגיע לתיקיית היעד שלנו, 8086\c.**
- **הפקודה "cd.." תשמש אותנו לחזרה אחורה בנתיב התיקיות**
- **למשל, אם אנחנו עכšíו ב"8086\c" ורשמנו "cd..", אז זה יחזיר אותנו לשורש כון c. כמובן, לנטיב \c.**

אז מה עכשווין?

גודל	סוג	תאריך שינוי	LOC
1 KB	קובץ CRF	12/06/2017 11:57	L.CRF
64 KB	ישום	08/06/2017 12:25	LINK
1 KB	Source Browser D...	18/06/2017 18:55	LINKTMP3
101 KB	ישום	10/06/2017 23:29	MASM
1 KB	ישום	18/06/2017 18:55	OBJ
1 KB	Assembler Source	09/02/2017 15:31	PROG1
1 KB	ישום	09/02/2017 15:31	PROG1
1 KB	Source Browser D...	09/02/2017 15:31	PROG1
1 KB	Assembler Source	10/02/2017 05:11	PROG1I
1 KB	ישום	10/02/2017 05:11	PROG1II
1 KB	Source Browser D...	10/02/2017 05:11	PROGRAM_
1 KB	Assembler Source	31/01/2017 18:37	PROGRAM_
1 KB	ישום	11/06/2017 00:05	PROGRAM_
1 KB	Source Browser D...	11/06/2017 00:05	TASM
105 KB	ישום	10/02/2017 05:11	TD
476 KB	ישום	11/06/2017 16:20	TMP
4 KB	Assembler Source	25/06/2017 15:22	TMP
1 KB	ישום	02/08/2017 09:56	TMP
1 KB	Source Browser D...	02/08/2017 09:56	TMP2
8 KB	Assembler Source	26/06/2017 22:48	TMP2
2 KB	ישום	17/09/2017 17:57	TMP2
1 KB	Source Browser D...	01/10/2017 17:47	TMP2
1 KB	ישום	26/06/2017 17:36	TMP2TIME
6 KB	Assembler Source	26/06/2017 15:02	tmp3

- נשמר את קובץ הטקסט בו רשםנו את הקוד
קובץ אסמלבי (סיומת mas).
- אנחנו נקמפל את הקובץ (באמצעות הקובץ MASM)
ואז יוצר לנו קובץ חדש עם סיומת obj
- נקשר את קובץ הobj (באמצעות הקובץ 'LINK')
ואז יוצר לנו קובץ חדש עם סיומת exe.
- כדי להריץ קובץ - פשוט רושמים את שם הקובץ
(במידה ואנחנו 'עומדים' בתיקייה שבה נמצא הקובץ)
- לכן, השלב השלישי יהיה ל כתוב פשוט את שם הקובץ.

העברה בשירותת הקומpileציה

כעת, נבצע בפועל את כל הפעולות עליון דיברנו

- פתחו את הXBOX DOS
- בהינתן תיקיית יעד (למשל, 8086), כתבו את השורות הבאות:

mount c c:\8086

c:

ml /Zm Prog1.asm ⇒ –compiling & Linkage
(CV) Prog1.exe ⇒ Run program (with 'cv' ⇒ Debug mode)

- שימושו לב שיש אפשרות להריץ את התוכנית דרך CV ולא באופן ישיר.
זה בעצם כמו שביז'ואל יכלתם להריץ ישירות, ויכלתם לדבג.
- CV זו בעצם תוכנת הדיבוג שלנו.

- כאשר אתם מבצעים את פקודת 'MASM', מתבצע הידור (קומפילציה).
- אם יש שגיאות בקוד, הן יופיעו בדרך כלל באופן מפורש (בוגרמא משMAIL למשל, ניתן לראות שלא הוגדרו המשתנים שרשומים, כאשר בסוגרים יש את מספר השורה בה יש בעיה בקוד).
- כאשר אין שגיאות בקוד, יוחזר הפלט כמו בדוגמה מימין

```
C:\8086>masm tmp6.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987. All rights reserved.

Object filename [tmp6.OBJ]:
Source listing [NUL.LST]:
Cross-reference [NUL.CRF]:
tmp6.asm(36): error A2009: Symbol not defined: M_SECONDS
tmp6.asm(43): error A2009: Symbol not defined: M_SECONDS
tmp6.asm(45): error A2009: Symbol not defined: SECONDS
tmp6.asm(46): error A2009: Symbol not defined: M_SECONDS
tmp6.asm(48): error A2009: Symbol not defined: SECONDS
tmp6.asm(50): error A2009: Symbol not defined: MINUTES
tmp6.asm(51): error A2009: Symbol not defined: SECONDS
tmp6.asm(60): error A2009: Symbol not defined: MINUTES
tmp6.asm(71): error A2009: Symbol not defined: SECONDS
tmp6.asm(79): error A2009: Symbol not defined: M_SECONDS

51558 + 448602 Bytes symbol space free

0 Warning Errors
10 Severe Errors
```

```
C:\>cd 8086

C:\8086>masm tmp2.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987. All rights reserved.

Object filename [tmp2.OBJ]:
Source listing [NUL.LST]:
Cross-reference [NUL.CRF]:

51558 + 448602 Bytes symbol space free

0 Warning Errors
0 Severe Errors

C:\8086>
```

קובץ LST

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

```
Z:\>SET BLASTER=A220 I7 D1 H5 T6
Z:\>mount c c:\

Mounting c:\ is NOT recommended. Please mount a (sub)directory next time.
Drive C is mounted as local directory c:\

Z:\>c:
C:\>cd 8086
C:\8086>masm tmp6.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987. All rights reserved.

Object filename [tmp6.OBJ]:
Source listing [NUL.LST]: MY_LST_FILE
Cross-reference [NUL.CRF]:

51194 + 465350 Bytes symbol space free

0 Warning Errors
0 Severe Errors

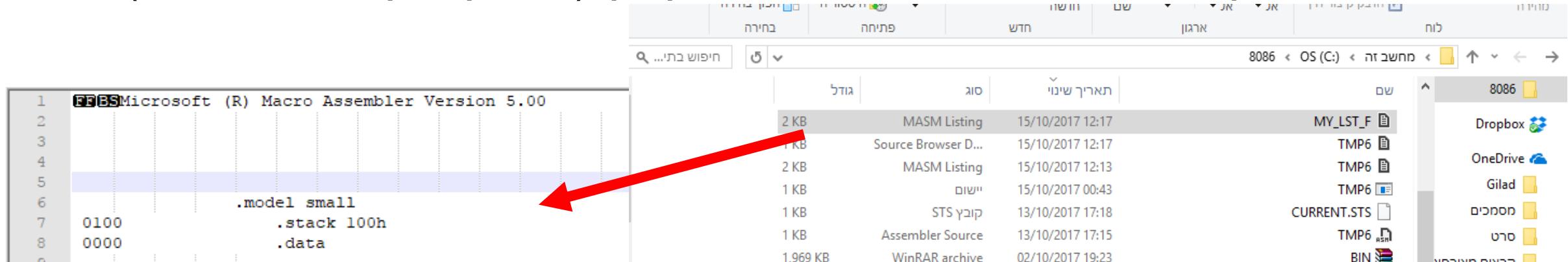
C:\8086>_
```

- את שאר השלבים נבע באותו אופן כמו שהזכרנו [כאן](#)

- ישנה אפשרות לקבל בקובץ אחד את הקוד עם פלט הקומpileציה (ובין היתר יפרט את השגיאות בקוד).

- במקום ללחוץ 3 פעמים Enter, בפעם השנייה שנטבקש להזין קלט, נכניס שם כלשהו, זה יהיה שם קובץ LST שלנו.

• נכנסים לתיקיית היעד שלנו ופתחחים את הקובץ (מודול דרך Notepad+)



• ניתן לראות משמאל את מיקום הקוד בזיכרון
ביחס לสมగנט הקוד (CS)

כרגע הקוד ללא שגיאות.

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

zmbv.dll zmbv.inf tmp42.asm TMP2.asm TMP6.asm TMP.ASM TMP6.LST MY_LST.FLST

1 FFB5Microsoft (R) Macro Assembler Version 5.00 10/15/17 12:27:46 Page 1-1

```

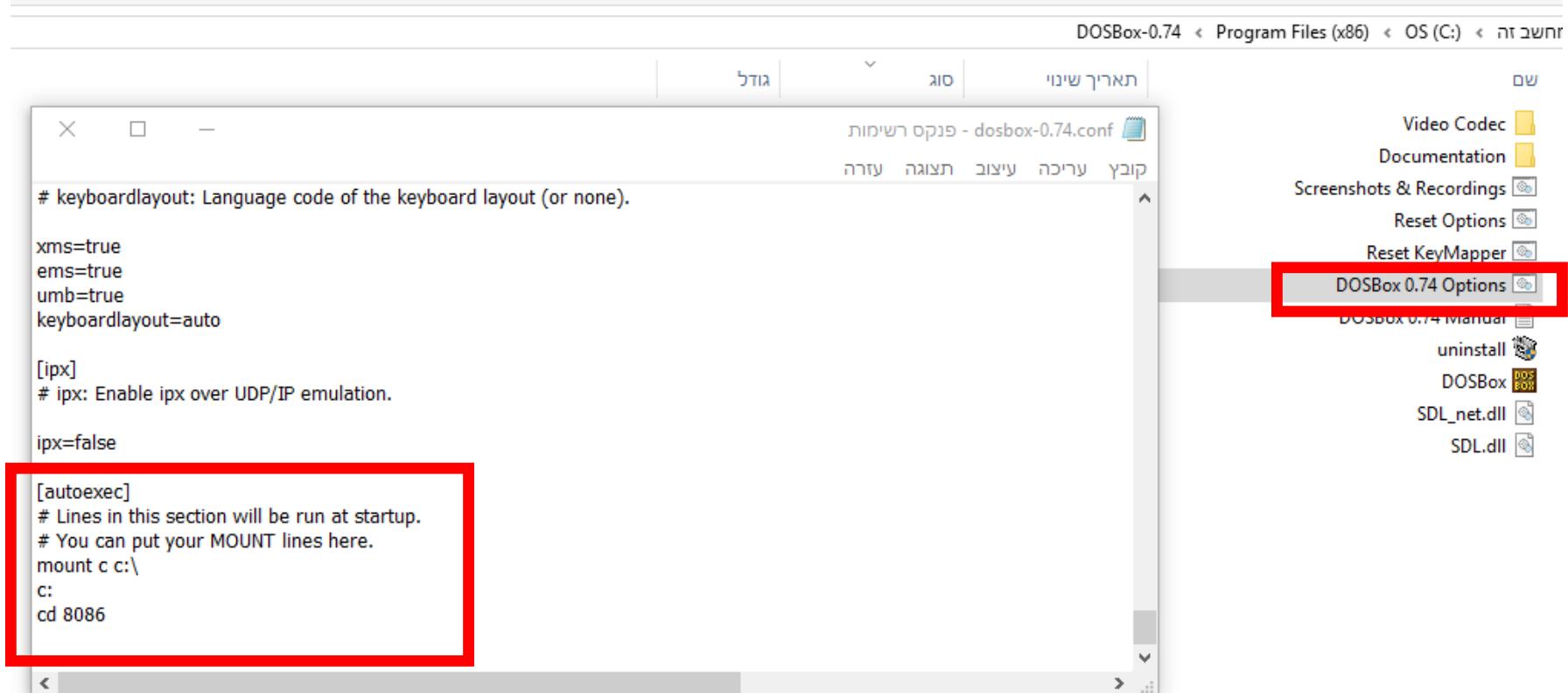
6 .model small
7 0100 .stack 100h
8 0000 .data
9
10 0000 .code
11 0000 B8 ---- R      mov ax, @data
12 0003 8E D8          mov ds, ax
13
14 0008 8A C3          mov al, bx
15 tmp6.asm(12): warning A4031: Operand types must match
16
17 000A B4 4C          mov ah, 4ch
18 000C CD 21          int 21h
19 000E end
20
21 FFB5Microsoft (R) Macro Assembler Version 5.00 10/15/17 12:27:46
22 Symbols-1
23
24
25 Segments and Groups:
26
27           Name        Length  Align  Combine Class
28
29 DGROUP . . . . . . . . . . . . GROUP
30 _DATA . . . . . . . . . . . 0000 WORD PUBLIC 'DATA'
31 _STACK . . . . . . . . . . . 0100 PARA STACK 'STACK'
32 _TEXT . . . . . . . . . . . 000E WORD PUBLIC 'CODE'
33
34 Symbols:
35

```

- כעת, נשים לב, שכאשר שינו את האוגר המקובל מאא ל-א (דבר הגורר שגיאת קומpileציה), קיבלנו הערת בקובץ שלנו, במקום המתאים.

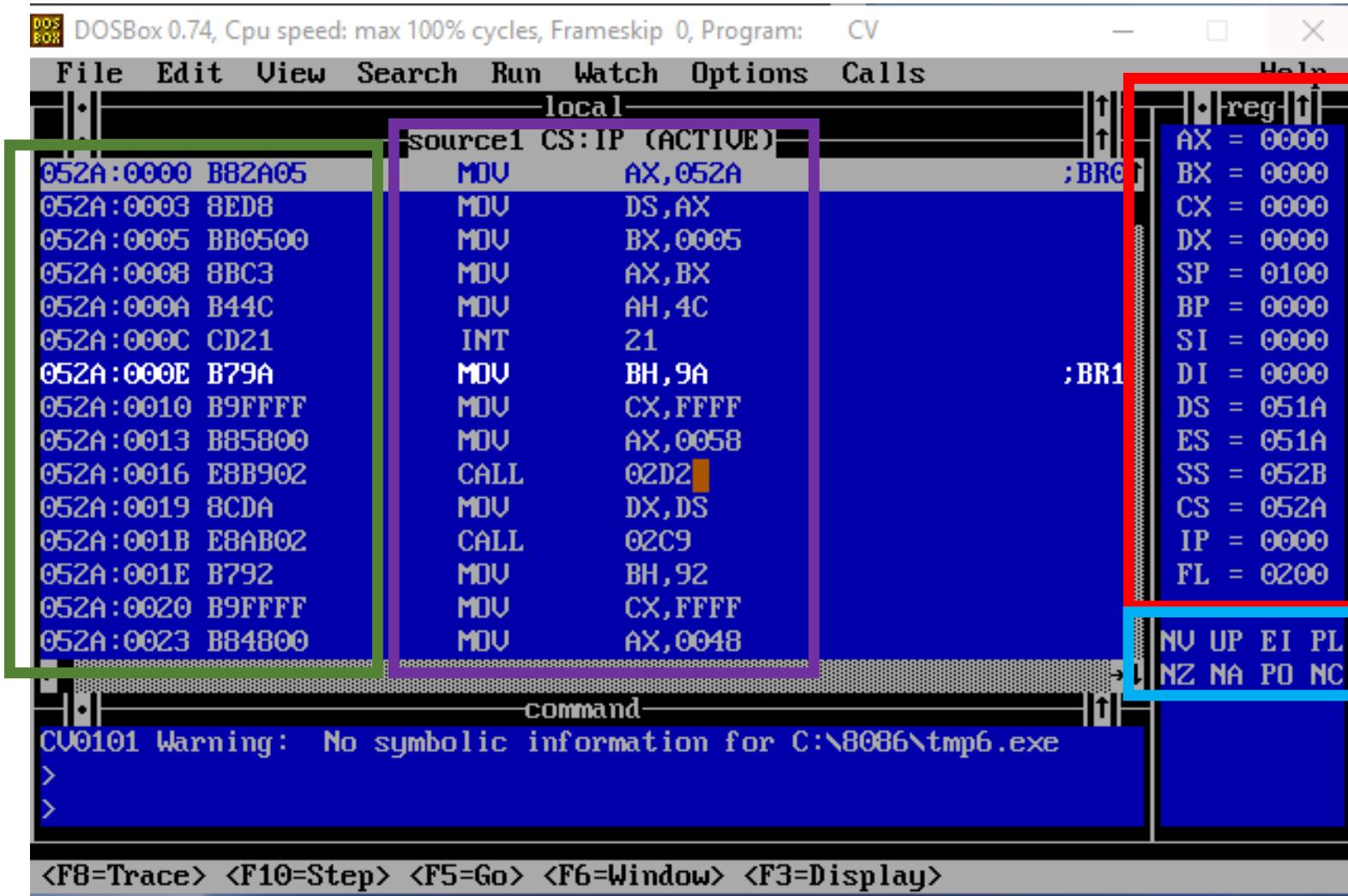
טיפ – השימוש בקונפיגורציה!

- בכל כתיבת קוד יהיה צורך בכתיבת הפקודות שוב ושוב, ישנה אופציה להזין מראש את הפקודות הנ"ל או לפחות את חלקן.
- היכנסו לתיקיה של התוכנה, לחצו על הקובץ המסומן בתמונה, כתבו את הפקודות הרלוונטיות במקום שמסומן באדום.



- עצה, בכל פעם שתיכנסו למערכת, הפקודות הללו יבוצעו באופן אוטומטי.

The CV debugger



- מצב האוגרים

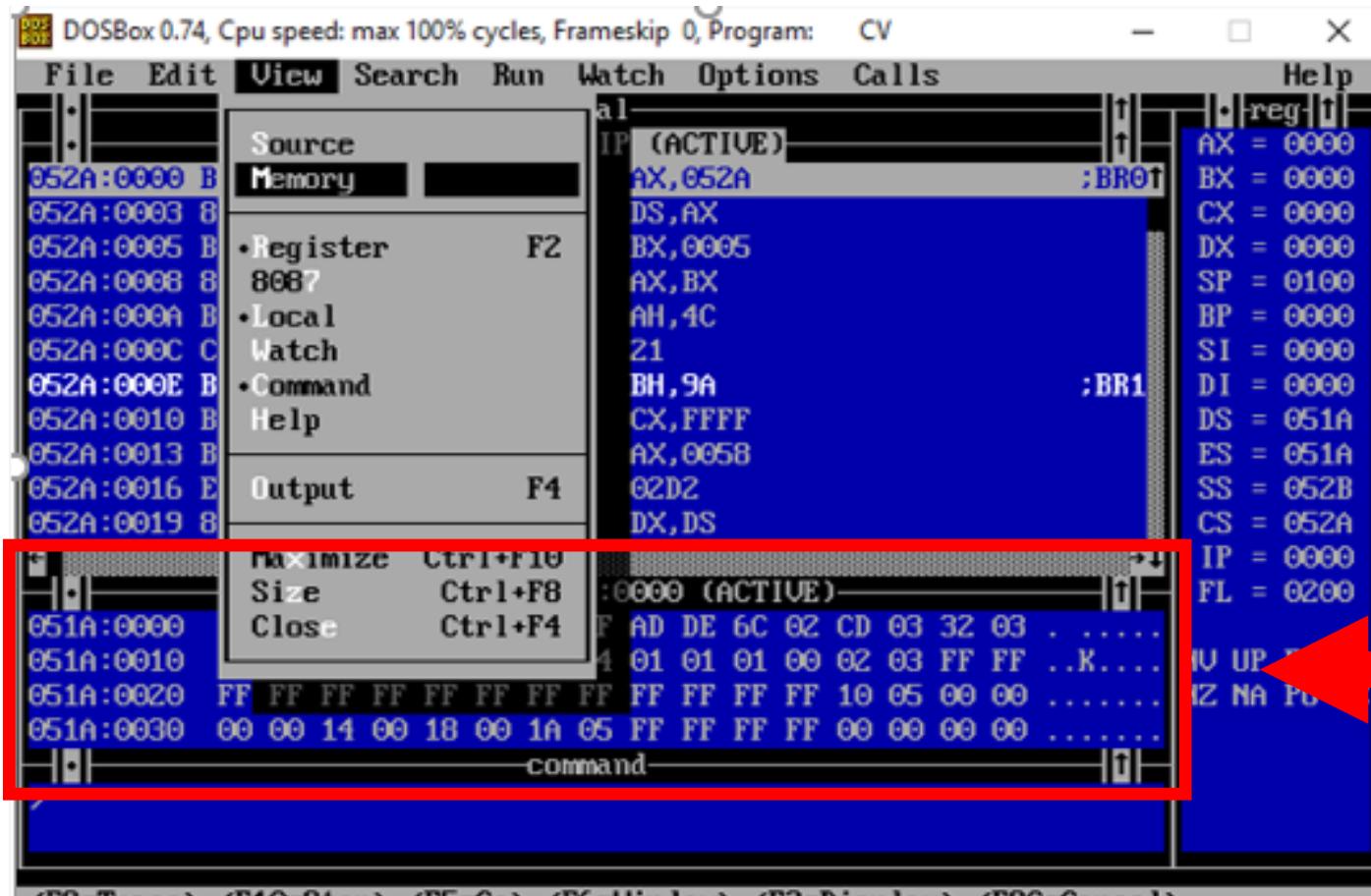
- פקודות התוכנית
(הנכחית מסומנת באפור)

- Address + Opcode •

- אוגר הדגלים

CAFIAH B'ZICHRON

View ⇒ Memory •



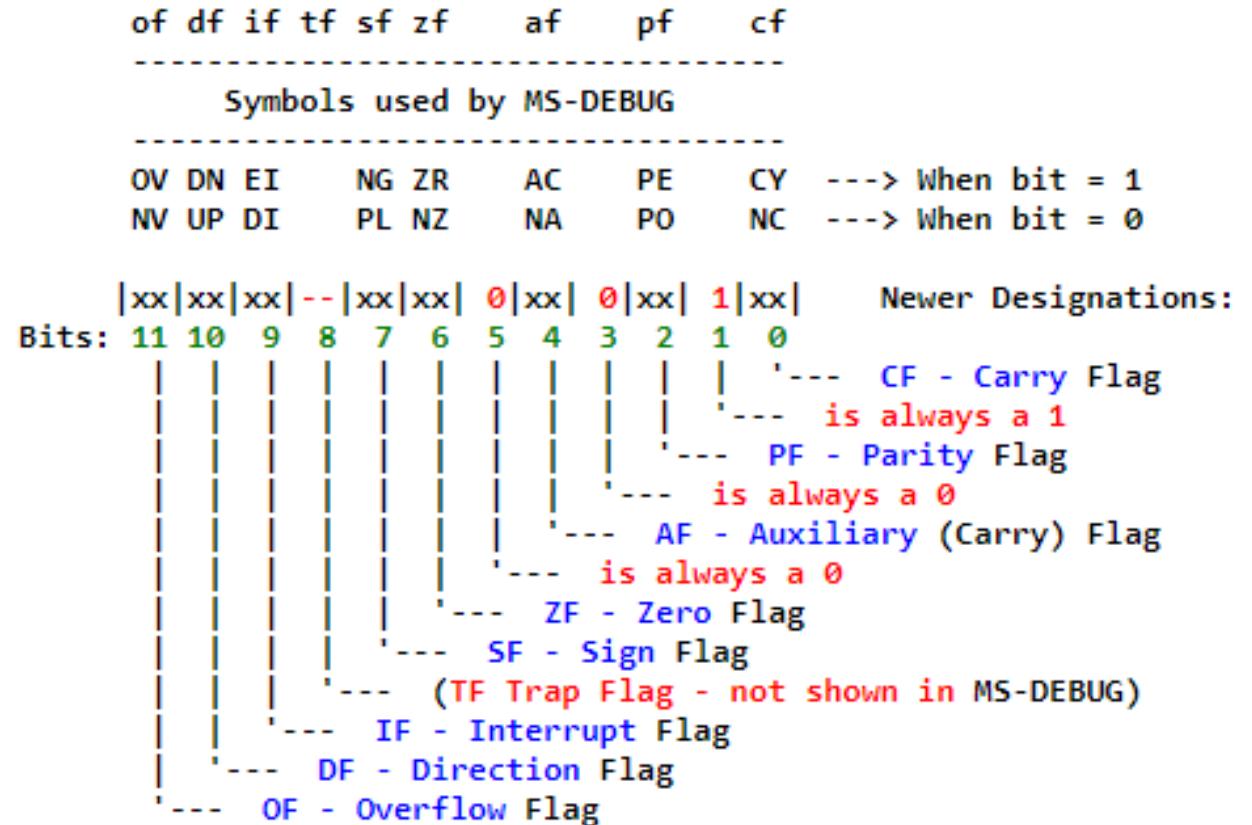
לחצני קיזור

- F4 - חזרה למסך ה DOSBOX (תצטרכו כדי לבדוק הדפסות לזכרון המסך)
- F5 – הרצת התוכנית (Go)
- F8 – הרצת פקודה אחת בקוד (Trace)
- F9 – הכנסת Breakpoint
- F10 – 'כניסה לעומק' – כמו שkopצים לשגרה בקובץ אחר (ho in)
- במידה ונתקע – ללחוץ על כפתור ה windows ולחזור DOSBOX – סגירת הCV וחזור ל DOSBOX Alt+f4

מקרה – דגלים

(חזרה)

- NV – Overflow flag
- UP – Direction flag
- DI – Interrupt\enable flag
- PL – Sign flag
- NZ – Zero flag
- NA – Adjust flag
- PO – Parity flag
- NC – Carry flag



- Source: <http://thestarman.pcministry.com/asm/debug/8086REGs.htm>

מיקרו-מעבדים ושפת אסמלר

תרגול מס' 3 – פקודות ומבנה תוכנה

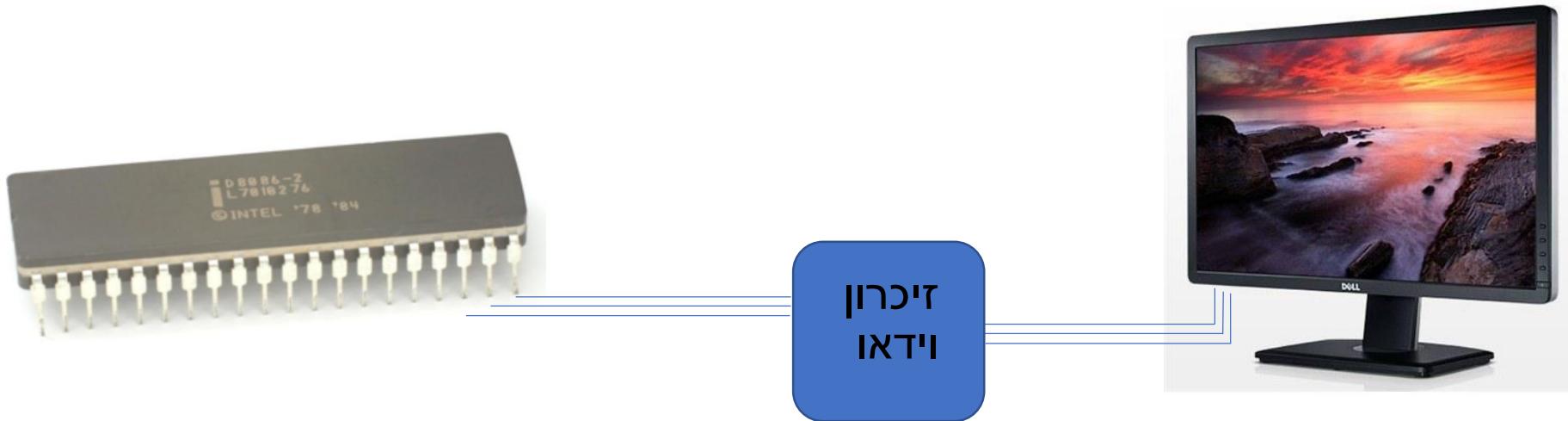
מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן



מהו מעבד

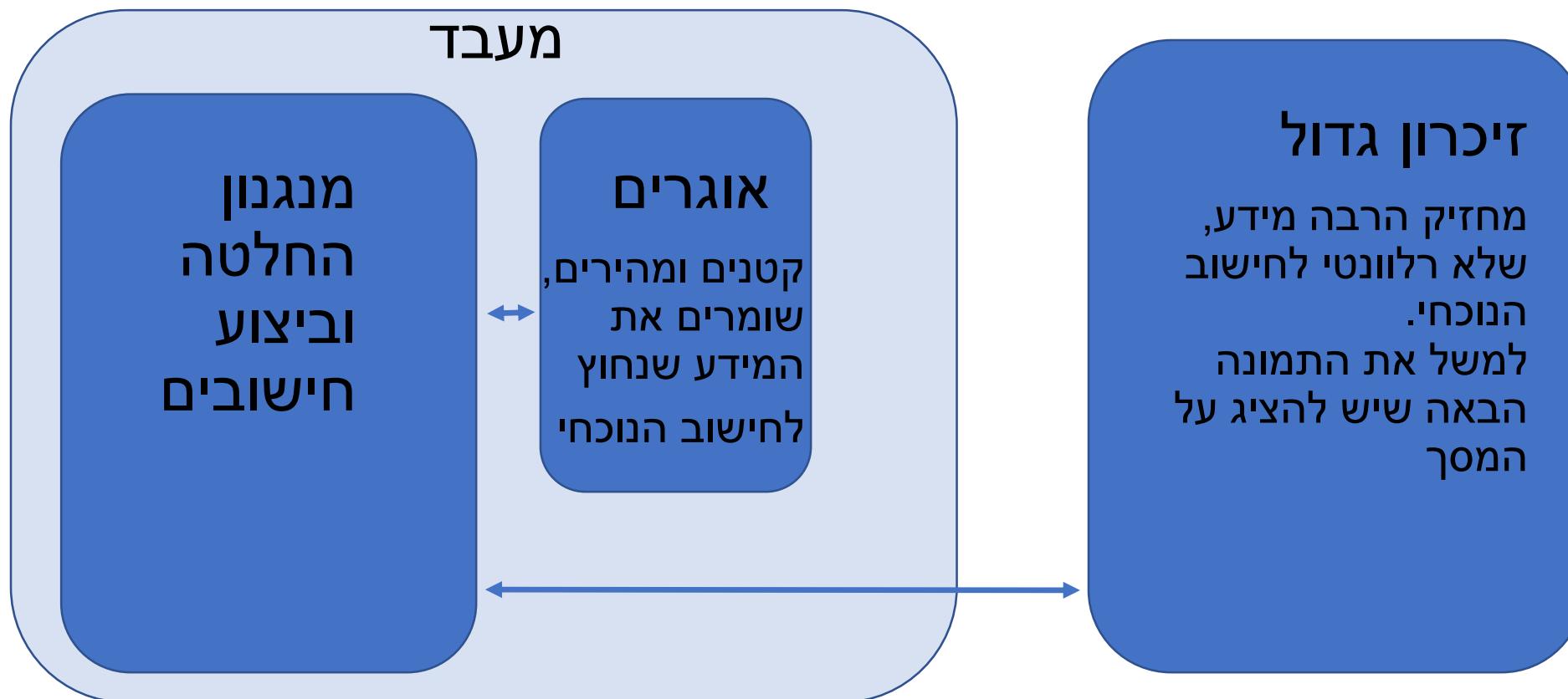
- מעבד הוא רכיב שמבצע חישובים ומחליט מה יבצע הרכיבים השונים שמחוברים למחשב, למשל, מה יציג המסך.
- כיצד המעבד אומר למסך מה להציג? המסך הוא אוסף של פיקסלים, המעבד קובע את ערכו של כל פיקסל ואומר אותו למסך.



- הדריך שהמעבד אומר למסך מה להציג היא ע"י כתיבת הערכים של הפיקסלים לזיכרון משותף, אז המסך קורא את הערכים מהזיכרון ומציג אותם.

כיצד המעבד מבצע חישובים

- המעבד מורכב מיחידות בקרה שמחלitas איזה חישוב לבצע, ויחידות שביצעות את החישוב בפועל (כמו ALU), ייחדות החישוב מביצעות פועלות על מידע שנמצא בתוך האוגרים או בתוך הזיכרון



אוגרים

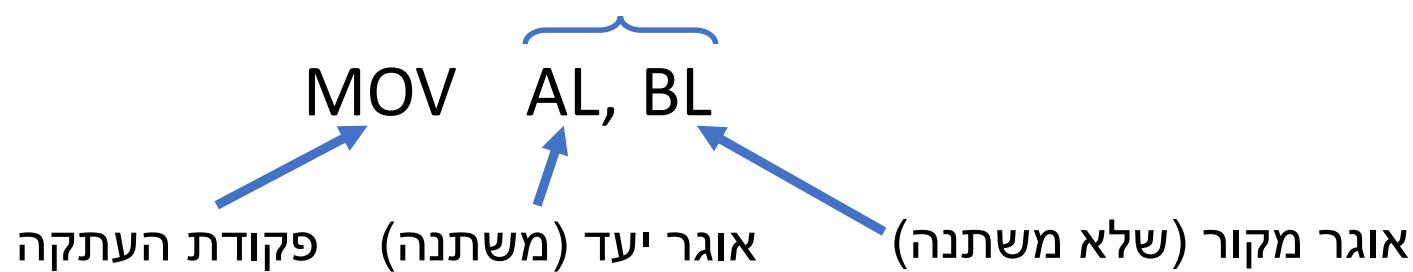
- **אוגר** הוא יחידה פיזית שיושבת במעבד, והמעבד עושה פעולות חישוב על המידע שנמצא באוגרים.
- **זיכרון** יושב מחוץ למעבד, וכדי שהמעבד יוכל לבצע פעולות על מידע מהזיכרון, צריך תחילה להביא אותו לאחד האוגרים.



פקודות

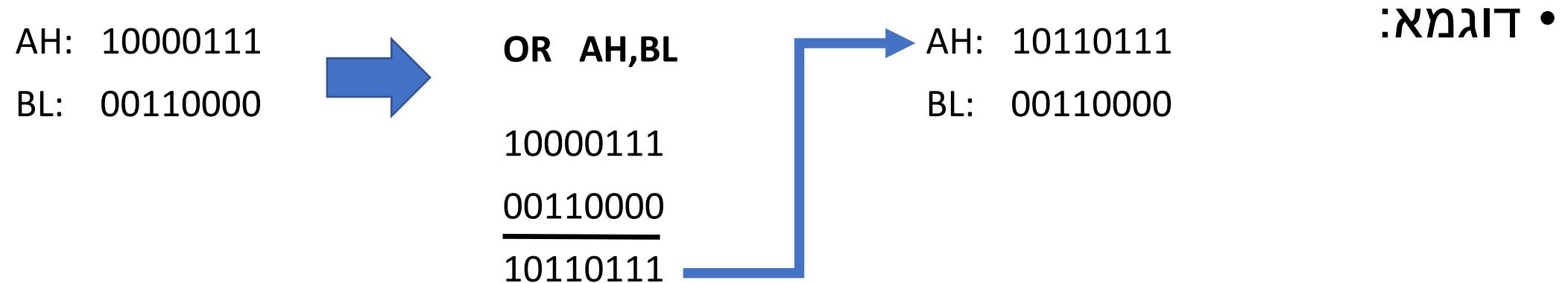
- פקודות יכולות להתבצע על אוגרים שלמים (...X,BX) או על חלקים של אוגרים (...AH,AL)
- פקודה מורכבת מטייאור הפקודה (...ADD,MOV) ומאופרנדים, כאשר יכולים להיות 0, 1 או 2 אופרנדים בפקודה.
- כאשר יש 2 אופרנדים בפקודה, הראשון הוא היעד (למשל האגר שאלוי מעתיקים) והוא ישתנה אחרי הפקודה, והשני הוא המקור (למשל האגר שמננו מעתיקים) והוא לא ישתנה אחרי הפקודה.

- דוגמא:



פקודת OR

- פקודת OR לוגי, בין **ביטים** של שני אוגרים (bitwise), התוצאה נשמרת באוגר היעד (שכתוב ראשון)



- שימוש אפשרי: למשל כאשר רוצים להدلיק ביטים מסוימים באוגר AH בלי לשנות את הביטים האחרים

פקודות נוספות עם 2 אופרנדים

- **פקודות אריתמטיות: מתייחסות לנთון כמספר**

ADD **ax, bx** חיבור 2 מספרים של 16 סיביות כאשר התוצאה נשמרת באוגר שכתוב ראשון

SUB **cx, dx** חיסור שני מספרים

MUL / DIV מאפשרות הכפלת \ חילוק של מספרים unsigned

|MUL / **|**DIV מאפשרות הכפלת \ חילוק של מספרים signed

- **פקודות לוגיות: מתייחסות לנთון כאוסף של ביטים**

AND **ax, bx** פעולה לוגית בין הביטים של המספרים כאשר התוצאה נשמרת באוגר שכתוב ראשון

XOR **dl, bh** XOR בין הביטים של המספרים

פקודות עם אופרנד אחד

- פקודות אריתמטיות שמבצעות פעולה על נתון יחיד:

INC **ax** $(AX := AX + 1)$ הוספה 1 למספר

DEC **ax** $(AX := AX - 1)$ חיסור 1 מהמספר

NEG **al** הפיכת סימן של מספר בשיטת המשלים ל 2

10111000  01000111 + 00000001

- פקודות לוגיות שמבצעות פעולה על נתונים יחיד:

NOT **ah** הפיכת הביטים של המספר

10111000  01000111

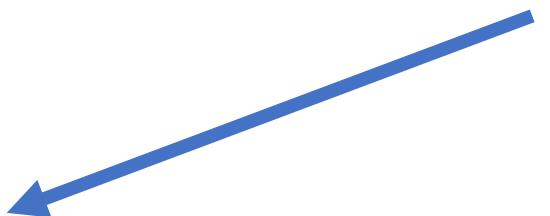
מבנה תוכנית בסיסית

```
.model small
.stack 100h
.code
    ; setting data segment
    mov ax, @data
    mov ds, ax

    ; code
    mov bx, 0
    mov ax, bx
    add ax, bx
    mul bh, al
    inc ax

    ; return to OS
    mov ax, 4c00h
    int 21h
end
```

הפקודות שלנו רשומות בחלק זהה



- מה יקרה אם נריץ את התוכנית הזאת?

טבלת ASCII

- קידוד של אותיות וסימנים אחרים לקוד מספרי

Decimal	Hex	Char
32	20	Space
33	21	!
34	22	"
35	23	#
36	24	\$
37	25	%
38	26	&
...

Decimal	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
...

כתיבה למסך

- כתיבה למסך מבוצעת ע"י כתיבה לאיזור הזיכרון שמתחל בכתובת $B800h$
- כדי לכתוב לאזור זה, נעדכן אוגר סגמנט (למשל ES) לערך $B800h$:

```
; setting extra segment to screen memory
mov ax, 0b800h
mov es, ax
```

- הפקודות מכוננות את אוגר ES להצביע על תחילת אזור הזיכרון המשויך
למסך.
- כעת ניתן לכתוב לתא הראשון במסך ע"י הפקודה

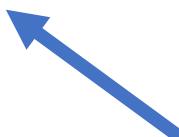
```
mov es:[0000], ax
```

רקע התו וערך התו

- המסר מחולק ל 80×25 מקומות שונים – כל אחד נשלט ע"י כתובת אחרת
- כל 2 תאים (bytes) רציפים בזיכרון המסר מאפיינים تو ייחיד
- תא אחד מכיל את הערך ה ASCII
- התא השני מכיל מידע על הצבע \ רקע \ הבוהב \ הדגשה

```
;setting ax to represent 'A' on green background
mov al, 65      ;'A' asci code
mov ah, 46d     ;green background code

;writing to screen memory
mov es:[140h], ax
```



כתבה לשגר ES מצביע עליה, בתוספת h 140 בתים (bytes) של היסט

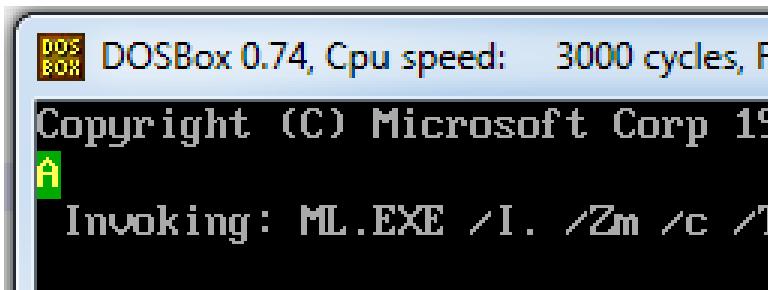
תכנית שכותבת 'A' למסך, במלואה

```
.model small
.stack 100h
.code

    mov ax, 0b800h          ; screen memory
    mov es, ax               ; set es to point at screen memory
    mov al, 65                ; 'A' in ASCII
    mov ah, 46d              ; green background
    mov es:[0A0h], ax        ; write to screen memory

; return to OS
    mov ax, 4c00h
    int 21h

end
```



הערות שמסבירות
את הנעשה בתוכנה

קוד תרגילי הבית
חייב להיות מתועד!

פקודת JMP

- לפעמים נרצה לקפוץ למקומות שונים בקוד.
- בשביל זה, משתמש בפקודה: LABEL JMP, בשונה מהפקודות הקודמות הפקודה הזאת משפיעה על זרימת הקוד, ולכן שייכת לפקודות בקרה

```
.model small
.stack 100h
.code

        mov ax, 0b800h
        mov es, ax          ;setting extra segment to screen memory
        mov ax, 2E41h          ;setting ax to represent 'A' on green background

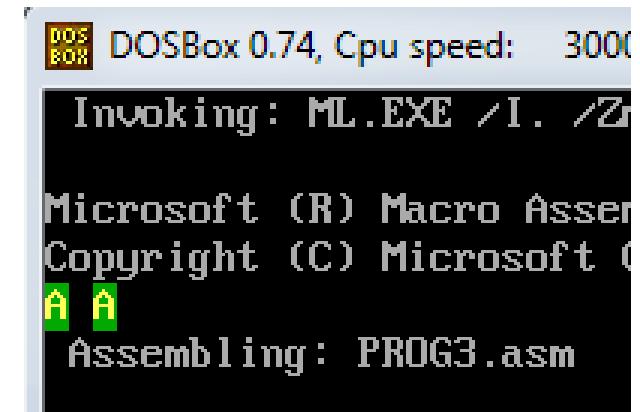
        mov es:[320h+0h],ax
        jmp L1
        mov es:[320h+2h],ax
        ;print first 'A'
        ;jump to L1
        ;print second 'A'

L1:   mov es:[320h+4h],ax
        ;print third 'A'
        ;return to OS
        mov ax, 4c00h
        int 21h

end
```

פקודת קפיצה

"לייבל" – יעד
הקפיצה



שימוש במשתנים

- כדי להציגות מקום שמור בזיכרון (לשימוש התוכנה) מגדרים **משתנה**
- המשתנים יושבים בסגמנט זיכרון מיוחד שהמחשב מנסה עבורם

```
.model small
.data
    CHR  DW  2E41h ;char we want to write
    OFST DW  320h   ;offset from start of screen memory
.stack 100h
.code
    mov ax, 0b800h
    mov es, ax
```

כדי להפוך את הקוד ליותר מסודר וקריא, נשתמש במשתנים, שיכילו בתוכם ערכים שלא נרצה לבזבז עליהם אוגר באופן קבוע, אך נרצה אותם זמינים

שימוש במשתנים

- כדי שהמעבד ידע להשתמש במשתנים, צריך לאותחל את אוגר DS לכתובה שהמחשב הקצה בשבייל המשתנים של התוכנית הנוכחית

```
.model small
.data
    CHR  DW 2E41h ;char we want to write
    OFST DW 320h ;offset from start of screen memory
.stack 100h
.code
    ;setting data segment
    mov ax, @data
    mov ds, ax

    ;setting extra segment to screen memory
    mov ax, 0b800h
    mov es, ax
```

קבעת ערכו של @data מתרחשת אחרי הקומpileציה, בזמן
טעינה התוכנית לזיכרון

הוראה מיוחדת ששואלת את
ה-loader היכן נמצא סגמנט הנתונים

שימוש במשתנים

CHR DW 2E41h
OFST DW 320h

- נכתב את התוכנית להציג 3 פעמים 'A' למסך תוך שימוש במשתנים

העתקת ערך ההיסט
לאוגר BX על מנת
לבצע בו שימוש

קידום ההיסט ב2 בתים
(bytes)

```
; setting ax to represent 'A' on green background
mov ax, CHR

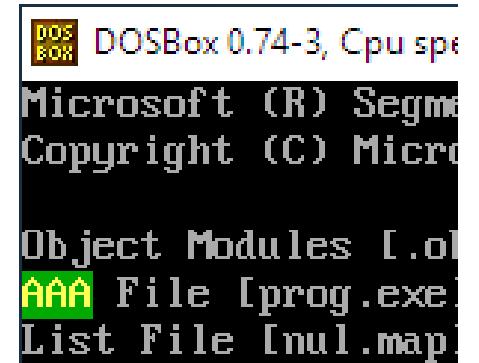
mov bx, OFST
mov es:[bx], ax ;print first 'A'
add bx, 2h
mov es:[bx], ax ;print second 'A'
add bx, 2h
mov es:[bx], ax ;print third 'A'

mov OFST, bx

; return to OS
mov ax, 4C00h
int 21h
```

end

שמירת הערך החדש של ההיסט
חזרה למשתנה



הקצת מתנה

ניתן להקצות מתנים בגודלים שונים: בית, מילה, מילה-כפולה

מבנה הפקודה: **VAR_NAME DB/DW/DD VALUE**

X1 dB 5d
Variable name Define Byte Decimal value

X2 dB 0FFh

לפני מספר בהקוצה שמתחל באות, יש
להוסיף אף כדי לציין שזה מספר

X3 dB ~~1012h~~

המספר גדול מ 8 ביטים ולכן הוא פשוט
יחתר

מערך

- הגדרת מערך בסegment DATA:

.data

ARRAY0 DB 2h, 5h, 1h, 2h, 2h

Array name
גודל תא
במערך -
byte

אם כתבנו כאן 5
איברים אז יוקצה
מערך בגודל 5 בתים

ARRAY1 DW 20d dup (?)

20 איברים

לא אתחול

- <http://www.asciiitable.com/>

מיקרו-מעבדים ושפת אסמלר

תרגול מס' 4 – זיכרון המעבד

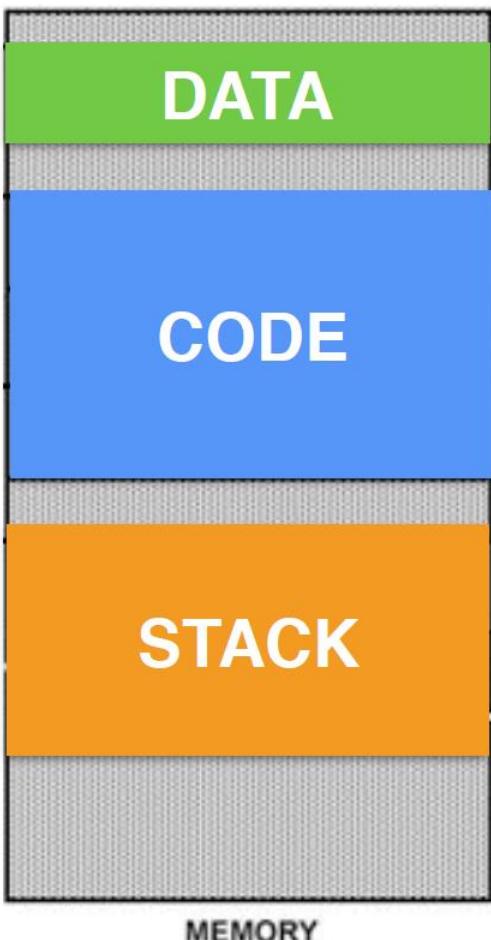
מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן



זיכרון המעבד

- מרחב הזיכרון של המעבד הוא 2^{20} , כאשר גודל אוגר הוא 16 ביט.
- נדרש 4 ביטים נוספים => אוגר סגמנט.



