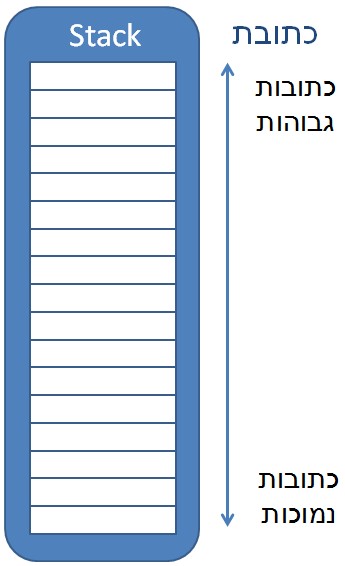
# פעולות ומחסנית

## מחסנית

push & pop



המחסנית (stack), היא סגמנט בזיכרון, המשמשת לאחסון של משתנים לזמן קצר.

המחסנית מנוהלת בדרך הדומה מאוד לדרך שבה מנוהלת מחסנית תחמושת של רובה אוטומטי. הכדורים נטענים במחסנית תחמושת בזה אחר זה, ודוחפים זה את זה דרך הפתח הנמצא בראש המחסנית. פריקת הכדורים מן המחסנית נעשית גם היא דרך הפתח, אך סדר הוצאת הכדורים הפוך מסדר הכנסת הכדורים: זה שנכנס-אחרון-יוצא-ראשון

LIFO – Last in First Out.

כל דחיפה של כדור למחסנית התחמושת מזיזה את כל הכדורים שכבר נמצאים בה, ומפנה

מקום לכדור החדש. פתח מחסנית התחמושת (ראש המחסנית) נשאר, כמובן, קבוע במקומו.

בדומה למחסנית התחמושת, גם המחסנית שבזיכרון המחשב מנוהלת בדרך המאפשרת

להכניס ולהוציא נתונים בשיטת נכנס-אחרון-יוצא-ראשון.

הכנסת נתון למחסנית שבזיכרון אינה גורמת להזזת הנתונים הקודמים. בִמקום זה, ראש המחסנית זז, כלומר המקום שאליו נכנס הנתון הבא משתנה, כדי שהנתון החדש שייכנס לא ייכתב על הנתון הקודם ויהרוס אותו.

הכנסת נתון למחסנית נקראת דחיפה - Pushing,

הוצאת נתון מן המחסנית נקראת שליפה - Popping

מעקב אחר מיקומו של ראש המחסנית, המשתנה עם כל דחיפה ושליפה, נעשה על-ידי האוגר SP שנקרא מצביע המחסנית, ותפקידו להצביע בכל רגע נתון על הכתובת שאליה נדחף הנתון האחרון. (SP מכיל כתובת יחסית לאוגר .SS)

שימו לב 🎔: כאשר משתמשים במחסנית, יש להימנע משליפת נתונים מכתובות שלא הוכנסו

אליהן נתונים. כדי למנוע מצב של שליפת נתונים אקראיים, כיוון שלכתובת הזו במחסנית לא הוכנס כל נתון במהלך העבודה עם המחסנית.

המצב הזה מכונה גלישת מחסנית ( Stack overflow ), שכן ראש המחסנית גולש בשליפה הזו מעבר למיקומו הראשוני, לאזור בזיכרון שאינו שייך למחסנית.

בהגדרת המחסנית עלינו להקפיד להקצות מספיק מקום לסגמנט המחסנית כדי שיהיה אפשר לנהל את כל הנתונים הדרושים.

לסיכום, המעקב אחר מיקומו של ראש המחסנית נעשה באוגר sp, כל פעולת דחיפה מקטינה באופן אוטומטי את ערכו של האוגר SP , וכל פעולת שליפה מגדילה את הערך הזה.

מחסנית

* המחסנית (stack), היא סגמנט בזיכרון, המשמשת לאחסון של משתנים לזמן קצר.
* המחסנית מנוהלת בדרך הדומה מאוד לדרך שבה מנוהלת מחסנית תחמושת של רובה אוטומטי.
* הכדורים נטענים במחסנית תחמושת בזה אחר זה, ודוחפים זה את זה דרך הפתח הנמצא בראש המחסנית.
* פריקת הכדורים מן המחסנית נעשית גם היא דרך הפתח, אך סדר הוצאת הכדורים הפוך מסדר הכנסת הכדורים: זה שנכנס-אחרון-יוצא-ראשון
* LIFO – Last in First Out.

Stack

בדומה למחסנית התחמושת, גם המחסנית שבזיכרון המחשב מנוהלת בדרך המאפשרת להכניס ולהוציא נתונים בשיטת   
LIFO – Last in First Out נכנס-אחרון-יוצא-ראשון.

הכנסת נתון למחסנית שבזיכרון אינה גורמת להזזת הנתונים הקודמים. בִמקום זה, ראש המחסנית זז, כלומר המקום שאליו נכנס הנתון הבא משתנה, כדי שהנתון החדש שייכנס לא ייכתב על הנתון הקודם ויהרוס אותו.