- קיימים 4 מצביעי סגמנט:
 - CS קוד
 - SS מחסנית
 - DS נתונים
 - ES אקסטרה
- כל אוגר סגמנט מצביע על מקטע ("סגמנט") רציף בזכרון של $k = 2^{16}$ כתובות
- כאשר פונים לזכרון, הכתובת מורכבת מהרכיבת 2 מספרים:
למצביע תחילת הסגמנט מוסיפים את הכתובת היחסית

אוגר הסגמנט + אוגר היחס = כתובת פיזית

חישוב כתובת פיזית/מוחלטת

- נכיה:

- כתובת סגמנט: $9CDB_{16} = 1001110011011011_2$

- כתובת יחסית בתוך הסגמנט: $0E8E_{16} = 0000111010001110_2$

- כדי לחשב את הכתובת בזיכרון, נבצע חיבור בין הסגמנט להיסט, כאשר מוסיפים לכתובת הסגמנט 4 אפסים מימין:

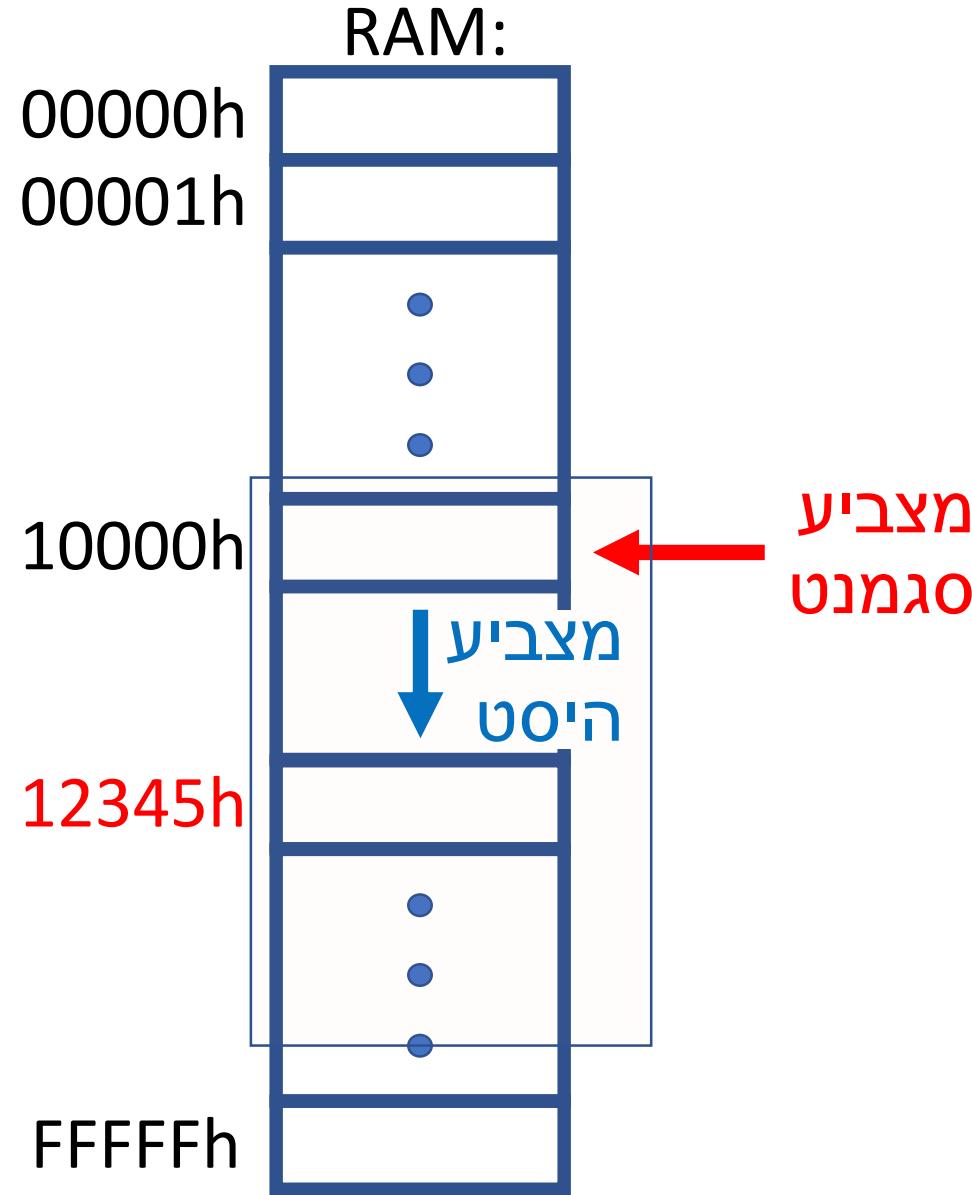
כתובת סגמנט

+

כתובת היסט

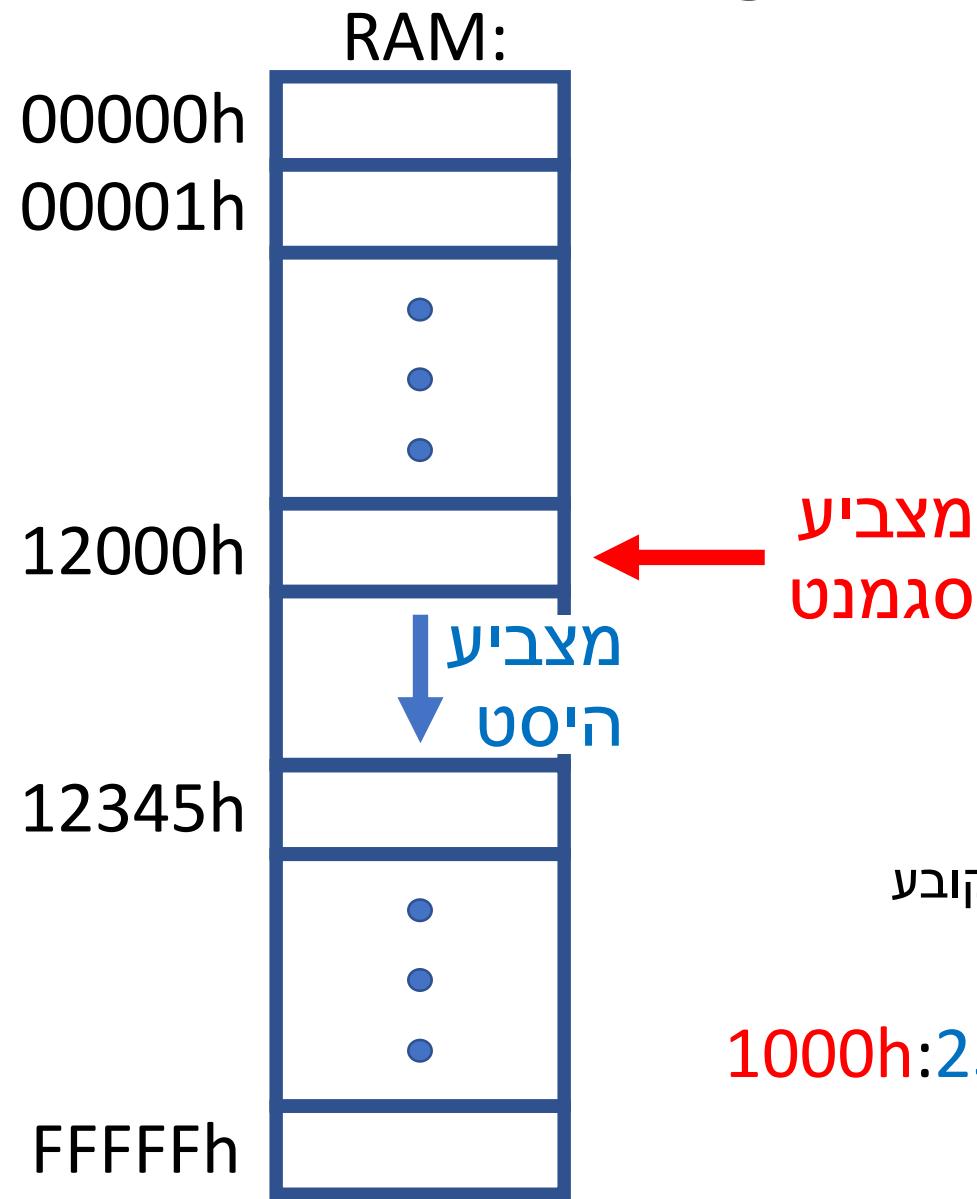
$$\begin{array}{r} 10011100110110110000 \\ + \quad 0000111010001110 \\ \hline 10011101110000111110 \end{array}$$

יצוג כתובות פיזיות בשיטה segment:offset



- כיצד נפנה לכתובת h12345?
 - תחילה נגדיר מצביע סגןנט, למשל h10000, למשל h10000:
`mov ds, 1000h`
 - אז נגדיר מצביע היאט:
`mov bx, 2345h`
 - ביחד אפשר לכתוב: (DS:BX) 1000h:2345h
- את האפס החסר המעבד מוסיף
בעצמו בחישוב הכתובת הפיזית

יצוג כתובות פיזית בשיטה segment:offset



- כמה דרכי יש לרשום את הכתובת הפיזית ?segment:offset 12345h

- עבור ערכים שונים של אורך סגמנט, נאלץ את ערך ההיסט בהתאם.

- לכתובת אחת יש 4 ייצוגים שונים cff:seg.

מקובע
1200
0345

1000h:2345h \leftrightarrow 1200h:0345h \leftrightarrow 1030h:2045h

- דוגמה:

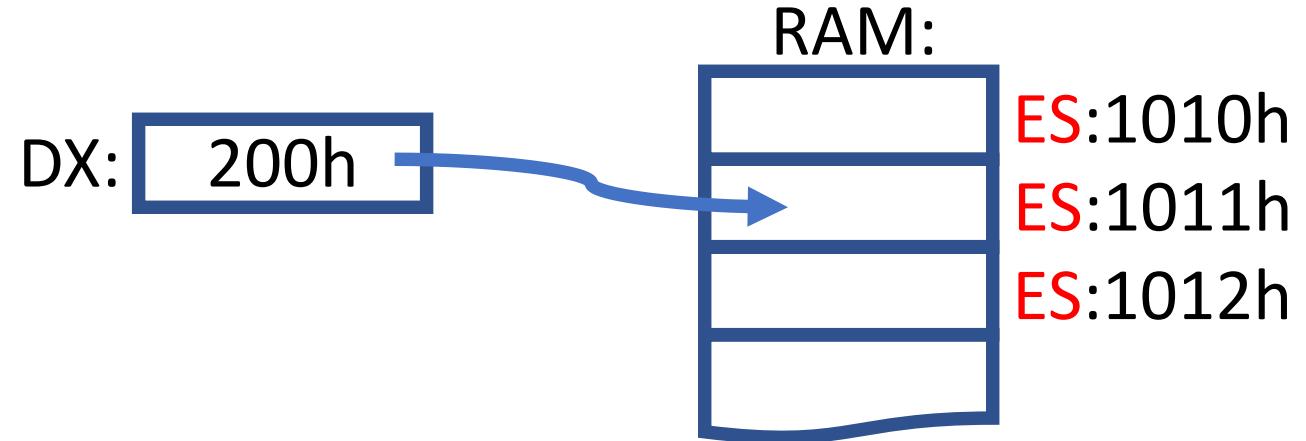
מעון ישיר

```
mov AX, 0b800h
```

```
mov ES, AX
```

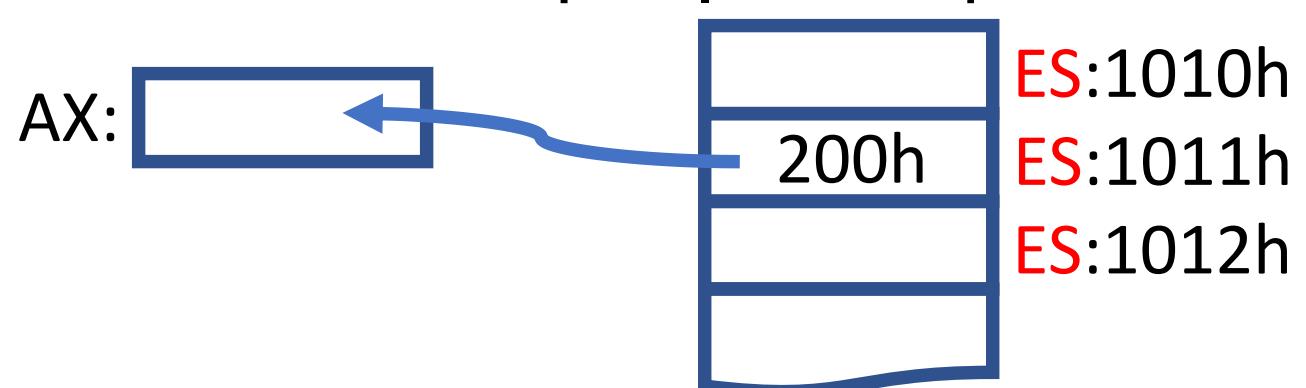
```
mov ES:[1011h], DX
```

- העתקה מאגר לזיכרון



```
mov AX, ES:[1011h]
```

- העתקה מזיכרון לתוך אגר

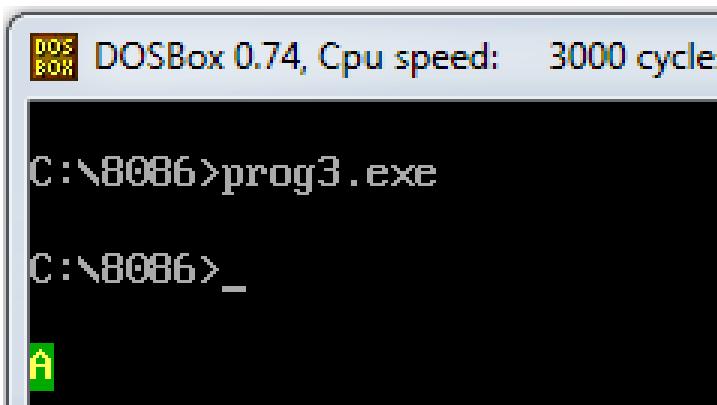


אלוץ מקטע

```
.model small
.stack 100h
.code
    ;setting ds to data segment
    mov ax, @data
    mov ds, ax

    ;setting extra segment to screen memory
    mov ax, 0b800h
    mov es, ax

    ;setting ax to represent 'A'
    mov ax, 2E41h
```



```
;writing to screen memory
mov es:[320h], ax
;writing to data memory
mov ds:[322h], ax

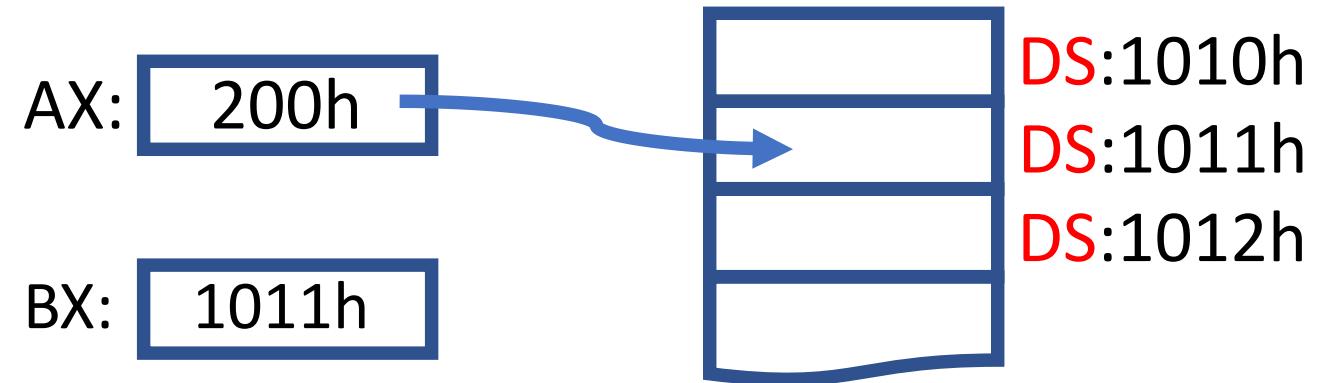
;return to OS
mov ax, 4c00h
int 21h
end
```

שיטות מעון – מעון עקיף

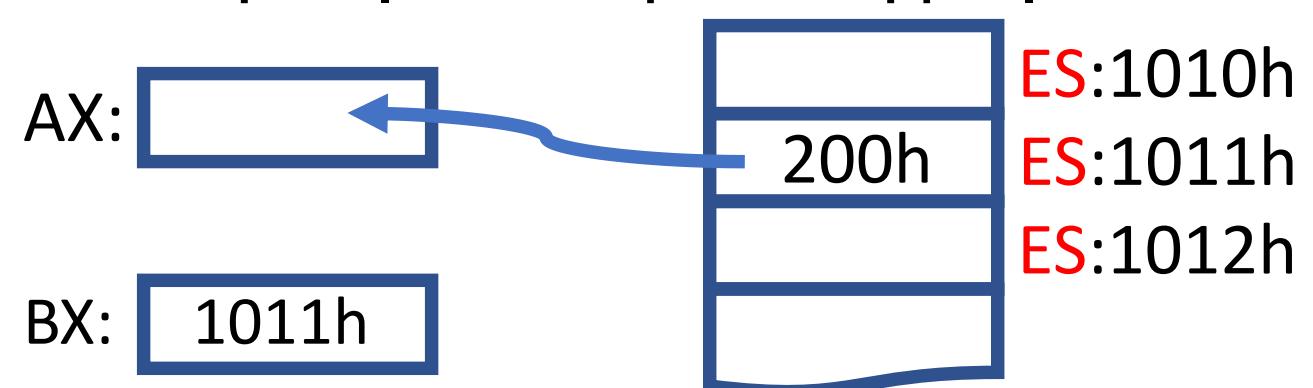
- מעון עקיף: העתקה מאוגר לזכרון דרך מצביע (אוגר אחר)

mov [BX], AX

הסיבה שאפשר להשתמש ב BX בתור מצביע, בלי לציין סגמנט, היא שלכל אוגר היסט יש אוגר סגמנט דיפולטיבי (DS עבר AX)



mov AX, ES:[BX]



מעון עקיף (מצבייע) מול מעון מיידי (לתוכ אגר)

```
.model small  
.stack 100h  
.code  
    ;setting ds to data segment  
    mov ax, @data  
    mov ds, ax  
  
    ;initialize AX  
    mov ax, 2E41h
```

הסמנט הדיפולטי
מעבר היסט BX הוא
סמנט הנתונים

```
;set bx to 320h offset  
mov bx, 320h  
  
;writing to data memory  
mov [bx], ax  
;writing to bx register  
mov bx, ax  
  
;return to OS  
mov ax, 4c00h  
int 21h
```

end

תפקידי אוגרים

SEGMENT:

CS
DS
SS
ES

מצבי סגמנט

BASE:

BP
BX

מצבי
בסיס
היחס

INDEX:

SI
DI

מצבי
אינדקס
היחס

mov DS : [BX + SI + 4h], ax

בסיס ואינדקס

- הצורך באוגר בסיס ואוגר סגמנט נובע מפעולת מעבדה עם מערכים
- דוגמה: מקטע הנתונים מתחילה בכתובת **1000h**
היסט תחילת המערך במקטע הנתונים הוא **200h**
- כיצד נפנה לתא מס' 2 במערך?

```
mov ax, 1000h ;writing number directly to DS not allowed
```

```
mov ds, ax      ;setting data segment to 10000h
```

```
mov bx, 200h    ;setting base offset - array start
```

```
mov si, 2h      ;setting array index
```

```
mov ds:[bx+si], ax
```



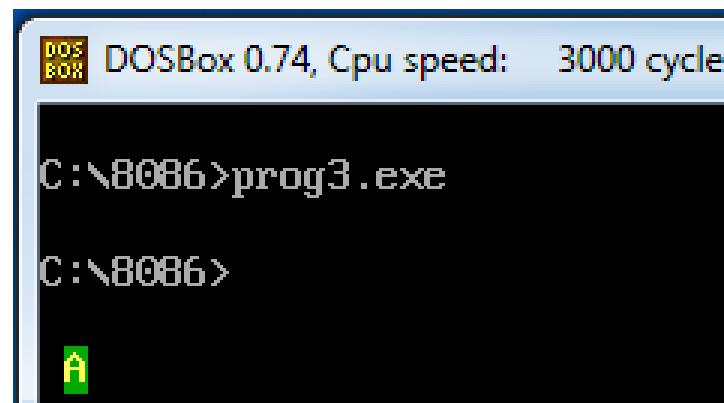
כדי לעבור על כל המערך,
נשנה את SI בולולאה

פקודת JNZ

• JNZ: אם תוצאה הפעולה האחרונות לא אפס, אז קופז.

```
.model small
.stack 100h
.code
START:
    ;setting extra segment to screen memory
    mov ax, 0b800h
    mov es, ax

    ;'A' on green background
    mov ax, 2E40h
```



```
        mov bx, 20h
        sub bx, 10h
        jnz L1
L1:   mov es:[320h+0h], ax
      mov es:[320h+2h], ax
      ;return to OS
      mov ax, 4c00h
      int 21h
end START
```

BX = 20h - 10h = 10h

תוצאה הפעולה הקודמת שונה מאפס, אך קופז

לולאה

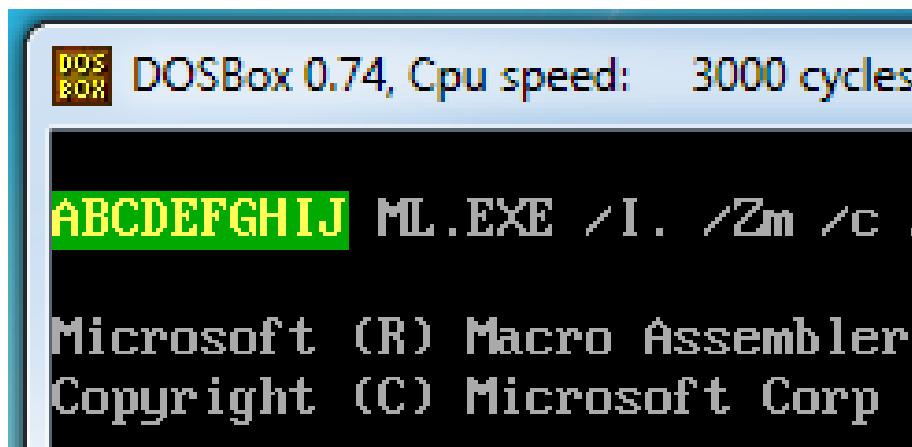
- פקודת קפיצה מותנית מאפשרת לנו לכתוב לולאות.
- מבנה בסיסי של לולאה:

```
mov cx, 10d
L1: ; Loop Content goes
      ; here
sub cx, 1
jnz L1
; Code to run after loop
```

כל איטרציה CX יורד ב1.
לאחר 10 פעמים תוצאה פועלות החישור היא 0
והקפיצה חוזרת ל L1 לא תבוצע.

הדרפסת א'ב

```
.model small
.stack 100h
.code
START:
    ;setting extra segment to screen memory
    mov ax, 0b800h
    mov es, ax
```



```
        mov ah, 2Eh ; Green background
        mov al, 41h ; 'A'
        mov bx, 140h ; Screen offset

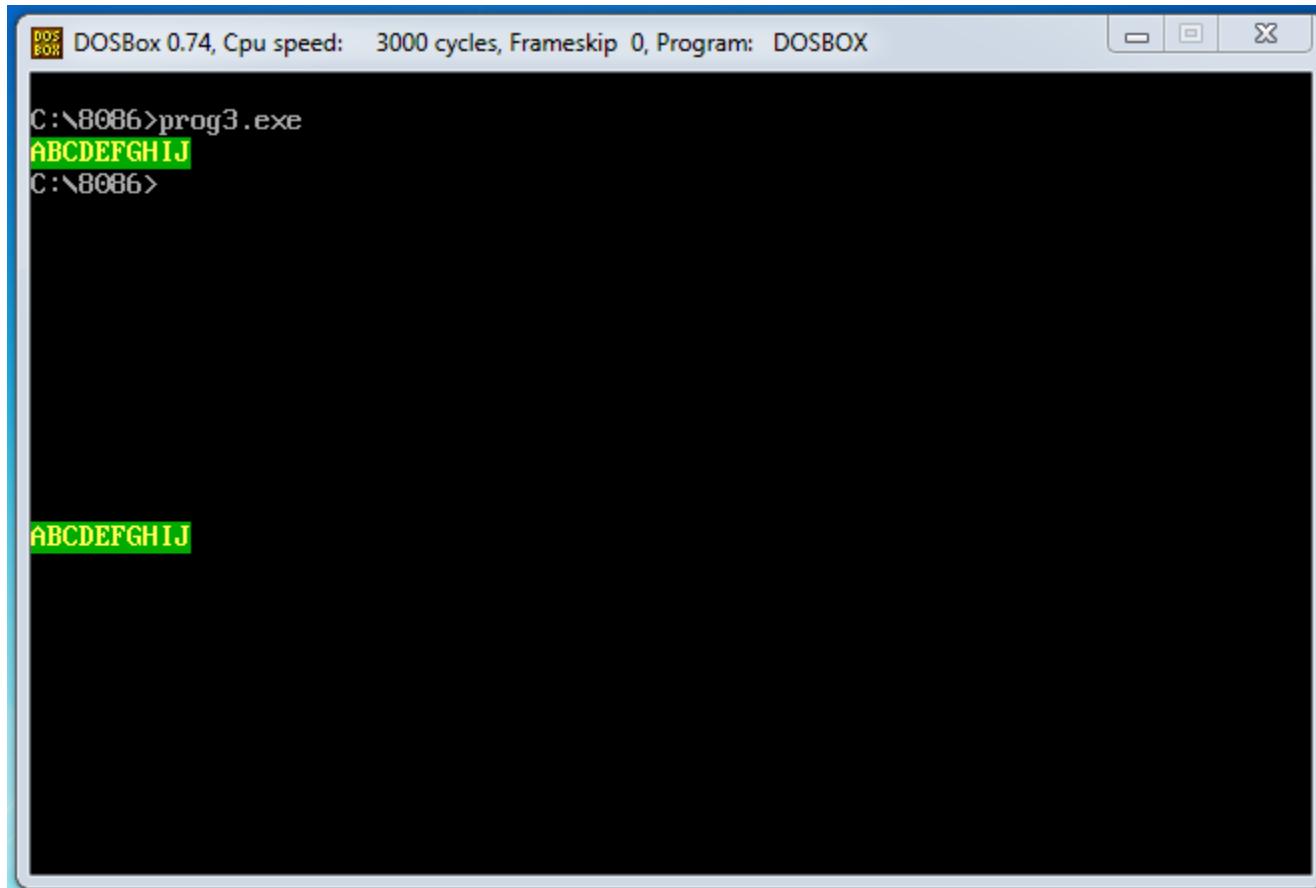
        mov cx, 10d
L1:
        mov es:[bx], ax
        inc al
        add bx, 2h

        dec cx
        jnz L1

; return to OS
        mov ax, 4C00h
        int 21h
end START
```

הוספה שורה

- נניח ונרצה להדפיס את ה א"ב גם באמצע המסר, כיצד נוכל לבצע זאת ע"י הוספת שורה בודדת לתוכנית?



הדרפסת א'ב

```
mov ah, 2Eh ; Green background
mov al, 41h ; 'A'
mov bx, 140h ; Screen offset

mov cx, 10d

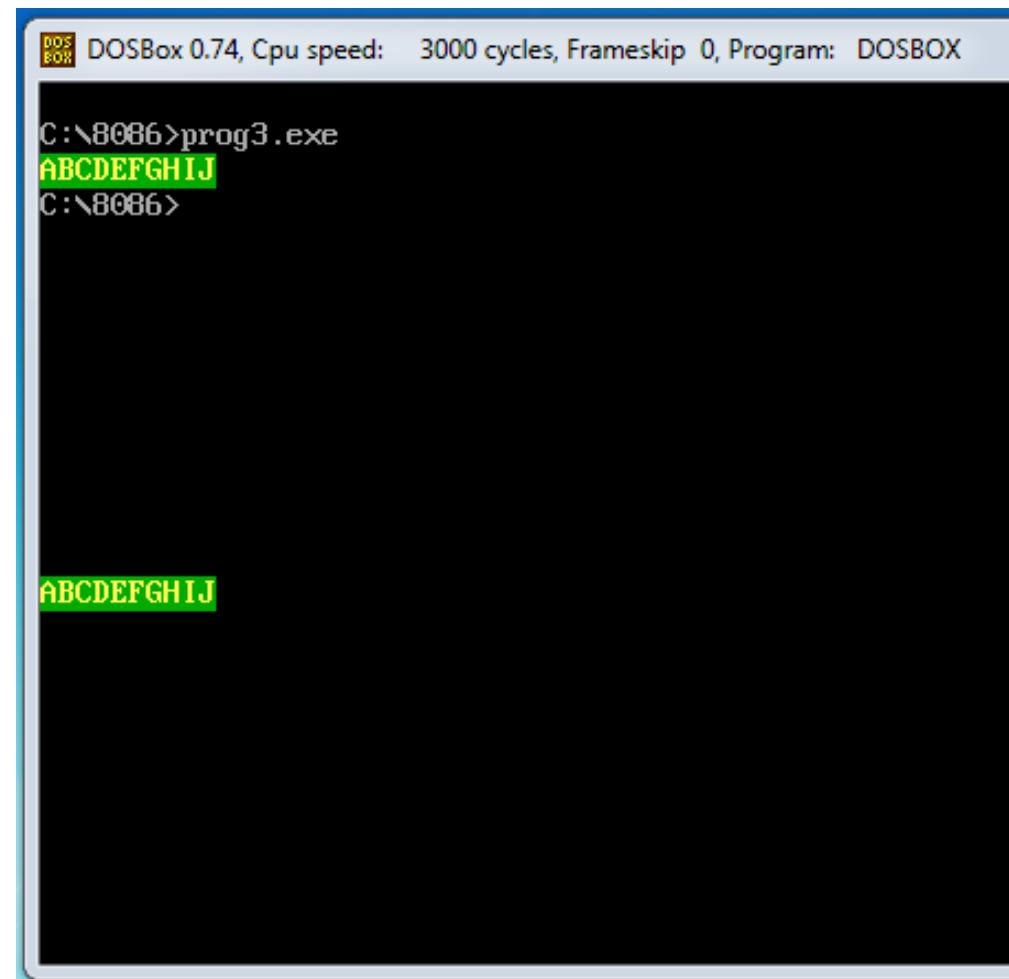
L1:
    mov es:[bx], ax
    mov es:[bx+780h], ax
    inc al
    add bx, 2h

    dec cx            $160d \cdot 12 = 1920d$ 
    jnz L1           $= 780h$ 

; return to OS
mov ax, 4C00h
int 21h

end START
```

- כיצד נבצע את הדרפסה:



מערך

- הגדרת מערך בסegment DATA:

.data

ARRAY0 DB 2h, 5h, 1h, 2h, 2h

Array name
גודל תא
במערך -
byte

אם כתבנו כאן 5
איברים אז יוקצה
מערך בגודל 5 בתים

ARRAY1 DW 20d dup (?)

20 איברים

לא אתחול

מיקרו-מעבדים ושפת אסמלר

תרגול מס' 5 – מבנה תכנית בסיסית

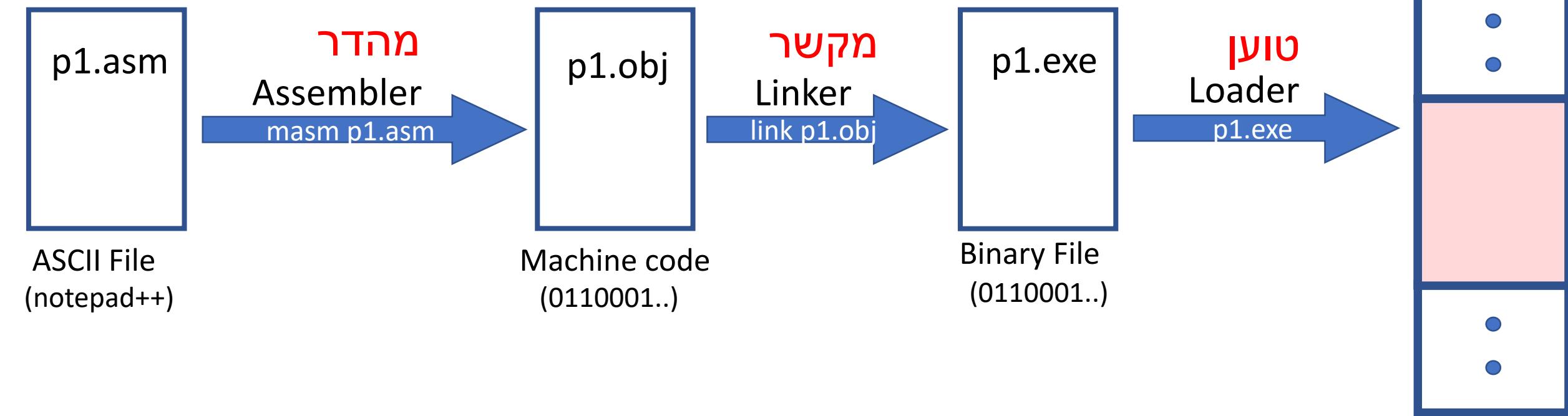
מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן



שרשרת קומpileציה והרצה

RAM:



- עד שלא הגענו לשלב ה-Load לא נוכל לדעת היכן הסגמנטים ממוקמים פיזית

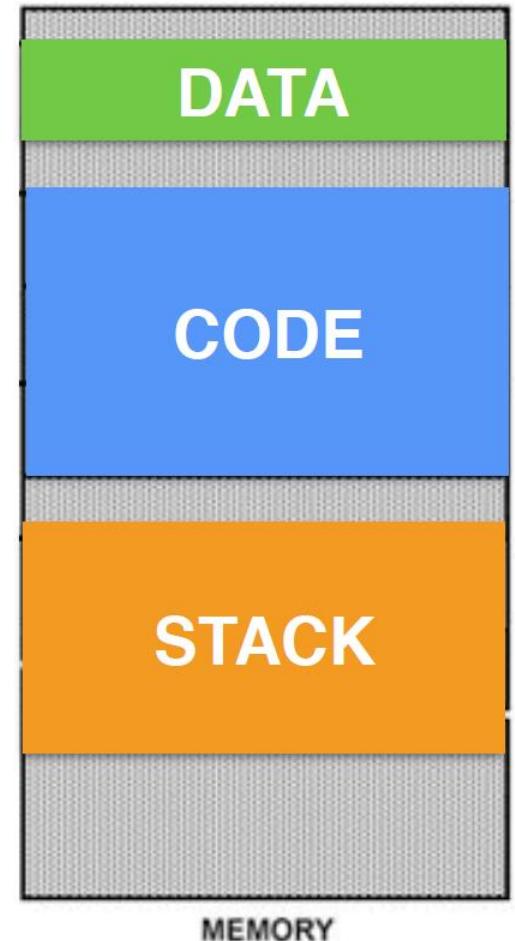
מבנה תכנית בסיסית

```
.model small
.data
    ARRAY1 DW 20d dup (?)           ←
.stack 100h
.code
PROG_CODE:
    ;setting data segment
    mov ax, @data
    mov ds, ax

    mov ax, 6546d
    mov ds:[140h], ax

    ;return to OS
    mov ax, 4c00h
    int 21h
END PROG_CODE
```

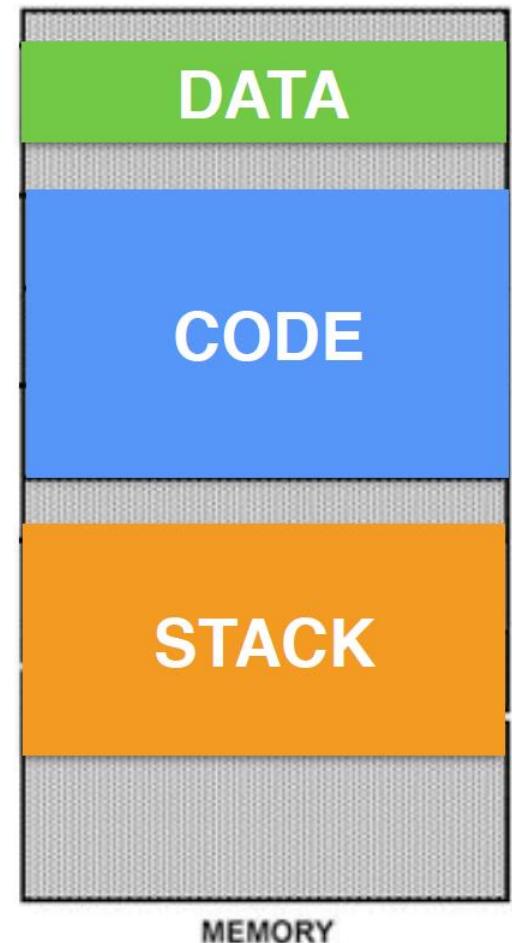
הגדרת המשתנים
שנמצאים בקטע
הנתוניים



מבנה תכנית בסיסית

```
.model small  
.data  
    ARRAY1 DW 20d dup (?)  
.stack 100h  
.code  
PROG_CODE:  
    ;setting data segment  
    mov ax, @data  
    mov ds, ax  
  
    mov ax, 6546d  
    mov ds:[140h], ax  
  
    ;return to OS  
    mov ax, 4c00h  
    int 21h  
END PROG_CODE
```

הגדרת מחסנית בגודל 100h
תאים של byte
מחסנית היא מקטע זיכרון
שמשתמשים בו על מנת
לשמר את מצב המערכת
כאשר עוברים בין תהליכיים
(מצביע למחסנית נתען
אוטומטית ע"י loader)



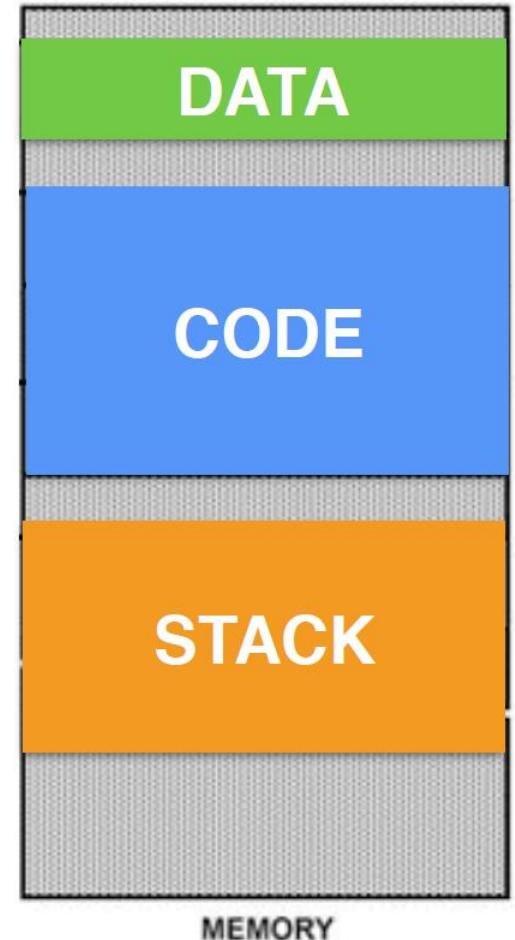
מבנה תכנית בסיסית

```
.model small
.data
    ARRAY1 DW 20d dup (?)
.stack 100h
.code
PROG_CODE:
; setting data segment
mov ax, @data
mov ds, ax

mov ax, 6546d
mov ds:[140h], ax

; return to OS
mov ax, 4c00h
int 21h
END PROG_CODE
```

מה שכתוב אחרי `.code` נשמר לתוכה סגמנט הזכרן שמיועד לקוד
PROG_CODE הוא "לייבל" עבור הקומpileר שמצין את תחילת הקוד שצריך להריץ

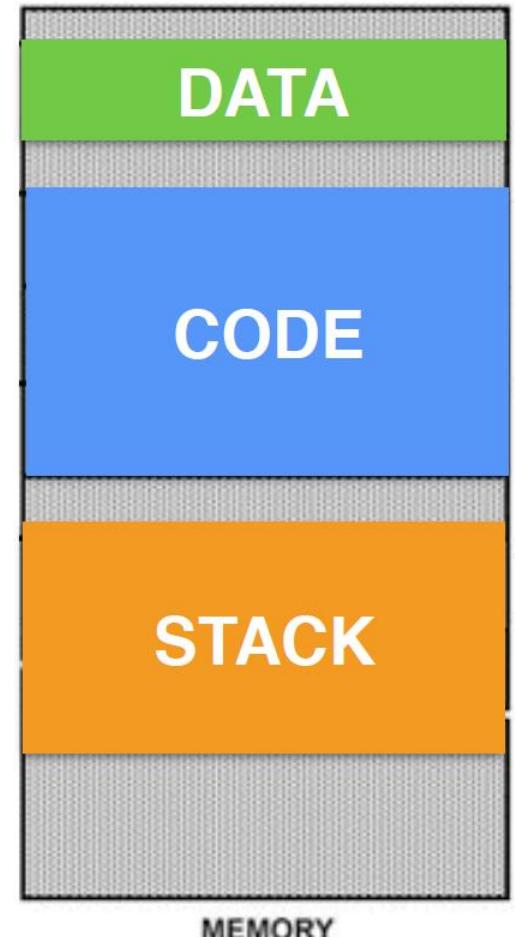


מבנה תכנית בסיסית

```
.model small  
.data  
    ARRAY1 DW 20d dup (?)  
.stack 100h  
.code  
PROG_CODE:  
    ;setting data segment  
    mov ax, @data  
    mov ds, ax  
  
    mov ax, 6546d  
    mov ds:[140h], ax  
  
    ;return to OS  
    mov ax, 4c00h  
    int 21h  
END PROG_CODE
```

הראה מיוחדת שسؤالת את המחשב היכן נמצא סגמנט הנתונים, כך שפקודות מכונה שפוניות לזכרון יפנו למקום הנכון

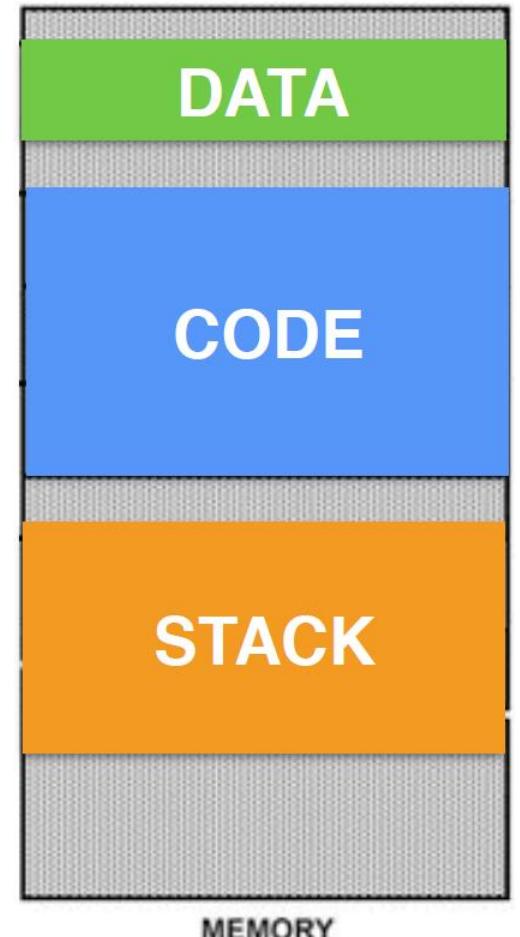
קבעת ערכו של `@data` מתרחשת אחרי הקומpileציה, בזמן טעינת התוכנית לזכרון



מבנה תכנית בסיסית

```
.model small  
.data  
    ARRAY1 DW 20d dup (?)  
.stack 100h  
.code  
PROG_CODE:  
    ;setting data segment  
    mov ax, @data  
    mov ds, ax  
  
    mov ax, 6546d  
    mov ds:[140h], ax  
  
    ;return to OS  
    mov ax, 4c00h  
    int 21h  
END PROG_CODE
```

העדכנים שעשינו לזכרון בזמן ריצת התוכנה ישמרו מצב האוגרים, לא.
הפעלת פסיקה שמחזירה את השליטה למערכת הפעלה.



CV Debugger

- כדי להפעיל את CV debugger, תחיליה נקמפל כרגע:

ml /Zm /Zi Prog3.asm

- ואז נרץ את התוכנה שנו דרכו ה debugger

CV Prog3.exe

```
C:\8086>ml /Zm Prog3.asm
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

Assembling: Prog3.asm

Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]: Prog3.obj
Run File [Prog3.exe]: "Prog3.exe"
List File [nul.map]: nul
Libraries [.lib]:
Definitions File [nul.def]:

C:\8086>CU Prog3.exe
```

תכנית עם שגיאות סינטקטיות

- עוד בשלב הקומpileציה קיבל הודעה של שגיאה, שמכונות אותנו לשורה עם השגיאה:

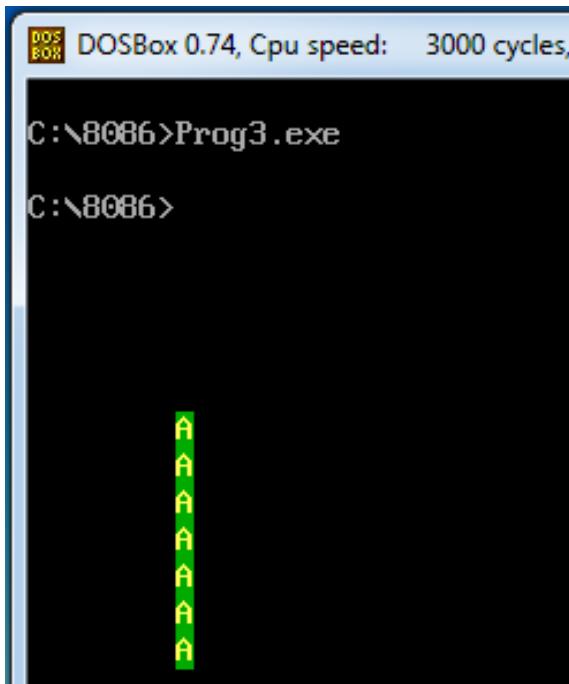
```
1 .model small
2 .stack 100h
3 .code
4 START:
5 ;setting extra segment to screen memory
6 mov ax, 0b800h
7 mov es, ax
8
9 mov ds, es
10
11 mov ax, 4c00h
12 int 21h
13 end START
```

לא ניתן להעתיק בין
אוגרי סגמנט

```
C:\8086>ml /Zm Prog3.asm
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

Assembling: Prog3.asm
Prog3.asm(9): error A2070: invalid instruction operands
```

תכנית שמדפסה עמודה של 'A' על המסך



- נרצה להדפס על המסך עמודה של האות A בצלע יрок.

- נרצה עמודה בגובה 7 שורות:
 - בשביל זה נגדיר קבוע (מומלץ לשים ב-header)

```
N_lines EQU 7h
```

- בנוסף נרצה את העמודה במקום ה9 בשורה 8
 - נקבע את היחסט במסך ל $160d \cdot 8 + 8d \cdot 2 = 1296d = 510h$

```
.code
```

```
    mov bx, 510h
```

תכנית שמדפסה עמודה של 'A' על המסך

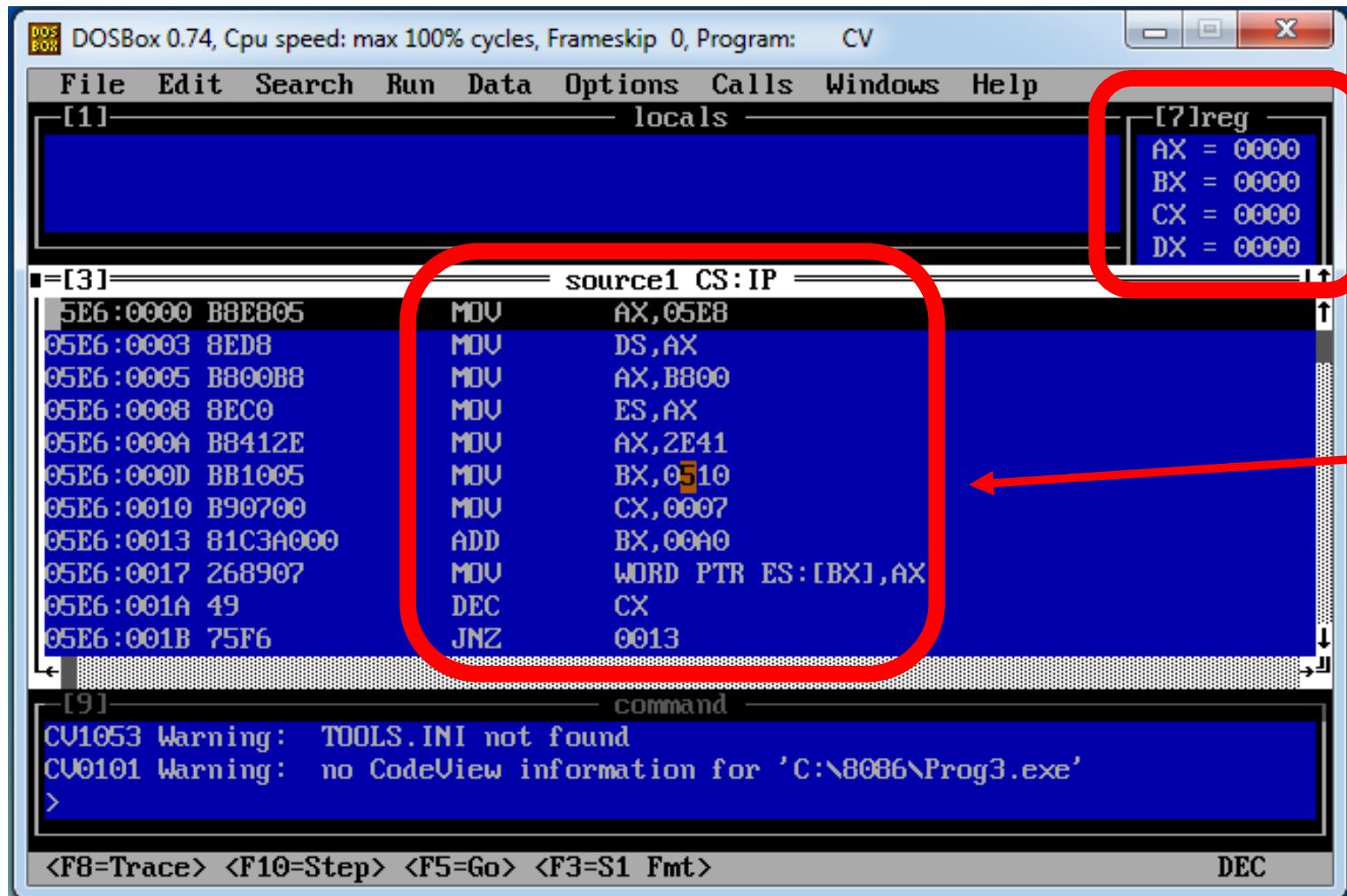
```
.model small  
  
N_lines EQU 7h  
  
.stack 100h  
.code  
START:  
    ; setting data segment  
    mov ax, @data  
    mov ds, ax  
  
    ; screen memory  
    mov ax, 0b800h  
    mov es, ax  
  
    ; 'A' on green background  
    mov ax, 2E41h
```

הגדרת קבוע

```
; set offset to line 8 position 9  
mov bx, 510h  
  
; set loop counter to N_lines  
mov cx, N_lines  
  
L2: add bx, 00a0h ; forward bx 1 line  
    mov es:[bx], ax ; write to screen  
    dec cx  
    jnz L2  
  
    mov ax, 4c00h  
    int 21h  
end START
```

הרצה התוכנית עם CV debugger

CV Prog3.exe



מצב האוגרים

פקודות התוכנית

הרצה התוכנית עם CV debugger

```
===== source1 CS:IP =====
MOV AX,05E8
MOV DS,AX
MOV AX,B800
MOV ES,AX
MOV AX,2E41
MOV BX,0510
MOV CX,0007
ADD BX,00A0
WORD PTR ES:[BX],AX
DEC CX
JNZ 0013
```

```
.code
START:
; setting data segment
mov ax, @data
mov ds, ax
; screen memory
mov ax, 0b800h
mov es, ax
```

בזמןטעינת התוכנית ה Loader
טען במקום @data את הכתובת
שהקצתה לSEGMENT DATA (שהיא
05E8 בΡίζה זו)

הרצה התוכנית עם CV debugger

The screenshot shows the DOSBox environment with the CV debugger running. The assembly code window displays instructions from address 05E6:0000 to 05E6:001B. The instruction at 05E6:0003 is highlighted in blue: `MOV DS,AX`. The registers window shows the state of the CPU registers after the instruction was executed. Red arrows point from the highlighted instruction in the assembly window to the value '05E8' in the AX register of the registers window, and another arrow points from the same instruction to the text 'F10 Step' in the status bar.

Address	OpCode	Instruction
05E6:0000	B8E805	MOV AX,05E8
05E6:0003	8ED8	MOV DS,AX
05E6:0005	B800B8	MOV AX,B800
05E6:0008	8EC0	MOV ES,AX
05E6:000A	B8412E	MOV AX,2E41
05E6:000D	BB1005	MOV BX,0510
05E6:0010	B90700	MOV CX,0007
05E6:0013	81C3A000	ADD BX,00A0
05E6:0017	268907	MOV WORD PTR ES:[BX],AX
05E6:001A	49	DEC CX
05E6:001B	75F6	JNZ 0013

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: CV

File Edit Search Run Data Options Calls Windows Help

[1] locals [7]reg

AX = 0000
BX = 0000
CX = 0000
DX = 0000

[3] source1 CS:IP

AX,05E8
DS,AX
AX,B800
ES,AX
AX,2E41
BX,0510
CX,0007
BX,00A0
WORD PTR ES:[BX],AX
CX
0013

command

CV1053 Warning: TOOLS.INI not found
CV0101 Warning: no CodeView information for 'C:\8086\prog3.exe'

<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt>

DEC

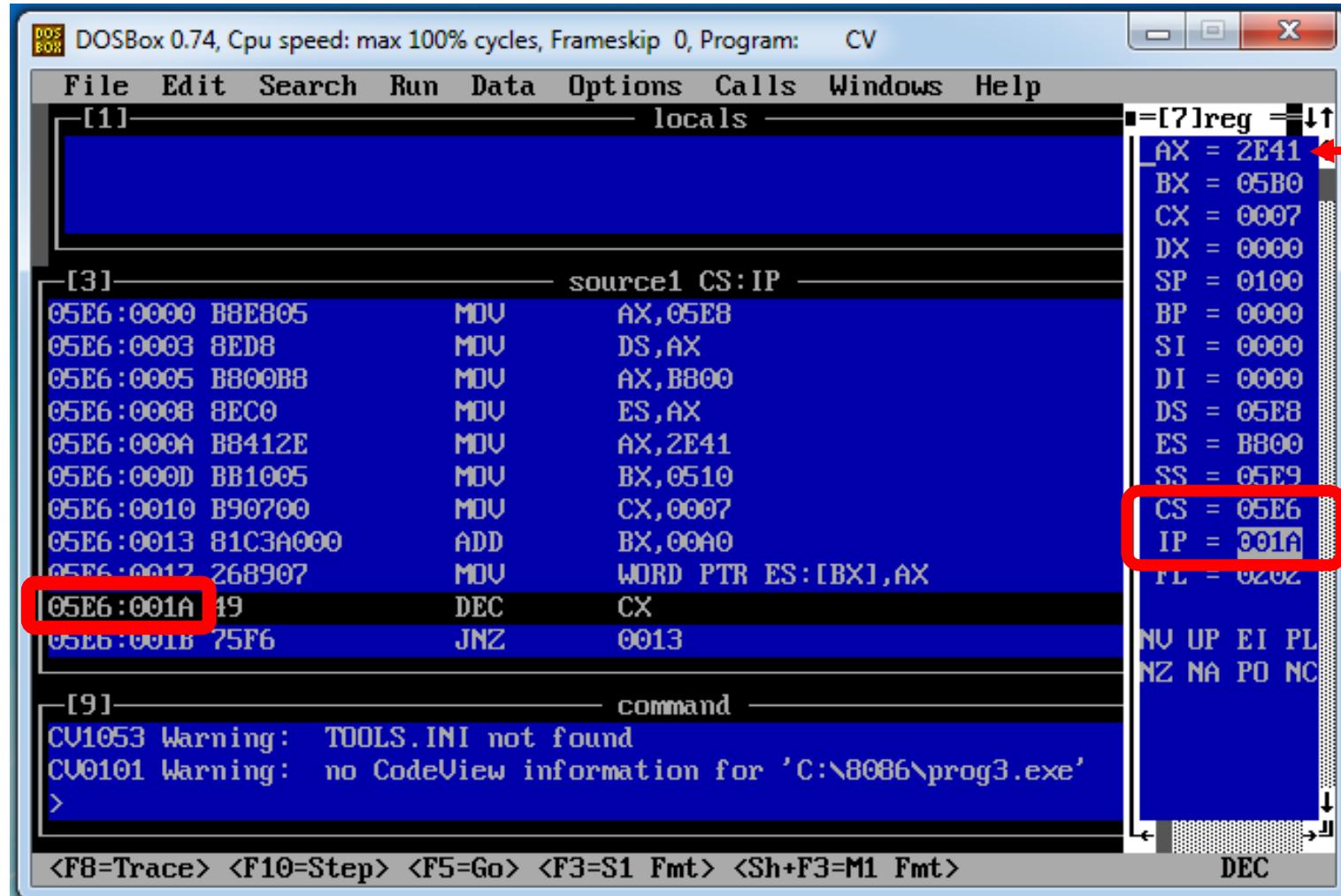
כדי לקדם את התוכנית צעד אחד, נלחץ F10

לאחר ביצוע הפקודה
הראשונה, באוגר AX
נמצא המספר 05E8

הפקודה שתתבצע
אחרי שנלחץ F10
פעם נוספת

מצב האוגרים

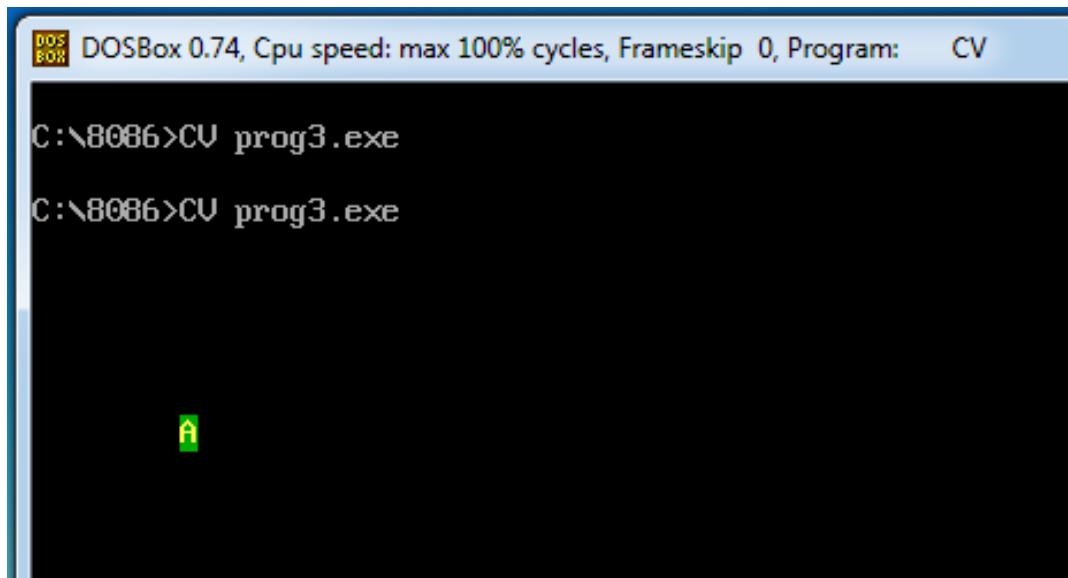
- כדי לראות את מצב האוגרים בהרחבה, נלחץ עם הוכבר על האזור של



'A' on green background

האוגרים
מרכיבים את
הכתובת של
הפקודה הנוכחית

הצגת הפלט

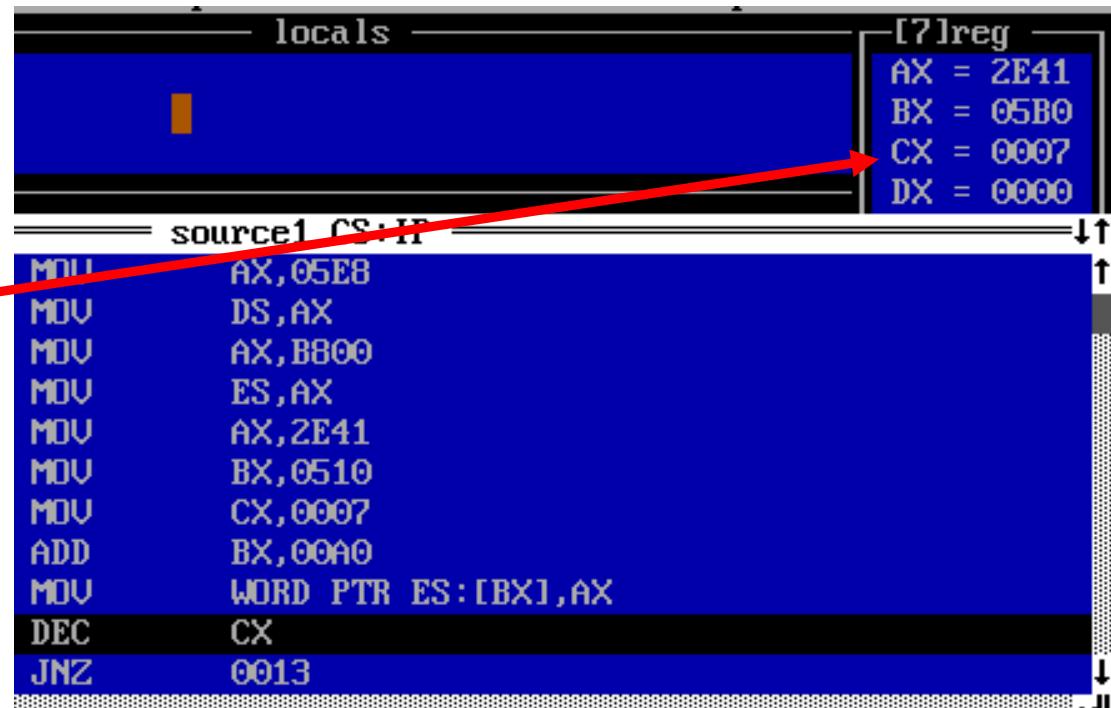


```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: CV  
C:\8086>CV prog3.exe  
C:\8086>CV prog3.exe
```

אנחנו נמצאים בסיבוב
הראשון של הלולאה, אחרי
הכתיבה לזכרון המסר.
לכן על המסר מופיע A ייחיד

- כדי להציג את המסר עם הפלט נלחץ F4

- כדי לחזור ל debugger נלחץ Enter

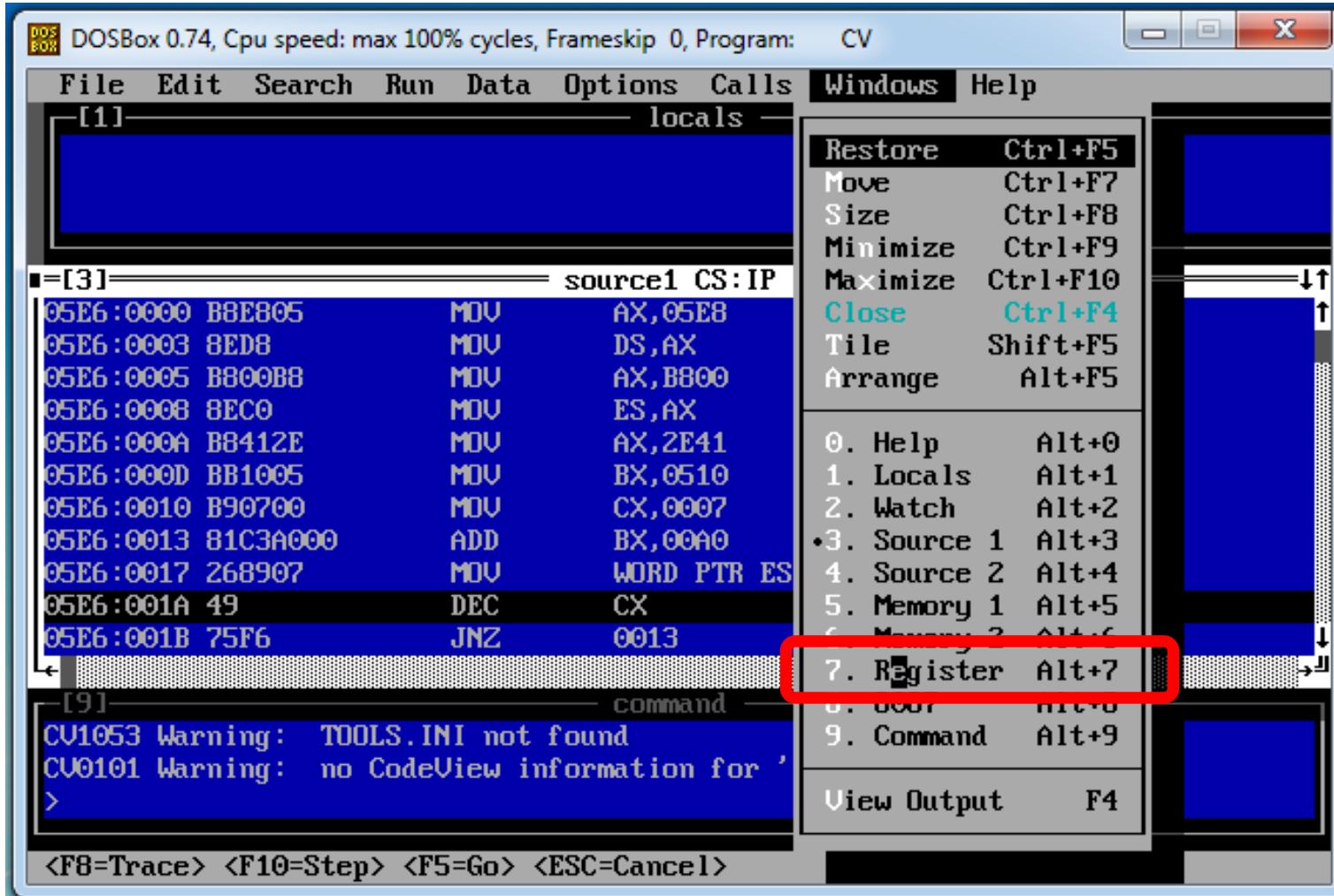


locals		[?]reg
		AX = 2E41
		BX = 05B0
		CX = 0007
		DX = 0000

source1 CS:IP	
MOV	AX,05E8
MOV	DS,AX
MOV	AX,B800
MOV	ES,AX
MOV	AX,2E41
MOV	BX,0510
MOV	CX,0007
ADD	BX,00A0
MOV	WORD PTR ES:[BX],AX
DEC	CX
JNZ	0013

הזרת\הוספת חלונות

- מה נעשה אם נעלם לנו חלון האוגרים למשל? או שנרצה לראות את תוכן הזיכרון?



- ניתן לפתח את החלונות, ולבצע פקודות מהסרגל הכלים:

פקודת הקפיצה

source1 CS:IP			
05E6:0013	81C3A000	ADD	BX,00A0
05E6:0017	268907	MOV	WORD PTR ES:[BX],AX
05E6:001A	49	DEC	CX
05E6:001B	75F6	JNZ	0013
05E6:001D	B8004C	MOV	AX,4C00
05E6:0020	CD21	INT	21

לייל L2 הוא
כתובת פיזית
בסמנט CODE

```
L2: add bx,00a0h ; forward bx 1 line
      mov es:[bx],ax ; write to screen
      dec cx
      jnz L2
```

תרגיל

```
mov ax, @data  
mov ds, ax  
mov bx, 05H  
mov [bx - 1], 0CDH  
mov [bx + 1], 0EFH
```

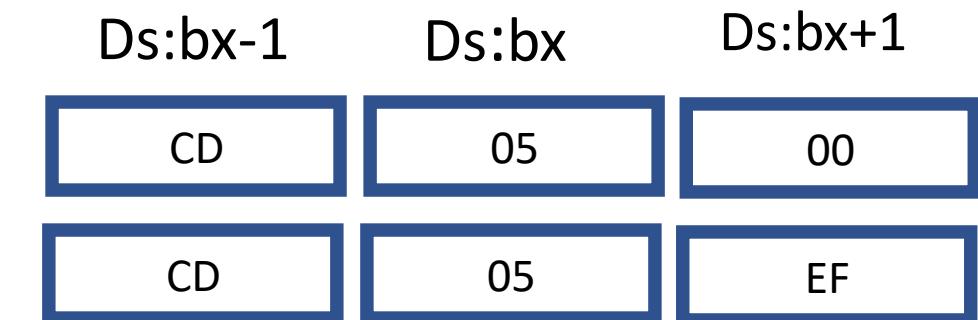
??

- בහינתו הקוד הנ"ל:

- מה יקרה אם הפקודה הריקה בסוף תהיה:

- *mov BYTE PTR [bx], 1005H*
- *mov WORD PTR [bx], 5H*
- *mov BYTE PTR [bx], 5H*

לא מתקין! – הגדרנו בית ושםנו מספר בגודל 'מילה'



כתיבות לזכרון

- נשלב את קטע הקוד שלנו בתכנית, ונשתמש בCV debugger על מנת
לצפוף בזיכרון

```
.model small
.data
    ARRAY1 DW 20h dup (1)
.stack 100h
.code
START:
    ;setting data segment
    mov ax, @data
    mov ds, ax
```

```
    mov bx, 16h
    mov [bx-1], 0CDh
    mov [bx+1], 0EFh

    mov WORD PTR [bx], 5h
    mov BYTE PTR [bx], 0Ah

    mov ax, 4C00h
    int 21h
end START
```

מפתח הזיכרון

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: CV

[3] — source1 CS:IP —

05E6:0000 B8E705	MOV	AX,05E7
05E6:0003 8ED8	MOV	DS,AX
05E6:0005 BB1600	MOV	BX,0016
05E6:0008 C747FFCD00	MOV	WORD PTR [BX-01],00CD
05E6:000D C74701EF00	MOV	WORD PTR [BX+01],00EF
05E6:0012 C7070510	MOV	WORD PTR [BX],1005
05E6:0016 C6070A	MOV	BYTE PTR [BX],0A
05E6:0019 B8004C	MOV	AX,4C00
05E6:001C CD21	INT	21
05E6:001E 0100	ADD	WORD PTR [BX+SII],AX
05E6:0020 0100	ADD	WORD PTR [BX+SII],AX

■[5] — memory1 b DS:0 —

05E7:0000 EF 00 C7 07 05 10 C6 02 0A B8 00 4C CD 21	01 . •♦ =•¤. L=!
05E7:000E 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05E7:001C 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05E7:002A 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05E7:0038 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt> <Sh+F3=M1 Fmt>

DEC

ערכים התחלתיים
של תא זיכרון

אזור שהוקצה
למערך

מפתח הזיכרון

The screenshot shows the DOSBox interface with two windows. The top window, titled '[3] source1 CS:IP', displays assembly code:

```
05E6:0000 B8E705    MOV     AX,05E7
05E6:0003 8ED8      MOV     DS,AX
05E6:0005 BB1600    MOV     BX,0016
05E6:0008 C747FFCD00 MOV     WORD PTR [BX-01],00CD
05E6:000D C74701EF00 MOV     WORD PTR [BX+01],00EF
05E6:0012 C7070510  MOV     WORD PTR [BX],1005
05E6:0016 C6070A    MOV     BYTE PTR [BX],0A
05E6:0019 B8004C    MOV     AX,4C00
05E6:001C CD21      INT     21
05E6:001E 0100      ADD     WORD PTR [BX+SI],AX
05E6:0020 0100      ADD     WORD PTR [BX+SI],AX
```

The instruction at address 05E6:0008 is highlighted with a red box. The bottom window, titled '[5] memory1 b DS:0', shows the memory dump:

Address	Value
05E7:0000	EF 00 C7 07 05 10 C6 07 0A B8 00 4C CD 21 0.
05E7:000E	01 00 01 00 01 00 01 CD 00 00 01 00 00 01 00 00
05E7:001C	01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00
05E7:002A	01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00
05E7:0038	01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00

A red arrow points from the highlighted byte 'CD' in the memory dump to the highlighted instruction in the assembly window. The memory dump shows the byte 'CD' at address 05E7:000E, which corresponds to the memory location specified in the assembly code: WORD PTR [BX-01].

כתיבת WORD
 בשיטת little
 endian

bx-1 = 15h נכתב בסegment DATA במקום 'CD'

מפתח הזיכרון

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: CV

[3] — source1 CS:IP —

05E6:0000 B8E705	MOV	AX,05E7
05E6:0003 8ED8	MOV	DS,AX
05E6:0005 BB1600	MOV	BX,0016
05E6:0008 C747FFCD00	MOV	WORD PTR [BX-011],00CD
05E6:000D C74701EF00	MOV	WORD PTR [BX+011],00EF
05E6:0012 C7070510	MOV	WORD PTR [BX],1005
05E6:0016 C6070A	MOV	BYTE PTR [BX],0A
05E6:0019 B8004C	MOV	AX,4C00
05E6:001C CD21	INT	21
05E6:001E 0100	ADD	WORD PTR [BX+SII],AX
05E6:0020 0100	ADD	WORD PTR [BX+SII],AX

[5] — memory1 b DS:0 —

05E7:0000 EF 00 C7 07 05 10 C6 02 00 B8 00 4C CD 21 0.	EF 00 C7 07 05 10 C6 02 00 B8 00 4C CD 21 0.
05E7:000E 01 00 01 00 01 00 01 CD 00 EF 00 03 01 00 0.	EF 00 C7 07 05 10 C6 02 00 B8 00 4C CD 21 0.
05E7:001C 01 00 01 00 01 00 01 00 01 00 01 00 01 00 0.	EF 00 C7 07 05 10 C6 02 00 B8 00 4C CD 21 0.
05E7:002A 01 00 01 00 01 00 01 00 01 00 01 00 01 00 0.	EF 00 C7 07 05 10 C6 02 00 B8 00 4C CD 21 0.
05E7:0038 01 00 01 00 01 00 01 00 01 00 01 00 01 00 0.	EF 00 C7 07 05 10 C6 02 00 B8 00 4C CD 21 0.

<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt> <Sh+F3=M1 Fmt> DEC

האזור בזיכרון
שהשתנה בעקבות
הפקודה מסומן
באפור

'EF' נכתב בסegment DATA במקום bx+1 = 17h

מפתח הזיכרון

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: CV

File Edit Search Run Data Options Calls Windows Help

[3] ————— source1 CS:IP —————

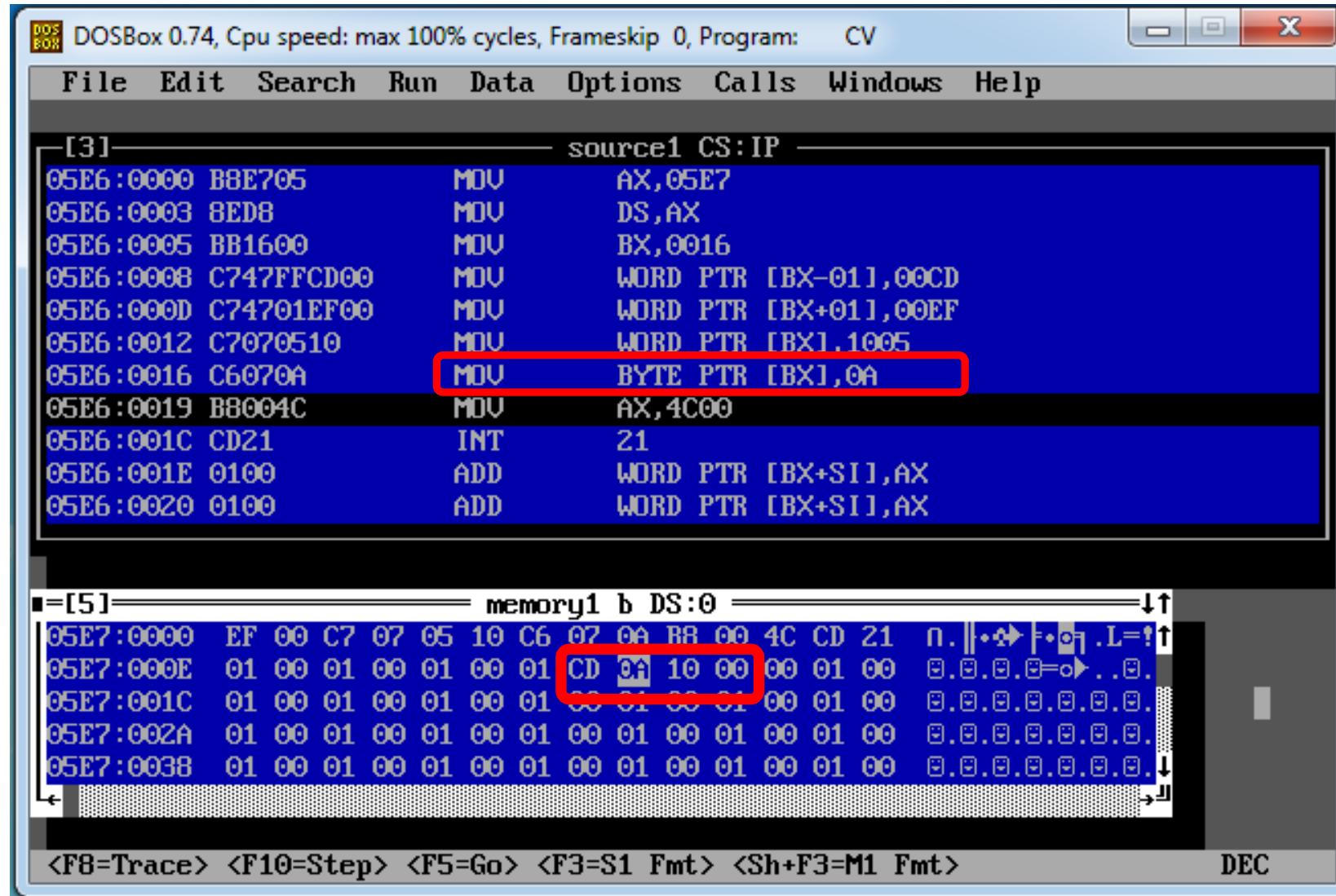
05E6:0000 B8E705	MDU	AX,05E7
05E6:0003 8ED8	MDU	DS,AX
05E6:0005 BB1600	MDU	BX,0016
05E6:0008 C747FFCD00	MDU	WORD PTR [BX-01],00CD
05E6:000D C74701EF00	MDU	WORD PTR [BX+01],00FF
05E6:0012 C7070510	MDU	WORD PTR [BX],1005
05E6:0016 C6070A	MDU	BYTE PTR [BX],0A
05E6:0019 B8004C	MDU	AX,4C00
05E6:001C CD21	INT	21
05E6:001E 0100	ADD	WORD PTR [BX+SI],AX
05E6:0020 0100	ADD	WORD PTR [BX+SI],AX

= [5] ————— memory1 b DS:0 ————— ↑↑

05E7:0000 EF 00 C7 07 05 10 C6 02 0A B8 00 4C CD 21 0.	•♦→ f•o7 .L=!
05E7:000E 01 00 01 00 01 00 01 CD 05 10 00 00 01 00 0.	0.0.0.0.=♦ .0.
05E7:001C 01 00 01 00 01 00 01 00 01 00 01 00 01 00 0.	0.0.0.0.0.0.0.
05E7:002A 01 00 01 00 01 00 01 00 01 00 01 00 01 00 0.	0.0.0.0.0.0.0.
05E7:0038 01 00 01 00 01 00 01 00 01 00 01 00 01 00 0.	0.0.0.0.0.0.0.

<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt> <Sh+F3=M1 Fmt> DEC

מפתח הזיכרון



מיקרו-מעבדים ושפת אסמבילר

Debug – מס' 6

מיקרו מעבדים ואסמבילר – 83255, תש"פ

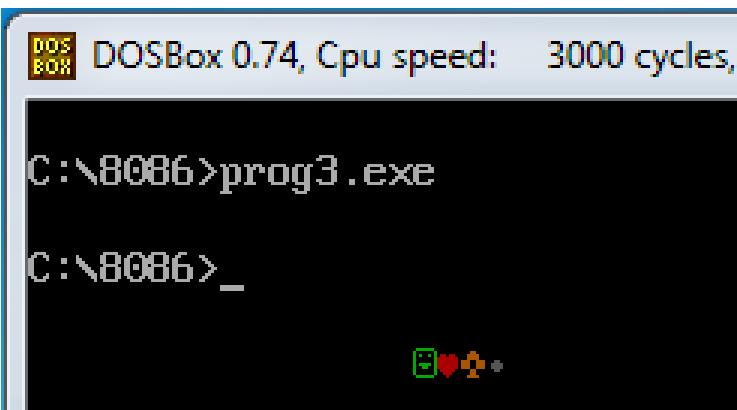
הפקולטה להנדסה, אוניברסיטת בר-אילן



דיבאג (de-bug) – ניפוי שגיאות

- נתון הקוד הבא, שמטרתו היא להדפיס למסך את הספרות מ-1 עד 9.

```
.model small
.data
.stack 100h
.code
HERE:
; Build a code that counts from 1 to 9
; and prints to screen
    mov ax, 0B800h
    mov es, ax
    mov si, 340h ;screen offset
```



```
        mov cx, 9
        mov ax, 1

PrntLOOP:
        mov es:[si], ax

        inc ax ; move to next char to print
        inc si ; move to next place on screen

        dec cx
        jnz PrntLOOP

        mov ax, 4c00h
        int 21h
end HERE
```

חזרה – הפעלת CV debugger

- כדי להפעיל את CV debugger, תחיליה נקמפל כרגע:

C:> ml /Zm Prog3.asm

- ואז נפעיל את תוכנת ה-CV ונטען אליה את התוכנה שקייינטנו:

C:> CV Prog3.exe

```
C:\8086>ml /Zm Prog3.asm
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

Assembling: Prog3.asm

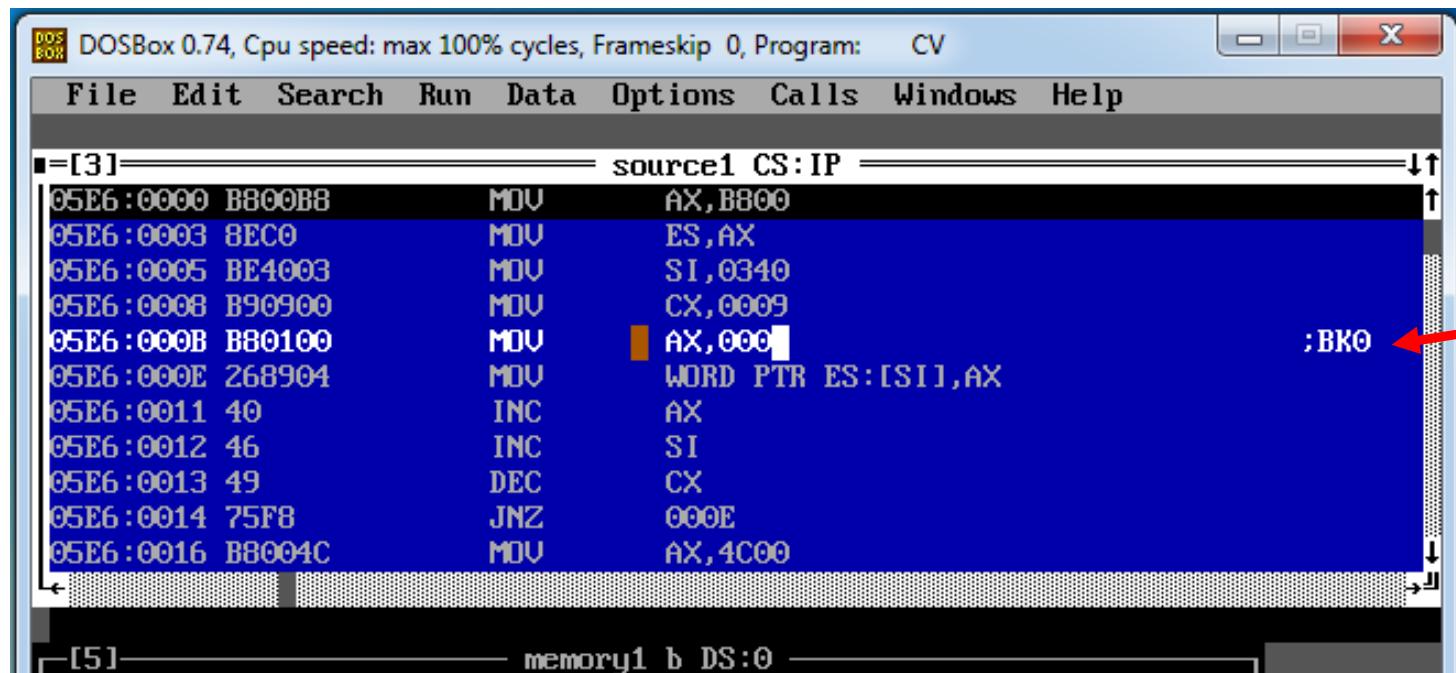
Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]: Prog3.obj
Run File [Prog3.exe]: "Prog3.exe"
List File [nul.map]: nul
Libraries [.lib]:
Definitions File [nul.def]:

C:\8086>CV Prog3.exe
```

Breakpoint & Run

- כדי לא להתעכ卜 על פקודות שאנו יודעים שהן תקינות, נשים נסימן בטור הראשונה של הקוד החדש ונריץ את התוכנה עד לנקודה זו.



Double click
השורה כדי לשים בה
breakpoint

```
mov cx, 9
mov ax, 1
```

```
PrntLOOP:
    mov es:[si], ax
```

```
inc ax ; move to next char to print
inc si ; move to next place on screen

dec cx
jnz PrntLOOP
```

Breakpoint & Run

- כדי להריץ עד לשורה שבה שמננו את ה breakpoint נלחץ F5

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: CV

File Edit Search Run Data Options Calls Windows Help

■=[3]— source1 CS:IP —↑↑

05E6:0000 B800B8	MOV	AX,B800
05E6:0003 BEC0	MOV	ES,AX
05E6:0005 BE4003	MOV	SI,0340
05E6:0008 B90900	MOV	CX,0009
05E6:000B B80100	MOV	AX,0001 :BKO
05E6:000E 268904	MOV	WORD PTR ES:[SI],AX
05E6:0011 40	INC	AX
05E6:0012 46	INC	SI
05E6:0013 49	DEC	CX
05E6:0014 75F8	JNZ	000E
05E6:0016 B8004C	MOV	AX,4C00

[5]— memory1 b DS:0 —↑↑

05D6:0000 CD 20 FF 9F 00 EA FF FF AD DE 96 02 3F 04 = f.º i [0?♦]
05D6:000E 97 03 3F 04 DD 0B 3F 04 BB 05 01 01 01 00 à?♦ ??♦[0?♦]

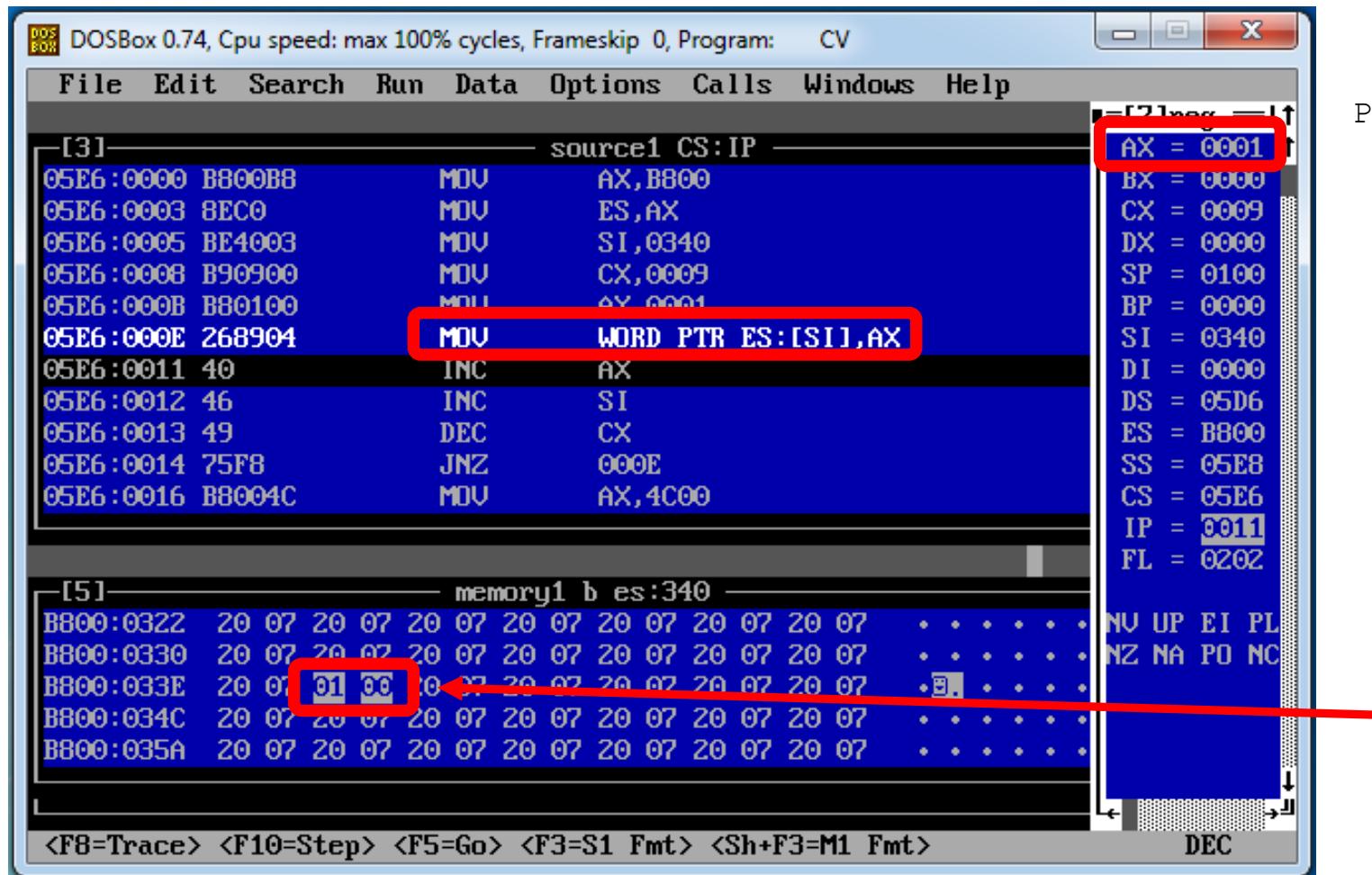
[9]— command —↑↑

>
BP# 0 - Break at: "0x05E6:0x000B"
>

<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt> <Sh+F3=M1 Fmt> DEC

דיבאג

- כעת נפתח את חלון האוגרים ונתמקד בקוד הולאה שלנו על מנת למצוא את הביעות שבתכנה.



```
PrntLOOP:
    mov es:[si], ax

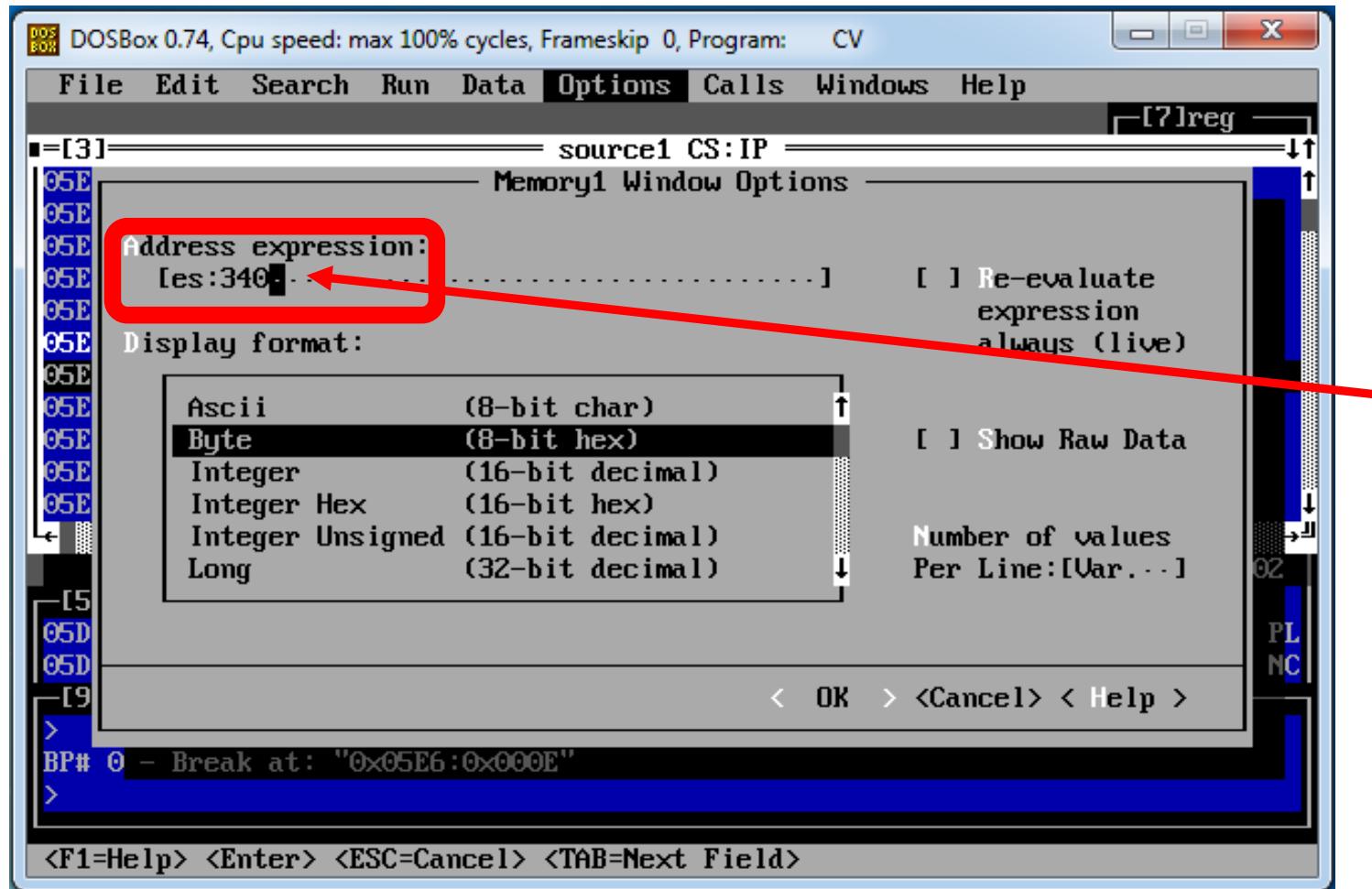
    inc ax ; move to next char to print
    inc si ; move to next place on screen

    dec cx
    jnz PrntLOOP
```

נבחן שהעתקת AX ל זיכרון,
העתקנו 2 bytes כאשר AL
קבע את הסימן המודפס,
-AH קבע את התא האחראי
לרקע וצבע, להיות 0.

כיוון תצוגת הזיכרון לסגמנט ES

- כדי לכוון את תצוגת הזיכרון לכתובת מסויימת נבחר "Memory Window" בלשונית "Options" בסרגל הכלים



- נלחץ עם העכבר על הכתובת שמופיעה ב Address expression ונכתב את הכתובת הרצiosa

תיקון ראשוני

- נעדכן את התוכנה שלנו להשתמש ב AL בלבד.

```
mov cx, 9
mov al, 1

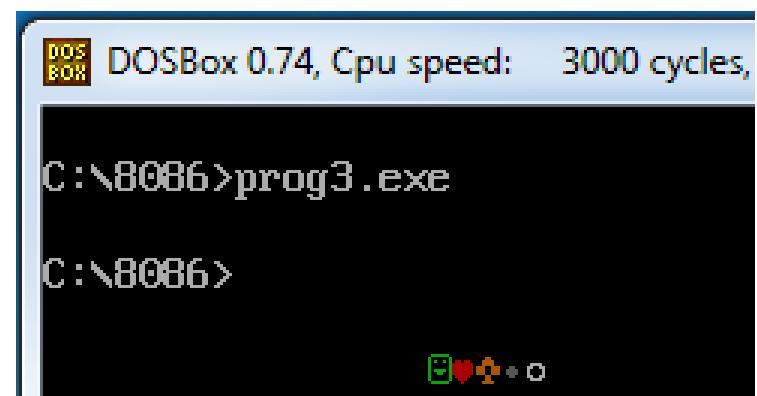
PrntLOOP:
    mov es:[si], al

    inc ax ; move to next char to print
    inc si ; move to next place on screen

    dec cx
    jnz PrntLOOP

    mov ax, 4c00h
    int 21h
end HERE
```

הפלט למסך השתנה קצר,
אך עדין לא מה שרצינו



דיבאג

- נרץ את התוכנה עד לכתיבת הראשתה למסר, ונלחץ על F4 כדי להציג את הפלט למסר

The screenshot shows two windows from the DOSBox 0.74 emulator. The top window displays assembly code in the source1 CS:IP pane:

Address	OpCode	Operands
05E6:0000	MOV	AX,B800
05E6:0003	MOV	ES,AX
05E6:0005	MOV	SI,0340
05E6:0008	MOV	CX,0009
05E6:000B	MOV	AL,01
05E6:000D	MOV	BYTE PTR ES:[SI],AL
05E6:0010	INC	AX
05E6:0011	INC	SI
05E6:0012	DEC	CX
05E6:0013	JNZ	000D
05E6:0015	MOV	AX,4000

The bottom window displays a memory dump in the memory1 b es:340 pane. A red arrow points to the byte at address B800:033E, which is highlighted in yellow and has the value 01.