הכנסת נתון למחסנית נקראת דחיפה - Pushing,

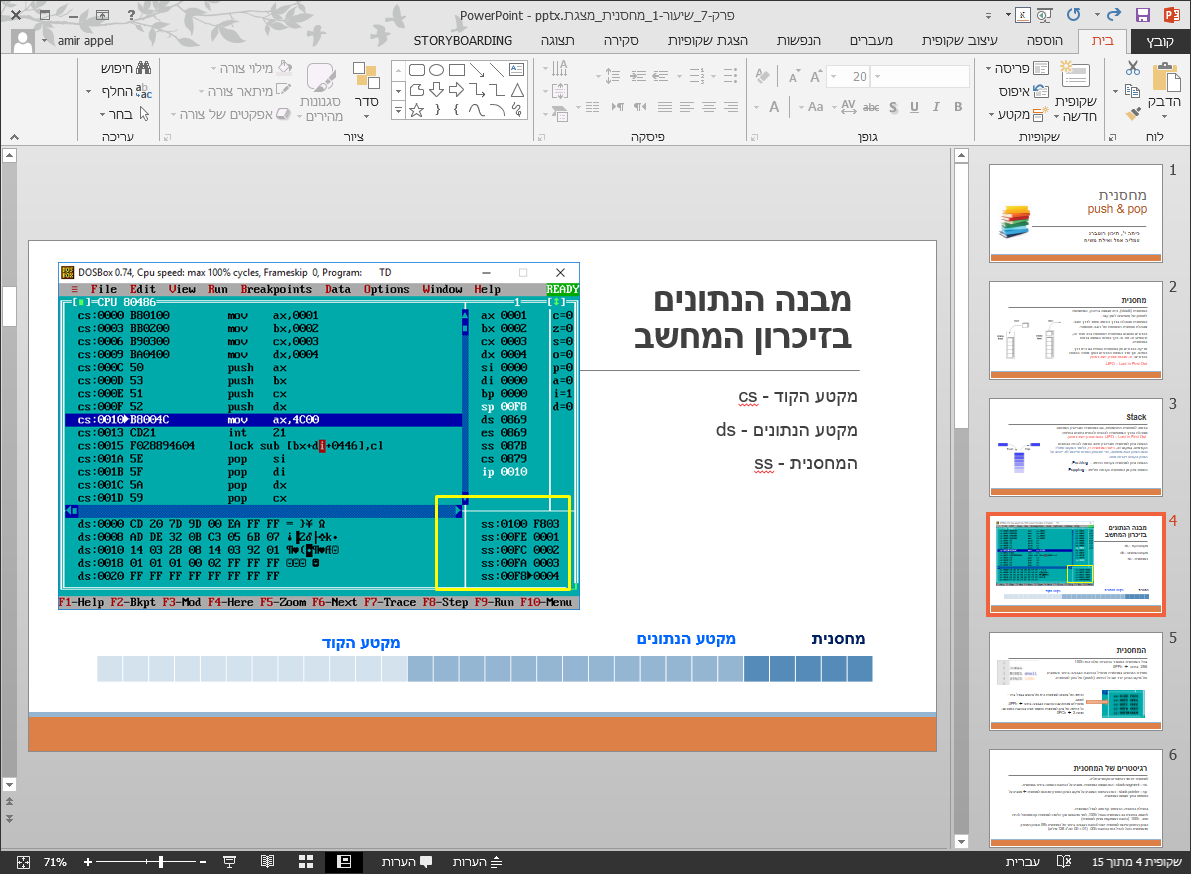
הוצאת נתון מן המחסנית נקראת שליפה – Popping

מבנה הנתונים בזיכרון המחשב

מקטע הקוד - cs

מקטע הנתונים - ds

המחסנית - ss



כמו כל סגמנט בזיכרון, המחסנית היא אזור בזיכרון שמתחיל בכתובת כלשהי ותופס גודל מוגדר של זיכרון.

גודל האזור בזיכרון שמוקצה למחסנית נקבע על-ידי המתכנת בתחילת התוכנית. הקצאת המקום נעשית בדרך הבאה:

STACK number of bytes

לדוגמה, כדי להקצות מחסנית בגודל 256 בתים, נגדיר - כפי שמוגדר גם בקובץ base.asm:

STACK 100h

רגיסטרים של המחסנית

למחסנית יש שני רגיסטרים שקשורים אליה.

stack segment – ss - הוא סגמנט המחסנית. מצביע על הכתובת הנמוכה ביותר במחסנית.

stack pointer – sp - הוא הרגיסטר המצביע על מיקום הנתון האחרון שהוכנס למחסנית ⇦ מצביע על האופסט בתוך סגמנט המחסנית.

בתחילת התוכנית, הרגיסטר sp שווה לגודל המחסנית.

לדוגמה בתוכנית בה המחסנית בגודל 100h, לפני שהכנסנו ערך כלשהו למחסנית sp מאותחל להיות שווה 100h (כתובת הממוקמת מחוץ למחסנית)

* הנתון הראשון שיכנס למחסנית יכנס לכתובת הגבוהה ביותר של המחסנית 0ffh והנתון האחרון שהמחסנית תוכל להכיל הוא הכתובת 00h. (01 ו- 00 סה"כ 128 מילים)

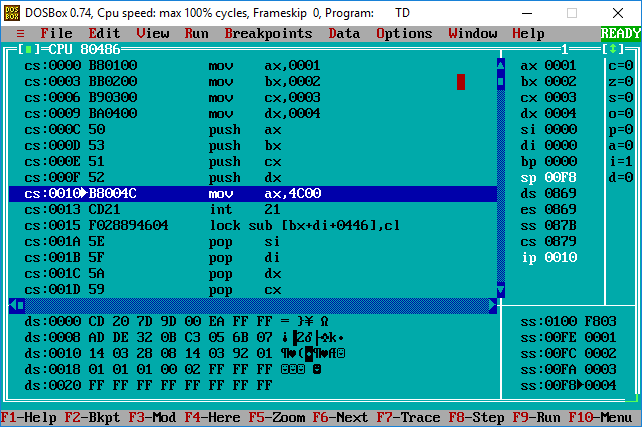
mov ax, 01

mov bx, 02

* mov cx, 03
* mov dx, 04
* push ax
* push bx
* push cx
* push dx

עם כל דחיפה של נתון למחסנית הביטים הגבוהים יכנסו לכתובת הגבוהה והביטים הנמוכים לכתובת הנמוכה.