At the bottom of the interface, there are several keyboard shortcut keys: <F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt> <Sh+F3=M1 Fmt> DEC.

The screenshot shows the DOSBox command prompt window with the following text:
C:\>8086>CU prog3.exe

נבחן שכתבנו בזיכרון את
הערך המספרי 01, בעוד
שכדי להציג על המסך דריש
הערך ה ASCII של התו

תיקון נוסף

- נתחל את AL בערך ה ASCII של 1.

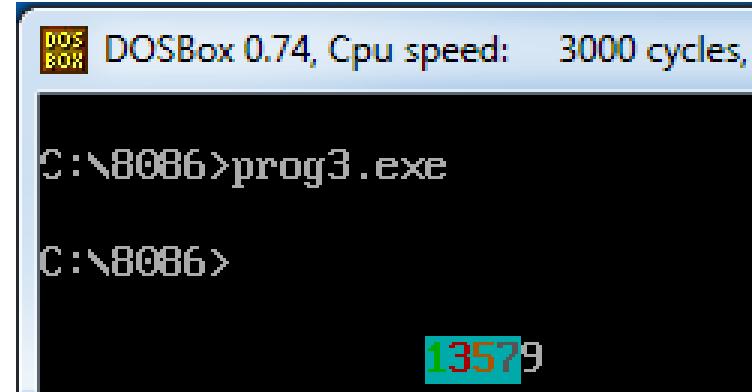
```
mov cx, 9
mov al, '1'

PrntLOOP:
    mov es:[si], al

    inc ax ; move to next char to print
    inc si ; move to next place on screen

    dec cx
    jnz PrntLOOP

    mov ax, 4c00h
    int 21h
end HERE
```

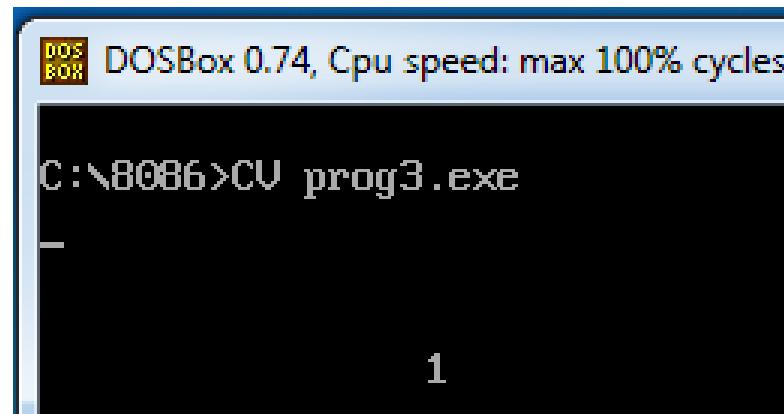


הפלט למסך הוא כעט מספרים,
אר לא בקפיצות של 2 וצבעים
משתנים

mov al, '1'

mov al, 31h

דיבאָג



- נרץ את התכנה עד להדפסה הראשונה, ונציג את הפלט.

[3]		source1	CS:IP
05E6:000B	B001	MOV	AL,01
05E6:000D	268804	MOV	BYTE PTR ES:[SI],AL
05E6:0010	40	INC	AX ;BK0
05E6:0011	46	INC	SI

- נמשיך עם ריצת התכנית, להדפסה הבאה, ונבחין שבסביב השני כתבנו לתא השם בעוד ש-2 כתובות רציפות אחראיות עלתו במסך.

**כדי להמשיך עם
הטכנית עד
להדפסה השנייה
פשוט נלחץ F5**

```
■=[5]===== memory1 b es:340 =====  
B800:0314 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07 . . . . .  
B800:0322 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07 . . . . .  
B800:0330 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07 . . . . .  
B800:033E 20 07 31 32 20 07 20 07 20 07 20 07 20 07 20 07 20 07 .12 . . . .  
B800:034C 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07 . . . . .
```

דיבאג

- נريץ את התוכנה שוב, והפעם לפני הסבר השני של הלוואה נעדכן ידנית את ערכו של אוגר IS ל-2 כתובות קדימה

The screenshot shows the DOSBox interface with several windows:

- Assembly Window:** Shows assembly code for source1 CS:IP. The instruction at address 05E6:000D is highlighted: `MOV BYTE PTR ES:[SI],AL`.
- Registers Window:** Shows register values. The CX register is highlighted with a yellow box, and the SI register is highlighted with a red box.
- Memory Dump Window:** Shows memory starting at address B800:0325. The bytes at B800:033F (31 07 32 07) are highlighted with a red box.
- Command Line Window:** Shows the command `C:\>CU prog3.exe`.
- Bottom Bar:** Contains keyboard shortcuts: F8=Trace, F10=Step, F5=Go, F3=S1 Fmt, Sh+F3=M1 Fmt.

8=CX מכאן שאנו חנו
באייטרציה השנייה

The command line window displays the output of the program:

```
DOSBox 0.74, Cpu speed: max 100% cycles,  
C:\>CU prog3.exe
```

עדכון ערכו של אוגר בזמן הרצה ב Debugger

- נלחץ על האוגר ונכתב את הערך הרצוי



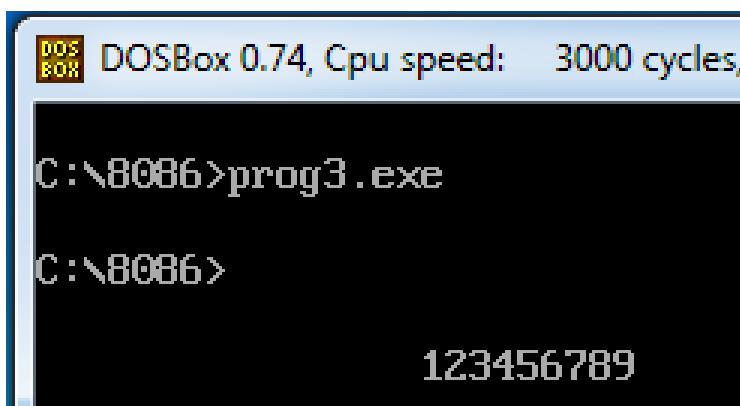
■=[7]reg =↓↑
AX = B832 ↑
BX = 0000
CX = 0008
DX = 0000
SP = 0100
BP = 0000
SI = 0041
DI = 0000
DS = 05D6
ES = B800
SS = 05E8
CS = 05E6
IP = 000D
FL = 0202

■=[7]reg =↓↑
AX = B832 ↑
BX = 0000
CX = 0008
DX = 0000
SP = 0100
BP = 0000
SI = 0042
DI = 0000
DS = 05D6
ES = B800
SS = 05E8
CS = 05E6
IP = 000D
FL = 0202

תכנית נכונה

- נעדכן את התכנה לקדם את IS ב-2 אינדיקטים.

```
.model small
.data
.stack 100h
.code
HERE:
; Build a code that counts from 1 to 9
; and prints to screen
    mov ax, 0B800h
    mov es, ax
    mov si, 340h ;screen offset
```



```
        mov cx, 9
        mov al, '1'

PrntLOOP:
        mov es:[si], al

        inc ax ; move to next char to print
        add si, 2h ; move to next place on screen

        dec cx
        jnz PrntLOOP

        mov ax, 4c00h
        int 21h
end HERE
```

איך "מתקנים" וכותבים תוכנה באSEMBLER ?

בנייה תוכנה – תיאור הבעייה

- נרצה לבנות תוכנה שمدפסה למסך את רצף האותיות הבא:

A BB CCC DDDD ...

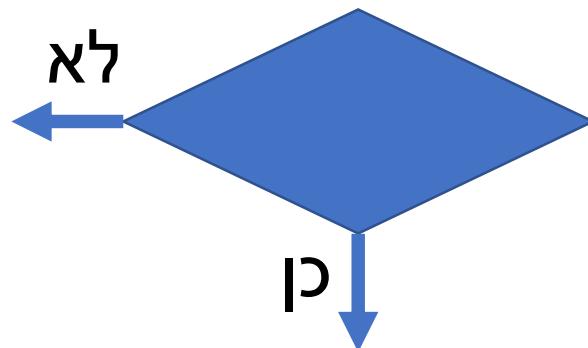
- קלומר מספר המופעים של כל אותו הוא האינדקס של ב-א"ב
- נכתב תוכנה שעושה את ההפסה עבר 4 האותיות הראשונות של ה-א"ב
- השלב הראשון הוא **להבין היטב את הדרישות:**
"מה התוכנה צריכה לעשות?"
- למרות שזאת תוכנה לא מסובכת, אם נתחיל ישר לכתב קוד בלי תוכנן,
אנו עלולים להסתבר.

תרשים זרימה

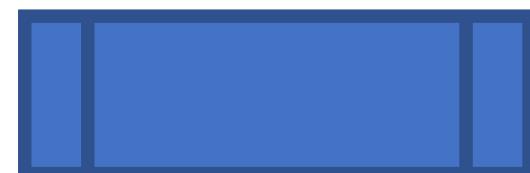
מצביע על הפעולה
הבא לבי嘱



מצין פעולה בקרה-
בדיקה תנאי כלשהו



מצין קטע של תכנית
שלא מפורט כאן



מצין תחילת או
סיום של תרשימים



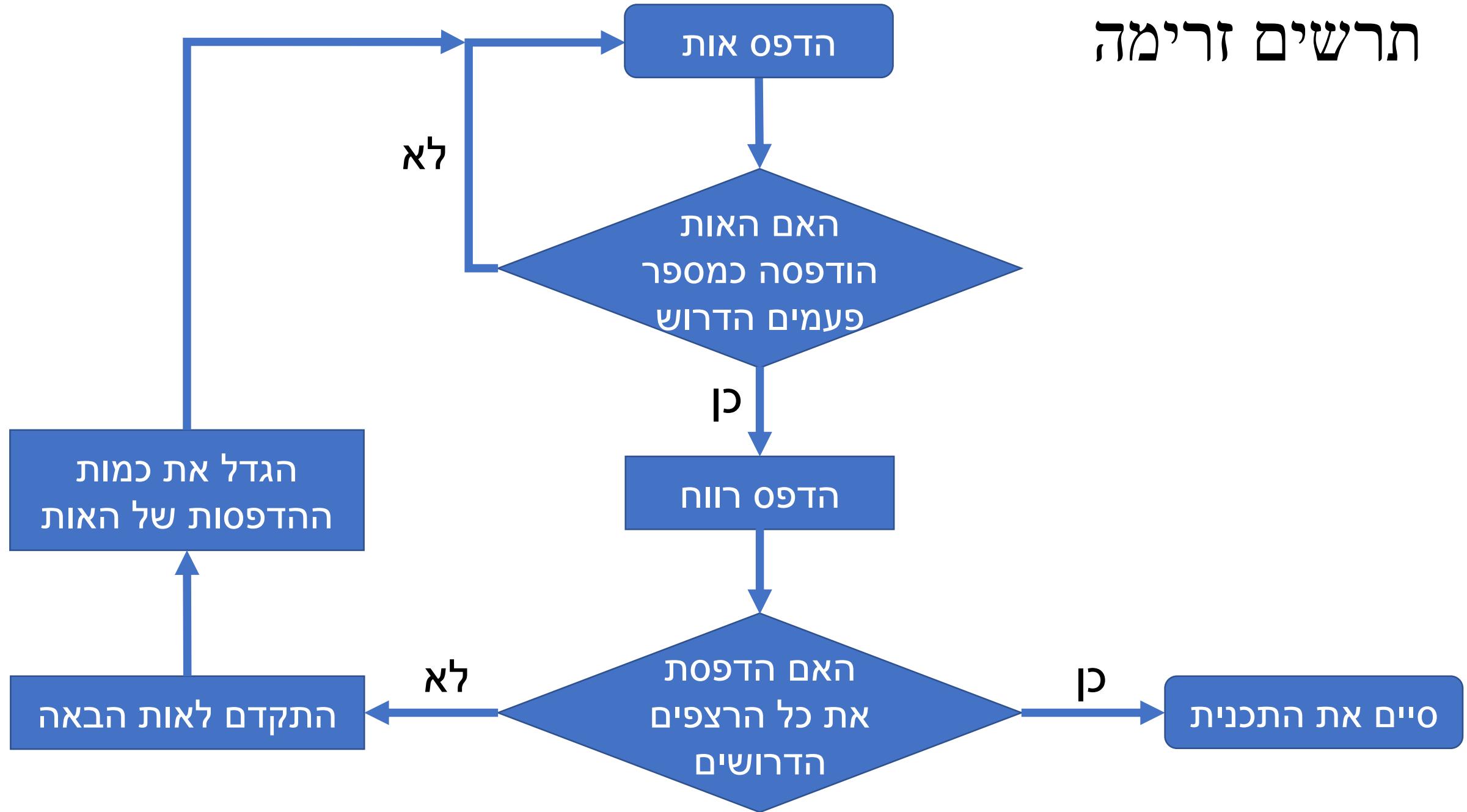
מצין פעולה של
קלט או פלט נתונים



מצין ביצוע הוראה
כמו השמה/חישוב...



תרשים זרימה



משתנים ולולאות

- נגידר משתנים (אגרים) ואת הפקידים שלהם בתכנית

A – איזה אות מופיעים

C – כמה אותיות שנשאר להדפס

B – מספר הפעמים שצריך להדפיס את האות

D – מספר הפעמים שנשאר להדפיס את האות הנוכחית

S – מקום (היט) ההדפסה במסמך

- אם צריך, נגידר גם משתנים / מערכיים בזיכרון וקבע את תפקידם

lolaoth

- הוכנה תדרוש שימוש ב-2 lolaoth:

1) lolaoת חיצונית שתעביר על כל האותיות

- lolaoת החיצונית מקדמת את AX, BX ורצה כמספר הפעמים שמוגדר ב CX

2) lolaoת פנימית שתדפיס כל אות את מספר הפעמים הדרש

- מדפסה למסך כמספר הפעמים ש CX מאותחל אליו

איזה אות מדפסים – AX

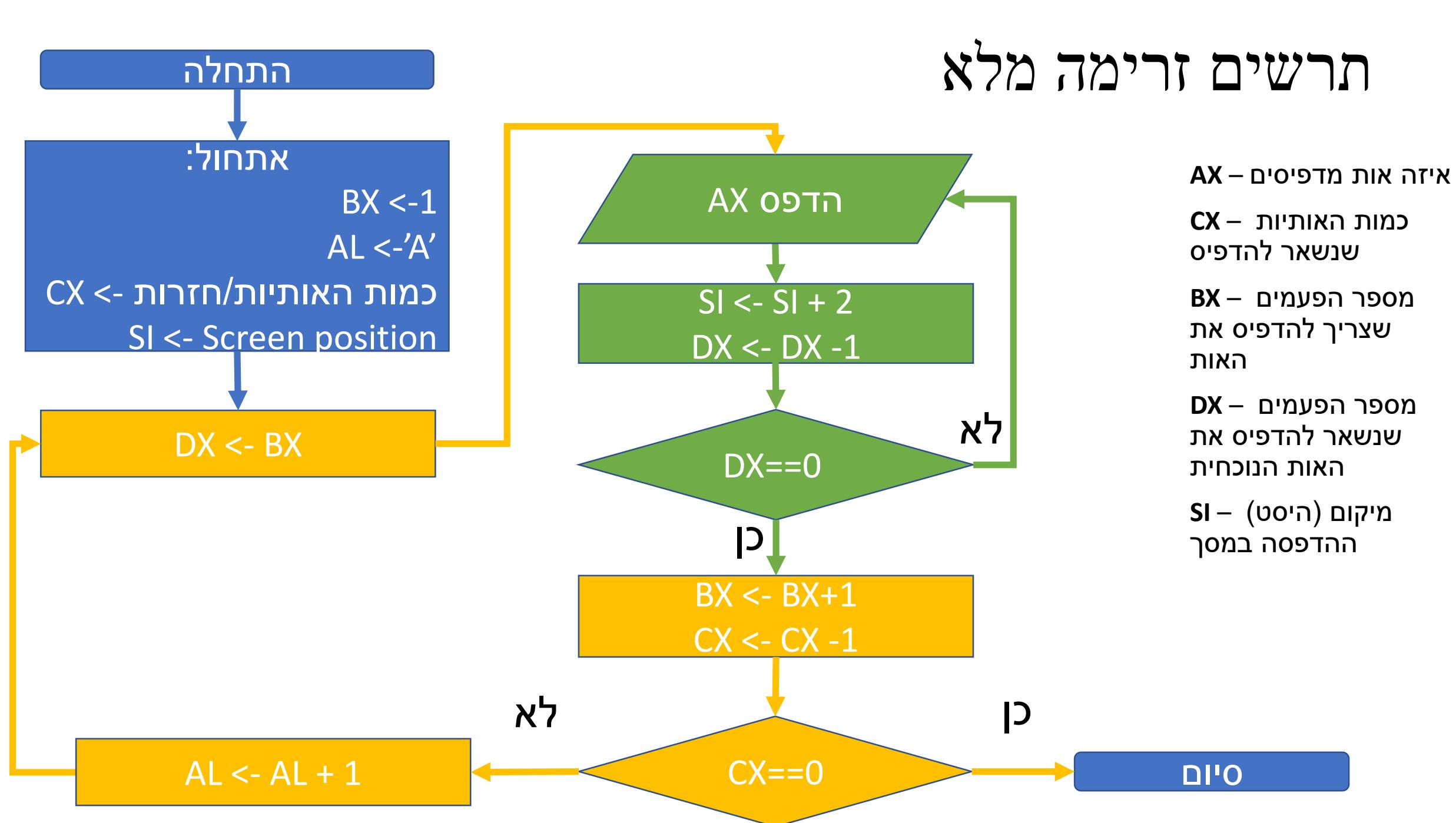
כמויות האותיות שנשאר להדפיס – CX

מספר הפעמים שצריך להדפיס את האות – BX

מספר הפעמים שנשאר להדפיס את האות הנוכחית – DX

מקום (היסט) ההדפסה בمسך – IS

תרשים זרימה מלא



כתבת קוד

```
.model small
.data
.stack 100h
.code
START:
    mov ax, 0b800h
    mov ds, ax ←
    ;Initial values of outer loop variables
    mov al, 'A' ← first letter to print is 'A'
    mov cx, 4h ; Print for first 4 letters
    mov bx, 1h ; First letter should appear 1 time
    ;Initial screen offset
    mov si, 340h
```

- בשלב הבא "נקודד" את תרשימים הזרימה לפקודות אסמבליר

כיוון שבתכנות הزاد אנו חנו לא מגדירים משתנים, אפשר לכוא את DS להציבו ל זיכרון המסר

גם כאן השתמש ב AL כדי לכתוב ל זיכרון את התו הרצוי בלי לשנות את הרקע

כתיבת קוד

LettersLoop:

```
mov dx,bx ;we dont want to change bx in the inner loop
```

PrintLoop:

```
mov ds:[si],al ;write to screen  
add si,2h ;forward screen offset  
dec dx ;decrease prints counter  
jnz PrintLoop
```

```
inc ax ;move to next letter  
inc bx ;increase appearance number
```

```
dec cx
```

```
jnz LettersLoop
```

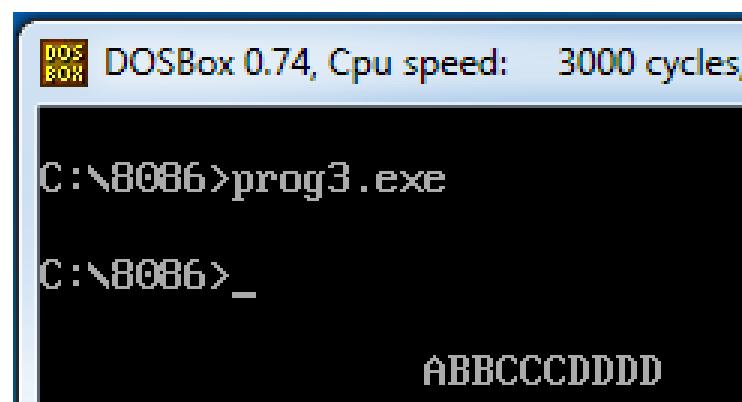
```
mov ax, 4c00h
```

```
int 21h
```

```
end START
```

לולאה
פנימית

אם נשתמש
ישירות ב BX
בתוך הלולאה
הפנימית, אז
נבד את המידע
שמור בו



איזה אות מדפסים – AX

כמה האותיות שנשאר להדפיס – CX

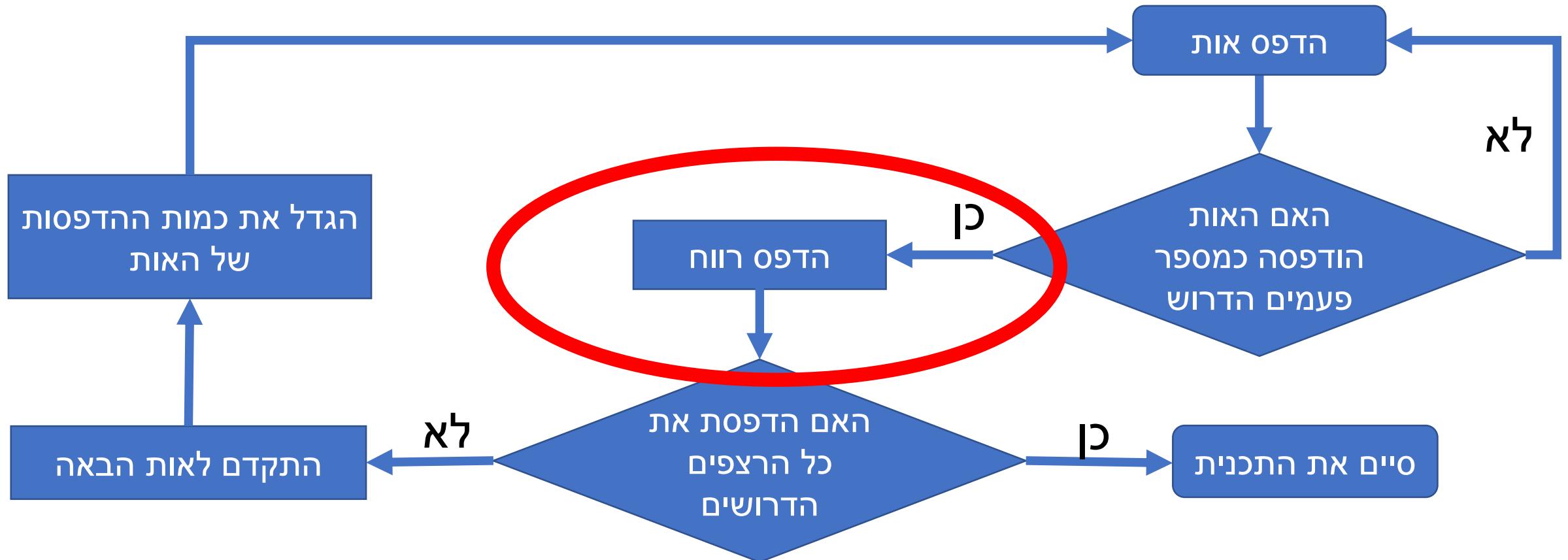
מספר הפעמים שצריך להדפיס את האות – BX

מספר הפעמים שנשאר להדפיס את האות הנוכחי – DX

מקום (היקט) ההדפסה ב מסך – SI

הוספה רוחחים

- הקוד שלנו עושה את מה שרצינו, פרט לרווחים.
- נוסיף את הדפסת הרוחחים בלולאה החיצונית, בהתאם לתרשים הזרימה שלנו



הוספה רוחחים

LettersLoop:

```
    mov dx,bx ;we don't want to change bx in the inner loop
```

PrintLoop:

```
    mov ds:[si],al ;write to screen
    add si,2h ;forward screen offset
    dec dx ;decrease prints counter
    jnz PrintLoop
```

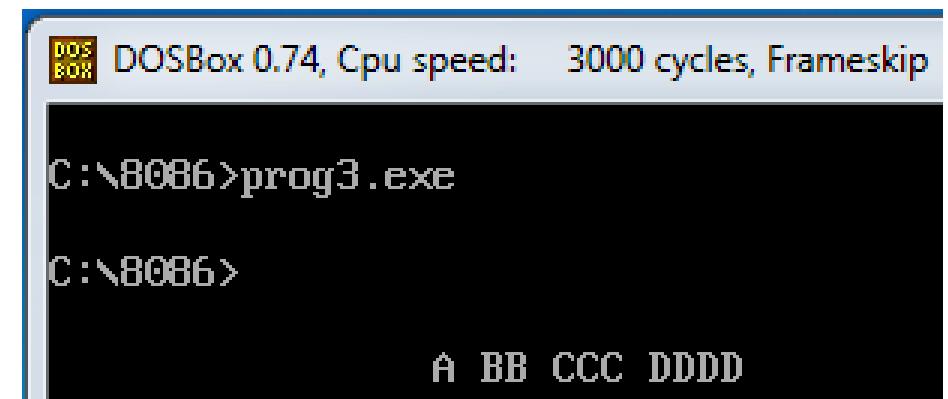
```
    mov ds:[si],' ' ;write space to screen
    add si,2h ;forward screen offset
```

```
    inc ax ;move to next letter
    inc bx ;increase appearance number
```

```
    dec cx
    jnz LettersLoop
```

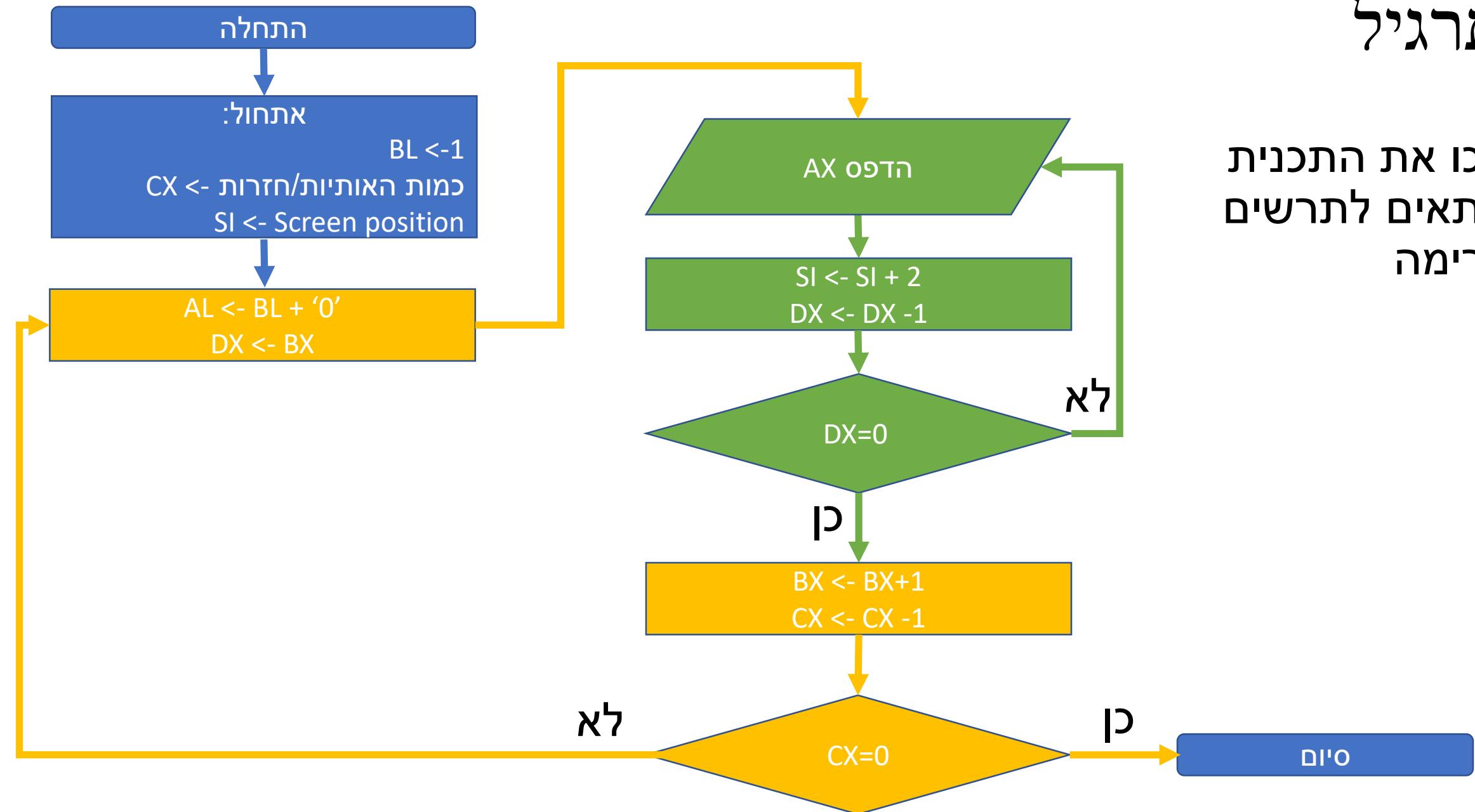
```
    mov ax, 4c00h
    int 21h
```

```
end START
```



תרגיל

ערכו את התכנית
להתאים לתרשימים
הזרימה



מיקרו-מעבדים ושפת אסמלר

תרגול מס' 7 – אוגר הדגלים והתנויות

מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן



אוגר הדגלים

- אוגר הדגלים הוא אוגר מיוחד שאפשר לבדוק אותו לאחר הפקודה, כדי לגלות אם התרחש משהו מיוחד בזמן הפקודה, למשל overflow בפקודת ADD

OF: Overflow

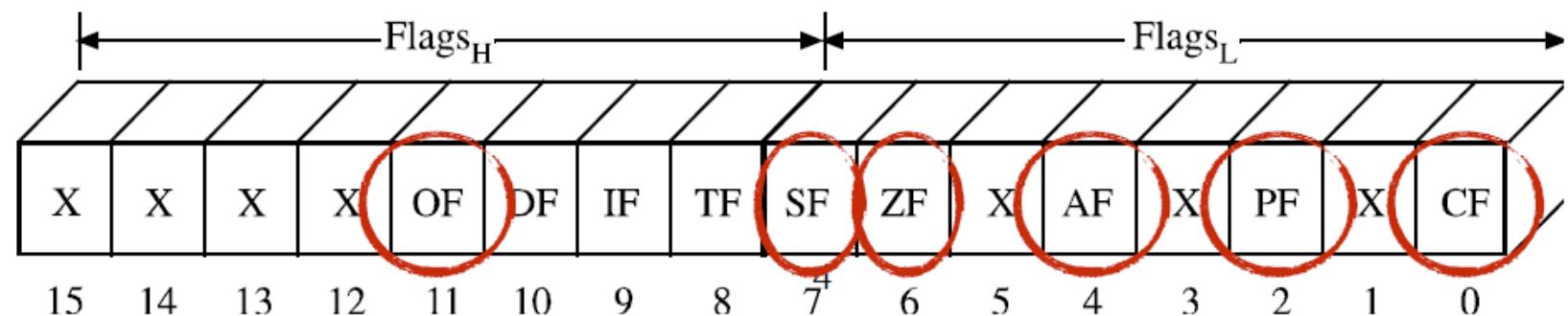
SF: Sign Flag (1 if result is negative)

ZF: Zero Flag

AF: Carry/Borrow on bit 3

PF: Parity Flag (Even number of 1's)

CF: Carry Flag



Zero Flag

- כשר נבצע פעולה חיסור (או כל פעולה אריתמטית אחרת), נוכל לקבל תוצאה שהיא 0.
- במקרה זה נדלק דגל ZF של המעבד.

mov bx, 20h

$$BX = 20h - 10h = \textcolor{red}{10h}$$

sub bx, 10h

$$ZF = 0$$

sub bx, 10h

$$BX = \textcolor{red}{10h} - 10h = 0$$

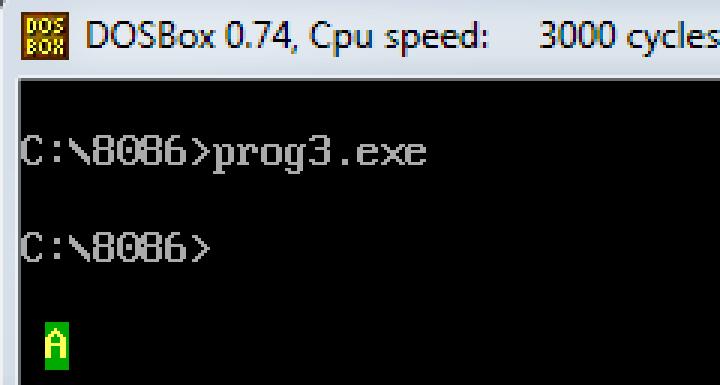
$$ZF = 1$$

פקודת ZJ ופקודת NZ

• אמ הדגל ZF דולק אז קפוץ.

```
.model small
.stack 100h
.code
START:
    ;setting extra segment to screen memory
    mov ax, 0b800h
    mov es, ax

    ;'A' on green background
    mov ax, 2E40h
```



```
mov bx, 20h
sub bx, 10h
jz L1
sub bx, 10h
jz L2
L1: mov es:[320h+0h], ax
L2: mov es:[320h+2h], ax

;return to OS
mov ax, 4c00h
int 21h
end START
```

BX = $20h - 10h = 10h$
ZF = 0

BX = $10h - 10h = 0$
ZF = 1

• אמ דגל ZF כבוי אז קפוץ.

לולאה ע"י שימוש בZF

- פקודת קפיצה מותנית מאפשרת לנו לכתוב לולאות.
- מבנה בסיסי של לולאה:

```
mov cx, 10d
L1: ; Loop Content goes
      ; here
dec cx
jnz L1
; Code to run after loop
```

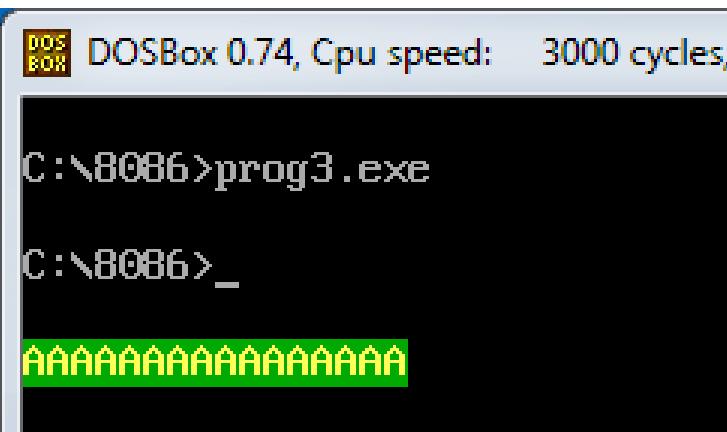
כל איטרציה CX מופחת ב-1.
לאחר 10 פעמים תוצאה פועלות החישור היא 0
והקפיצה חוזרת ל L1 לא תבוצע.

פקודת LOOP

- הולאה תבצע כמספר הפעמים שרשום באוגר CX

```
.model small
.stack 100h
.code
START:
    ;setting extra segment to screen memory
    mov ax, 0b800h
    mov es, ax

    ;'A' on green background
    mov ax, 2E40h
```



```
        mov bx, 320h ;screen offset

L1:   mov cx, 10h ;loop counter
      mov es:[bx], ax
      add bx, 2h
      loop L1

      ;return to OS
      mov ax, 4c00h
      int 21h
end START
```

loop L1

sub cx, 1
jnz L1

פקודת CMP

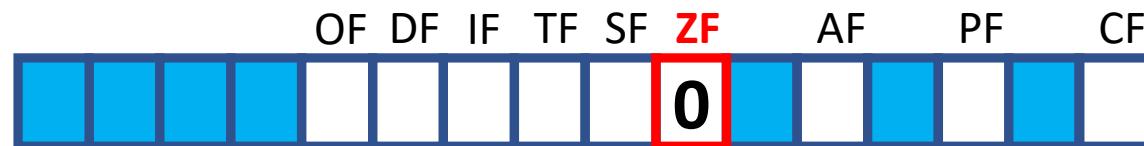
- פקודת בקירה לבדיקה>If חסמים בין שני אוגרים. הפקודה לא משנה את האוגרים שעליהם מבוצעת, אך משנה את אוגר הדגלים.
- כאשר נכתב **H CMP AL,BH** המעבד יבצע את הפעולה SUB AL,BH מבלתי לשמר את התוצאה ב AL. בהתאם לשינוי הדגלים נדע האם שווים, או מי גדול/קטן ממי

mov ax, 3Fh

mov bx, 00h

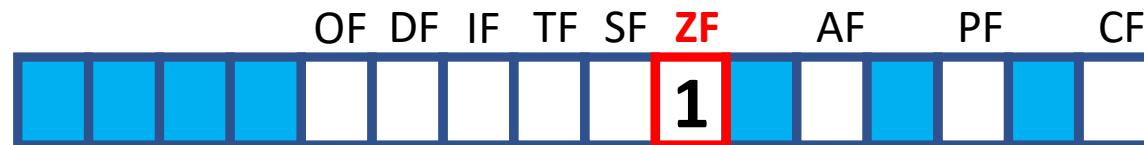
cmp ax, bx

- לדוגמה – אם ערכי האוגרים שוויים, דגל ZF יידלק



mov bx, 3Fh

cmp ax, bx



JB מול JB

- 2 הפקודות מבצעות קפיצה במקרה שהאופrnd הראשון ב-CMP קטן מהשנ
- JB מניח שפקודת CMP הופעלה על שני מספרים UNSIGNED

mov al, 00111111b

mov bl, 11111111b

cmp al, bl

jb L1

הפקודה בודקת את CF
כדי להכיר אם לקפוץ

אשר המספרים UNSIGNED הער
של BL יותר גדול משל AL ולכן נקפוץ

JB מול JL

- JL מניח שה-CMP הzbוצע על שני מספרים SIGNED ביצוג משלים ל-2

```
mov al, 00111111b
```

```
mov bl, 11111111b
```

```
cmp al, bl
```

```
jl L1
```

הפקודה בודקת את הדגלים $OF \neq SF$ כדי להכריע אם לkapoz

כasher המספרים SIGNED הערך של AL יותר גדול משל BL וולכן לא נקפוז

JB מול LJ

- האם הפקודה שבאה לפני LJ/JB חייבת להיות CMP?
- תשובה: לא
 - כל פקודה שמשנה את אוגר הדגלים יכולה לקבוע את דגלי הקפיצה.
 - אך קיימת הנחה חבוייה ש כדי הפקודה תעבור צפוי (With Jump) הפקודה שלפני אמורה להיות השוואה או חיסור

sub al, bl

jb L1

JB מול JL

- האם מבחינת אוגר הדגלים, זה משנה מה הפקודה שבאה אחרי פקודת ?CMP

mov ax, 00111111b

mov bx, 11111111b

cmp al, bl

ZF=0, OF=0, CF=1, SF=1



L1 **jb** L1 L2 **jl** L2

- תשובה: לא

- בפעולה אחת CMP מצליח לשנות את כל הדגלים באופן כזה שמתאים לכל פקודת הקפיצה שיכולה לבוא אחריו (גם הדגלים שרלוונטיים לחישוב UNSIGNED, וגם אלו שרלוונטיים ל贤וצאת SIGNED כמו SF)

BL מולTL

- האם נקפוץ כתוצאה מהפקודות:

```
mov al, 00111111b
```

```
mov bl, 11111111b
```

```
cmp al, bl
```

```
mov bl, 00000011b
```

```
jb L1
```

- תשובה: כן
- פקודת MOV לא משנה את אוגר הדגלים, שכן עדין נקפוץ בגלל שערך הדגלים לא השתנו

לולאת WHILE

- נרצה לכתוב תוכנית שمدפסה אותיות מ A עד E ללא שימוש באוגר מונה CX

```
AL = 'A'  
while AL < 'F' {  
    print AL  
    AL = AL + 1  
}
```

- בשביל בדיקת תנאי הלולאה בלי לשנות את תוכן האוגרים נשתמש בפקודת CMP בשילוב עם פקודת JB

```
cmp al, 'F'  
jb L1
```

מדליק את CF כאשר AL בעל ערך
קטן מערך האסקי של האות 'F'
מכבה את CF אחרת.

לולאת WHILE

```
.model small
.stack 100h
.code
START:
    ;setting extra segment to screen memory
    mov ax, 0b800h
    mov es, ax

    ; 'A' on green background
    mov ax, 2E41h
```

DOSBox 0.74, Cpu speed: 3000 cycles,

```
C:\8086>prog3.exe
C:\8086>
ABCDE
```

WHILE

```
        mov bx, 320h ;screen offset
L1:
    mov es:[bx], ax
    add bx, 2h
    inc al
    cmp al, 'F'
    jb L1
    ;return to OS
    mov ax, 4c00h
    int 21h
end START
```

תנאי IF

- נוכל להשתמש בפקודות קפיצה מותנית גם בשביל כתיבת תנאי If-Else

```
cmp OPERAND1 , OPERAND2
```

```
jnz LabelFalse
```

```
; Do if True
```

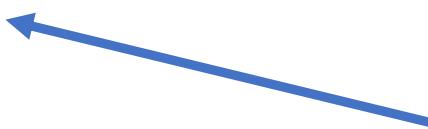
```
jmp LabelAlways
```

```
LabelFalse:
```

```
; Do if False
```

```
LabelAlways:
```

```
; Do After if
```

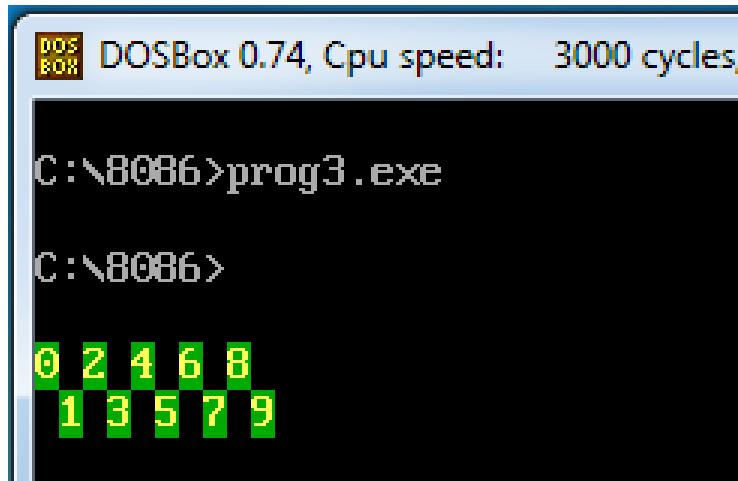


במקרים שבהם ישנו זריקה לפקודה CMP
אחרת כמו INC,DEC,ADD...
שמדליה דגל באוגר הגדלים.

גם את פקודת הקפיצה אפשר
להחליף בכל פקודה קפיצה אחרת
כמו ...JB,JL

תנאי IF

- נרצה להדפיס את המספרים הזוגיים בשורה ראשונה ומספרים לא זוגיים בשורה מתחת



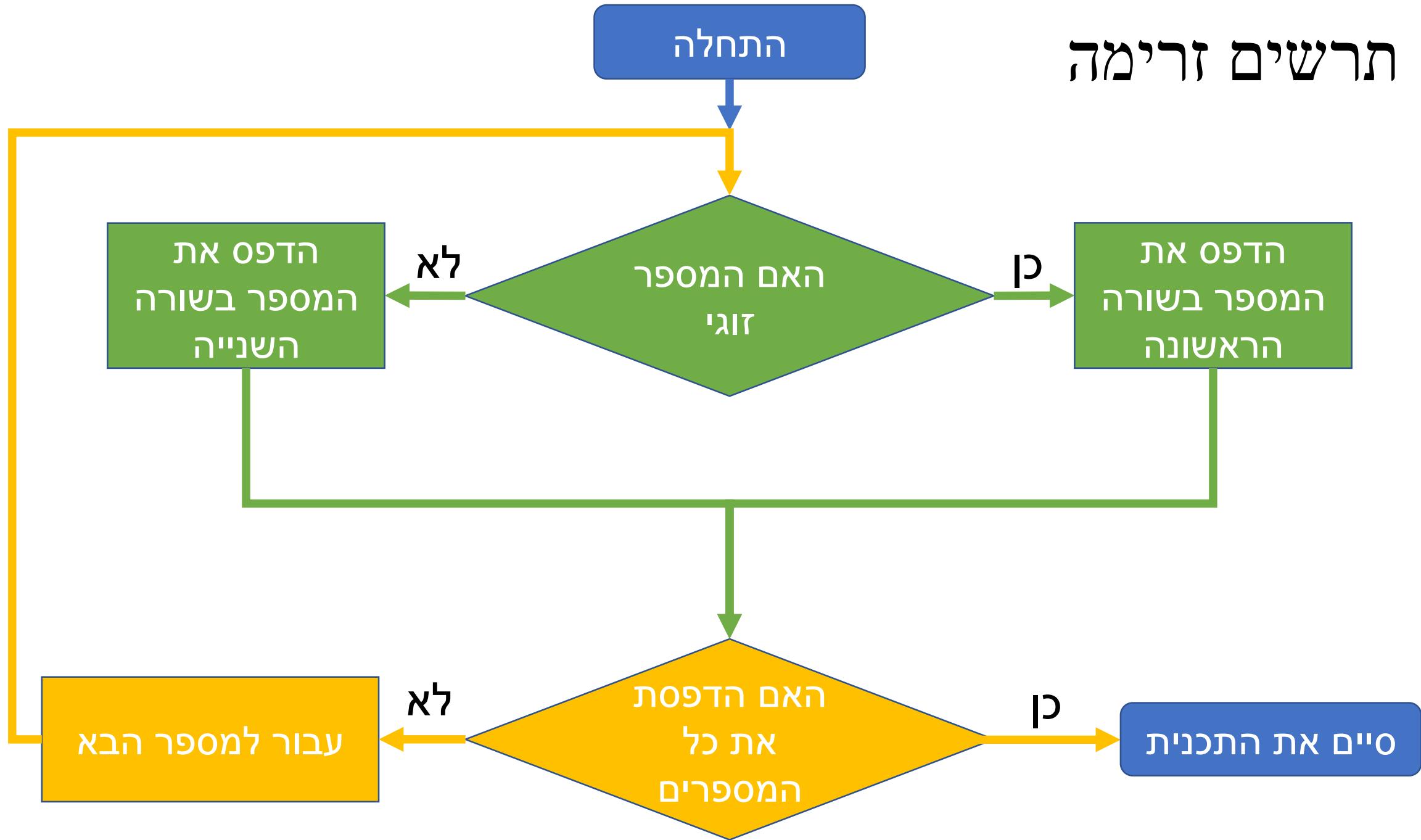
- נאתחל את ES להצביע לזיכרון המסך
ואת AX למספרה '0' על רקע ירוק:

```
.model small
.stack 100h
.code
START:
        ;setting extra segment to screen memory
        mov ax, 0b800h
        mov es, ax

        ; '0' on green background
        mov ax, 2E30h

        ;Screen offset
        mov bx, 320h
```

תרשים זרימה



תנאי IF

`mov dx, 1h`

פעולות בהדפסת מספר זוגי

{
`mov es:[bx], ax`
`dec dx`

פעולות בהדפסת מספר אי-זוגי

{
`mov es:[bx+0a0h], ax`
`inc dx`

נגידיר את AX להיות דגל האם המספר שכרגע מדפיים הוא זוגי או אי זוגי.

0=DX משמעתו שכרגע מדפיים מספר אי זוגי.

1=DX משמעתו שכרגע מדפיים מספר זוגי.

נשייר את 0 לזוגיים (AX מאותחל ל-0)

תנאי IF

אם DX הוא אפס אז
נקפוץ לפעולות שעושים
בהדפסת מספר אי-זוגי

```
cmp dx, 0
jz LabelOdd
```

LabelEven:

```
mov es:[bx], ax
dec dx
```

אם DX שונה מ-0 אז
לא נקפוץ, ונגיע לקוד
שעושים בהדפסת
מספר זוגי

LabelOdd:

```
mov es:[bx+0a0h], ax
inc dx
```

תנאי IF

```
cmp dx, 0  
jz LabelOdd
```

LabelEven:

```
mov es:[bx], ax  
dec dx  
jmp LabelAlways
```

נוסיף את פקודת הקפיצה כדי לא לבצע את הפעולות של הדפסת מספר אי-זוגי

LabelOdd:

```
mov es:[bx+0a0h], ax  
inc dx
```

בסוף כל הדפסה (גם של מספר זוגי וגם של אי-זוגי) נרצה לקדם את AL לאות הבאה, ולהתקדם בהיסט המסר

LabelAlways:

```
add bx, 2h  
inc al
```

תנאי IF

```
mov dx, 1h  
mov cx, 10d  
L1:  
    cmp dx, 0  
    jz LabelOdd  
        mov es:[bx], ax  
        dec dx  
    jmp LabelAlways
```

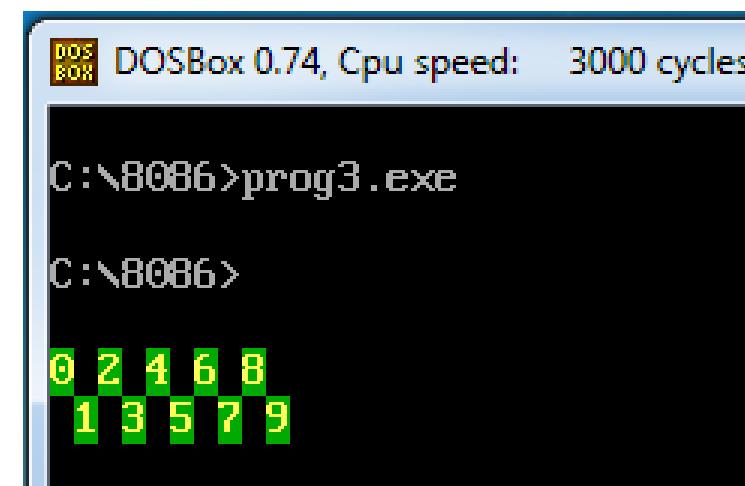
LabelOdd:

```
    mov es:[bx+0a0h], ax  
    inc dx
```

LabelAlways:

```
    add bx, 2h  
    inc al  
    loop L1
```

נבע את ההוראות
שלאן בולאה



```
; return to OS  
mov ax, 4c00h  
int 21h  
end START
```

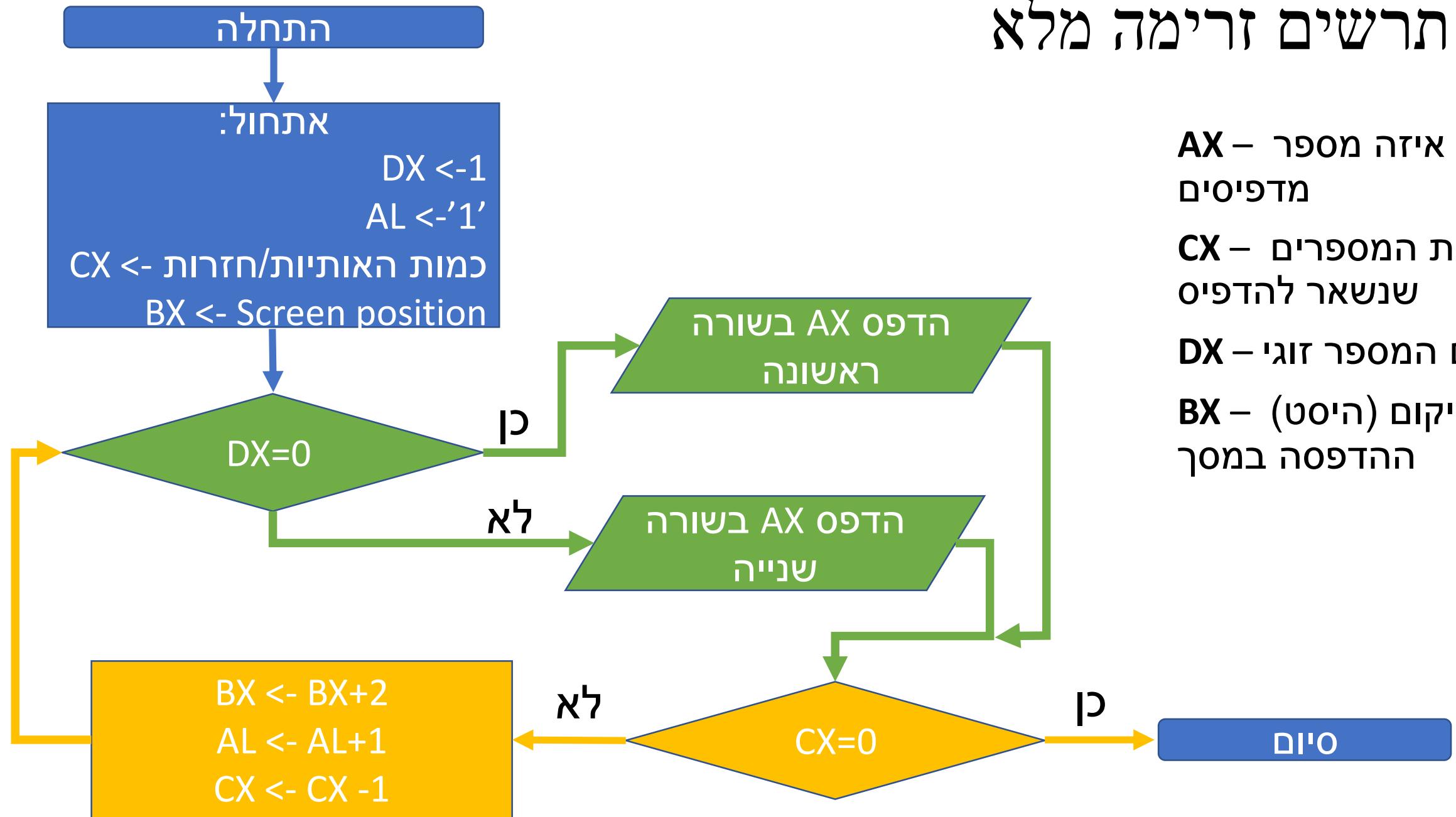
תרשים זרימה מלא

איזה מספר – AX מדפים

כמויות המספרים – א' שנסחר להדפס

האם המספר זוגי –

מקום (היסט) – X הדרופה במסר



מיקרו-מעבדים ושפת אסמלר

תרגול מס' 8 – המחסנית, הנחיות מהדר

מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן

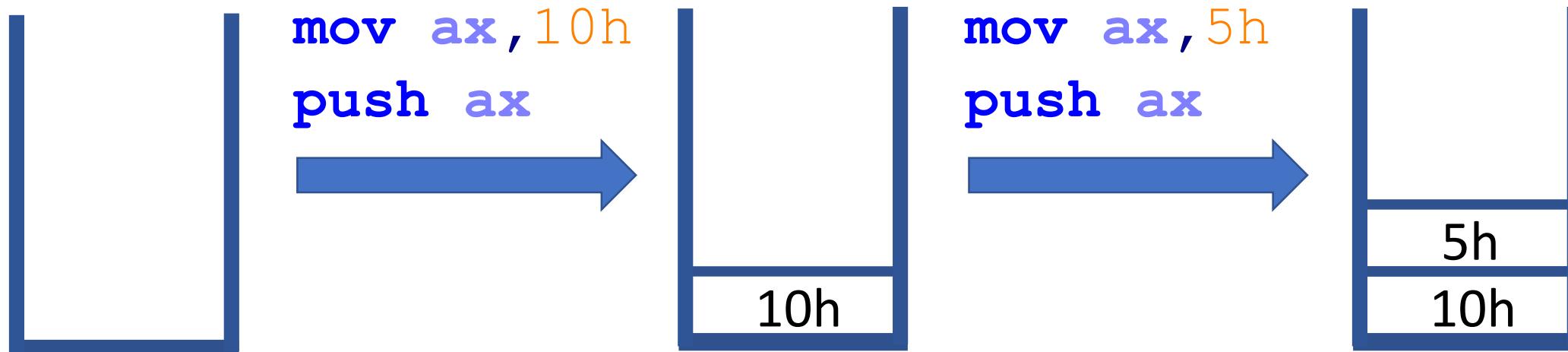


מחסנית

.stack 100h

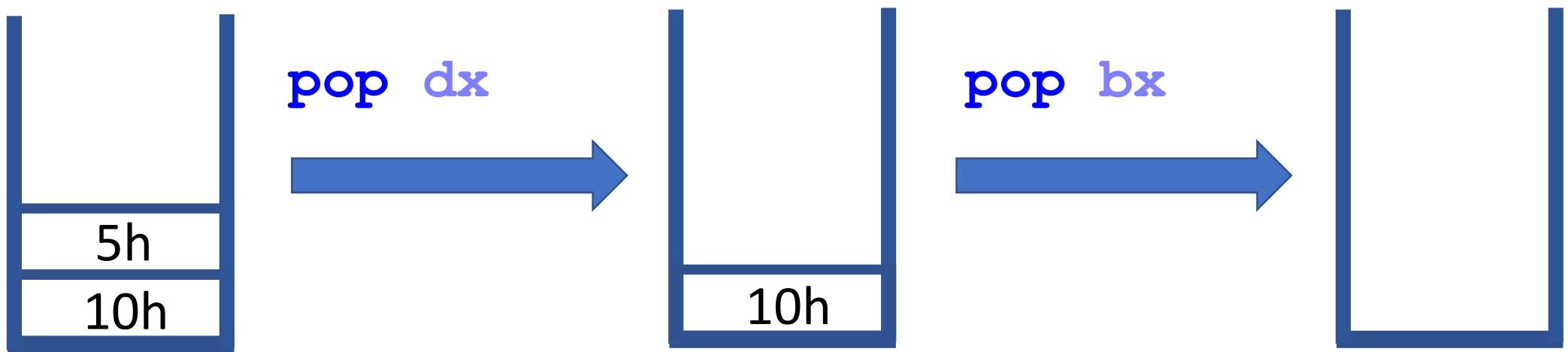
- הגדרת מחסנית בגודל 100h

- המחסנית היא Last In First Out המכוון לראש המחסנית



מחסנית

- מוציאים איברים מראש המחסנית (Last In)



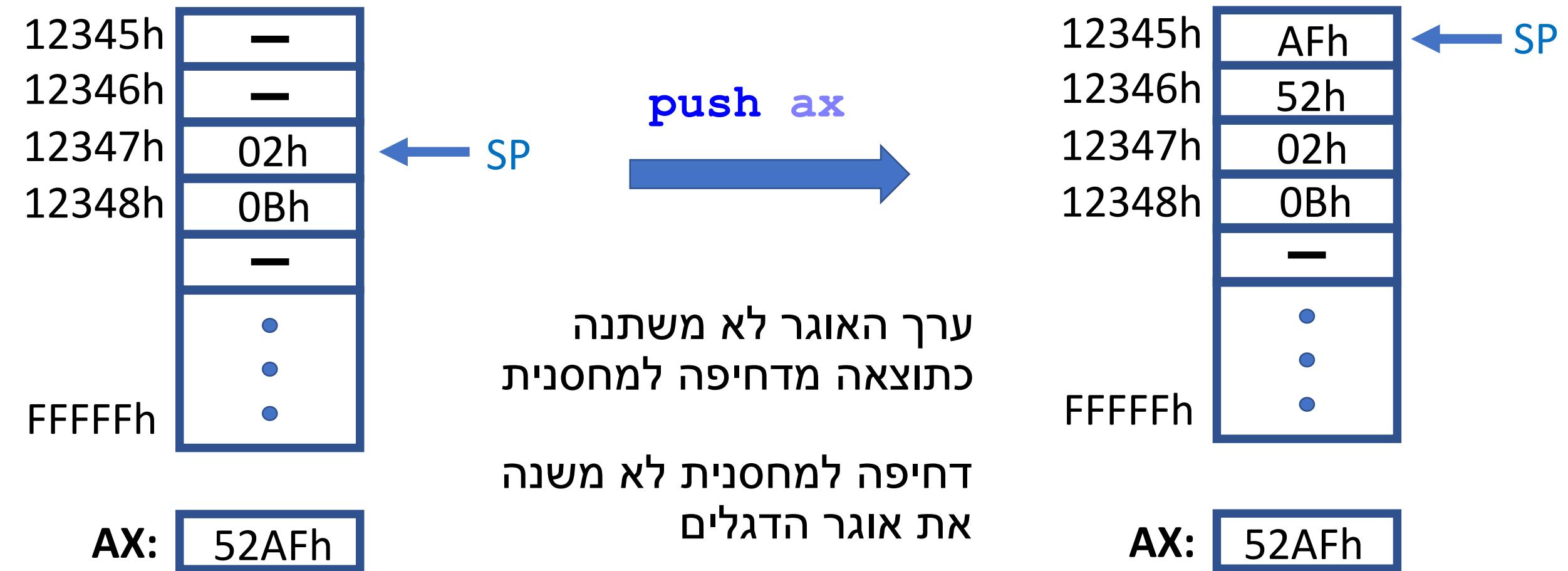
DX: 5h

DX: 5h

BX: 10h

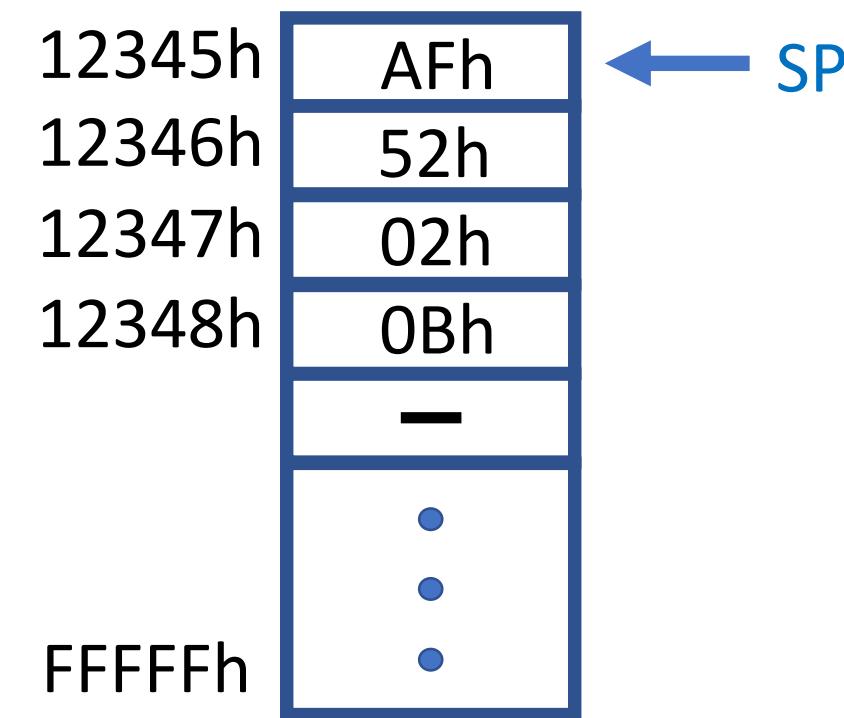
מחסנית

- בפועל המחסנית היא אזור בזיכרון, וכך למש את תכונת LIFO משתמש באוגר SP כמצבי לראש המחסנית



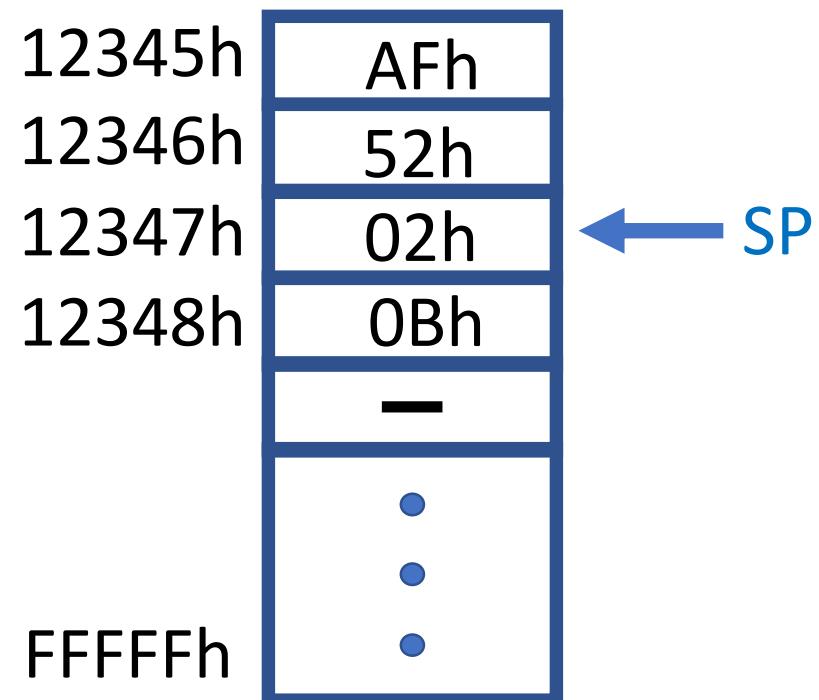
מחסנית

- בפועל המחסנית היא אזור בזיכרון, וכך למש את תכונת LIFO משתמש באוגר SP כמצבי לראש המחסנית



שליפה מהמחסנית לא
מוחקמת דברים מהזיכרון

BX: 12B0h



BX: 52AFh

מחסנית

- הפקודות שאחרי PUSH ולפני POP לא משפיעות על ערכו של CX לאחר פקודה POP

mov cx, 30h

CX = 30h

push cx

CX = 30h

sub cx, 30h

CX = 0h

add cx, 20h

CX = 20h

inc cx

CX = 21h

pop cx

CX = 30h

לולאה כפולה ע"י שימוש במחסנית

```
sub cx, 1  
jnz L1
```

```
loop L1
```

- פקודת LOOP תמיד עובדת עם CX

- כיצד נוכל למשולש לולאה כפולה ע"י שימוש בפקודת LOOP?

```
mov cx, 10h
```

```
L1:
```

```
push cx
```

```
; Inner Loop
```

```
pop cx  
loop L1
```

שומרים במחסנית את ערך CX השיר לlolאה החיצונית

לולאה כפולה ע"י שימוש במחסנית

- תכנית שمدפסה מושלש של ספרות, כאשר בכל שורה מספר המופעים של הספרה שווה לערך הספרה

1
22
333
4444

```
.model small
.stack 100h
.code
START:
    ; setting extra segment to screen memory
    mov ax, 0b800h
    mov es, ax

    ; 'A' on green background
    mov al, '1'
    mov ah, 2Eh

    ; Screen offset
    mov di, 340h
```

mov al, 31h

לולאה כפולה ע"י שימוש במחסנית

```
mov cx, 5h  
L1:  
    push cx ; save outer loop counter
```

```
    mov cl, al }  
    sub cx, 30h }
```

המרת ערך ASCII של המספר
השמור בAX למספר עצמו

```
L2:  
    mov es:[di], ax  
    add di, 2h  
loop L2
```

```
inc al  
add di, 0a0h ;move to next line
```

```
pop cx ;pop outer loop counter  
Loop L1
```

באייטרציה ראשונה:

AL = '1' = 31h

CX = AL - 30h = 1h

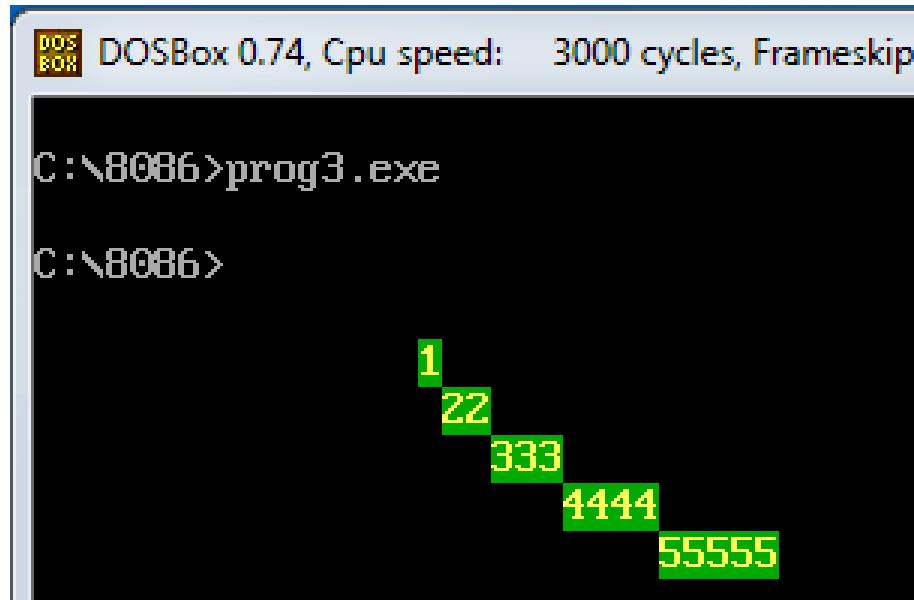
באייטרציה שנייה:

AL = '2' = 32h

CX = AL - 30h = 2h

```
;return to OS  
mov ax, 4c00h  
int 21h  
end START
```

לולאה כפולה ע"י שימוש במחסנית



הסיבה להזחות, היא שככל מחרוזת
של הלולאה הפנימית מזיזים את DI
קדימה בתוך השורה, ולא מזיזים
אותו חזרה לתחילת השורה
בלולאה החיצונית

- נרץ את התכנית ונקבל:

```
mov cx, 5h
L1:
    push cx ; save outer loop counter

    mov cl, al
    sub cx, 30h

    L2:
        mov es:[di], ax
        add di, 2h
    loop L2

    inc al
    add di, 0a0h ; move to next line

    pop cx ; pop outer loop counter
Loop L1
```

לולאה כפולה ע"י שימוש במחסנית

```
mov cx, 5h
L1:
    push cx ; save outer loop counter

    mov cl, al
    sub cx, 30h

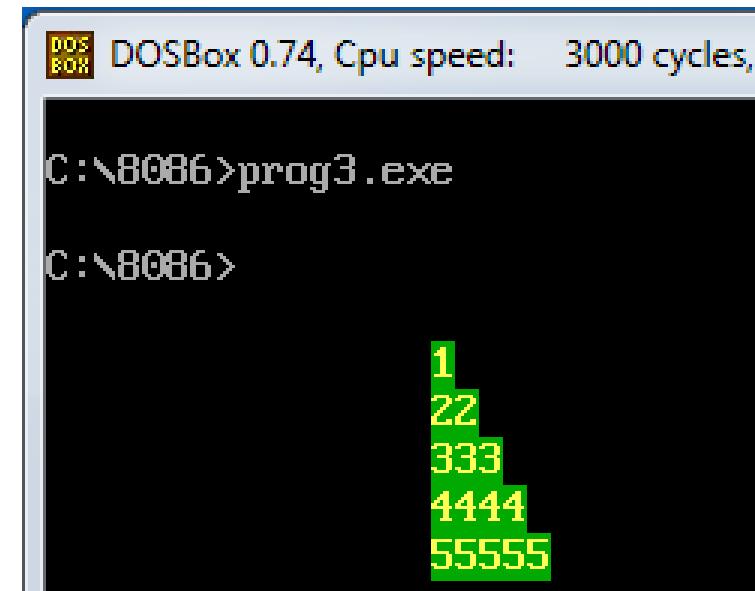
    push di ←
L2:
    mov es:[di], ax
    add di, 2h
loop L2
pop di

inc al
add di, 0a0h ; move to next line

pop cx
Loop L1
```

נשמר את הערך של CX לתוך
המחסנית לפני הקידום שלו
בלולאה.

בסוף הלולאה, נעשה POP
לCX שמצוין לתחילת השורה.



לולאה כפולה ע"י שימוש במחסנית

```
mov cx, 5h  
L1:  
    push cx ; save outer loop counter
```

```
    mov cl,al  
    sub cx, 30h  
    push cx
```

```
L2:  
    mov es:[di], ax  
    add di, 2h  
loop L2
```

```
inc al  
add di, 0a0h ;move to next line
```

```
pop cx  
pop cx  
Loop L1
```

- תרגיל: ערכו את התכנית להשתמש בדחיפות של CX בלבד

נתחיל לרשום את מספר האיטרציות של הלולאה הפנימית לתוך המחסנית

כיוון שביצענו 2 פקודות PUSH, נצטרך גם 2 פקודות POP כדי להחזיר את הערך של CX למונח של הלולאה החיצונית

לולאה כפולה ע"י שימוש במחסנית

```
mov cx, 5h
```

L1:

```
push cx ; save outer loop counter
```

```
mov cl,al
```

```
sub cx,30h
```

```
push cx ; save number of inner loop iterations
```

L2:

```
mov es:[di], ax
```

```
add di,2h
```

```
loop L2
```

```
inc al
```

```
add di, 0a0h ;move to next line
```

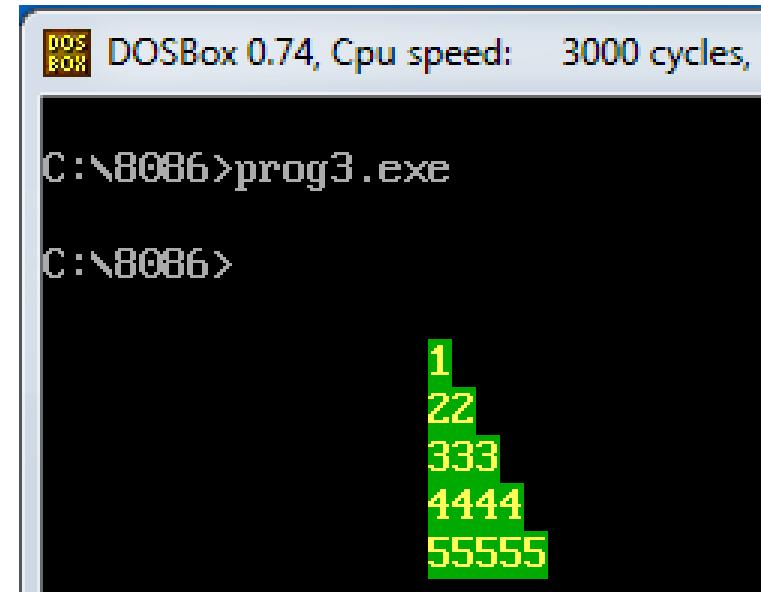
```
pop cx
```

```
shl cx,1
```

```
sub di,cx
```

```
pop cx
```

```
Loop L1
```

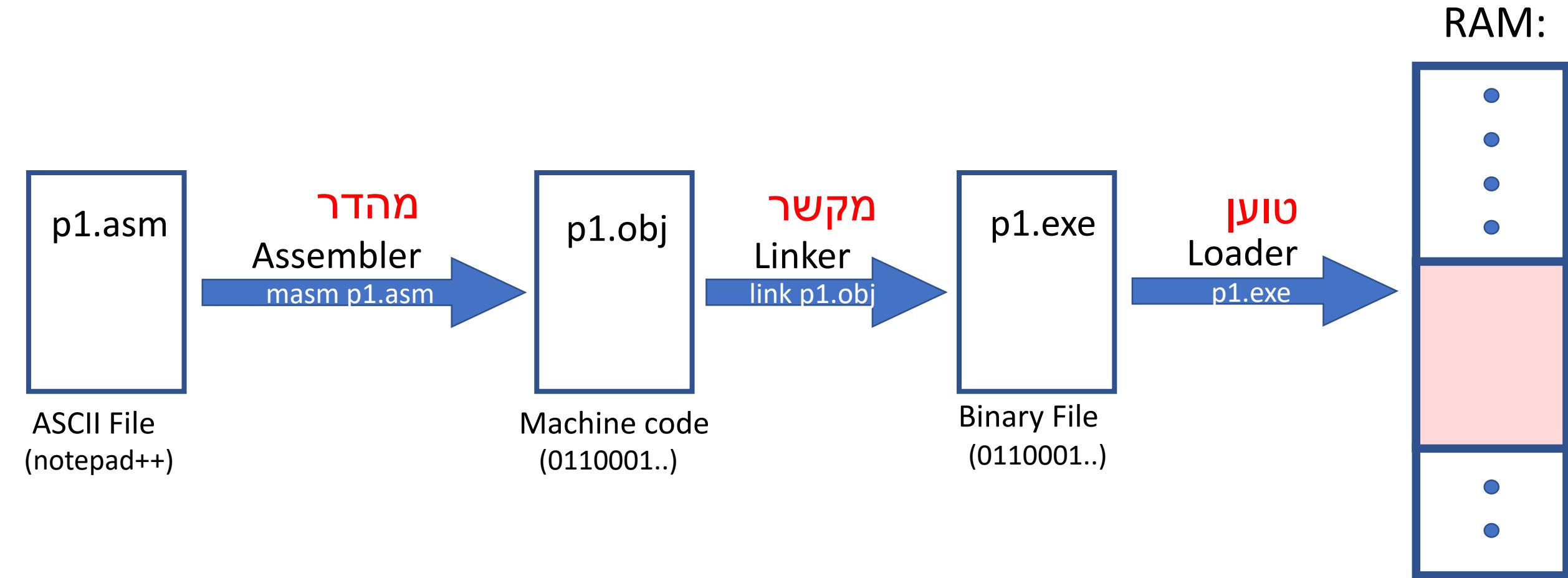


הזה שמאלה בביט אחד של מספר
בינארי שקול להוגית לכפול ב-2.

נרצה לכפול את CX ב-2 כי כל
איטרציה של הלולאה הפנימית אנחנו
מקדמים את DI ב 2 bytes

הנחיות מהדר...

שרשרת קומpileציה והרצה



שלב האסמבלר

- המהדר מתרגם את הקובץ ASCII לפקודות מכונה
- חלק מהשירותים המופיעות בקובץ ASCII הן לא פקודות של ממש, הן הנחיות למהדר

p1.asm

```
.model small
N EQU 4h
.stack 100h
.code
START:
    mov ax, @data
    mov ds, ax
    mov cx, N
L1: sub cx, 1h
    jnz L1
    .exit
end START
```

מהדר
Assembler
masm p1.asm

p1.obj

```
B8E705
8ED8
B90400
83E901
75F8
B44C
CD21
```

MOV	AX,05E7
MOV	DS,AX
MOV	CX,0004
SUB	CX,01
JNZ	0005
MOV	AH,4C
INT	21

פקודה מול הנטית מעבד

`mov es, ax`

- פקודה היא הוראה למעבד שמתבצעת בזמן ריצה
- פקודה מתרגמת לשפת מכונה (instruction) ע"י המהדר



- הנטית מהדר, היא הוראה למהדר שמתבצעת בזמן הקומpileציה
- מנהה את המהדר כיצד לבנות את הקובץ הבינארי (ולא דווקא מתרגמת לשפת מכונה)

`N EQU 4h`

כאשר המהדר רואה `lines_N` בקוד, הוא יודע להחליף

אותו ב7 (למשל בפקודה `mov N, cx`)



דוגמאות נוספות להבנת מהדר (control-flow directives)

.model **small**

- הנחיה כיצד לחלק התכנית לSEGMENTS

.stack 100h

- הנחיה להקצת מחסנית של 100h

proc **near**

- הנחיה שאומרת שקובעת אם בסיום הפקציה נחזיר לקוד המבוצע בפקפית

proc **far**

near או בפקפית far

- נזכיר בהרצאה ותרגולים הבאים...

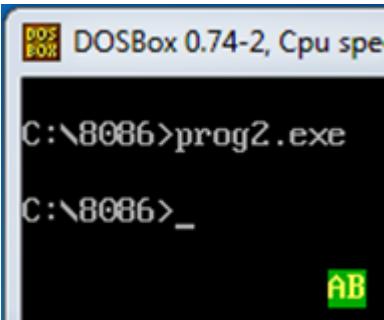
מакרו

- מאקרו זהה הינה לkompiiler למקם מסוים בлок נתון של קוד

```
.model small
.stack 100h

write_letters MACRO
    mov es:[510h], 2E41h ;write to screen
    mov es:[512h], 2E42h ;write to screen
ENDM
```

הkompiiler מחליף את השורה
בתוכן של המاكרו



```
.code
START:

;setting data segment
mov ax, @data
mov ds, ax

;screen memory
mov ax, 0b800h
mov es, ax

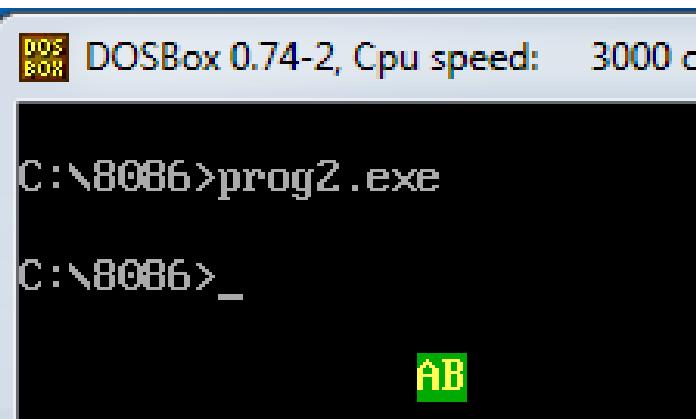
write_letters

    mov ax, 4c00h
    int 21h
end START
```

מакרו עם פרמטרים

```
.model small  
.stack 100h  
  
set_es MACRO pSeg  
    mov ax, pSeg  
    mov es, ax  
ENDM
```

כאשר המהדר מחליף את הליבל בפקודות המופיעות בתוך המакרו, הוא גם מחליף כל מופיע של Seg בפרמטר שקיבל



```
.code  
START:  
    ;setting data segment  
    mov ax, @data  
    mov ds, ax  
  
    set_es 0b800h  
  
    write_letters  
  
    mov ax, 4c00h  
    int 21h  
end START
```

מאקרו מבצע החלפת מחרוזות

```
set_bx_and_es MACRO reg  
    mov bx, reg  
    mov es, reg  
ENDM
```

```
.code  
  
set_bx_and_es ax  
  
set_bx_and_es 05h
```

- כאשר הקומפיילר "ראה" מאקרו הוא מבצע:
 1. החלפת מחרוזות (כולל הפרמטרים)
 2. תרגום הקוד המתkeletal לאסמבילר
- תקינות הקוד המתkeletal תלויות בפרמטר שהעבכנו למאקרו

8BD8	MOV	BX,AX
8EC0	MOV	ES,AX

```
Assembling: prog.asm  
prog.asm(49): error A2070: invalid instruction operands  
    set_bx_and_es(2): Macro Called From  
    prog.asm(49): Main Line Code
```

הנחיות EXIT ו STARTUP

```
.model small
.stack 100h
.data
    greenA DW 2E41h

.code
    .startup
        mov ax, 0b800h
        mov es, ax

        mov ax, greenA
        mov es:[340h], ax

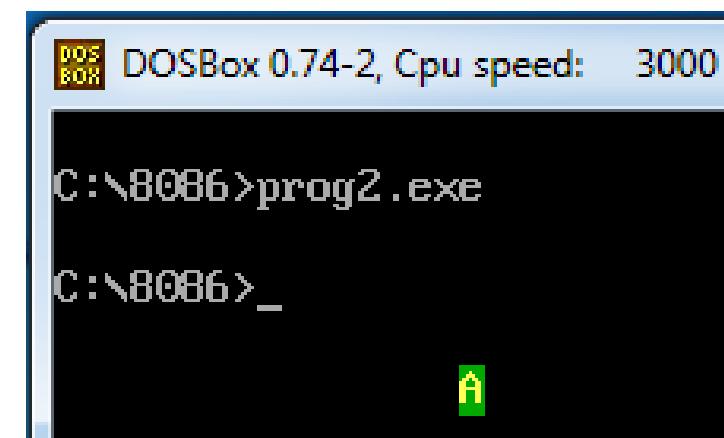
    .exit
end
```

הנחייה שמקמת את הקוד שמאתחל
את אוגרי הסגמנט בתחילת התכנית

```
mov ax, @data
mov ds, ax
:
```

מקם את הקוד
שאחראי לסיום
התכנית

```
mov ax, 4c00h
int 21h
```



מַאֲקָרֶו הָוֹא לֹא פּוֹנְקָצִיה!

- גם מַאֲקָרֶו וגם פּוֹנְקָצִיה - מקום ייחידי שמרכז את הפוקודות
- אם רוצים לשנות - משנים במקום ייחיד
- הוראת מַאֲקָרֶו "פִּיזִית" מחליפה את הליבל המופיע בקוד, בפקודות המופיעות במאקו.
- כלומר, אם נכתב את המַאֲקָרֶו 10 פעמים, זה יכולוشرשمنו את בלוק הקוד 10 פעמים.
- ביצוע פּוֹנְקָצִיה, זה קפיצה למקום אחר בסגמנט, וביצוע ההוראות המופיעות במקום זה.
- אם נקרא לפּוֹנְקָצִיה 10 פעמים, התוכנה תמיד ת Kapoor לבצע את אותו הקוד
- מכאן שביצוע של מַאֲקָרֶו הוא יותר מהיר כיון שאין את הקפיצה

גם ליבל הוא הנחיה מהדר

L1: sub cx, 1h
jnz L1

- **לייבל – מוחלף בכתובת הפיזית בזמן הידור- קישור**

■=[3]=	source1 CS:IP =
05E6:0008	83E901 SUB CX, 01
05E6:0008	75FB JNZ 0008

- **הגדרת משתנה – הקצאת תא בזכרון וקריאה לו בליבל מסוים**

.data
VR DB 5
.code
mov al, VR

■=[5]=	DS	memory1 b ds:0x0000 =
05E7:0000	B4 4C CD 21 05	89 44 0A 33 C0 89 44

■=[3]=	source1 CS:IP =
05E6:0005	A00400 MOV AL,BYTE PTR [0004]

מעון עקייף (מצבייע)

הנחיית OFFSET

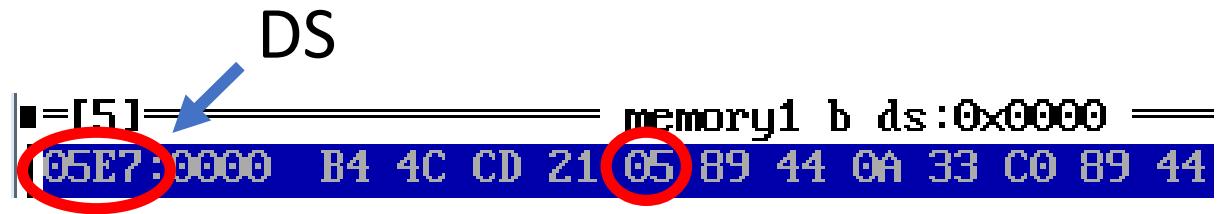
- הנחיה לנקח את הכתובת של המשתנה (ולא את הערך שלו)

.data

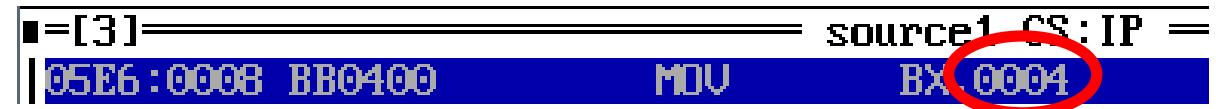
VR DB 5

.code

mov bx, offset VR



המשנה VR ממוקם בבית הרביעי
מתחלת סגמנט הנתונים



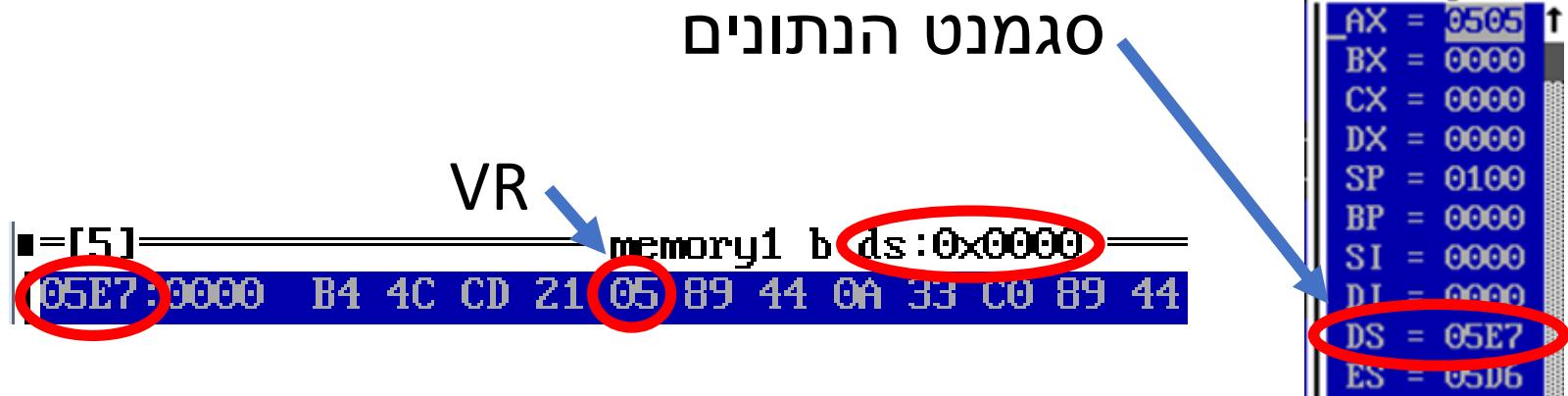
הכתובת של המשתנה VR (4) מועתקת לBX

הנחיית SEG

- הנחייה ללקחת את ערך הsegment שבו נמצא המשתנה

.data

VR DB 5



.code

mov bx, seg VR

■=[3]= source1 CS:IP =
| 05E6:0008 BBE705 MOV BX 05E7

הנחית DUP

ARRAY0 DB 2h, 5h, 1h

- הנחית להקצת מערך עם 3 תאים, כל תא הוא בגודל byte ("ARRAY0" וلتת לו את השם "ARRAY0")

ARRAY1 DB 2h dup (5)

לפני שהמادر מתחילה לבנות את קובץ jbo הוא מחליף את הפקודה DUP במספרים

- הנחית DUP אומרת לקומpileר לשכפל את 2h מספר הפעמים שרשום בסוגרים (5)

ARRAY0 DB 2h, 2h, 2h, 2h, 2h

מיקרו-מעבדים ושפת אסמלר

תרגול מס' 9 – שגרות

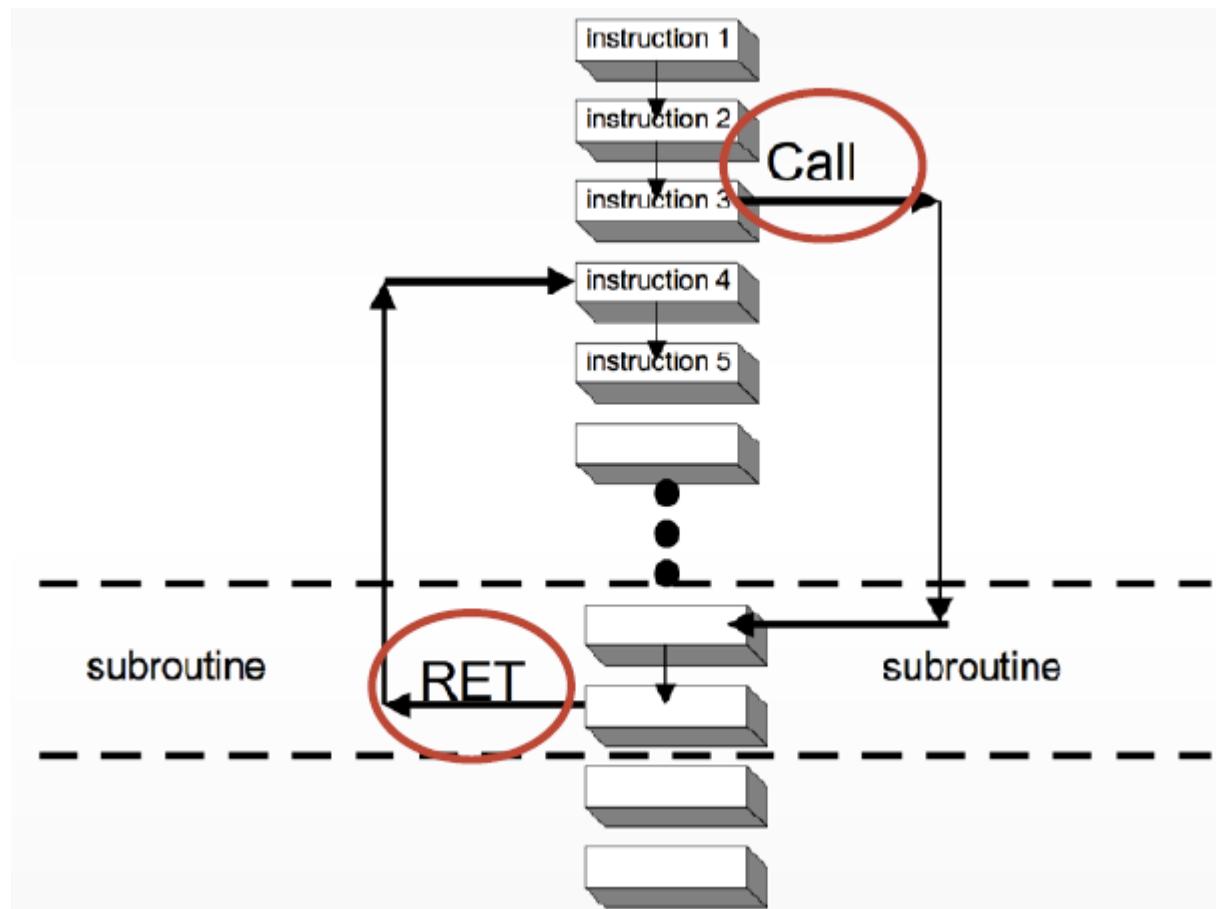
מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן



שגרות (פונקציות)

- כארש יש איזהו קטע קוד ש מורכב ממספר שורות, ואנחנו משתמשים בו יותר מפעם אחת בתוכנית, נרצה להוציא אותו לפונקציה.



INSTRUCTION 3
קריאה לפונקציה

מבנה של תכנית עם פונקציה

```
.model small  
.stack 100h  
.code  
HERE:
```

```
mov ax, 0B800h  
mov es, ax
```

```
call print_A
```

```
mov al, 'B'  
mov es:[342h], al
```

```
mov ax, 4c00h  
int 21h
```

```
print_A proc  
    mov al, 'A'  
    mov es:[340h], al  
    ret  
print_A endp
```

```
end HERE
```

הקריאה לפונקציה היא חלק מפקודות התכנית

קריאה לפקודות סיום התכנית

הקוד של הפונקציה נמצא באזורי שאלוי התכנית לא תגיע ללא קריאה מפורשת לפונקציה



מבנה של תכנית עם פונקציה

```
.model small  
.stack 100h  
.code  
  
print_A proc  
    mov al, 'A'  
    mov es:[340h], al  
    ret  
print_A endp  
  
HERE:  
    mov ax, 0B800h  
    mov es, ax  
  
    call print_A  
  
    mov al, 'B'  
    mov es:[342h], al  
  
    mov ax, 4C00h  
    int 21h  
  
end HERE
```

- הפונקציה יכולה להיות ממוקמת גם באזור אחר של סegment הקוד.