ה – SP יצביע על הכתובת הנמוכה של המילה במחסנית

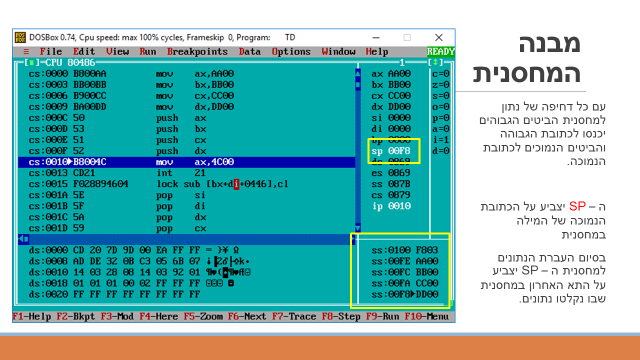


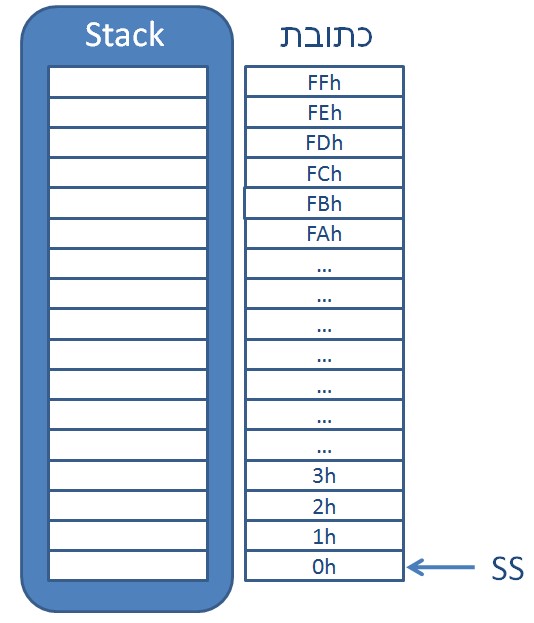
|  |  |
| --- | --- |
| **נתון** | **כתובת במחסנית** |
| 00  01 | FF  FE |
| 00  02 | FD  FC |
| 00  03 | FB  FA |
| 00  04 | F9  F8 |

עם כל דחיפה של נתון למחסנית הביטים הגבוהים יכנסו לכתובת הגבוהה והביטים הנמוכים לכתובת הנמוכה.

ה – SP יצביע על הכתובת הנמוכה של המילה במחסנית

בסיום העברת הנתונים למחסנית ה – SP יצביע על התא האחרון במחסנית שבו נקלטו נתונים.



למחסנית יש שני רגיסטרים שקשורים אליה.

stack segment – ss - הוא סגמנט המחסנית. מצביע על הכתובת הנמוכה ביותר במחסנית.

stack pointer – sp - הוא מצביע המחסנית. מצביע על האופסט בתוך סגמנט המחסנית.

בתחילת התוכנית, הרגיסטר sp שווה לגודל המחסנית.

בדוגמה שנשתמש בה, מחסנית בגודל 100h, לפני שהכנסנו ערך כלשהו למחסנית sp מאותחל להיות שווה h100.

במחסנית בגודל 100h, כלומר 256 בתים, מרחב הכתובות אינו מגיע עד 100h אלא הוא בין 0 ל - offh. (הערך ההתחלתי של מצביע על כתובת שהיא בדיוק בית אחד מעל קצה המחסנית).

המחסנית מנוהלת בשיטת LIFO – Last In First Out,

כלומר הערך שנכנס אחרון הוא הראשון לצאת מהמחסנית.

לפני הכנסה של נתונים למחסנית, ערכו של sp יורד ולאחר הוצאה של נתונים מהמחסנית ערכו של sp עולה.

פקודת PUSH

פקודת push גורמת להכנסה של ערך למחסנית.

הפקודה נכתבת כך:

push operand

מה יבוצע?

- ערכו של sp יירד בשתיים: 2-sp=sp.

- ערכו של האופרנד יועתק למחסנית, לכתובת ss:sp.

ערכו של sp תמיד יורד בשתיים עם פקודת push, כלומר הוא מצביע על כתובת שרחוקה שני בתים מהכתובת האחרונה עליה הצביע.

המשמעות היא שאפשר לדחוף למחסנית רק משתנים בגודל של שני בתים – word. כל ניסיון לבצע push לכמות אחרת של בתים – יוביל לשגיאה.

push ax

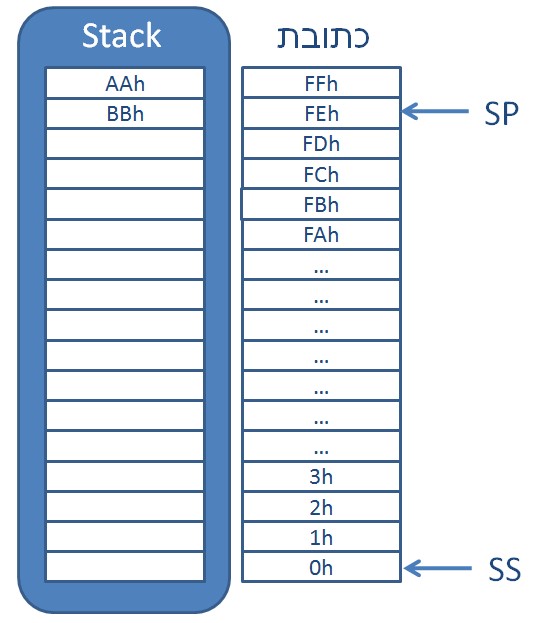
push 10

push [var]

הפקודה הראשונה תדחוף למחסנית את ax.

הפקודה השניה תדחוף למחסנית את הערך 10 (בצורתו כ-word, לא כ־byte).

הפקודה השלישית תדחוף למחסנית את תוכן המשתנה [var] – בתנאי שהוא מגודל word.



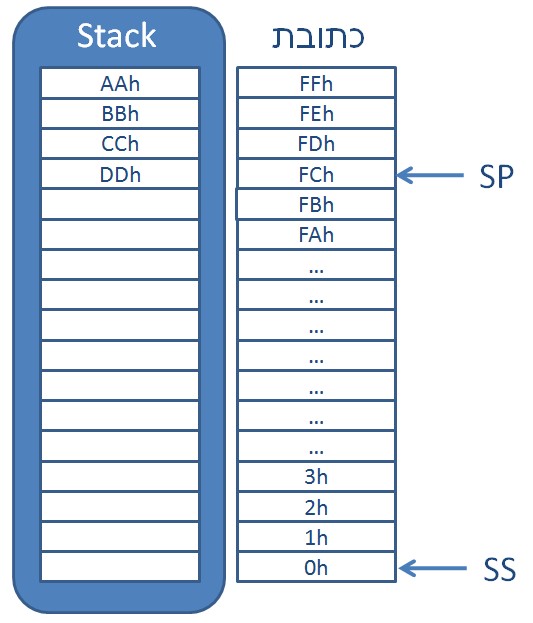
מה יהיה מצב המחסנית לאחר ביצוע הפקודות הבאות?

mov ax, 0AABBh

push ax

הביטים הגבוהים נכנסים לכתובת הגבוהה של המחסנית.