הקוד מתייחס כאן, لكن ניתן גם לכתוב את הפונקציה לפני הליבל (עדין בסegment הקוד)

הפקודה HERE end מגדרה את הליבל שבו מתחילה התכנית

מבנה של תכנית עם פונקציה

- כאשר אנחנו כתבים תוכנית שימושת בפונקציות, האם אנחנו חייבים להקצות מקום לSEGMENT המחסנית?
- תשובה: כן
 - פקודת CALL מבצעת PUSH IP
 - ופקודת RET מבצעת POP IP

```
.model small  
.stack 100h
```

העברה נתון לפונקציה דרך אוגר

ABCDE

JKLMN

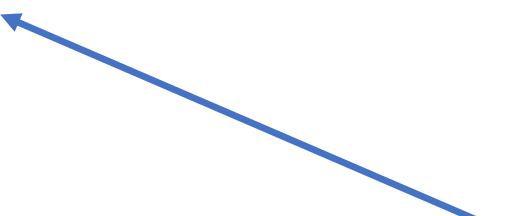
- נרצה לכתוב תכנית שמדפיסה:
- זהה למשה הדפסת האות A ו5 אותיות העוקבות שלה, ירידת שורה
והדפסת האות L ו5 אותיות העוקבות שלה.
- ע"מ לא לכתוב קוד פעמיים, אנחנו נגדיר שגרה שמקבלת את ראשונה
ומדפיסה 5 אותיות שלאחריה

העברת נתונים לפונקציה דרך אוגר

- נכתב פונקציה שמדפיסה את 5 האותיות העוקבות לאות שנמצאת באוגר AL, החל מהמקום שוגדר ע"י האוגר DI

```
; Inputs:  
; AL - starting letter (ascii)  
; DI - starting screen position  
; Outputs: None
```

```
print_SEQ proc  
    mov cx, 5h  
    L1:  
        mov es:[di], al  
        inc al  
        add di, 2h  
    Loop L1  
    ret  
print_SEQ endp
```



נכתב Header שגדיר
למשתמש מה הקלטים
והפלטים של הפונקציה

העברה נתון לפונקציה דרך אוגר

- למעשה אנחנו "מעבירים" לפונקציה את AL ו DI בתור פרמטרים.

```
.model small
.stack 100h
.code
HERE:
    mov ax, 0B800h
    mov es, ax

    mov di, 340h
    mov al, 'A'
    call print_SEQ

    add di, 0a0h
    add al, 9d
    call print_SEQ

    mov ax, 4c00h
    int 21h
```

הדפסת רצף ראשון

אחרי הדפסת הרצף
הראשון נוסיף 0A0h
להיסט המסר כדי לרדת
שורה

ונקדם את AL ב 9 אותיות
ע"י פעולה חיבור (לאות)

```
print_SEQ proc
    mov cx, 5h
    L1:
        mov es:[di], al
        inc al
        add di, 2h
    Loop L1
    ret
print_SEQ endp
end HERE
```

העברת נתונים לפונקציה דרך אוגר

```
.model small
.stack 100h
.code
HERE:
    mov ax, 0B800h
    mov es, ax

    mov di, 340h
    mov al, 'A'
    call print_SEQ

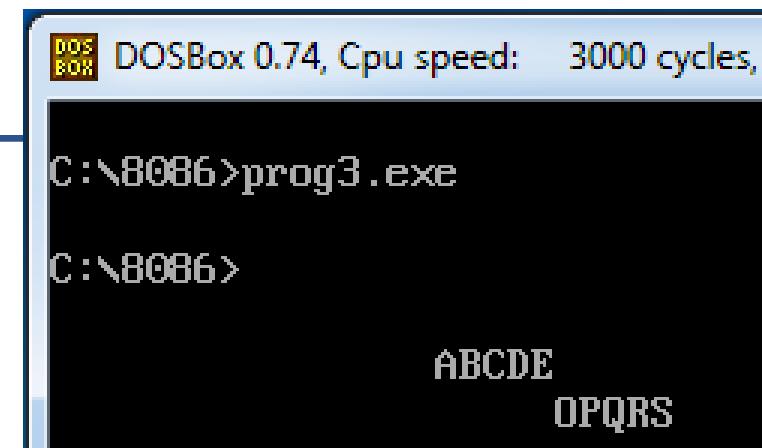
    add di, 0a0h
    add al, 9d
    call print_SEQ

    mov ax, 4c00h
    int 21h
```

ירידת שורה ב DI וקידום
של AL ב 9 אותיות
קדימה (לאות ג)

- הסיבה להזחה היא שהפונקציה משנה את הערכים של CX,DI,AL,CX
מקומות ימינה, ושל AL ב 5 אותיות קדימה

```
print_SEQ proc
    mov cx, 5h
    L1:
        mov es:[di], al
        inc al
        add di, 2h
    Loop L1
    ret
print_SEQ endp
end HERE
```

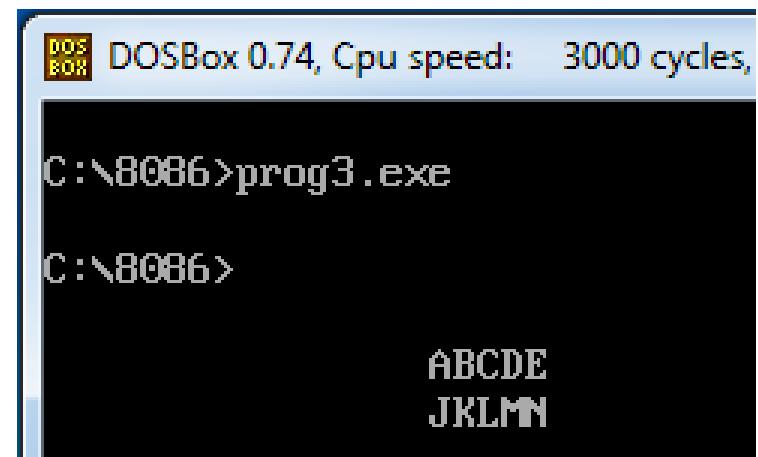


העברת נתונים לפונקציה דרך אוגר

- כדי לא ליצור צורך במעקב תמידי אחרי מה הפונקציה משנה ומה לא, בתחילת הפונקציה תמיד נדחוף למחסנית את הערבים ההתחלתיים של האוגרים שלא מוגדרים כ-"פלט" ובסיום הפונקציה נעשה להם POP

```
print_SEQ proc  
    push cx  
    push ax  
    push di  
  
    mov cx, 5h  
    L1:  
        mov es:[di], al  
        inc al  
        add di, 2h  
    Loop L1  
  
    pop di  
    pop ax  
    pop cx  
    ret  
print_SEQ endp
```

שליפה מהמחסנית
בסדר הפוך להכנסה



העברת נתונים דרך המחסנית

```
.model small  
.stack 100h  
.code  
HERE:
```

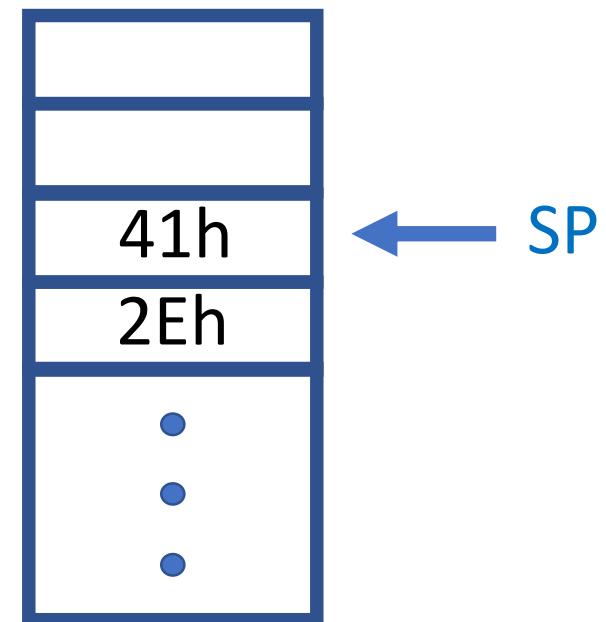
```
mov ax, 0B800h  
mov es, ax
```

```
mov ax, 2E41h ;first letter ('A')  
push ax
```

```
mov ax, 4C00h  
int 21h
```

- נדרש לכתוב תוכנית שמדפסה שני תווים על המסך.
- נעשה זאת ע"י כתיבת פונקציה שמקבלת כפרמטר את התווים להדפסה - דרך המחסנית.
- נתחיל מהקוד של התוכנית:

מצב המחסנית לאחר
דחיפת האות הראשונה



העברת נתונים דרך המחסנית

```
.model small  
.stack 100h  
.code  
HERE:
```

```
mov ax, 0B800h  
mov es, ax
```

```
mov ax, 2E41h ;first letter ('A')  
push ax
```

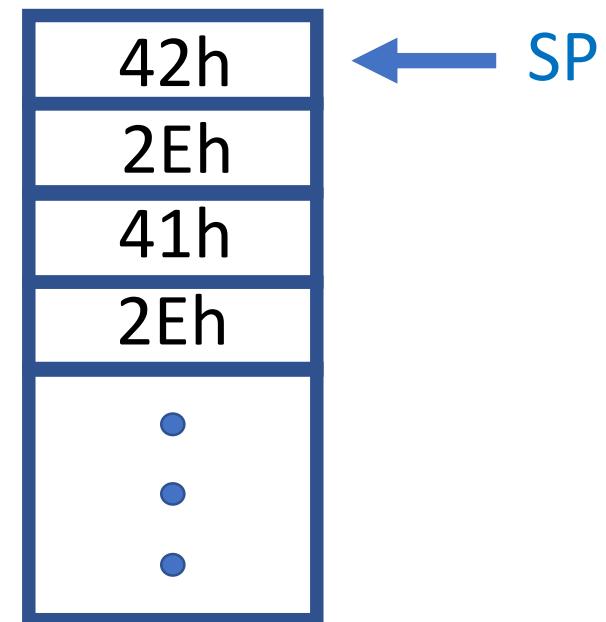
```
mov ax, 2E42h ;second letter ('B')  
push ax
```

```
call print_2_Letters
```

```
mov ax, 4C00h  
int 21h
```

- נדרש לכתוב תוכנית שמדפסה שני תוים על המסך.
- נעשה זאת ע"י כתיבת פונקציה שמקבלת כפרמטר את התווים להדפסה - דרך המחסנית.
- **נתחיל מהקוד של התוכנית:**

מצב המחסנית לאחר
דחיפת האות השנייה

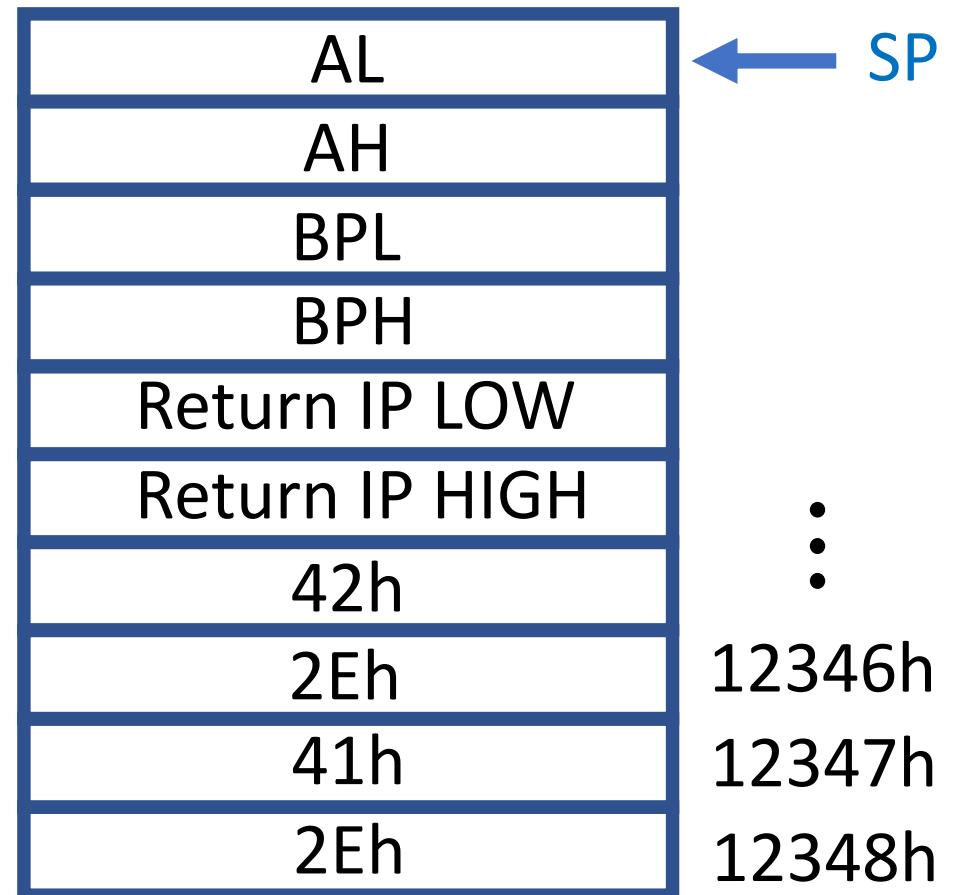


העברת נתונים דרך המחסנית

```
; Inputs:  
; Two top stack elements  
print_2_Letters proc  
    push bp  
    push ax  
  
    ; Printing Code ←  
  
    pop ax  
    pop bp  
    ret 4  
  
print_2_Letters endp
```

- נתחל את כתיבת הפונקציה מדחיפת האוגרים שנשתמש בהם בתוך הפונקציה

מצב המחסנית
בנק' זא



end HERE

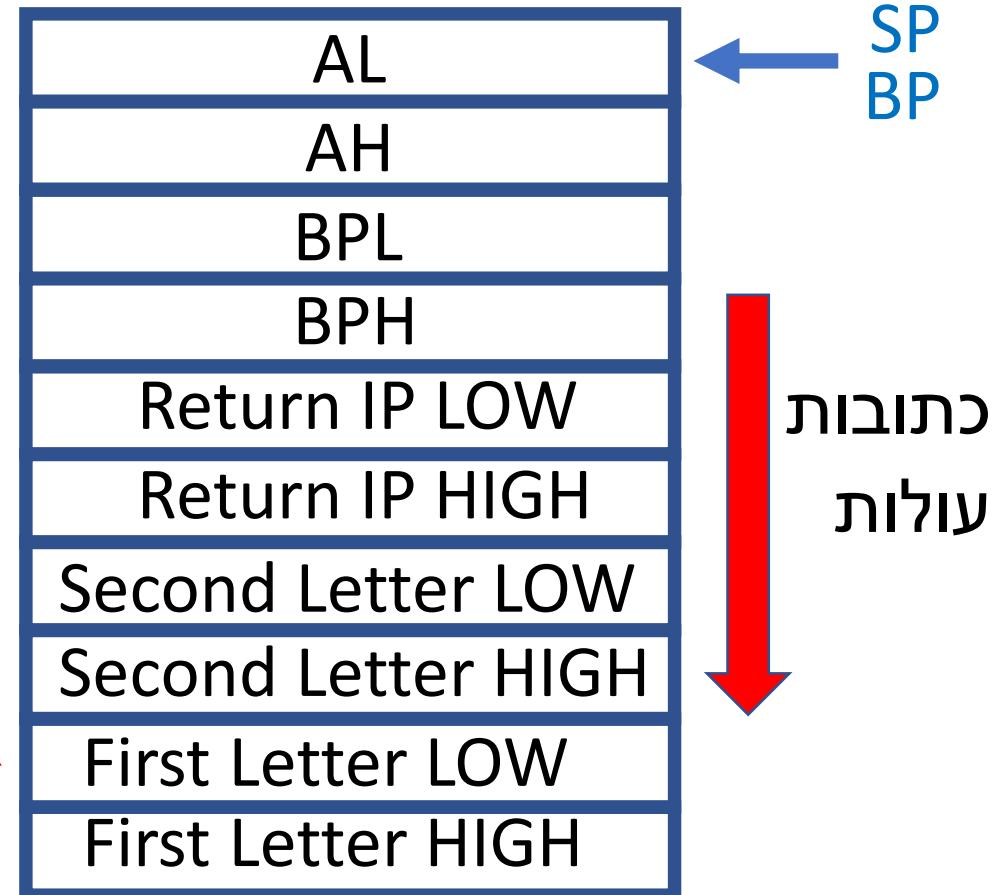
העברה נטוונ דרכ המחסנית

```
; Inputs:  
; Two top stack elements  
print_2_Letters proc  
    push bp  
    push ax  
  
    mov bp, sp  
  
    mov ax, [bp+8d]  
    mov es:[340h], ax  
  
    pop ax  
    pop bp  
    ret 4  
  
print_2_Letters endp  
  
end HERE
```

לא ניתן להשתמש ב SP
ישירות למעון לזכרון

האות הראשונה
נמצאת בכתב
SP + 8

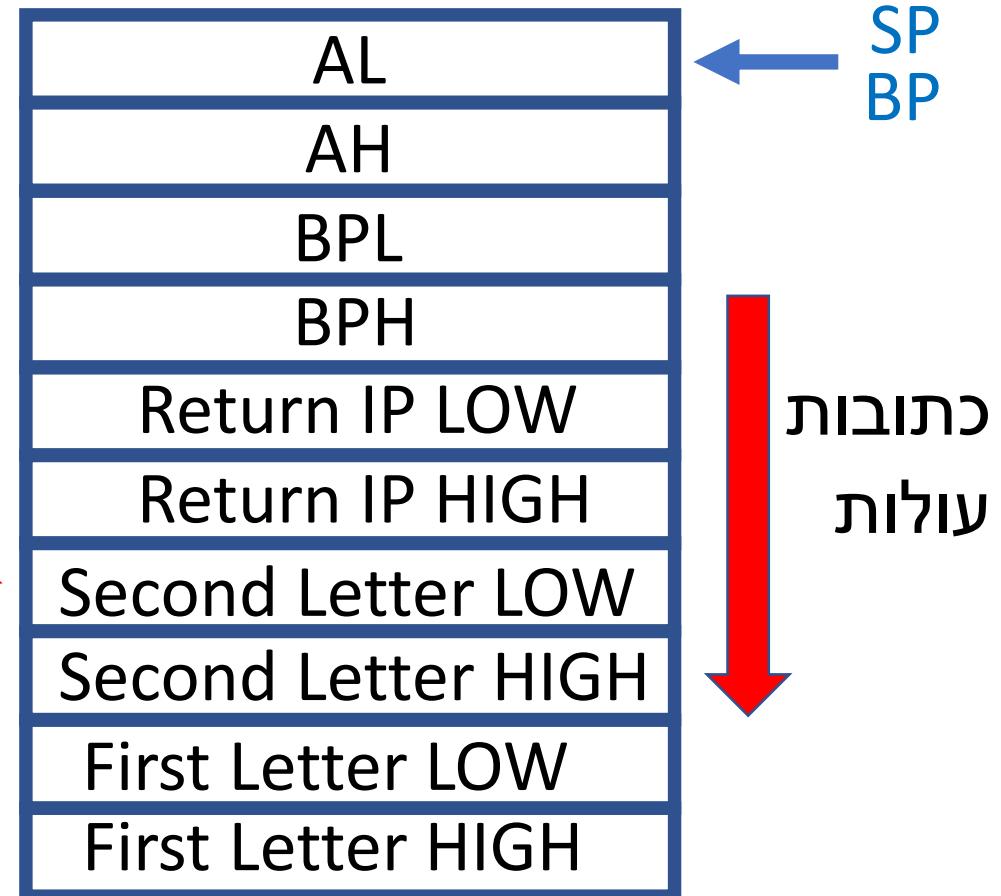
- הדפסת האות הראשונה:



העברת נתונים דרך המחסנית

```
; Inputs:  
; Two top stack elements  
print_2_Letters proc  
    push bp  
    push ax  
  
    mov bp, sp  
  
    mov ax, [bp+8d] ; first letter location  
    mov es:[340h], ax  
  
    mov ax, [bp+6d] ; second letter location  
    mov es:[342h], ax  
  
    pop ax  
    pop bp  
    ret 4  
print_2_Letters endp  
end HERE
```

- כתיבת הפונקציה:



העברה נתוֹן לְרַד הַמִּחְסָבִית

```
; Inputs:  
; Two top stack elements  
print_2_Letters proc
```

push bp
push ax

; Printing Code

pop ax
pop bp
ret 4

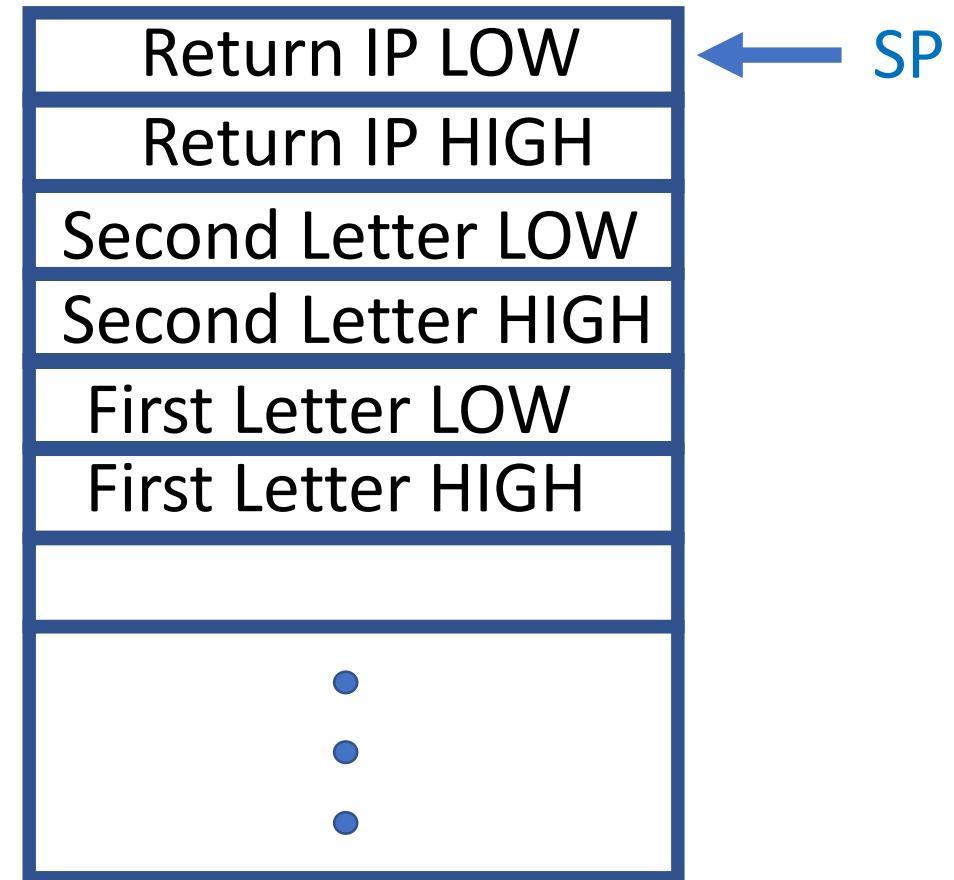
print 2 Letters endpoint

end HERE

פקודת RET 4 תקדם את SP ב 4 מקומות
קדימה (לאחר הוצאת ה PO), ותגרום
ל"העלמת" הנתונים שהעבכנו לפונקציה

מצב המחסנית אחרי החזרת האוגרים לערם התחלתי

new SP after
ret 4



פקודת ret

- האם אנחנו חייבים את פקודת ret בסיום הפונקציה?

תשובה: לא

- פקודת ret אחראית לכתובת הקוראת בסיום השגורה. אנו יכולים לkapoz חזרה גם ע"י פקודות אחרות.

שליפת כתובת החזרה
מהמחסנית וקפיצה אליה

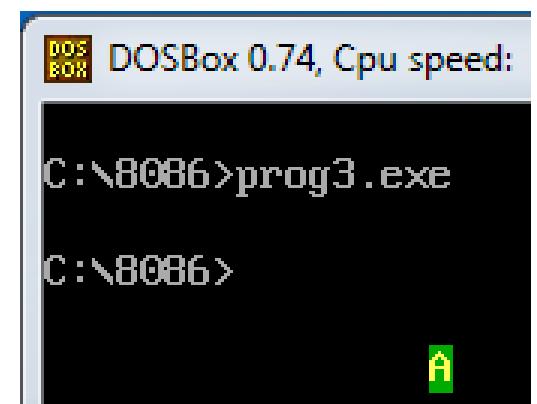
```
.model small
.stack 100h
.code
HERE:
    mov ax, 0B800h
    mov es, ax

    mov ax, 2E41h ; 'A'
    call print_Letter

    mov ax, 4C00h
    int 21h
```

```
; Inputs:
; AX - letter to print
print_Letter proc
    mov es:[340h],ax
    pop ax
    jmp ax
print_Letter endp
```

end HERE



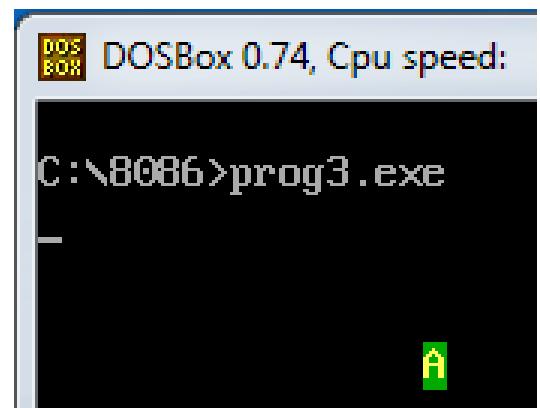
פקודת ret

```
.model small  
.stack 100h  
.code  
  
print_Letter proc  
    mov es:[340h],ax  
print_Letter endp  
  
HERE:  
    mov ax, 0B800h  
    mov es, ax  
  
    mov ax, 2E41h ; 'A'  
    call print_Letter  
  
    mov ax, 4C00h  
    int 21h  
  
end HERE
```

- מה יקרה בתכנית הבאה:

- התכנית תתקע בלולאה אין סופית:
- פקודת call תגרום לקריצה לקוד הפונקציה.
- כיוון שאין קריצה חוזרת בסיום הפונקציה, התכנית תמשיך לשורה הבאה שהיא תחילת התכנית .

ההדפסה למסך תקרלה, אך DOSBOX "יתקע" כיוון שהתכנית נמצאת בלולאה אין סופית ולא מגיעה לפקודות הסיום תכנית



החזרת ערך מן הפקציה

- נכתב תוכנית שימושת בפקציה שמחשבת את כתובת היסט המסר.
- הפקציה מקבלת את היסט הקודם דרך המחסנית, ומחזירה את היסט החדש בראש המחסנית.

```
.model small  
.stack 100h  
.code  
HERE:  
    mov ax, 0B800h  
    mov es, ax  
    mov di, 338h ;initial offset  
  
    push di ←  
    call get_offset  
    pop di ←  
  
    mov es:[di], 2E41h ; print 'A'  
  
    mov ax, 4C00h  
    int 21h
```

דחיפת הכתובת הישנה למחסנית
שליפת הכתובת החדשה מהמחסנית

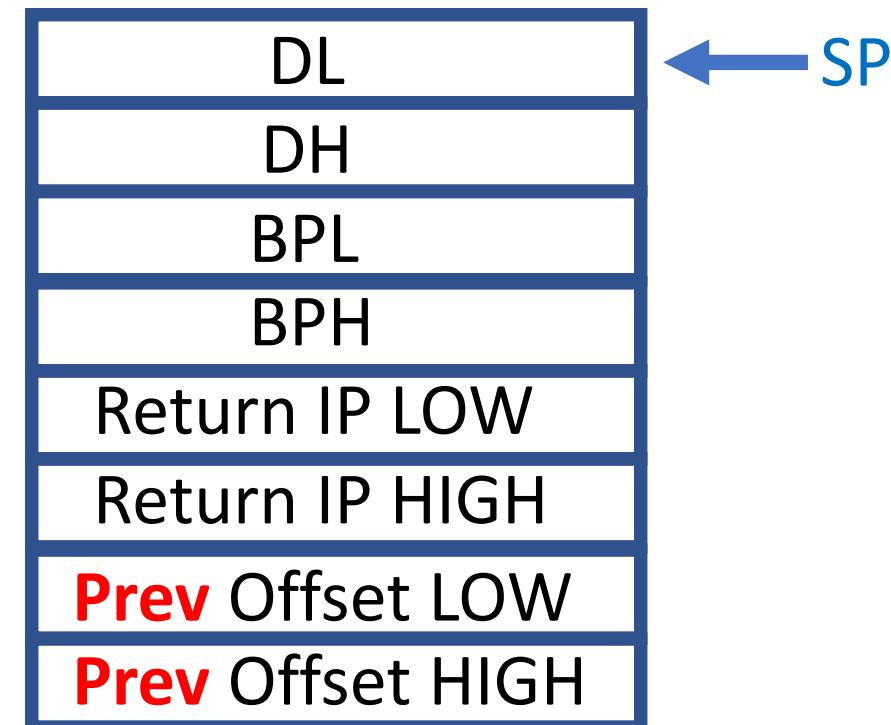
החזרת ערך מן הפונקציה

- הפונקציה לוקחת את הערך הישן מהמחסנית וכותבת את החדש במקום

```
; Input: Top stack element  
; Output: Top stack element  
get_offset proc  
    push dx  
    push bp  
  
    mov bp, sp  
    mov dx, [bp+6d]  
    add dx, 250h  
    mov [bp+6d], dx  
  
    pop bp  
    pop dx  
    ret  
get_offset endp  
end HERE
```

←
מצב המחסנית
בנק' זו

; get previous offset
; update offset
; save new offset

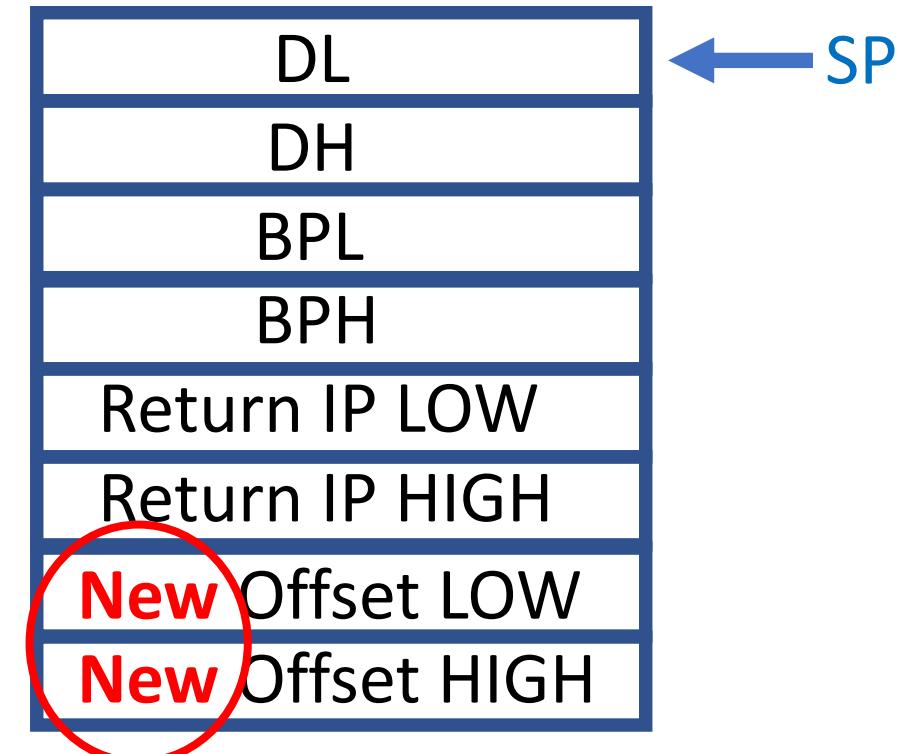


החזרת ערך מן הפונקציה

- הפונקציה לוקחת את הערך הישן מהמחסנית וכתבת את החדש במקום

```
; Input: Top stack element  
; Output: Top stack element  
get_offset proc  
    push dx  
    push bp  
  
    mov bp, sp  
    mov dx, [bp+6d]      ; get previous offset  
    add dx, 250h          ; update offset  
    mov [bp+6d], dx       ; save new offset  
  
    pop bp  
    pop dx  
    ret  
get_offset endp  
end HERE
```

מצב המחסנית
בנק' זו



מיקרו-מעבדים ושפת אסמלר

תרגול מס' 10 – התקני קלט/פלט, פסיקות תוכנה

מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן

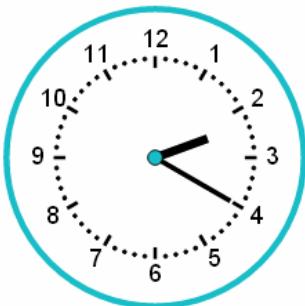


גישה להתקן חיצוני

`mov ax, 0B800h`
`mov es, ax`

- ע"י אזור בזיכרון
- למשל גישה למסר עי פניה לזכרון

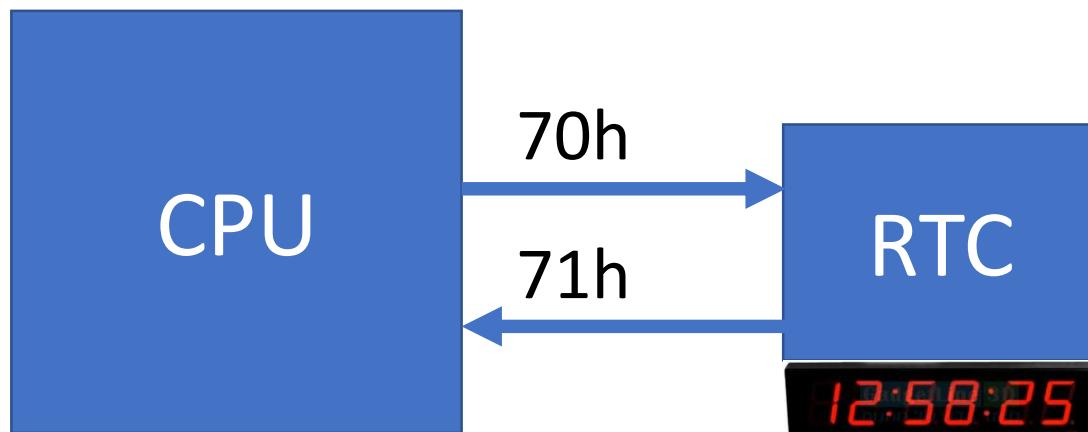
- פורט קלט/פלט ייעודי, הפורט מחובר להתקן חיצוני, למשל שעון או מקלדת
- כאשר נרצה לשאול את השעון מה השעה, נכתב לפורט
- כאשר נרצה לקרוא את התשובה, נקרא מהפורט



- גישה לפורט:
 - קוראים מפורט ע"י פקודת **IN**
 - כתבים לפורט ע"י פקודת **OUT**

Real-Time Clock (RTC)

- אחד הנקודות החיצונית שהמעבד יכול לתקשר איתם הוא שעון זמן אמת (RTC)
- השעון מקבל "בקשות מידע" דרך פורט שמספרו $70h$



- השעון מחזיר תשובה בפורט $71h$

`out 70h, al`

`in al, 71h`

איך נדע לעבד עם רכיב אחר?

- השעון הוא רכיב חיצוני, וכך למד איר מתקשרים אותו צרי לגשת למפרט שפורסם ע"י היצרן
- ניתן גם למצוא את תיאור הרכיב באינטרנט.
- למשל באתר: http://stanislavs.org/helppc/cmos_ram.html



CMOS RTC - Real Time Clock and Memory (ports 70h & 71h)

Programming Considerations:

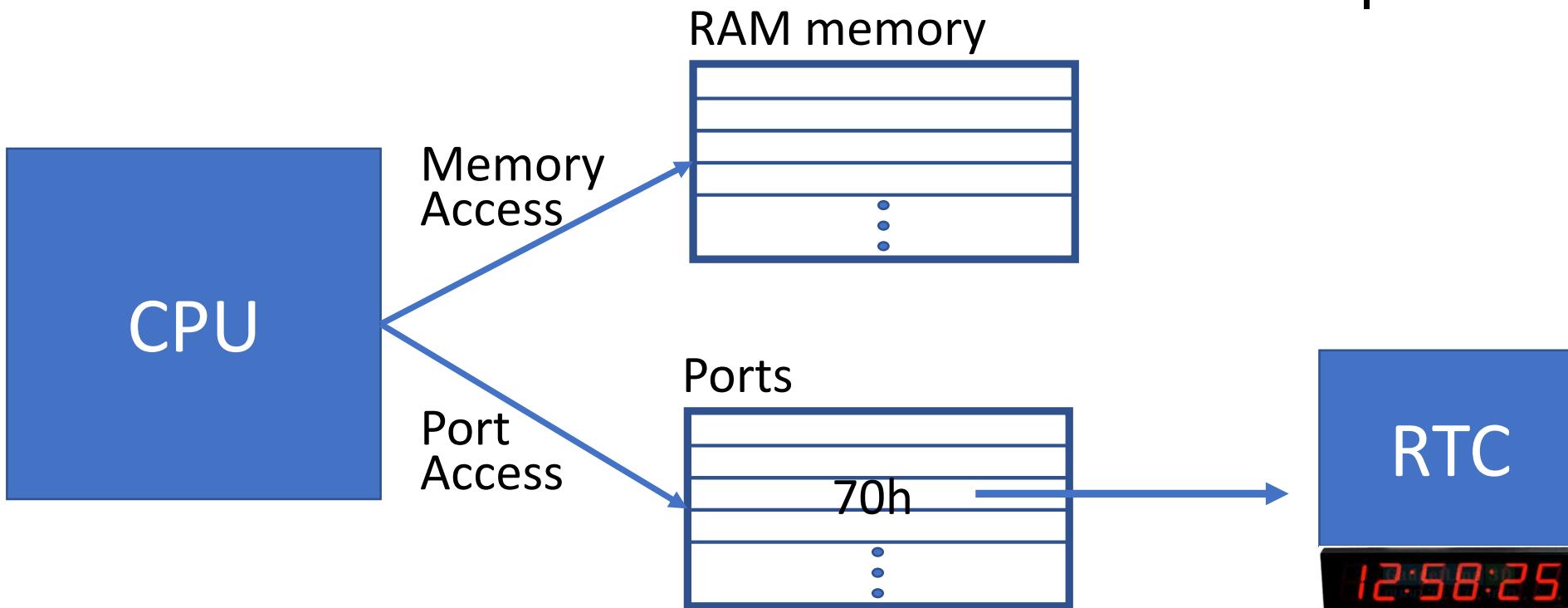
Reg#	Description
00	RTC seconds
01	RTC seconds alarm
02	RTC minutes
03	RTC minutes alarm
04	RTC hours
05	RTC hours alarm
06	RTC day of week
07	RTC day of month
08	RTC month
09	RTC year
0A	RTC Status register A:

Write CMOS address to read or write to port 70h
Read/write port 71h to get/set data

האתר נותן למד שהגישה לרכיב מתבצעת ע"י קריית תא זיכרון ברכיב ש מכיל את המידע הרצוי, בפרט המידע אודות השעה נמצאת בתא 04 של הרכיב

איך נדע לעבד עם רכיב אחר?

- למה דוחק אפורט $h70$?
- תשובה:
 - כאשר מעבד פונה לפורט מסוים, הוא מבצע כתיבה או קרייה **מכתובות הרכיב**.
 - תהליך הכתיבה לכתובות במרחב כתובות הרכיבים דומה לכתיבה לתא זיכרון למרחב כתובות הזיכרון



Real-Time Clock (RTC)

```
.model small  
.stack 100h  
.code  
HERE:
```

```
    mov al, 04h  
    out 70h, al
```

```
    in al, 71h  
    mov bl, al
```

```
    add bl, 30h
```

```
    mov ax, 0B800h  
    mov es, ax  
    mov es:[340h], bl
```

```
    mov ah, 4ch  
    int 21h
```

```
end HERE
```

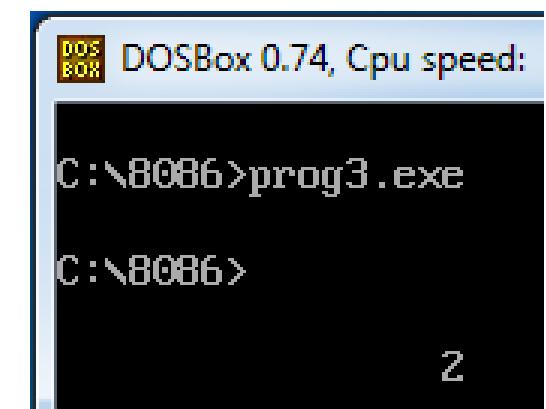
- נכתוב תוכנית שمدפסה את השעה ע"י תקשורת עם RTC דרך הפורטים המתחברים אליו

שליחת קוד בקשה עבור שעה דרך פорт 70h

קריאה שעה דרך פорт 71h

העברה לASCII

הדפסת השעה שהתקבלה



Real-Time Clock (RTC)

```
.model small
.stack 100h
.code
HERE:
    mov al, 04h
    out 70h, al

    in al, 71h
    mov bl, al

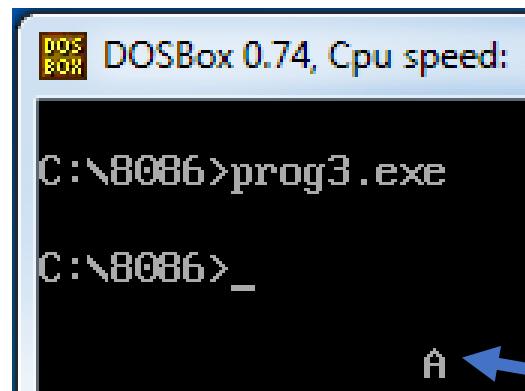
    add bl, 30h

    mov ax, 0B800h
    mov es, ax
    mov es:[340h],bl

    mov ah, 4ch
    int 21h

end HERE
```

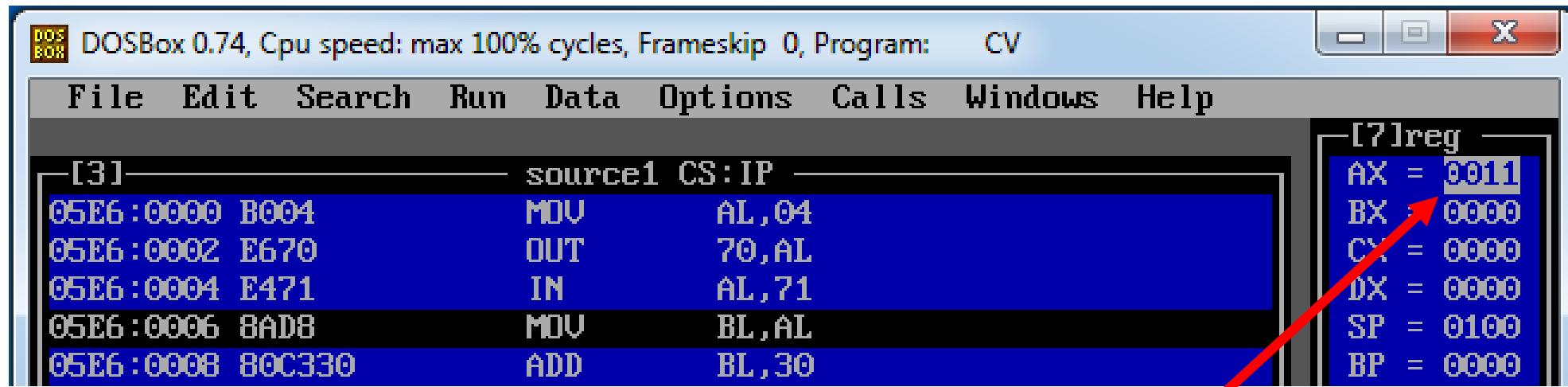
- תרגיל:
- מה יציג עבור השעה 11 אם משתמש בקוד זה?



קיבלו מספר הקוסה
כלשהו, שלא מייצג נכון
את השעה (0Ah=10d)

Real-Time Clock (RTC)

- כדי להבין את התוצאה שקיבלנו, נכנס לדיבאגר על מנת לבדוק את תוכן האוגר AL בזמן הדפסה למסך



The screenshot shows the DOSBox interface with assembly code and register values.

Assembly code window:

source1 CS:IP
05E6:0000 B004 MOV AL,04
05E6:0002 E670 OUT 70,AL
05E6:0004 E471 IN AL,71
05E6:0006 8AD8 MOV BL,AL
05E6:0008 80C330 ADD BL,30

Registers window:

[7]reg
AX = 0011
BX = 0000
CX = 0000
DX = 0000
SP = 0100
BP = 0000

A red arrow points from the CX = 0000 entry in the registers window to the IN AL,71 instruction in the assembly code window.

- הערך שMOVED מהשעון בפורט 71 הוא שני מספרים דצימליים.
- נדפיאו אותם ספרה-ספרה (כפי שעשינו בת'ב 2+3)

פסיקות תוכנה

- פסיקת תוכנה היא דרך מובנית לבצע קריאה לשירות/פונקציה תוך שמירת מצב המערכת
- הפעלת פסיקת תוכנה:
 - מבקשים מהמעבד להפעיל את הפסיקה ע"י הפקודה `N int`
 - לעיתים פסיקה אחת מאגדת מספר שירותים שונים – לפני הפעלת הפסיקה נשים בAH את מספר השירות המבוקש

`mov ah, 01h`

`int 21h`

פסיקות תכנה – הדפסת تو

```
mov ah, 01h  
int 21h  
; AL= 'KEY' ASCII code
```

- פסיקה לקבלת מקש מנקלדת:
- בפסיקת תכנה ah=01h קיימת שירותה int 21h לקריאת מקש מנקלדת
- בהפעלת השירות, הפסיקה שומרת את התו שהתקבל באוגר AL

```
mov dl, '*' ; ASCII to print  
mov ah, 02h  
int 21h
```

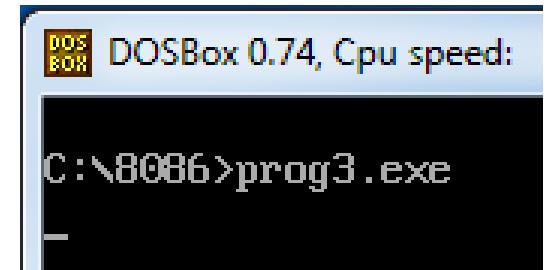
- פסיקה להדפסת תו למסך:
- שירות נוסף של הפסיקה ah=02h
- בהפעלת השירות הפסיקה מדפיסה למסך את התו שנמצא באוגר DL

תכנית שמקבלת ומדפיסה תוו ע"י שימוש בפסקיקות

```
.model small  
.stack 100h  
.code  
HERE:  
  
    mov ah, 01h  
    int 21h  
  
    mov dl, al  
  
    mov ah, 2h  
    int 21h  
  
    mov ah, 4ch  
    int 21h  
  
end HERE
```

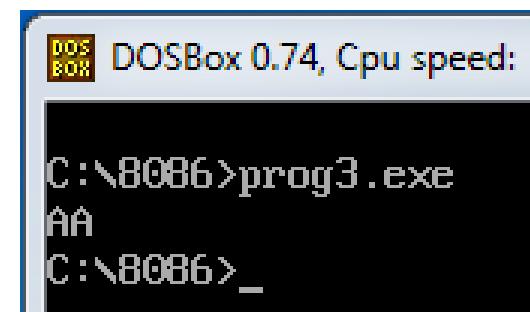
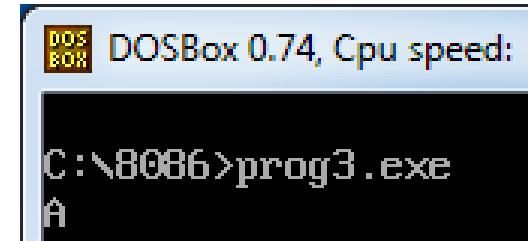
- נבנה תוכנית שמשתמשת בפסקיקות h 01h או h 02h על מנת לקלוט תוו, ולהציג אותו שוב למסך

התכנית מושהה, והפסקיקה ממחכה
ללחיצה על המקלדת



בסיום הפסקיקה הראשונה נעביר את
התו שהתקבל ביעד הפסקיקה (AL) לאוגר
המשמש כמקור לפסקיקה השנייה (DL)

הפסקיקה מדפיסה את הערך השמור
בDL



פסקה להדפסת מחרוזת למסך:

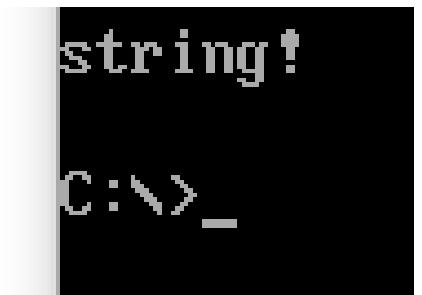
```
.model small  
.stack 100h
```

```
.data  
out_msg db 'String!', 0Ah, 0Dh, '$'  
.code  
mov ax, @data ;initializing  
mov ds, ax ;data segment  
  
mov dx, offset out_msg  
mov ah, 9h  
int 21h ;print request  
  
.exit  
end
```

- שירות נוסף של הפקה `int 21h` הוא `09h`

Diagram illustrating the string representation:

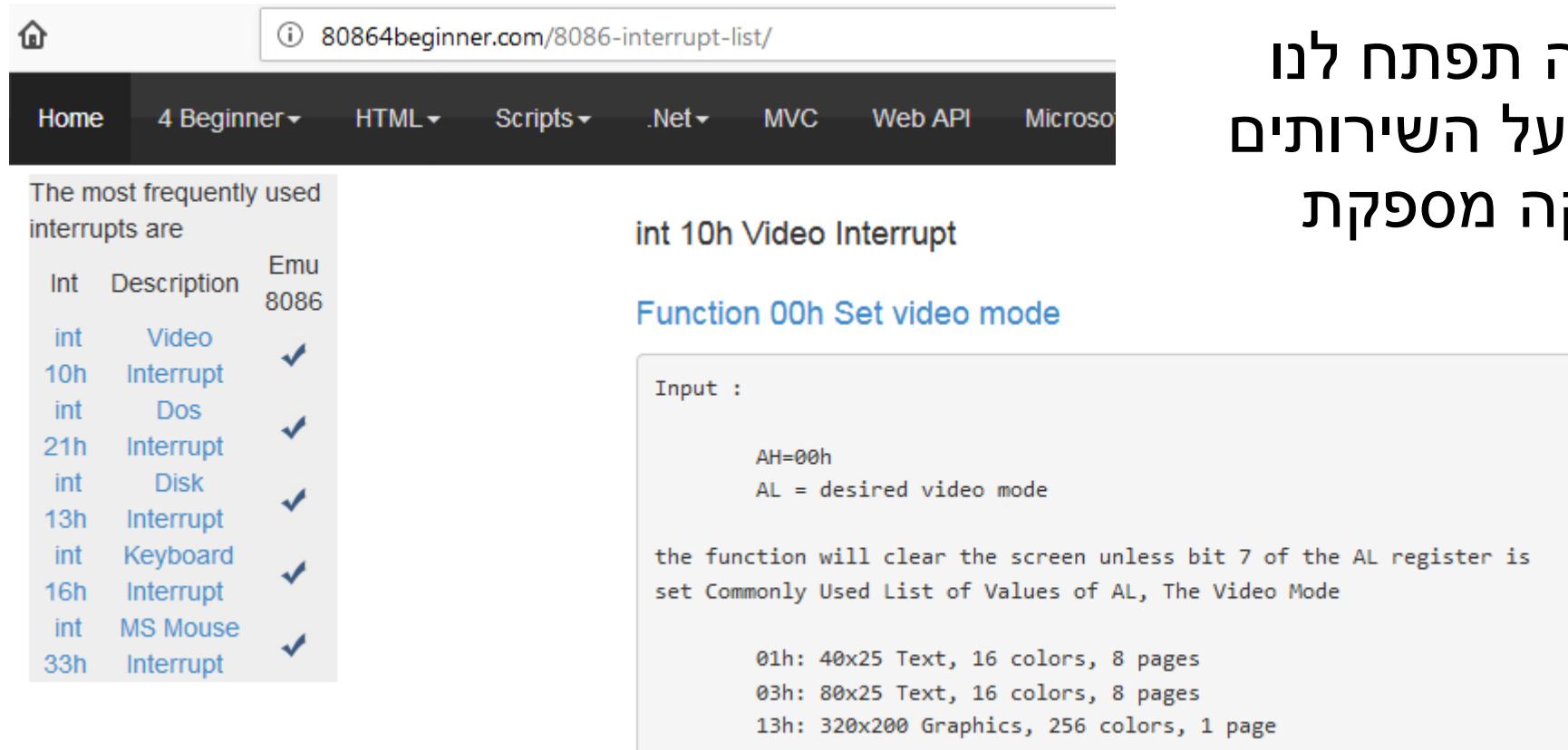
- '\$' (End of String)
- 0Dh (Carriage Return)
- 0Ah (New Line)



- בהפעלת השירות הפקה מדפסה למסך את המחרוזת שנמצאת בהיסט DX (ביחוס לสมגנט הנתוניים).

פסיקות נוספות

- מגוון הפסיקות הקיימות במעבד מופיעות בקישור:
<http://8086beginner.com/8086-interrupt-list/>



The screenshot shows a web browser window with the URL <http://8086beginner.com/8086-interrupt-list/> in the address bar. The page content is as follows:

The most frequently used interrupts are

Int	Description	Emu
8086		
int 10h	Video Interrupt	✓
int 21h	Dos Interrupt	✓
int 21h	Interrupt	✓
int 13h	Disk Interrupt	✓
int 16h	Keyboard Interrupt	✓
int 16h	Interrupt	✓
int 33h	MS Mouse Interrupt	✓

int 10h Video Interrupt

Function 00h Set video mode

Input :

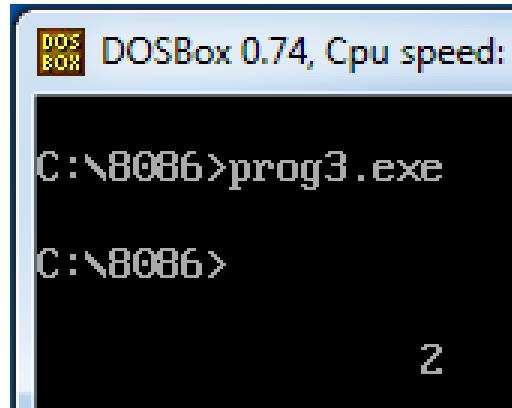
```
AH=00h
AL = desired video mode
```

the function will clear the screen unless bit 7 of the AL register is set
Commonly Used List of Values of AL, The Video Mode

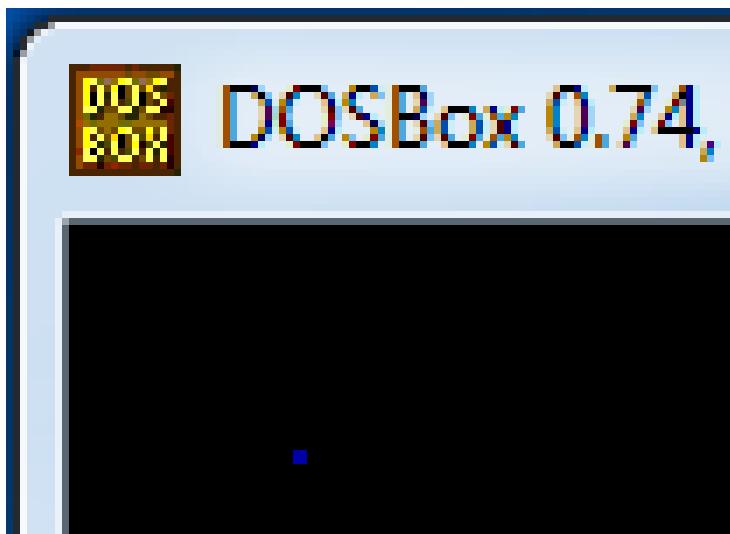
```
01h: 40x25 Text, 16 colors, 8 pages
03h: 80x25 Text, 16 colors, 8 pages
13h: 320x200 Graphics, 256 colors, 1 page
```

- לחייב על פסיקה תפתח לנו עמוד עם הסבר על השירותים השונים שהפסיקה מספקת

מעקב אחרי עכבר ע"י שימוש בפסיקות



- כאשר הדפנו למסר דרך זיכרון המסר, בחרנו איזהתו להדפיס ובאיזה מקום.
- ניתן להדפיס למסר גם ברמת הפיקסל הבודד.



מעקב אחרי עכבר ע"י שימוש בפסיקות

- בשביל להדפס בرمת הפיקסל הבודד תחילת צירר להעביר את התצוגה
למצב "Graphics" כפי שמוסבר בעמוד הפסיקה int 10h int 10h Video Interrupt

Function 00h Set video mode

Input :

AH=00h

AL = desired video mode

the function will clear the screen unless bit 7 of the AL register is set
Commonly Used List of Values of AL, The Video Mode

01h: 40x25 Text, 16 colors, 8 pages

03h: 80x25 Text, 16 colors, 8 pages

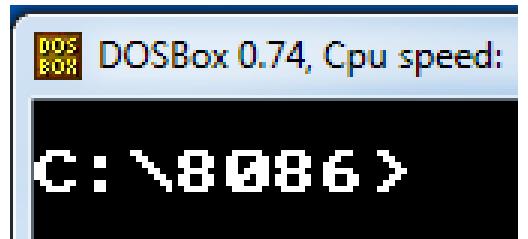
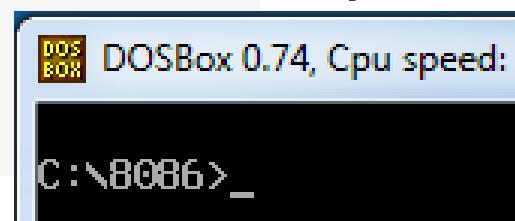
13h: 320x200 Graphics, 256 colors, 1 page

mov ah, 00h

mov al, 13h

int 10h

- הבדל בין תצוגה במסך גרפי למסך המצביע טקסט:



מעקב אחרי עכבר ע"י שימוש בפסקאות

- שירות 3h של הפסקה h 33h מחזיר את המיקום הנוכחי של העכבר

Int 33h MS Mouse Interrupt

Function 3 Get Mouse Position & Button Status

```
Input  
AX = 3  
Output  
BX = Button Status  
xxxx xxxx xxxx xMRL  
M=middle (if present) R=right L=left  
0= not pressed 1 = pressed  
CX = Horizontal Mouse Cursor Position  
DX = Vertical Mouse Cursor Position  
(div positions by 2 for med res  
graphics; div by 8 for text mode)
```

```
mov ax, 03h  
int 33h  
shr cx, 1
```

כדי לקבל את המיקום
הנכון במצב גרפי, צריך
לחلك ב 2

מעקב אחרי עכבר ע"י שימוש בפסיקות

- שירות 0Ch של הפסיקה int 10h מופיע פיקסל בודד במקום המבוקש

int 10h Video Interrupt

Write graphics pixel

Sets the color number of the specified pixel in graphics mode. Valid in all graphics modes.

Inputs

AH = 0Ch

BH = Display page.

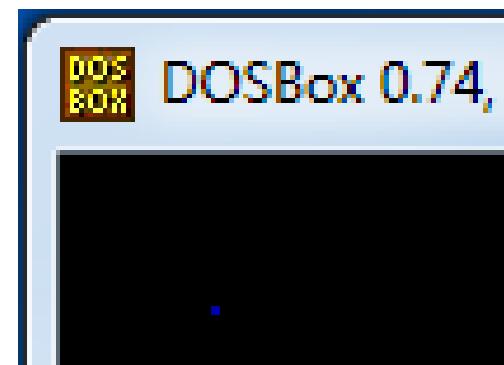
DX = Screen line (0 is top).

CX = Screen column (0 is leftmost).

AL = Color number.

mov cx, 10h
mov dx, 10h
mov al, 01h ;blue

mov ah, 0ch
int 10h



מעקב אחרי עכבר ע"י שימוש בפסיקות

- נשלב את הפסיקות לתכנית, שעוקבת אחרי העכבר בלבד אינסופית ומדפיסה נקודה כחולה במקום שבו הוא נמצא.

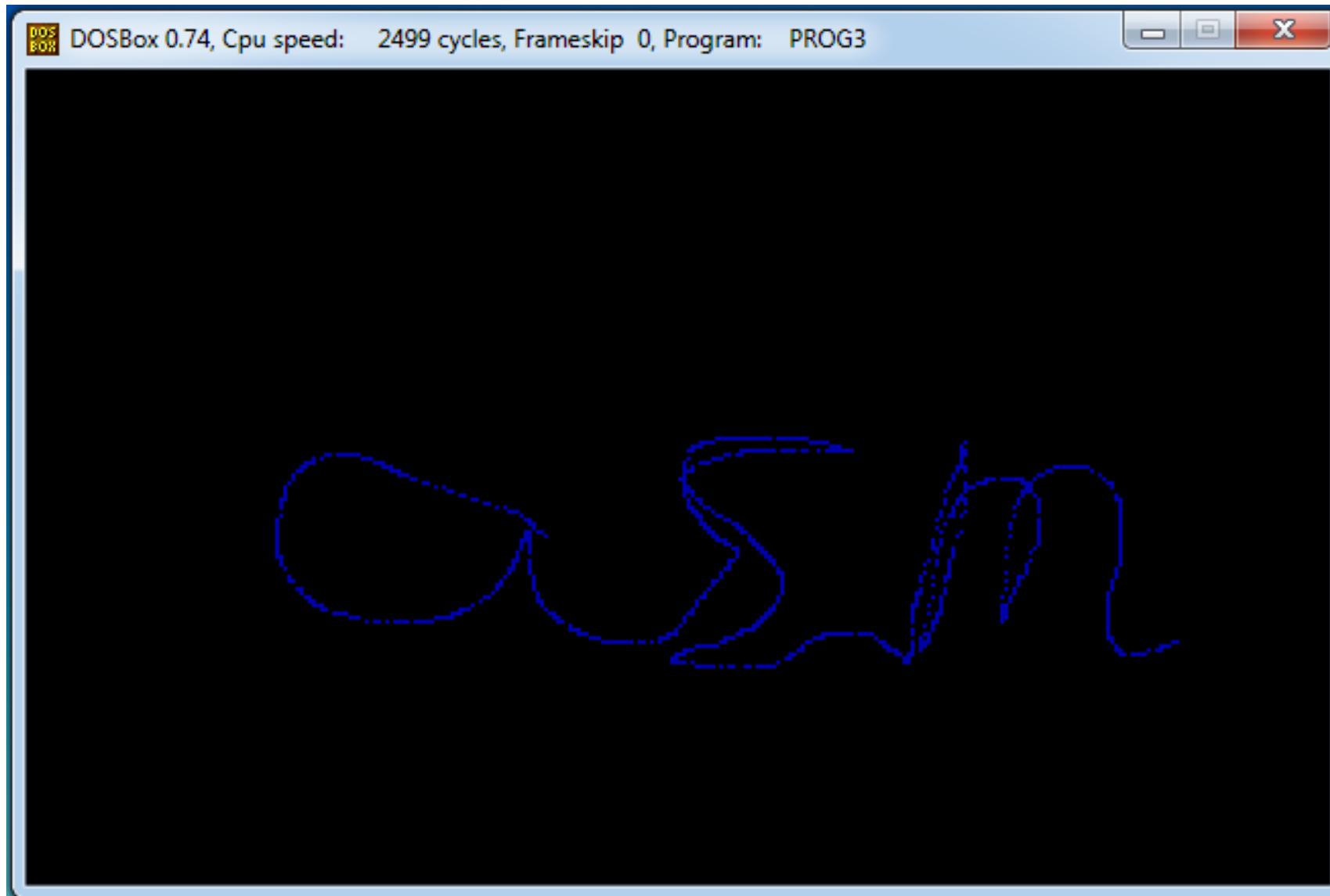
```
.model small  
.stack 100h  
.code  
HERE:  
    mov ah,00h  
    mov al,13h  
    int 10h
```

```
L1:  
    mov ax, 03h  
    int 33h  
    shr cx,1  
  
    mov al,01h ;blue  
    mov ah,0ch  
    int 10h  
  
    jmp L1  
  
    mov ah,4ch  
    int 21h  
end HERE
```

הפסיקה מחשירה את העמודה והשורה של מיקום העכבר באוגרים CX,DX

הפסיקה תדפיס את הפיקול במקומות שמוגדרים ע"י CX,DX שזה בדיקת האוגרים שליהם הפסיקה הקודמת החזירה את מיקום העכבר

מעקב אחרי עכבר ע"י שימוש בפסיקות



- במידה והעכבר לא מגיב, לחצו **CTRL+F10**

מיקרו-מעבדים ושפת אסמלר

תרגול מס' 11 – פסיקות חומרה

מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן



תזכורת - פסיקות תוכנה

- פסיקת תוכנה היא דרך מובנית לבצע קריאה לשירות/פונקציה תוך שבירת מצב המערכת
- הפעלת פסיקת תוכנה:
 - מבקשים מהמעבד להפעיל את הפסיקה ע"י הפקודה `N int`
 - לעיתים פסיקה אחת מאגדת מספר שירותים שונים – לפני הפעלת הפסיקה נשים בAH את מספר השירות המבוקש

`mov ah, 01h`

`int 21h`

פסקת תוכנה להדפסת מהרזה למסך

- נקבע את מיקום הסמן (המקום שבו מדפיסים) ע"י שירות **02h** של פסיקה **int 10h**

int 10h Video Interrupt

Function 02h Set cursor position

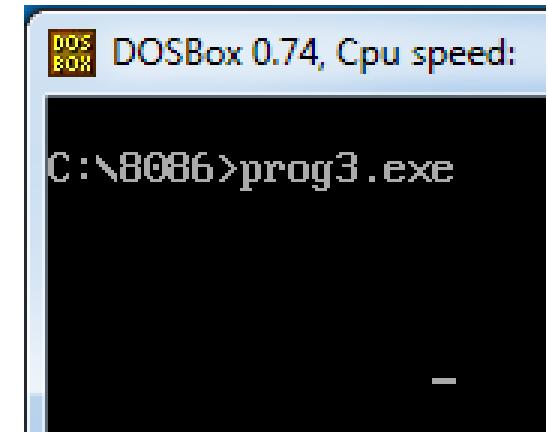
Input:
AH = 02h

BH = page number
0-3 in modes 2&3
0-7 in modes 0&1
0 in graphics modes

DH = row (00h is top)
DL = column (00h is left)

Return:
None

mov dh, 5h
mov dl, 10h
mov ah, 02h
int 10h



פסקת תוכנה להדפסת מחרוזת למסך

- נדפסו את המחרוזת במיקום הסמן ע"י שירות 09h של פסיקה 21h

int 21h Dos Interrupt

Function 09h- Write string to STDOUT

Sends the characters in the string to the standard output device.

Input: AH=09H

output: none

DS:DX = pointer to the character string ending with '\$'

המחרוזת צריכה להימצא
בסegment DATA ולהסתיים ב\$

```
.model small
.data
    msg db 'My string$'
```

.stack 100h

.code

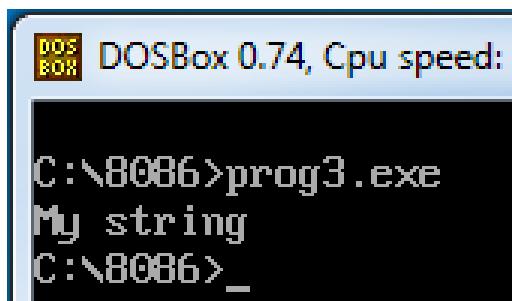
HERE:

```
mov ax, @data
mov ds, ax
```

```
mov dx, offset msg
```

```
mov ah, 09h
int 21h
```

הנחה offset אומרת
לקומפיילר לשמר
בAX את הכתובת
היחסית של המערך בתוך
Data segment

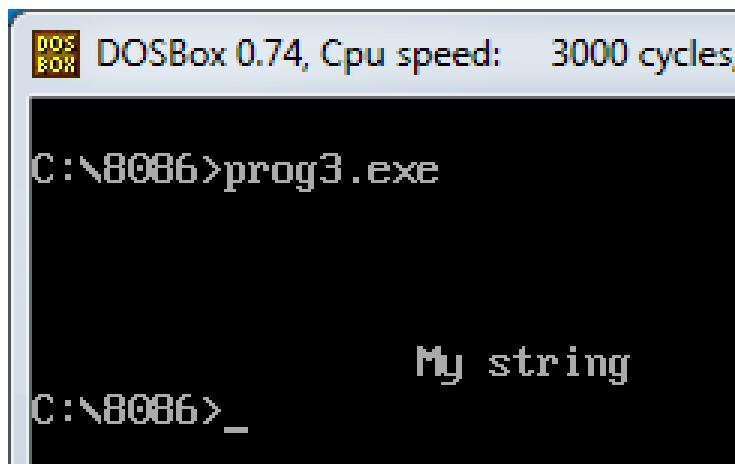


end HERE

פסקית תוכנה להדפסת מחרוזת למסך

- נשלב את הפקיות לתכנית שמדפיסה את המחרוזת "My String" בשורה 10h ובעמודה 15h

```
.model small
.data
    msg db 'My string$'
.stack 100h
.code
HERE:
    mov ax, @data
    mov ds, ax
```



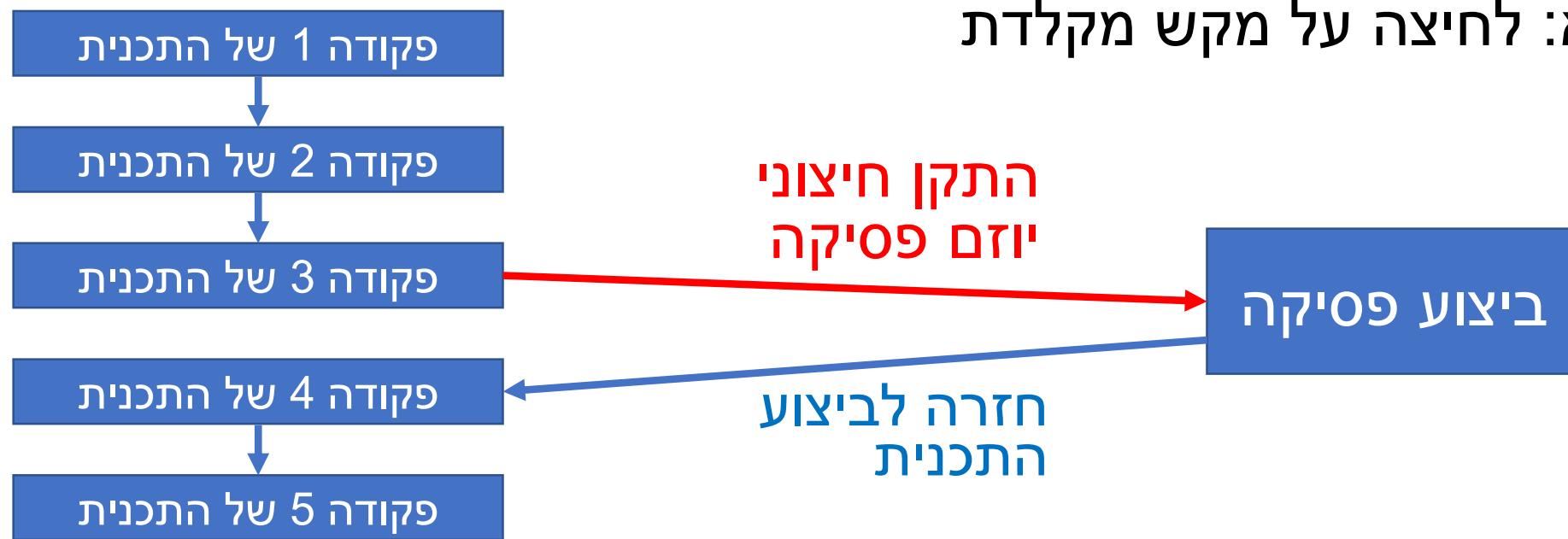
```
        mov dh,5h ;set carriage
        mov dl,10h
        mov ah,02h
        int 10h

        mov dx, offset msg ;print string
        mov ah, 09h
        int 21h

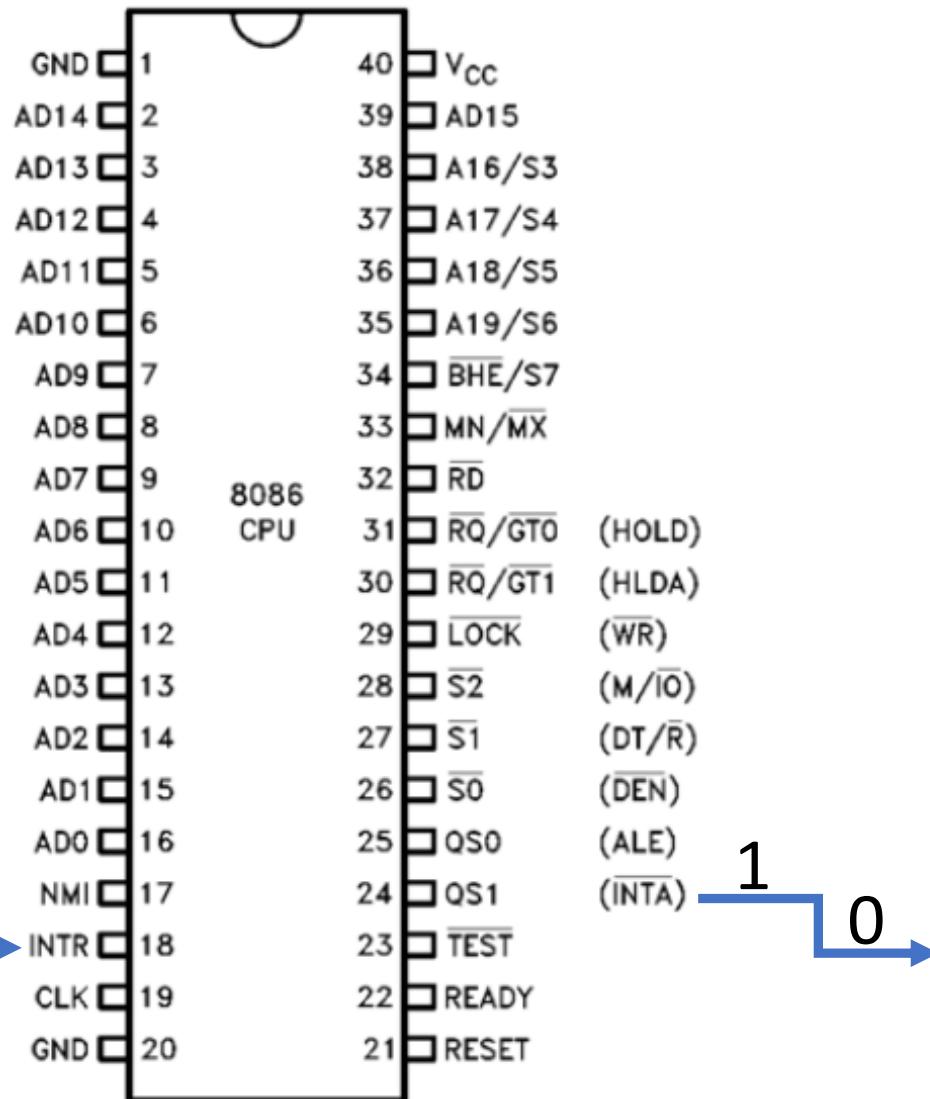
        mov ah,4ch
        int 21h
end HERE
```

פסקות חומרה

- פסקת חומרה היא "שגרה" שמתבצעת אוטומטית כתגובה לאירוע חומרה מסוים.
- בשונה מגישה לפורט קלט/פלט, או פסקת תוכנה, מי שיזם את הפסקה הוא התקן החיצוני.
- דוגמא: לחיצה על מקש מקלדת

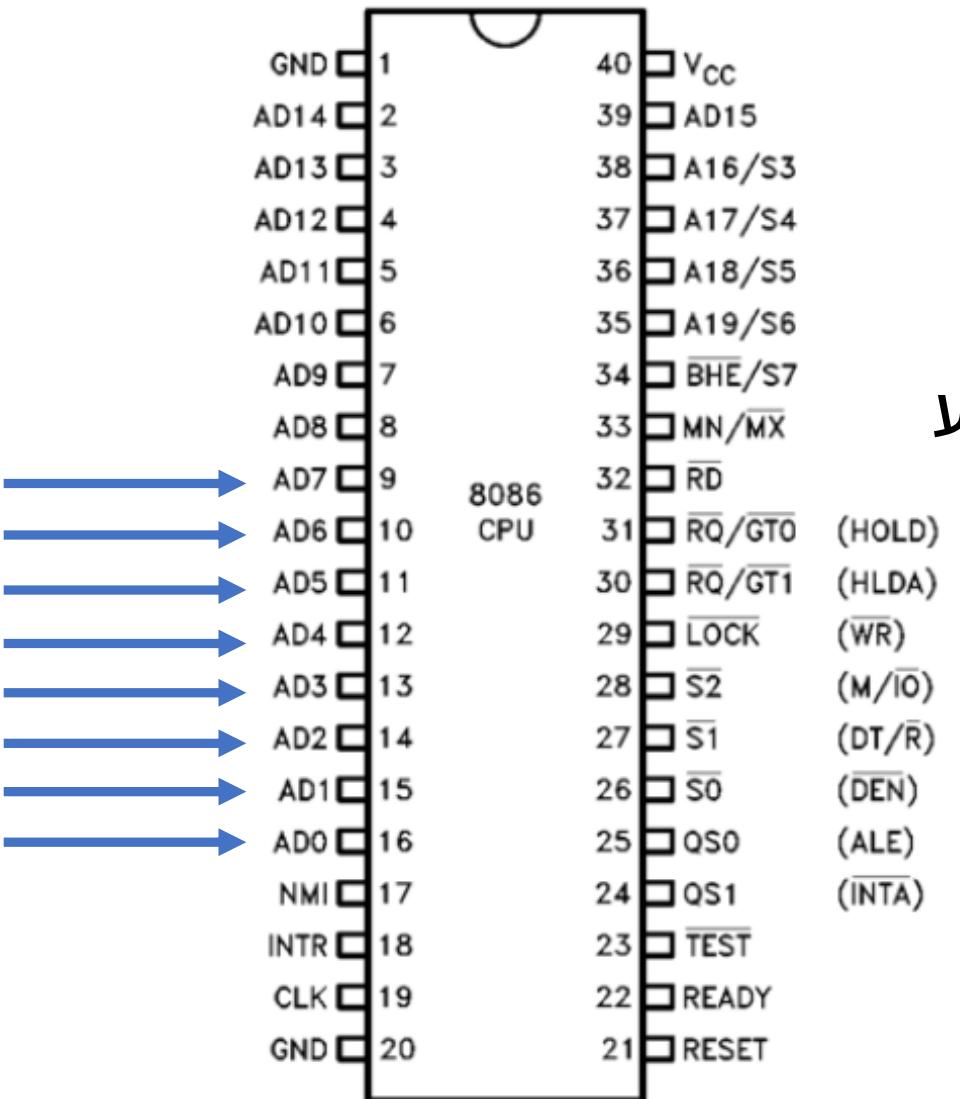


בקשת פסיקת חומרה ע"י התקן חיצוני



- כאשר התקן חיצוני רוצה לבקש פסיקה, הוא מעלה את קו INTR.
- המעבד בודק את ערכי הדגלים TF (Trap Flag) ○ IF (Interrupt Flag) ○
- אם ערך IF=1 (יתקבלו גם פסיקות שנן לא NMI), אז המעבד:
 1. שומר את CS:IP ואת אוגר הדגלים
 2. מאפס את TF ואת IF
 3. מאשר את הפסיקה ע"י הורדת קו INTA

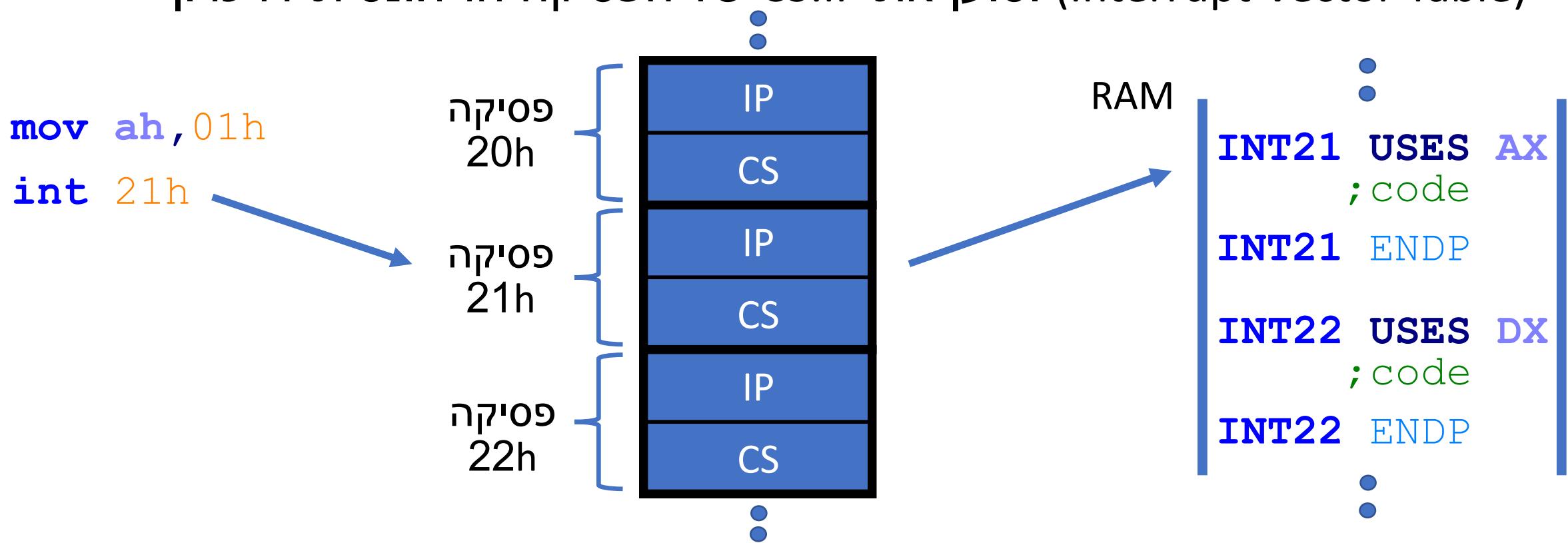
ביצוע הפסיקה ע"י המעבד



- המעבד מקבל את קוד הפסיקה מההתקן דרפסו הנטוניים (הורדת קו INT0 היא סימן עבור ההתקן לשדר את קוד הפסיקה)
- המעבד טוען את וקטור הפסיקה לIP ומבצע את קוד הפסיקה
- המעבד מגיע לפקודת RET ומוסים את הפסיקה ע"י
 1. טעינה חוזרת של אוגר הדגלים
 2. טעינה חוזרת IP:CS של המשך התוכנית

מיקום הפסיקה בזיכרון

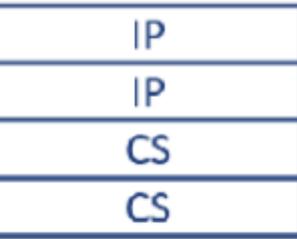
- כארש אנו כותבים `N int`, אנחנו אומרים למעבד איזה פסיקה לבצע.
- בAH קובעים איזה מהשירותים של הפסיקה רוצים להפעיל
- כדי לבצע את הפסיקה, המעבד בודק את כתובת הפסיקה בטבלת IVT
וטוען את IP:CS של הפסיקה הרלוונטית לזיכרון



גישה לטבלת INT דרך CV debugger

כתובות

כתובת	값
0h	INT 0 Divide by 0
3h	INT 1 Single step
4h	INT 2
Bh	INT 3
Ch	Breakpoint
Fh	...
...	...
4xN	INT N
4xN+3	...
3FCh	INT OFFh



- גודל טבלת INT הוא 4 בתים כפול 256. פסיקות אפשריות = $1024d = 400h$ או כלומר כתובות 0000 עד 3FFh.

- אפשר לבדוק את תוכן טבלת INT דרך מפת הזיכרון של הדיבאגר, ע"י זה שנכוון את תצוגת הזיכרון לכתובת 0000:0000

```
=[5]=----- memory1 b 0x0000:0x0000 -----  
0000:0000 8E 03 3F 04 08 00 70 00 1A 03 3F 04 08 .A?♦.p.→?♦.  
0000:000D 00 70 00 8E 03 3F 04 63 03 3F 04 78 03 .p.À?♦c?♦?♦  
0000:001A 3F 04 8E 03 3F 04 A5 FE 00 F0 D4 05 3F ?♦À?♦?♦Ñ.?=?  
0000:0027 04 55 FF 00 F0 60 10 00 F0 60 10 00 F0 ♦U.?=►.?=■.=  
0000:0034 69 10 00 10 80 10 00 F0 60 10 00 F0 F4 ►.?=Q►.?=►.?=
```

IP of int 0 CS of int 0

גישה לטבלה IVT דרך CV debugger

The screenshot shows the DOSBox CV debugger interface. The top window displays assembly code for the IVT table:

```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: CV
File Edit Search Run Data Options Calls Windows Help
[3] source1 CS:IP
043F:10F4 1E PUSH DS
043F:10F5 68A201 PUSH 01A2
043F:10F8 1F POP DS
043F:10F9 0AE4 OR AH,AH
043F:10FB 7410 JZ 110D
043F:10FD 80FC10 CMP AH,10
043F:1100 741C JZ 111E
043F:1102 80FC0B CMP AH,0B
043F:1105 7455 JZ 115C
043F:1107 1F POP DS
043F:1108 2EFF2EF010 JMP DWORD PTR CS:[10F0]
043F:110D C6068D26FF MOV BYTE PTR [268D],FF
```

The bottom window shows memory dump at address 0x0040:

```
[5] memory1 b 0x0000:0x0000
0000:0010 8E 03 3F 04 63 03 3F 04 78 03 3F 04 8E 03 3F 04 A?♦c?♦x?♦A??
0000:0020 A5 FE 00 F0 D4 05 3F 04 55 FF 00 F0 60 10 00 F0 N..?♦U .?♦.?
0000:0030 60 10 00 F0 60 10 00 F0 80 10 00 F0 60 10 00 F0 ???.??.??.??
0000:0040 F4 10 3F 04 00 11 00 F0 20 11 00 F0 40 11 00 F0 ???.??.??.??
0000:0050 A0 11 00 F0 B0 0B 3F 04 63 05 3F 01 20 12 00 F0 á??.??.??.??.?.
```

Annotations with red arrows point from the circled values in the assembly code and memory dump to the corresponding registers in the right-hand register window:

- From the circled `043F:10F4 1E` to the `CS = 043F` register entry.
- From the circled `F4 10 3F 04` to the `IP = 10F4` register entry.
- From the circled `0040` in the memory dump to the `IP = 10F4` register entry.

At the bottom of the debugger window, keyboard shortcuts are listed: <F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt> <Sh+F3=M1 Fmt> DEC.

אפשר לבחון את תוכן הפקיקה $int 10h$ ע"י קביעת הערכים של האוגרים CS:IP לערכים המופיעים בטבלה

פקודות הקפיצה לשירותים שונים כתלות ב-AH

CS:IP בתוך IVT במקום $4 \cdot 10h = 40h$

מיסוך (חסימת) פסיקות

- לפעמים נרצה לכבות את פסיקות החומרה בזמן ביצוע קטע קוד מסוים.
למשל בזמן שמתבצע קטע קוד שעורך את אחת הפסיקות.
- נעשה זאת ע"י העברת הדגל (Interrupt Flag) IF ל-0.
- קיימות 2 פקודות מיוחדות עבור כיבוי והדלקת דגל הפסיקות:
פקודת CLI כדי לחסום (למסר) פסיקות ($IF=0$)
פקודת STI כדי לאפשר פסיקות ($IF=1$)

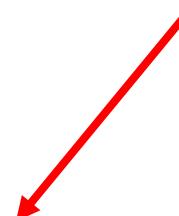
דוגמא: פסיקת חומרה בלחיצה על מקלדת

- פסיקת חומרה 9 int מתרחשת כאשר יש לחיצה על מקש מקלדת.

The screenshot shows a web browser window with the following details:

- Address Bar:** 80864beginner.com/8086-Interrupt-List/int-09h-Keyboard-Data-Ready-Interrupt.h
- Navigation Bar:** Home, 4 Beginner, HTML, Scripts, .Net, MVC, Web API, Microsoft Azure, Mobile
- Page Content:**
 - ### int 09h Keyboard Data Ready Interrupt
 - The keyboard generates an INT 9 every time a key is pushed or released
 - This is a hardware interrupt (IRQ 1) activated by the make or break of every keystroke.
 - . For ASCII keys, when a make code is encountered, the ASCII code and the scan code for the key are placed in the 32-byte keyboard buffer, which is located at 0:41Eh. The ASCII code and scan code are placed in the buffer at the location addressed by the Keyboard Buffer Tail Pointer (0:041Ch). The Keyboard Buffer Tail Pointer is then incremented by 2, and if it points past the end of the buffer, it is adjusted so that it points to the beginning of the buffer.

הפסיקה שומרת את התו
שנלחץ בבאפר בזיכרון



. For ASCII keys, when a make code is encountered, the ASCII code and the scan code for the key are placed in the 32-byte keyboard buffer, which is located at 0:41Eh. The ASCII code and scan code are placed in the buffer at the location addressed by the Keyboard Buffer Tail Pointer (0:041Ch). The Keyboard Buffer Tail Pointer is then incremented by 2, and if it points past the end of the buffer, it is adjusted so that it points to the beginning of the buffer.

Scan Codes

- הפזיקה 9 int שומרת במאפר (מערך זיכרון) את ה-Scan Code של התו שנשלח.

[טבלת הקודים נמצאת בקישור:](https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-6.0/aa299374(v=vs.60))

Key	Scan Code	ASCII or Extended↑			ASCII or Extended↑ with SHFT↑			ASCII or Extended↑ with CIRL			ASCII or Extended↑ with ALI					
		Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	
ESC	1 01	27	1B	ESC	27	1B	ESC	27	1B	ESC	1 01	NUL§				
!	2 02	49	31	1	33	21	!				120	78	NUL			
.	
B	48 30	98	62	b	66	42	B	2 02	SIX	48	30	NUL				
N	49 31	110	6E	n	78	4E	N	14 0E	SO	49	31	NUL				
M	50 32	109	6D	■	77	4D	M	13 0D	CR	50	32	NUL				
,	51 33	44	2C	,	60	3C	<			51	33	NUL§				

Scan Code עבר
האות b הוא 30h

גם שחרור מקש גורם
לפזיקה, אך ה Scan Code הוא שונה
שמרתקבל הוא שונה

כתבת שגרת פסיקה / הרחבת פסיקה קיימת

- טבלת ה-T70 היא אזרז בזיכרון שמכיל את כתובות שגורת הפסיקות.
- נוכל לשנות כתובת של פסיקה מסויימת בטבלה, לכתובת אחרת שבה נמצא המימוש שלנו עבור הפסיקה.
- בזמן שנערוך את טבלת T70 נרצה לחסום פסיקות ע"י הפקודה CLI.

הרחבת שגרת פסיקה

- נרחיב את הקוד של שגרת פסיקה 9, כך שבנוספ' למילוי באפר הקלט היא גם תציג כוכבית בצבע משתנה למסר כל פעם שיש לחיצה על מקש "b"

```
ISR_New proc near uses ax es
```

```
; Code to check for 'b'  
; and print '*'
```

```
prev_isr:  
int 80h
```

```
iret  
ISR_New endp
```

הוראה לקומpileר להוסיף פקודות שבתחלת השגרה יעשו PUSH לאוגרים ES, AX ובסיום השגרה יעשו להם POP

לאחר שהדפסנו את הcocbit, נרצה להפעיל את הפסיקה המקורית שתבצע את דחיפת התו לבאפר. התכנית הראשית צריכה לדאוג לשמר את וקטור הפסיקה המקורי במקום ריק (למשל, h 80) ב-D7V

בסיום הפסיקה המקורי היא תדועה לבקר הפסיקות שנגמר הטיפול בפסיקה, על מנת לאפשר טיפול בפסיקות נוספות.

הרחבת שגרת פסיקה

```
ISR_New proc near uses ax es
```

שמירת אוגר הדגלים מתבצעת באופן אוטומטי

in al, 60h

נקרא את ה Scan Code מהפורט
של המקלדת

cmp al, 30h

אם נקלט Scan Code מתאים
לתו ”b” אז לא נקפוץ ונגייע
לקוד של הדפסת “*”

jnz prev_isr

mov ax, 0B800h

mov es, ax

mov es:[340h], 2E2Ah

prev_isr:

int 80h

קריאה לפסיקה המקורית

iret

IRET גם דואג לדחיפה אוגר הדגלים בחזרה

```
ISR_New endp
```

עדכון טבלת IVT

HERE:

```
mov ax, 0h  
mov es, ax
```

טבלת וקטורי הפסיקות
מתחילת בכתובת 00000h

```
cli
```

כיבוי פסיקת זמן שינוי תוכן ה IVT

```
;moving Int9 into IVT[080h]  
mov ax,es:[9h*4] ;IP  
mov es:[80h*4],ax  
mov ax,es:[9h*4+2] ;CS  
mov es:[80h*4+2],ax
```

כל "וקטור" פסיקה תופס 4 bytes של הטבלה,
לכן כדי להגיע ל IP של פסיקה 9 נבצע את
הכפל $9h \cdot 4 = 36h$

```
;moving ISR_New into IVT[9]  
mov ax,offset ISR_New_int9  
mov es:[9h*4],ax  
mov ax,cs  
mov es:[9h*4+2],ax
```

```
sti
```

הדלקה חוזרת של הפסיקות
וללא אינסוף כדי לחכות להחיצה על
מקש 'b'

- קוד התכנית הראשית:

```
L1:  
    jmp L1  
  
    mov ah, 4ch  
    int 21h  
  
end HERE
```

```

.model small
.stack 100h
.code

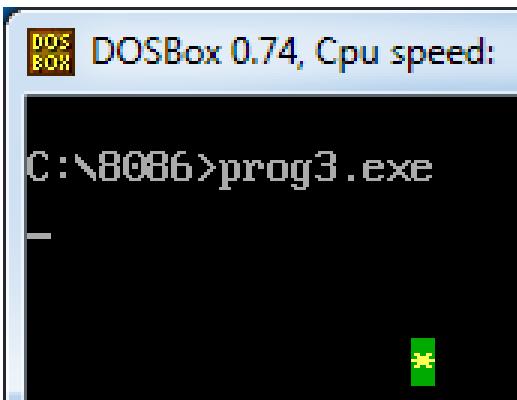
ISR_New_Int9 proc near uses ax es
    in al,60h ;read keyboard_port
    cmp al,30h
    jnz prev_isr ;check if 'b'

    mov ax, 0B800h ;print '*'
    mov es, ax
    mov es:[340h],2E2Ah

prev_isr:
    int 80h ;use the old interrupt

    iret
ISR_New_Int9 endp

```



קוד הרכנית בשלמותו

HERE:

```

        mov ax,0h      ; IVT is location is '0000' address of RAM
        mov es,ax

        cli ; block interrupts

        ;moving Int9 into IVT[080h]
        mov ax,es:[9h*4] ;copying old ISR9 IP to free vector
        mov es:[80h*4],ax
        mov ax,es:[9h*4+2] ;copying old ISR9 CS to free vector
        mov es:[80h*4+2],ax

        ;moving ISR_New_Int9 into IVT[9]
        mov ax,offset ISR_New_int9 ;copying IP of ISR_New to IVT[9]
        mov es:[9h*4],ax
        mov ax,cs ;copying CS of our ISR_New into IVT[9]
        mov es:[9h*4+2],ax

        sti ;enable interrupts

L1:
        jmp L1

        mov ah,4ch
        int 21h
end HERE

```

שירות 01 של פסיקה int 21h מול פסיקת חומרה 9

- שירות 01 של פסיקה (התוכנה) int 21h מוחזירה מקש שנלחץ - אם קיים בבאפר. אם לא קיים - "עוצרת" את ביצוע התוכנה עד שמקש יילחץ.

int 21h Dos Interrupt

Function 01h - read character from standard input

It will waits for a character to be read at the standard input device, then echoes the character to the standard output device and returns the character in AL.