ורגיסטר sp מצביע על הכתובת הנמוכה של המילה בה נשמר הנתון האחרון.



push 0CCDDh

הדחיפה הבאה הפחיתה את הכתובת שרגיסטר sp מצביע עליה ב -2

FE – 2 = FC

פקודת pop

פקודת pop היא הפקודה ההפוכה ל־push.

פקודה זו גורמת להוצאה של מילה (שני בתים) מהמחסנית והעתקה שלה לאופרנד היעד:

pop operand

פקודת pop מושכת מילה מראש המחסנית ומעתיקה ללאופרנד.

ערכו של sp יעלה ב־2.

דוגמאות לשימוש בפקודת pop:

pop ax

pop [var]

הפקודה הראשונה תעתיק לתוך ax את המילה שבראש המחסנית.

הפקודה השנייה תעתיק לתוך המשתנה את [var], את המילה שבראש המחסנית.

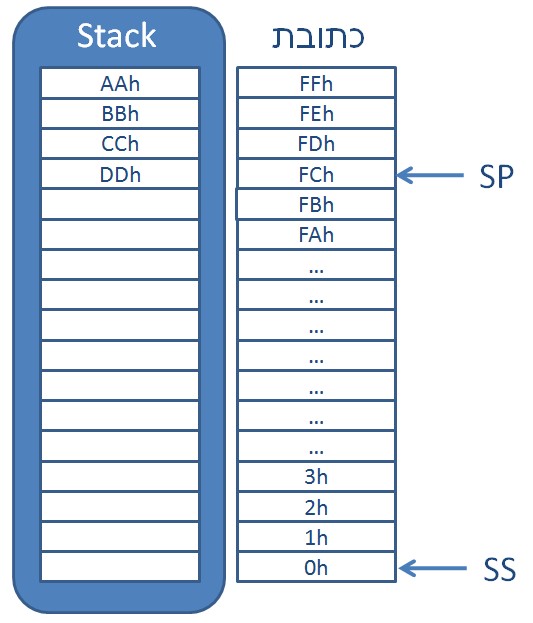
(כדי למנוע שגיאת קומפילציה ,var צריך להיות מטיפוס מילה).

* האופרנד חייב להיות בגודל מילה

pop al ;illegal

* אסור לאופרנד להיות קבוע

pop 5h ;illegal



פקודת POP – המשך

נתונה מחסנית. מה יקרה לאחר הפקודה הבאה?

pop bx

bx יקבל את הערך - 0CCDDh,

ערכו של sp יעלה בשתיים ויהיה שווה 0FEh

# פרוצדורות

תכנית מודולארית

מודולרית? ומדוע זה חשוב?

תכנית מודולרית היא תכנית שבנויה ממודולים – חלקים של קוד, שיש להם נקודת כניסה אחת ונקודת יציאה אחת, והם מבצעים פעולה מוגדרת.

מודול כזה נקרא פרוצדורה (Procedure) בשם העברי "תת תכנית" או פונקציה (Function).

בשפת אסמבלי אין הבדל מעשי בין פרוצדורה לפונקציה ולכן נתייחס לכול בתור פרוצדורות.

כשאנחנו כותבים קוד מודולרי – קוד שבנוי מפרוצדורות עם קוד שמקשר ביניהן – אנחנו מתכננים מראש איך לחלק את מה שהתוכנית שלנו צריכה לעשות לכמה פרוצדורות, כאשר לכל פרוצדורה יש תפקיד מוגדר.

**לעבודה בשיטה הזו יש כמה יתרונות:**

* קוד קצר יותר: (לא צריך לחזור על קטעי קוד.
* חלוקת הקוד לחלקים מקלה על מציאה ותיקן של באגים.
* קוד קראי יותר.
* יכולת לשף קוד (בין תכניות ובין תכנתים).

חסרונות:

* שיתוף קוד הוא פתח לבעיות
* כתיבת פרוצדורות דורשת השקעה מצד המתכנת.
* קריאה לפרוצדורה וחזרה דורשות משאבי מחשב.

פרוצדורה

פרוצדורה היא קטע קוד שיש לו כניסה אחת, יציאה אחת (רצוי), והוא מבצע פעולה מוגדרת.   
יש כמה רכיבים שהופכים קטע קוד לפרוצדורה:

* קוראים לפרוצדורה באמצעות הפקודה call.
* אפשר להעביר פרמטרים לפרוצדורה. לדוגמה, פרוצדורה שמחברת שני מספרים ומחזירה את סכומם – אפשר להעביר לה כפרמטרים שני משתנים – num1, num2.
* לפרוצדורה יש מנגנונים להחזרת תוצאות העיבוד. כלומר ,num1+num2 לא רק יחושב, אלא גם יועבר חזרה לתוכנית שזימנה את הפרוצדורה.
* פרוצדורה יכולה ליצור משתנים מקומיים (שמשמשים את הפרוצדורה בלבד) ולהיפטר מהם לפני החזרה לתוכנית הראשית .

פרוצדורה מגדירים או מיד בתחילת CODESEG, או בסופו של CODESEG. בפרוצדורה יהיה קטע קוד:

proc ProcedureName

… ;Code for something that the procedure does

ret ; Return to the code that called the procedure

endp ProcedureName

**פרוצדורה שמציירת תו למסך במיקום הסמן.**

IDEAL

MODEL small

Stack 100h

DATASEG

color dw 0Eh



CODESEG

proc drawCharacter

* mov ah, 9
* mov al, 2 mov bx, [color]
* mov cx, 1 int 10h
* ret

endp drawCharacter

start:

mov ax, @data

mov ds, ax

call drawCharacter

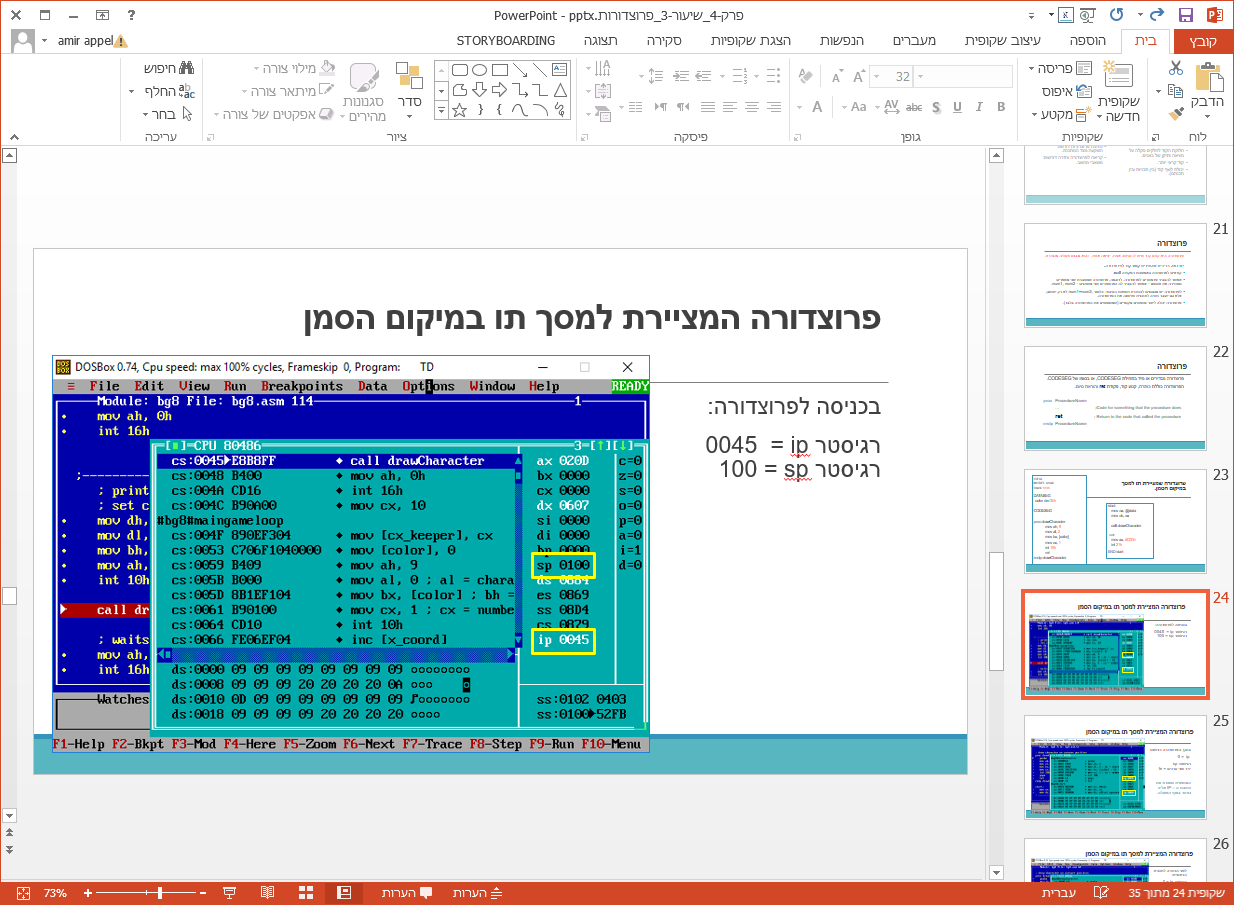
exit:

mov ax, 4C00h

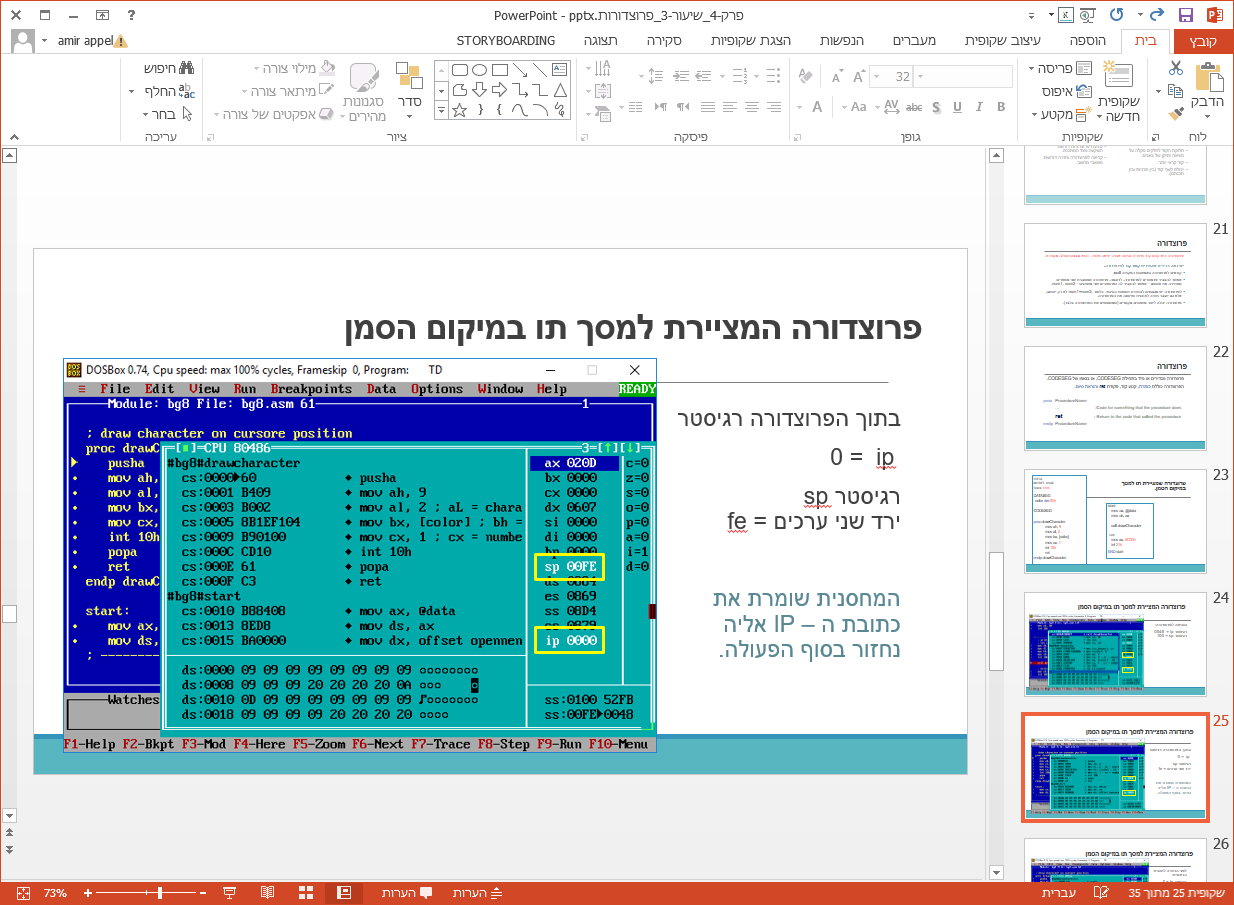
int 21h

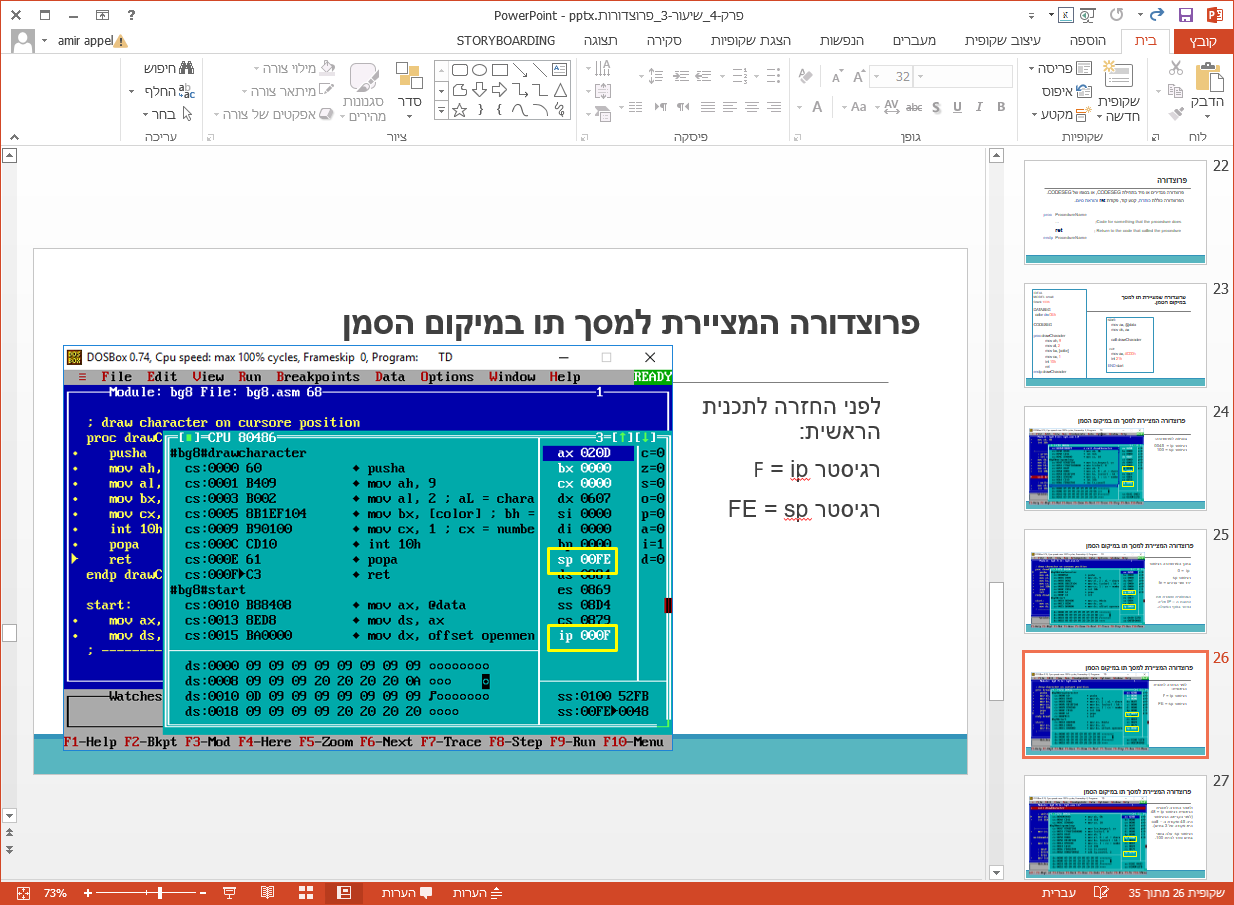
* END start

בכניסה לפרוצדורה רגיסטר ip = 45 רגיסטר sp = 100

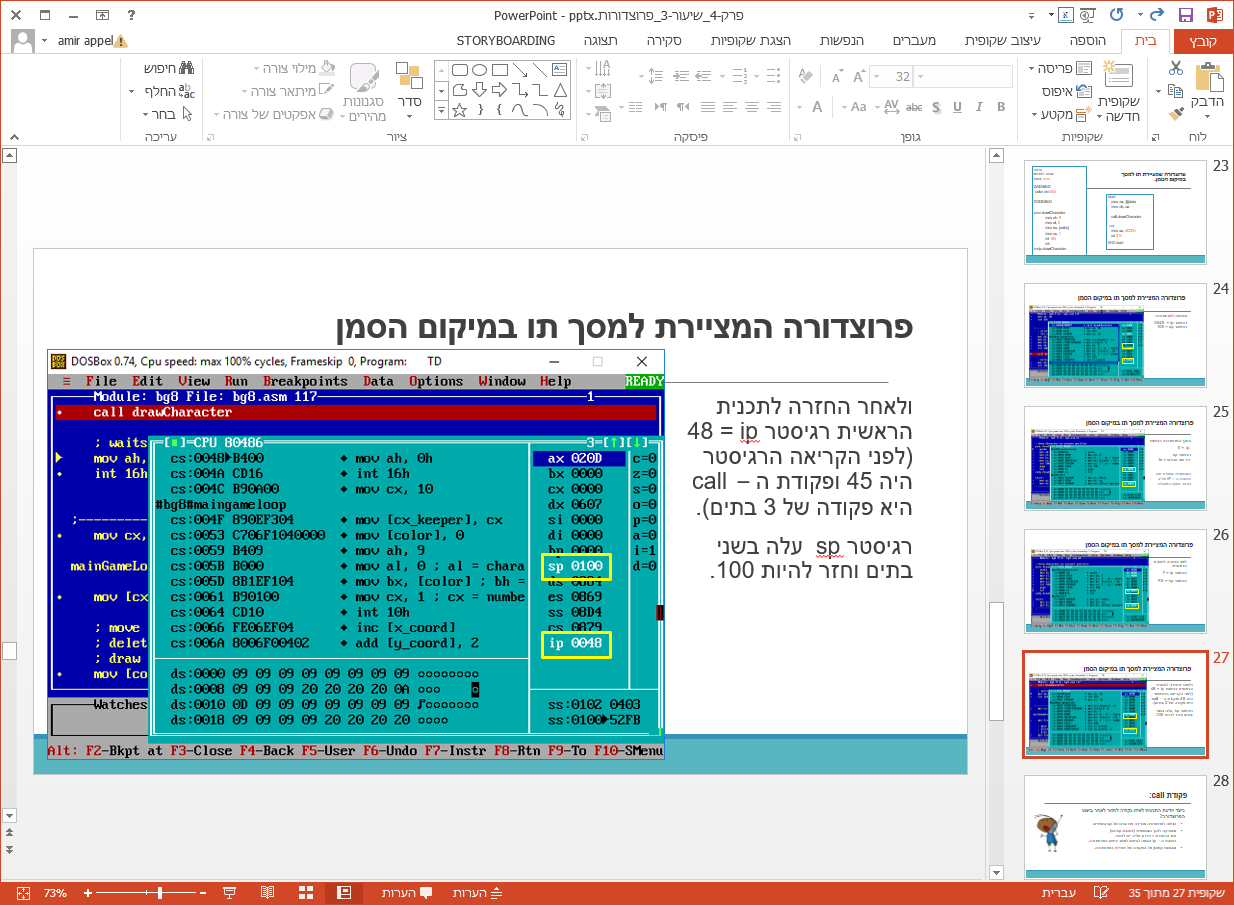


בתוך הפרוצדורה רגיסטר ip = 0 ורגיסטר sp ירד שני ערכים = fe



לפני החזרה לתכנית הראשית רגיסטר ip = F ורגיסטר sp = FE

ולאחר החזרה לתכנית הראשית רגיסטר ip = 48 (לפני הקריאה הרגיסטר היה 45 ופקודת ה – call היא פקודה של 3 בתים. ורגיסטר sp עלה בנשי בתים וחזר להיות 100.



פקודת call:

כיצד יודעת התכנית לאיזו נקודה לחזור לאחר ביצוע הפרוצדורה?

* כניסה לפרוצדורה מורידה את ערכו של sp בשתיים
* מעתיקה לתוך המחסנית (בכתובת ss:sp) את הכתובת בזיכרון אליה יש לחזור  
  כתובת ה – ip הבאה לביצוע לאחר ביצוע הפרוצדורה.
* מבצעת jump אל תחילת הפרוצדורה.

פקודת ret:

* מושכת מהמחסנית (הכתובת ss:sp) את הכתובת בזיכרון אליה יש לחזור.
* מעלה את ערכו של sp בשתיים.
* משנה את ה-ip אל הכתובת שנקראה מ-ss:sp, ובכך מחזירה את התוכנית לביצוע הפקודה שלאחר הפרוצדורה.

פרוצדורות near ו – far.

פרוצדורה יכולה להיות או באותו סגמנט עם הקוד שקורא לה, או בסגמנט אחר

* פרוצדורה near- באותו סגמנט
* פרוצדורה far- בסגמנט אחר

תוספת ההוראה far בכותרת הפרוצדורה

* למחסנית יידחף גם ה-ip וגם הסגמנט של כתובת החזרה
* ערכו של sp יירד ב-4 עם ביצוע call
* ערכו של sp יעלה ב-4 עם ביצוע ret

סביר שנשתמש רק בפרוצדורות near

proc ZeroMemory far

באופן מעשי ,כיוון שאנחנו מגדירים מודל זיכרון model small, בעל סגמנט קוד יחיד ,כל הפרוצדורות שנגדיר יהיו בתוך ה - CODESEGMENT.

# שימוש במחסנית לשמירת מצב התוכנית

לפנינו תכנית להדפסה של שלוש שורות, בכל שורה ארבעה תווי' X'.

בתוכנית פרוצדורה Print10X המדפיסה 4 X בשורה אחת.

proc Print10X