Input: AH = 01h

Output: AL = character from the standard input device

שירות 01 של פסיקה ah 21h מול פסיקת חומרה 9 int 21h

- כאר ש אנחנו רוצים לקלוט אות מהמקלדת, ע"י שירות 01 של ah int 21h מופעלות 2 פסיקות:

```
.model small  
.stack 100h  
.code
```

HERE:

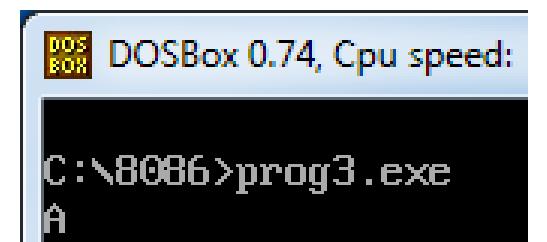
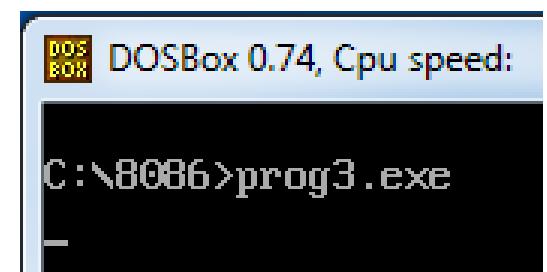
```
mov ah, 01h  
int 21h
```

התכנית מושהה, והפסיקה ממחכה
לקבלתתו בבאפר

```
mov ah, 4ch  
int 21h
```

לאחר לחיצה על המקלדת ומילוי
הבאפר, פסיקה ah 21h מדפיסה את התו
ומסתירה

end HERE



מיקרו-מעבדים ושפת אסמלר

תרגול מס' 12 – TSR, קישור, הידור והרצאה

מיקרו מעבדים ואסמלר – 83255, תש"פ

הפקולטה להנדסה, אוניברסיטת בר-אילן



```

.model small
.stack 100h
.code

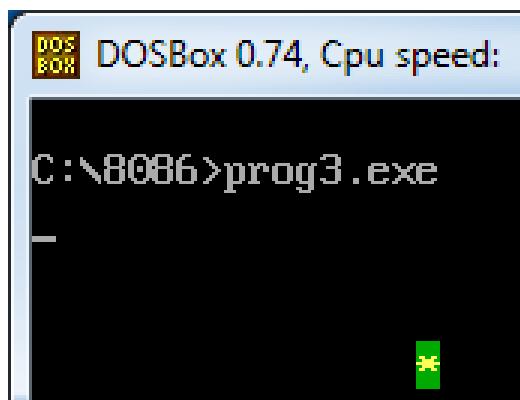
ISR_New_Int9 proc near uses ax es
    in al,60h ;read keyboard_port
    cmp al,30h
    jnz prev_isr ;check if 'b'

    mov ax, 0B800h ;print '*'
    mov es, ax
    mov es:[340h],2E2Ah

prev_isr:
    int 80h ;use the old interrupt

    iret
ISR_New_Int9 endp

```



תזכורת – הרחבת פסיקה 9

HERE:

```

        mov ax,0h      ; IVT is location is '0000' address of RAM
        mov es,ax

        cli ; block interrupts

        ;moving Int9 into IVT[080h]
        mov ax,es:[9h*4] ;copying old ISR9 IP to free vector
        mov es:[80h*4],ax
        mov ax,es:[9h*4+2] ;copying old ISR9 CS to free vector
        mov es:[80h*4+2],ax

        ;moving ISR_New_Int9 into IVT[9]
        mov ax,offset ISR_New_int9 ;copying IP of ISR_New to IVT[9]
        mov es:[9h*4],ax
        mov ax,cs ;copying CS of our ISR_New into IVT[9]
        mov es:[9h*4+2],ax

        sti ;enable interrupts

L1:
        jmp L1
        mov ah,4ch
        int 21h
end HERE

```

לולאה אינסופית כדי להשאיר
הקוד של הפסיקה החלופית
בזיכרון

Terminate and Stay Resident (TSR)

- כאשר אנחנו מושנים וקטור פסיקה, נרצה שהשגרה החדשה תשמר בזיכרון גם אחרי סיום ריצת התוכנית שטוענת אותה ומשנה את ה-IVT.
- למשל נרצה שכוכبية תודפס כל פעם שלחצים על 'b' בDOSBOX
- פסיקה 27h מאפשרת לנו לצאת מהתוכנית חזרה אל מערכת הפעלה, תוך כדי שמירה בזיכרון של חלק מקוד התוכנית.
(הקוד ישאר בזיכרון לנצח – עד כיבוי מתח)

The screenshot shows a web page with the URL vitaly_filatov.tripod.com/ng/asm/asm_011.8.html. The page title is "INT 27h (39) Terminate and Stay Resident". The description reads: "Terminates a program and leaves a specified portion installed in memory." Below this, under "On entry:", there are two columns: "CS" and "DX". The "CS" column has the description "Segment of PSP" and "Address at which next program may be loaded (i.e., highest address to stay resident + 1)".

- DX משמש כקלט לפסיקה ואומר כמה bytes בזיכרון צרי להשאר (החל מתחילה ה-PSP)

Terminate and Stay Resident (TSR)

- כדי לשמר בזיכרון באופן תמידי את קוד השגרה שלנו, נחליף את הלולאה ואת פסיקת סיום התכנית בפסיקת TSR: `int 27h`.
- כאשר משתמשים בפסיקה `h27`, הicy נוח לכתוב את התוכנה ב `tiny model`.
- במקרים של שימוש `tiny` התכנית הראשית מתחילה לרוץ מהפקודה הראשונה בסגמנט הקוד. לכן, נמוך את השגרה שלנו אחרי JMP כדי שתופיע בתחילת הקוד.
- כאשר כותבים תוכנית `tiny` השורה הראשונה בסגמנט הקוד צריכה להיות:
`org 100h`
- נדרש לבצע-link עם הפקודה `ml /AT <file>` או `link /TINY <file>` כדי ליצור +קישור ע"י

```
.model tiny
```

```
.code
```

```
org 100h
```

```
HERE:
```

```
jmp IVT_REPLACE
```

```
ISR_New proc near uses ax es
```

```
    in al,60h
```

```
    cmp al,30h
```

```
    jnz prev_isr
```

```
    mov ax, 0B800h
```

```
    mov es, ax
```

```
    mov es:[340h],2E2Ah
```

```
prev_isr:
```

```
    int 80h
```

```
    iret
```

```
ISR_New endp
```

Terminate and Stay Resident (TSR)

הוספת הפקודה `org 100h` במודול `tiny` כדי למתאם הקצתת מקום ל-PSP

הטכנית מדלגת על הקוד של הרוטינה בגל הקפיצה

יציאה מהטכנית ע"י שימוש ב `int 27h` שמקבל כפרמטר את `DX` עם המספר `h122h`, `122h` עם הערך `122h` bytes כלומר הפסיקה תשאיר בזיכרון את ה `122h` bytes הראשונים של הטענית, ותצא למערכת הפעלה

```
IVT_REPLACE:
```

```
    mov ax,0h
```

```
    mov es,ax
```

```
    cli
```

```
;moving Int9 into IVT[0a0h]
```

```
    mov ax,es:[9h*4]
```

```
    mov es:[80h*4],ax
```

```
    mov ax,es:[9h*4+2]
```

```
    mov es:[80h*4+2],ax
```

```
;moving ISR_New into IVT[9]
```

```
    mov ax,offset ISR_New
```

```
    mov es:[9h*4],ax
```

```
    mov ax,cs
```

```
    mov es:[9h*4+2],ax
```

```
    sti
```

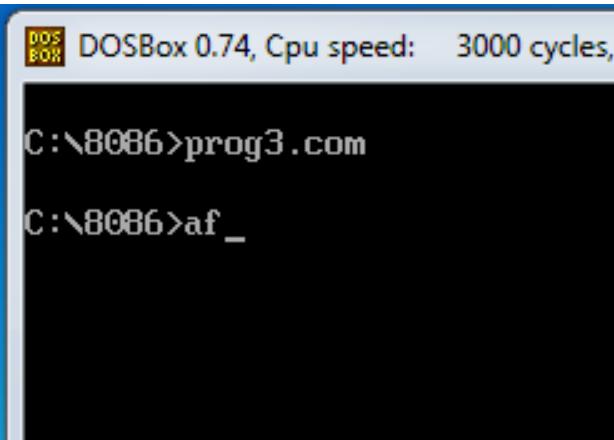
```
    mov dx, 122h
```

```
    int 27h
```

```
end HERE
```

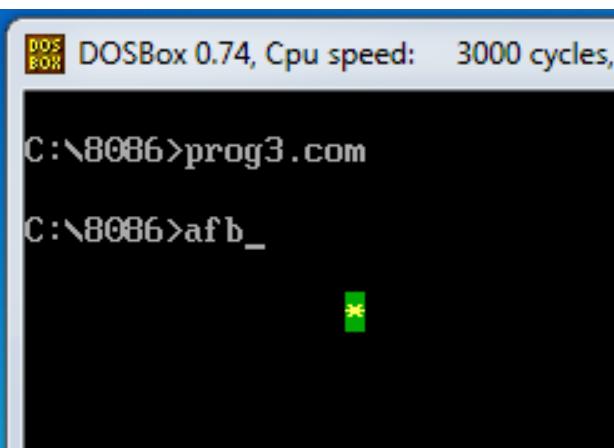
התנהגות ה DOSBOX לאחר ההרצאה

- כל עוד לא הופיעה האות 'b' ה DOSBOX מתנהג כרגע



```
DOSBox 0.74, Cpu speed: 3000 cycles,  
C:\8086>prog3.com  
C:\8086>af_
```

- כאשר מוקלדת האות 'b' ה DOSBOX מציג כוכבית למסך



```
DOSBox 0.74, Cpu speed: 3000 cycles,  
C:\8086>prog3.com  
C:\8086>af b_  
*
```

תכנית עם מספר קבועים

- קיימת אפשרות לבנות את התכנית שלנו ממספר קבועים, כאשר בכל קבוע יש חלק אחר של הקוד.

- כדי להציג משתנה או פונקציה מגיעים מקובץ אחר נשתמש ב `extern`

```
extern print_A:near
```

- כדי להפוך משתנה לפומבי (כלומר שקובץ אחר יוכל לגשת אליו) נשתמש ב- `public`

```
public print_A
```

תכנית עם מספר קבועים

- קובץ `p1.asm` משתמש בפונקציה `print_A` שמוגדרת בקובץ `p2.asm`

`p1.asm`

```
.model small
.stack 100h
.code
extern print_A:near
```

HERE:

```
call print_A
mov ah,4ch
int 21h
```

end HERE

הצורה על זה
שהתכוונית משתמשת
בפונקציה חיצונית
שמוגדרת בקובץ אחר

הצורה על זה ש `A` ש
היא פונקציה פומבית
שקבצים אחרים יכולים
לגשת אליה

`p2.asm`

```
.model small
.stack 100h
.code
public print_A

print_A proc near uses ax es
    mov ax, 0B800h
    mov es, ax
    mov es:[340h], 2E41h ; "A"
    ret
print_A endp
end
```

קימפול של תכנית עם מספר קבצים

- כדי לקימפול תכנית עם מספר קבצים, תחילה נקימפול כל קובץ בנפרד ע"י `masm`

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

```
C:\>masm p1.asm
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

Invoking: ML.EXE /I. /Zm /c /Ta p1.asm

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

Assembling: p1.asm

C:\>masm p2.asm
Microsoft (R) Macro Compatibility Driver
Copyright (C) Microsoft Corp 1993. All rights reserved.

Invoking: ML.EXE /I. /Zm /c /Ta p2.asm

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

Assembling: p2.asm

C:\>
```

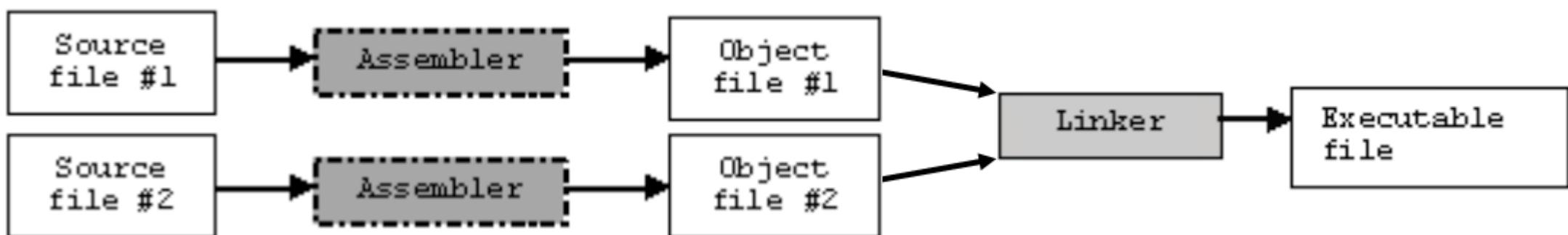
קישור של תכנית עם מספר קבצים

- נקשר בין הקבצים המוקומפלים ע"י link

```
DOSBOX DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX
C:\>link p1.obj p2.obj
Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Run File [p1.exe]: ←
List File [nul.map]:
Libraries [.lib]:
Definitions File [nul.def]:
```

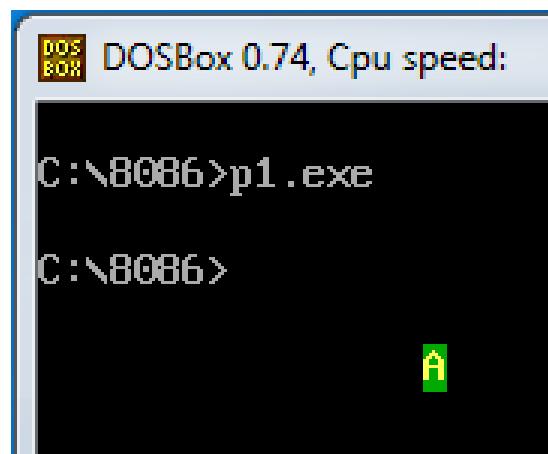
אם פשוט נלחץ 4 פעמים "Enter" באופציית של הلينקר, התכנית שמתקבלת תקרה p1.exe



הרצת התכנית

- נרץ את התכנית כרגע, ע"י p1.exe

- או CV p1.exe להפעיל את הדיבאגר



תכנית שהקוד שלה "מפוזר" בין קבצים

p1.asm

```
.model small
.data
    char dw 2E41h ; 'A'
public char
.code
HERE:
    mov ax, @data
    mov ds, ax

    mov ax, 0B800h
    mov es, ax

    extern CONT:near
    jmp CONT
end HERE
```

קובע שהתכנית מתחילה בתוויות HERE

מספיק להציג על
המחסנית באחד מן
הקבצים

כאשר מייבאים משתנה
מבחן, חייבים להציג
על הגודל שלו

המשך התכנית אחרי
קפיצה

הצירה שCONTnear
הוא יעד קפיצה שנמצא בקובץ אחר

p2.asm

```
.model small
.stack 100h
.data
extern char:word
.code
public CONT
CONT:
    mov ax, char
    mov es:[340h], ax

    mov ah, 4ch
    int 21h
end
```

קימפול והרצה

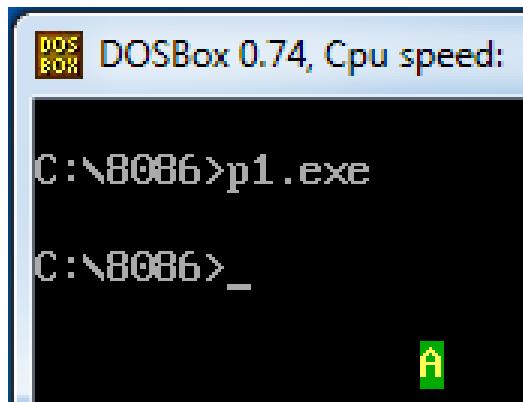
- גם כאן נקמפל כל קובץ בנפרד:

```
masm p2.asm
```

```
link p1.obj p2.obj
```

- נקשר בין הקבצים:

- ובהנחה שהשארכנו את אופציות ה-link הדיפולטיביות (לוחצים 4 פעמים enter) התכנית מקבלת נקראת p1.exe



אפשרויות בהרצה masm

- כדי לבדוק את האופציות שאיתן ניתן להפעיל את masm וכותב "?/asm"

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

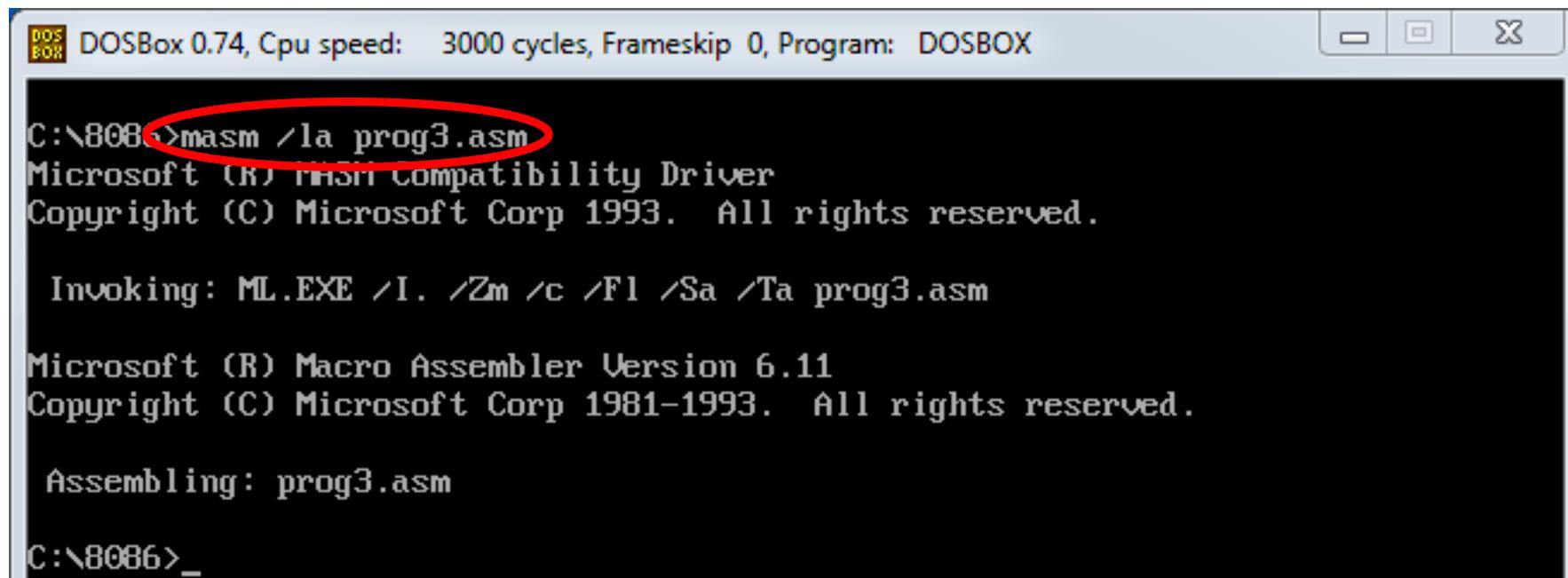
```
C:\>masm /?
usage: masm /options source(.asm),[out(.obj)], [list(.lst)], [cref(.crf)][;]

/c          Generate cross-reference
/D<sym>[=<val>] Define symbol
/e          Emulate floating point instructions and IEEE format
/I<path>   Search directory for include files
/l[a]       Generate listing, a-list all
/M{l|u|x}  Preserve case of labels: l-All, x-Globals, u-Uppercase Globals
/n          Suppress symbol tables in listing
/t          Suppress messages for successful assembly
/w{0|1|2}  Set warning level: 0-None, 1-Serious, 2-Advisory
/X          List false conditionals
/Zi         Generate symbolic information for CodeView
/Zd         Generate line-number information
```

אחד האופציות היא הוצאה קובץ LST. משמעות הסוגרים המרובעים היא שאפשר לכתוב a ואפשר לא לכתוב. לעומת גם "masm /la" וגם "/asm" הן פקודות תקין

קובץ LST

- קובץ LST הוא קובץ תיעוד לפעולות הקומפיילר.
- כדי לקבל אותו עברו קומפיילציה של הקובץ p1.asm (בנוסף לקובץ obj שמתיקבל) נכתב "masm /la p1.asm"



DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

```
C:\8086>masm /la prog3.asm
Microsoft (R) MS-DOS compatibility Driver
Copyright (C) Microsoft Corp 1993. All rights reserved.

Invoking: ML.EXE /I. /Zm /c /F1 /Sa /Ta prog3.asm

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

Assembling: prog3.asm

C:\8086>
```

A screenshot of a DOSBox window titled "DOSBox 0.74". The window shows the command "masm /la prog3.asm" being run in a DOS environment. The output of the assembly process is displayed below the command. The line "masm /la prog3.asm" is circled in red. The window has standard DOS-style icons in the title bar.

קובץ LST

p1.asm	13/6/2018 1:52 AM	ASM File	1 KB
P1.LST	13/6/2018 3:14 AM	LST File	2 KB
P1.OBJ	13/6/2018 3:14 AM	OBJ File	1 KB

- לאחר הרצת "masm /la p1.asm" בתקיית הרצה שלנו יופיע קובץ p1.lst

כתבת יחסית
בSEGMENT DATA

SEGMENT DATA

SEGMENT CODE

כתבת יחסית
בSEGMENT CODE

```

1 Microsoft (R) Macro Assembler Version 6.11          06/13/18 03:25:02
2 p1.asm                                              Page 1 - 1
3
4
5 .model small
6 0000 .data
7 0000 2E41      char dw 2E41h ;'A'
8
9 0000           public char
10 0000 .code
11 0000 HERE:
12 0003 B8 ----- R      mov ax, @data
13 0003 8E D8      mov ds, ax
14 0005 B8 B800      mov ax, 0B800h
15 0008 8E C0      mov es, ax
16
17 000A E9 0000 E      extern CONT:near
18 000A E9 0000 E      jmp CONT
19
20 end HERE

```

הקוד שכתבנו

פקודות מכונה

קובץ LST

p1.asm	13/6/2018 1:52 AM	ASM File	1 KB
P1.LST	13/6/2018 3:14 AM	LST File	2 KB
P1.OBJ	13/6/2018 3:14 AM	OBJ File	1 KB

- לאחר הרצת "masm /la p1.asm" יופיע קובץ p1.lst

```
1 Microsoft (R) Macro Assembler Version 6.11          06/13/18 03:25:02
2 p1.asm                                              Page 1 - 1
3
4
5           .model small
6 0000       .data
7 0000 2E41     char dw 2E41h ;'A'
8           public char
9 0000       .code
10 0000      HERE:
11 0000 B8 ----- R    mov ax, @data
12 0003 8E D8      mov ds, ax
13
14 0005 B8 B800    mov ax, 0B800h
15 0008 8E C0      mov es, ax
16
17           extern CONT:near
18 000A E9 0000 E   jmp CONT
19           end HERE
```

קריאה לSEGMENT שעדיין אין לו כתובת כל בשלב זה. מיקום SEGMENT ה DATA נקבע בשלב טיענת התכנית לזיכרון

שומר מקום עבור כתובת שמקורה בקובץ אחר שהתוכן שלו לא ידוע לקומpileר בשלב זה

קומפיילר (פקודת masm) ומיקום הsegmantים

```
.model small
.stack 100h
.data
    charA dw 2E41h ; 'A'
    charB dw 2E42h ; 'B'
    charC dw 2E43h ; 'C'
    charD dw 2E44h ; 'D'
    charE dw 2E45h ; 'E'
.code
HERE:
    mov bx,charD
    mov dx,charE
    mov ax,bx

    mov ah,4ch
    int 21h
end HERE
```

- בזמן הקומpileציה, הקומפיילר לא ידע
היכן ימוקמו הsegmantים.

Microsoft (R) Macro Assembler Version 6.11
prog3.asm Page 1 - 1

```
.model small
.stack 100h
.data
    0000      .data
    0000 2E41      charA dw 2E41h ; 'A'
    0002 2E42      charB dw 2E42h ; 'B'
    0004 2E43      charC dw 2E43h ; 'C'
    0006 2E44      charD dw 2E44h ; 'D'
    0008 2E45      charE dw 2E45h ; 'E'


---


    0000      .code
    0000      HERE:
    0000 8B 1E 0006 R      mov bx,charD
    0004 8B 16 0008 R      mov dx,charE
    0008 8B C3      mov ax,bx
    000A B4 4C      mov ah,4ch
    000C CD 21      int 21h


---


    end HERE
```

SEGMENT DATA CODE

קומפיילר (פקודת masm) ומיקום הsegmantים

```

.model small
.stack 100h

0000      .data
0000 2E41      charA dw 2E41h ;'A'
0002 2E42      charB dw 2E42h ;'B'
0004 2E43      charC dw 2E43h ;'C'
0006 2E44      charD dw 2E44h ;'D' // Line 6
0008 2E45      charE dw 2E45h ;'E'

0000      .code
0000
HERE:
0000 8B 1E 0006 R      mov bx,charD
0004 8B 16 0008 R      mov dx,charE
0008 8B C3              mov ax,bx

000A B4 4C              mov ah,4ch
000C CD 21              int 21h

end HERE

```

SEGMENT
DATA

SEGMENT
CODE

סיום הפקודה נמצא
בהתיקט של Eh+2=0Eh+2=0Ch
מתחלת סגמנט הקוד

- קובץ LST של הקומפיילר מתיכון למשתנים בהתאם יחסיות בסגמנט DATA

- בנוסף יש בקובץ "סיכום" של כמה כל סגמנט תופס בזיכרון

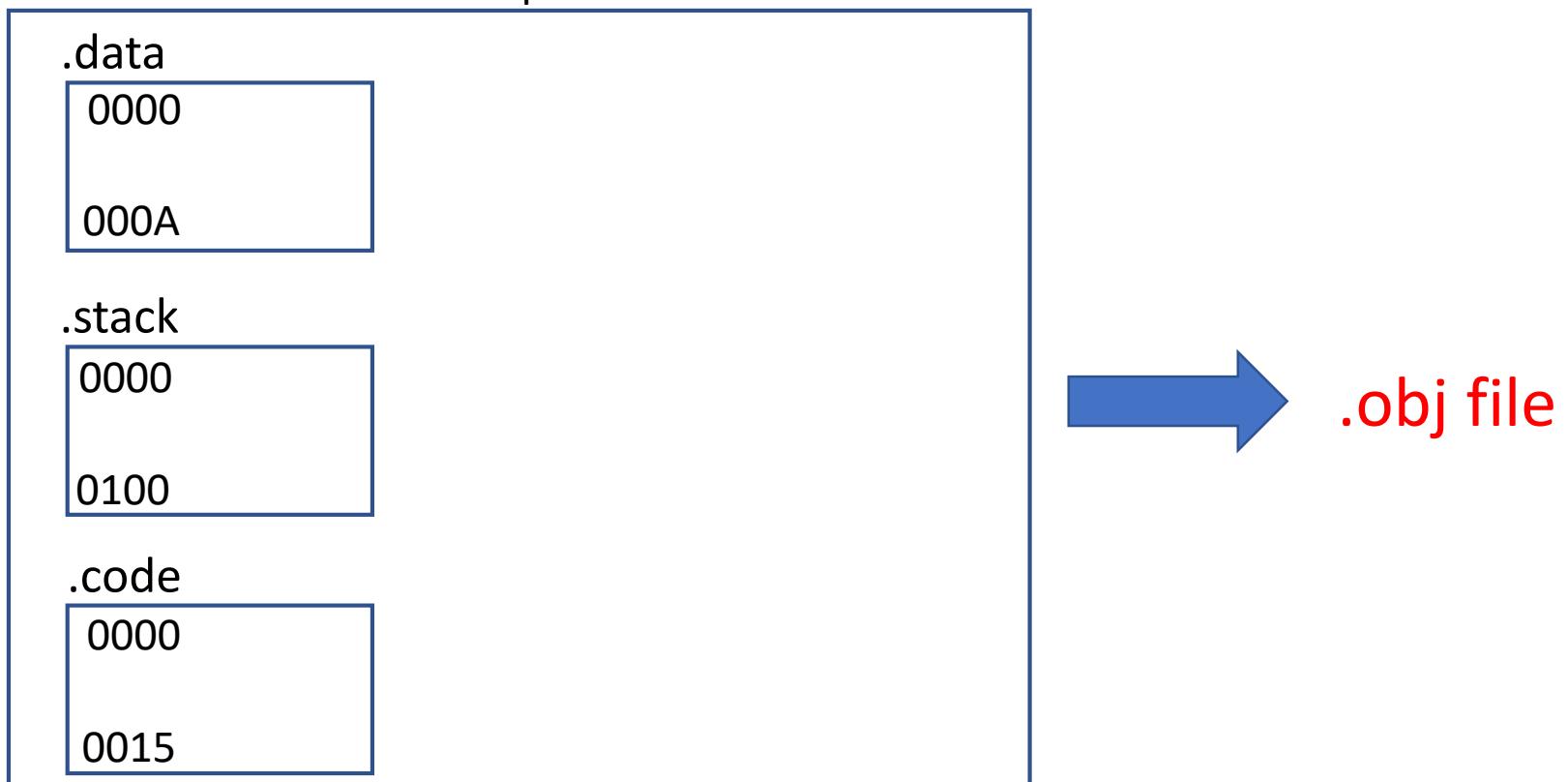
Segments and Groups:						
	Name	Size	Length	Align	Combine Class	
DGROUP	GROUP	
DATA	Word	Public	'DATA'
STACK	Para	Stack	'STACK'
TEXT	Word	Public	'CODE'

קורומפיבילר (פקודת masm) ומיקום הסגמנטים

Segments and Groups:

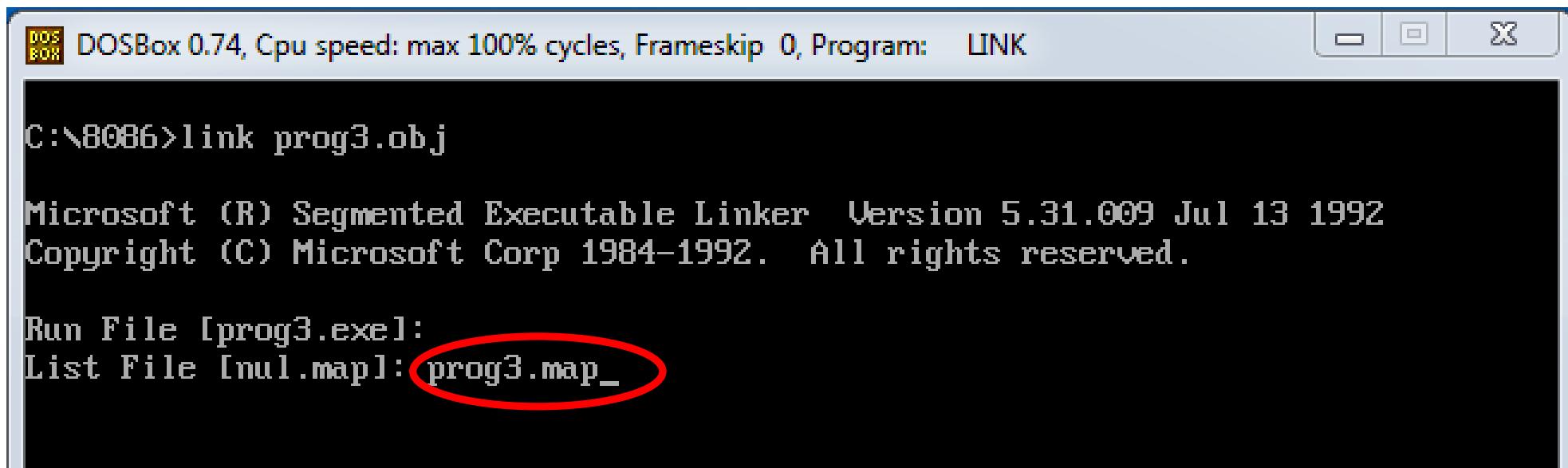
	Name	Size	Length	Align	Combine	Class
DGROUP	.	GROUP				
_DATA	.	16 Bit	000A	Word	Public	'DATA'
STACK	.	16 Bit	0100	Para	Stack	'STACK'
TEXT	.	16 Bit	000E	Word	Public	'CODE'

Compiler



קובץ MAP שמתקבל מ Linker

- כדי לקבל קובץ MAP של שלב הקישור, נכתבaira נרצה שהקובץ יקרא בשלב 2 של תהליך הקישור



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: LINK
C:\>link prog3.obj

Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Run File [prog3.exe]:
List File [nul.map]: prog3.map_
```

Name	Date modified	Type	Size
PROG3.MAP	17/6/2018 12:32 PM	MAP File	1 KB

קישור (פקודת link) ומיקום הsegmantים

- קובץ MAP של תהליך הקישור מכיל את המידע כיצד למקם את הsegmantים בזיכרון בזמן טעינה

Start	Stop	Length	Name	Class
000000H	0000DH	0000EH	_TEXT	CODE
0000EH	00017H	0000AH	_DATA	DATA
00020H	0011FH	00100H	STACK	STACK

Origin	Group
0000:0	DGROUP

Program entry point at 0000:0000

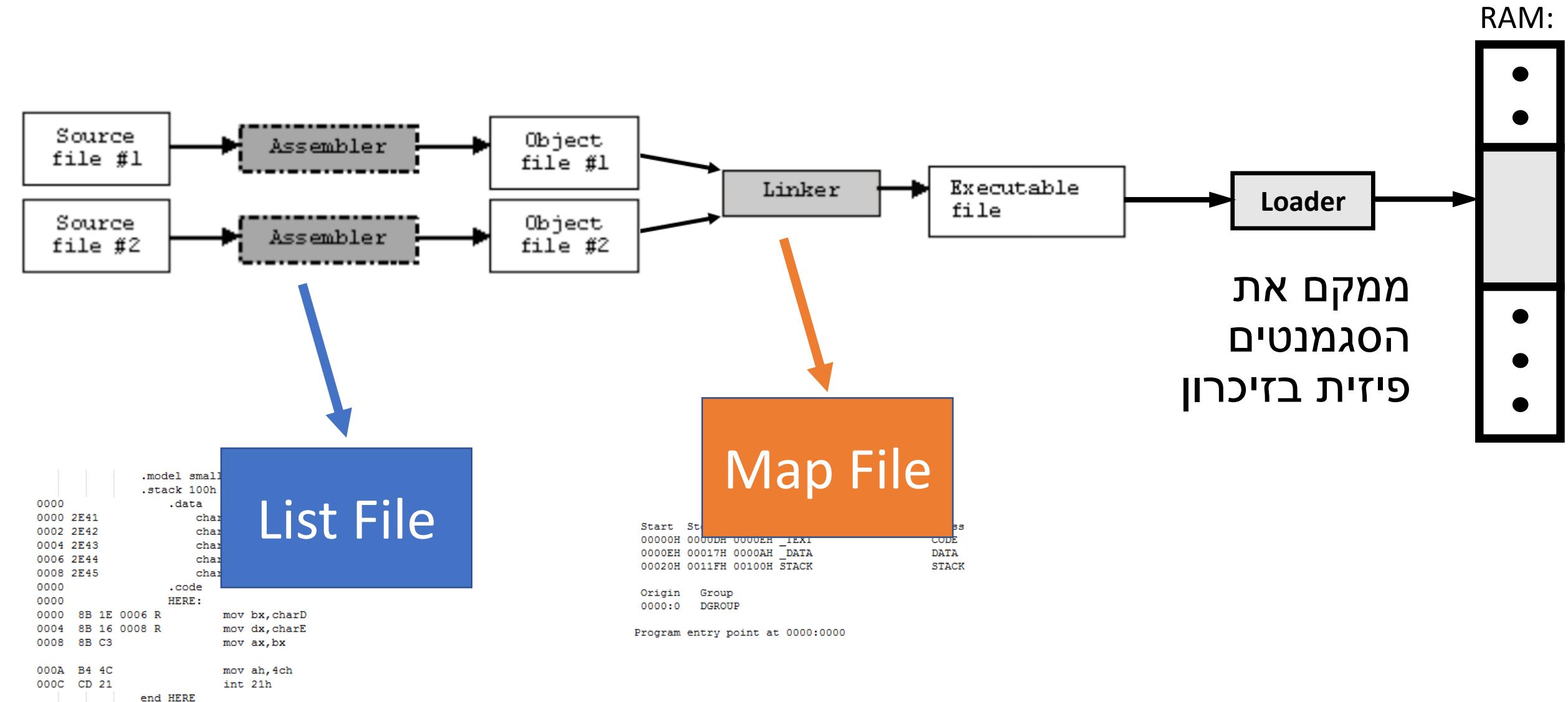
עדין מדובר בכתבאות לא פיזיות
(לא אמתיות)!

- מידע זה לא זמין בשלב הקומpileציה כיון שתוכן הsegmantים יכול לבוא מקבצים שונים

```
.model small          p1.asm
.code
HERE:
    ;start of the code
    extern CONT:near
    jmp CONT
end HERE
```

```
.model small          p2.asm
.code
public CONT
CONT:
    ;continue code
    mov ah,4ch
    int 21h
end
```

קובץ List וקובץ map



קישור (פקודת link) ומיקום הSegments

- נבחן שה linker מיקם את סגמנט DATA צמוד לSEGMENT CODE

Start	Stop	Length	Name	Class
000000H	0000DH	0000EH	_TEXT	CODE
000000H	00017H	0000AH	_DATA	DATA
00020H	0011FH	00100H	STACK	STACK

הכתובות המופיעות בקובץ הן
כתובות פיזיות של 5 ספרות.
כתובת הSEGMENT היא 4 ספרות
הראשונות.
הספרה החמישית היא היחס
ביצוג segment:offset

- תרגיל: מהו CS של סגמנט הקוד המתקבל?

• תשובה 0000h

- תרגיל: מהו DS של סגמנט DATA המתקבל?

• תשובה 0000h

קישור (PCODE) ומיקום הסגמנטים

```

.model small
.stack 100h
.data
    charA dw 2E41h ; 'A'
    charB dw 2E42h ; 'B'
    charC dw 2E43h ; 'C'
    charD dw 2E44h ; 'D'
    charE dw 2E45h ; 'E'
.code
HERE:
    mov ax, @data
    mov ds, ax
    mov bx, charD
    mov dx, charE

    mov ax, 0B800h
    mov es, ax
    mov ax,bx

    mov ah, 4ch
    int 21h
end HERE

```

נוסיף פקודות
לקטע הקוד על
מנת "לנפח" את
הגודל שלו

masm.lst:

	.model small	.stack 100h
0000	.data	charA dw 2E41h ; 'A'
0000		charB dw 2E42h ; 'B'
0002		charC dw 2E43h ; 'C'
0004		charD dw 2E44h ; 'D'
0006		charE dw 2E45h ; 'E'
0008		
0000	.code	
0000	HERE:	
0000		mov ax, @data
0003		mov ds, ax
0005		mov bx, charD
0009		mov dx, charE
000D		mov ax, 0B800h
0010		mov es, ax
0012		mov ax,bx
0014		mov ah, 4ch
0016		int 21h
		end HERE

חפיפה בין סגמנט קוד וסגמנט נתונים

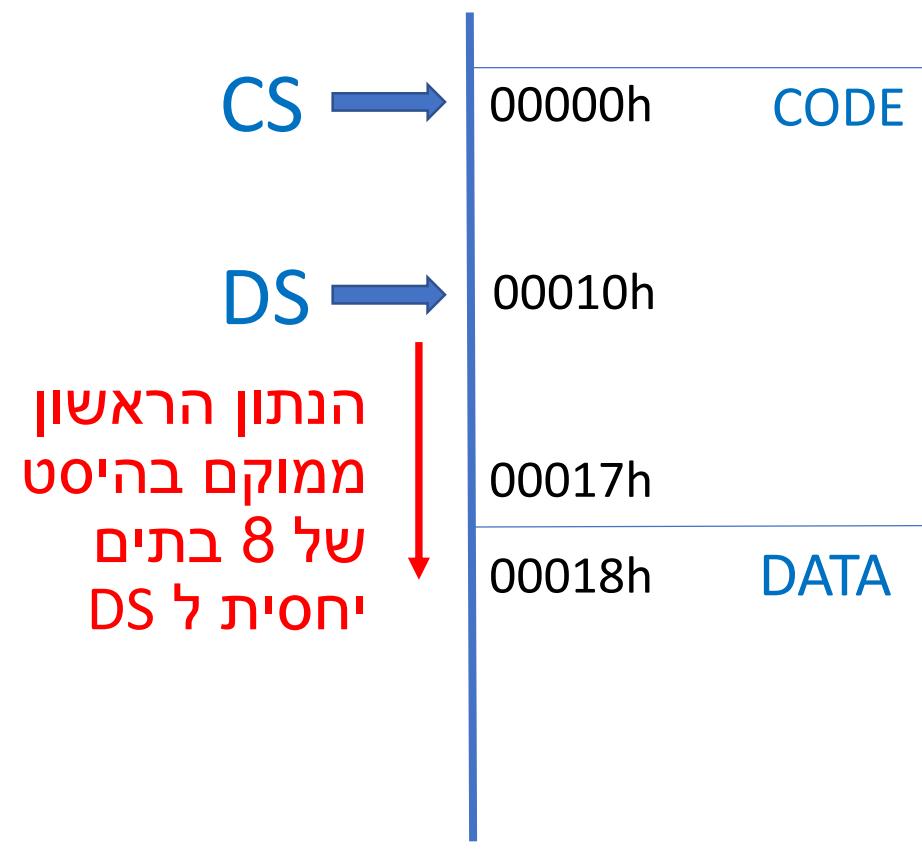
Start	Stop	Length	Name
00000H	00017H	00018H	_TEXT
00018H	00021H	0000AH	_DATA
00030H	0012FH	00100H	STACK

- אם נרץ link מקבל את קובץ map הבא:

- סגמנט DATA ממוקם פיזית אחרי סגמנט הקוד ומתחילה בכתובת h18000

CS = 00000h , DS = 00010h •

נחת הקומpileר שהסגמנטים לא חופפים היא לא נכונה.



קישור (פקודת link) ומיקום הסגמנטים

masm.lst

```
0000          .data
0000 2E41      charA dw 2E41h ;'A'
0002 2E42      charB dw 2E42h ;'B'
0004 2E43      charC dw 2E43h ;'C'
0006 2E44      charD dw 2E44h ;'D'
0008 2E45      charE dw 2E45h ;'E'

0000          .code
0000
0000 HERE:
0000 8B 1E 0006 R      mov bx,charD
```

אחרי פעולה של linking
הכתובת תהפוך ל
 $0006h + 0008h = 000Eh$ (=14 d)

מספר הבטים שיש להתקדם מאייפה
שDS מצביע עד לתחילת הנתונים באמת

- כדי לאפשר לリンקר לפצות על הנחה
שგיה זו, הקומpileר מסמן באות R
את המיקומות בהם הnicח שתחילת
סגןט DATA היא ללא היסט DS

Link.lst

Start	Stop	Length	Name
00000H	00017H	00018H	_TEXT
00018H	00021H	0000AH	_DATA
00030H	0012FH	00100H	STACK

טעינה הוכנית לזכרון

```
.model small  
.stack 100h  
.data  
    charA dw 2E41h ; 'A'  
    charB dw 2E42h ; 'B'  
    charC dw 2E43h ; 'C'  
    charD dw 2E44h ; 'D'  
    charE dw 2E45h ; 'E'
```

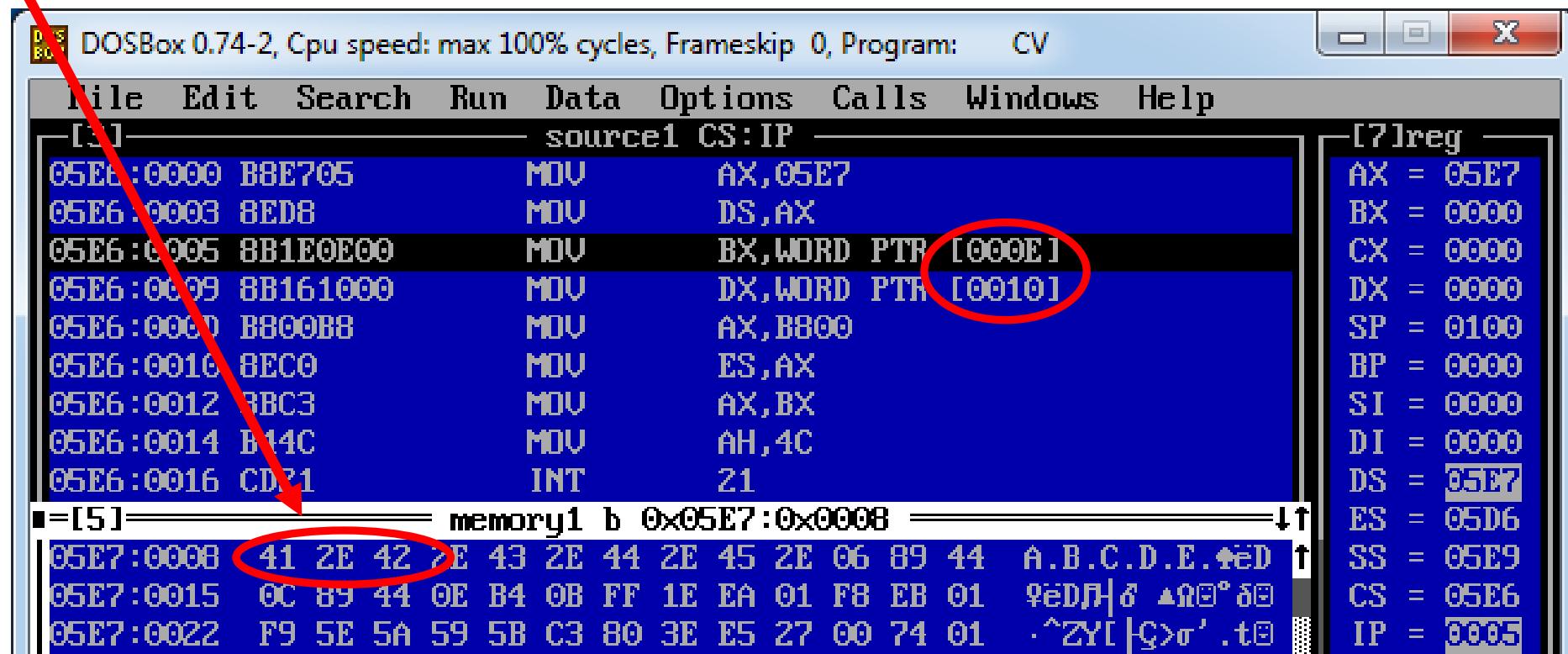
.code

HERE :

```
    mov ax, @data  
    mov ds, ax  
    mov bx, charD  
    mov dx, charE  
  
    mov ax, 0B800h  
    mov es, ax  
    mov ax,bx  
  
    mov ah, 4ch  
    int 21h
```

end HERE

- בסוף מטור CV debugger ניתן להבחין בהיסטים שנוטפו בפניהם לsegment DATA



טעינה התיכנית לזכרון

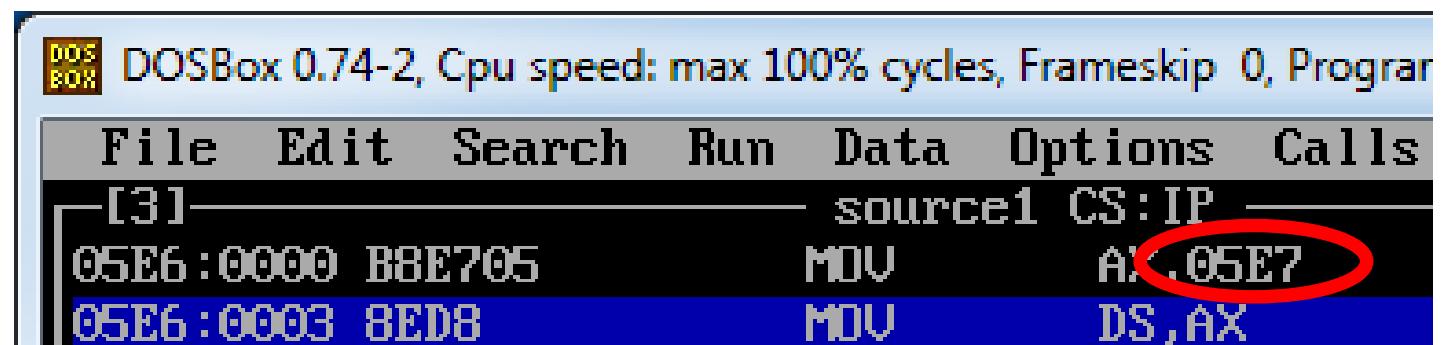
```
.model small
.stack 100h
.data
    charA dw 2E41h ; 'A'
    charB dw 2E42h ; 'B'
    charC dw 2E43h ; 'C'
    charD dw 2E44h ; 'D'
    charE dw 2E45h ; 'E'
.code
HERE:
    mov ax, @data
    mov ds, ax
    mov bx, charD
    mov dx, charE

    mov ax, 0B800h
    mov es, ax
    mov ax,bx

    mov ah, 4ch
    int 21h
end HERE
```

- בשלב LOAD המיקומים הפיזיים של הסגמנטים נקבעים, והמקומות החסרים בקוד מוחלפים בכתבאות אמתיות. את הכתובות של שלב זה ניתן לבדוק דרך CV Debugger

0000	B8	----- R	mov ax, @data
0003	8E	D8	mov ds, ax



כיצד ניתן לגלות מהו @data

- מיקומו של סגמנט הנתונים (@data) נקבע בשלב ה LOAD, ואין לנו קובץ Ist Überoor שלב זה.
- כיצד נוכל לגלות את מיקום סגמנט הנתונים (@data) ללא הפעלת התכנית ב ?CV debugger