mov cx, 4 ; 4 'X' in each line

PrintXLoop:

mov dl, 'X' ; Print the value stored in dl ('X')

mov ah, 2h

int 21h

loop PrintXLoop

ret

endp Print10X

התכנית הראשית בה הוראות לירידת שורה וזימון לפרוצדורה.

start:

אם היינו מריצים את התכנית הזו היינו מקבלים לולאה אין סופי של ציור x למסך.

גם בפרוצדורה וגם בתוכנית הראשית יש שימוש ברגיסטר cx.

הסבר: בתחילת התוכנית cx מאותחל ל־3. בתוך הפרוצדורה ערכו משתנה ל־4. ביציאה מהפרוצדורה ערכו הוא 0, ואז הפקודה loop Row מפחיתה את ערכו באחד והופכת את ערכו ל-65,535 (כזכור, זהו הייצוג ה-unsigned של מינוס אחד) . כיוון שתנאי העצירה של לולאת ה־row לא מתקיים (cx אינו שווה לאפס) היא ממשיכה לרוץ ולקרוא שוב

לפרוצדורה, ששוב מחזירה את cx עם ערך 0 וכך הלאה...

mov ax, @data

mov ds, ax

mov cx, 3 ; 3 lines of 'X'

Row:

call Print10X

mov dl, 0ah ; New line

mov ah, 2h

int 21h

loop Row

exit:

mov ax, 4C00h

int 21h

END start

לכן, אנו זקוקים למנגנון שיאפשר לנו לשמור את מצב הרגיסטרים בתוכנית לפני הכניסה לפרוצדורה, ולשחזר את הערכים של הרגיסטרים (אם נרצה בכך) לפני החזרה לתוכנית.

proc Print10X

push cx

mov cx, 4 ; 4 'X' in each line

PrintXLoop:

mov dl, 'X' ; Print the value stored in dl ('X')

mov ah, 2h

int 21h

בעזרת פקודות push ו־pop אפשר לשנות את הפרוצדורה Print10X. בכניסה לפרוצדורה צריך לשמור את רגיסטרים cx במחסנית, וביציאה מהפרוצדורה לשלוף את הערך שלו חזרה ל –cx מהמחסנית ולהחזיר את הרגיסטר למצבו טרם הפרוצדורה

loop PrintXLoop

pop cx

ret

endp Print10X

**קיימת פקודה המבצעת פעולת push ו – pop לכל הרגיסטרים יחד**

* לפני מקטע הנתונים יש להקיש את הפקודה:
* **p186**
* בפרוצדורה:
* פקודת push all - pusha
* בתחילת הפעולה
* פקודת pop all - popa
* לפני פקודת ret

תרגול:

## הפסיקות בתכנית שלנו

|  |  |
| --- | --- |
| פקודה להדפסת מחרוזת  (שם משתנה המחרוזת msg) | mov dx, offset msg  mov ah, 9h  int 21h |
| פקודה לקליטת תו מהמקלדת | mov ah, 0h  int 16h |
| כניסה לתצוגה גרפית  25\*40 | mov ax, 13h  int 10h |
| יציאה מתצוגה גרפית  80\*25 | mov ax, 2h  int 10h |
| מיקום הסמן על המסך | mov dh, [y\_cord] ; row  mov dl, [x\_cord] ; column  mov bh, 0 ; page number  mov ah, 2  int 10h |
| ציור תו על המסך במיקום הסמן | mov ah, 9  mov al, 2 ; al = character to display  mov bx, [color] ; bh = Background bl = Foreground  mov cx, 1 ; cx = number of times to write character  int 10h |

INT 16h / AH = 00h - get keystroke from keyboard (no echo).

return:

AH = BIOS scan code.

AL = ASCII character.

(if a keystroke is present, it is removed from the keyboard buffer).

## bg8.asm

בתכנית שלנו יש הרבה פסיקות שחוזרות על עצמן, בקוד התכנית.

נוכל לקצר את התכנית ולסדר אותה בשימוש בפרוצדורות.

העלו את הקובץ שיצרנו בשיעור שעבר (עם הלולאה שבה cx משמש מונה הלולאה).

שמרו את הקובץ בשם חדש והמשיכו לעבוד בקובץ החדש.

## יצירת פרוצדורות

את הפרוצדורות כותבים בתחילת מקטע הקוד – CODESEG

|  |
| --- |
| IDEAL  MODEL small  STACK 100h  **p186**  DATASEG |

ולפני התווית של ה – start:

על מנת שנוכל להשתמש בפקודות pusha ו popa בפרויקט שלנו הוסיפו בתחילת

הקובץ p186

1. צרו פרוצדורה המדפיסה למסך מחרוזת במקטע הקוד לפני תווית ה – start:.

; print a string on the screen assuming the string offset is in dx

**pusha** - הפקודה המשמרת את הערכים של הרגיסטרים לפני הכניסה לפעולה

proc printString

pusha

mov ah, 9h

פקודות הפסיקה להדפסת המחרוזת למסך

int 21h ;interrupt that displays a string

popa

**popa** - הפקודה המחזירה את הערכים לרגיסטרים לפני הכניסה לפעולה

ret

**ret** – מחזירה אותנו למיקום הנכון בתכנית.

endp printString

**בתכנית הרשאית שנו את הפקודות,**

במקום הפסיקה המדפיסה למסך את המחזורת, השאירו את ההוראה המעבירה לרגיסטר dx, את מיקום תחילת המחרוזת

במקום שורות הקוד זמנו את הפעולה:

; print openning screen

mov dx, offset openScreen ;printing the opening screen string

call printString

שמרו, הריצו ובדוק שהקוד תקין.

1. שנו את ההוראות הדפסת המחרוזת השנייה למסך (המחרוזת עם שמכם)

אל תשכחו לפני הזימון לפעולת ההדפסה להעביר לרגיסטר dx את מיקום תחילת המחרוזת בסגמנט הנתונים.

1. השתמשו בפרוצדורה להדפסת המסגרת של המשחק
2. צרו פרוצדורה הממתינה למקש ושומרת אותו למשתנה.

ראשית ניצור משתנה שישמור את הערך של המקש שהוקש בסגמנט הנתונים:

chr db 0

נוסיף את הפרוצדורה למקטע הקוד:

; reads a character into chr

proc readChr

pusha

; waits for character

mov ah, 0h

int 16h

mov [chr], al ; save character to [chr]

popa

ret

endp readChr

בתכנית הראשית מחקו את הפסיקה של המתנה למקש

וזמנו את הפעולה במקום.

; waits for character

call readChr

שנו בכל המקומות במשחק שלנו במקום הפסיקה של קליטת תו זימון לפרוצדורה.

1. הוסיפו תנאי הבודק האם הקשנו על מקש q (quit) כדי לצאת מהמשחק.

cmp [chr], 'q'

je end\_game

שמרו, הריצו ובדוק שהקוד תקין.

1. כדי לצייר את השחקן, ראשית ממקמים את הסמן ולאחר מכן מציירים את השחקן במיקום הסמן.  
   צרו שתי פרוצדורות:

* מיקום הסמן - setCursorePosition
* ציור השחקן - drawCharacter

בתכנית הראשית זמנו את הפעולות במקום הפסיקות.

תזכורת תבנית יצירת הפעולה (פרוצדורה):

proc setCursorePosition

pusha

**…**

popa

ret

endp setCursorePosition

מחקו את הפסיקות של מיקום הסמן וציור השחקן בפעם הראשונה על לוח המשחק, וציירו את השחקן ע"י זימון הפעולות למיקום הסמן וציור השחקן.

שמרו, הריצו ובדוק שהקוד תקין.

1. בתוך הלולאה שנו את הפקודות ל"מחיקת השחקן" , הזזת הסמן וציור השחקן מחדש לפקודות המזמנות פעולות.

**הדרכה:**

שנו את צבע השחקן לשחור

השתמשו בפעולה drawCaracter במיקום בו נמצא הסמן כדי למחוק את השחקן.

שנו את הערכים של מיקום השחקן על ציר ה x וציר ה – y

שנו את הצבע של השחקן.

זמנו את הפעולה למיקום הסמן setCursorePosition

זמנו את הפעולה לציור מחדש של השחקן drawCaracter

שימו לב 🎔, מאחר ואנחנו בכל פרוצדורה "דוחפים" למחסנית את הערכים של הרגיסטרים ו"מושכים" אותם בסוף הפרוצדורה, אין צורך לשמור את הערך של רגיסטר cx - מונה הלולאה למשתנה.

מהו אורך התכנית?