

קודגורו אקסטרים

או שפת סף כמשחק



גרסת "גבהים"



תוכן עניינים

3.....	1. הקדמה
3.....	2. על התחרות
3.....	3. התקנת תוכנות עזר להרצת מנוע המשחק
9.....	4. התחלה מחויכת
12.....	5. חזרה לחוקי המשחק
13.....	6. מרוב עצים רואים יער
15.....	7. קצת תיאוריה
15.....	רגיסטרים
17.....	פקודות
19.....	המחסנית
21.....	8. תרגילי כספות
22.....	9. קריאת שורדים מתחרים
23.....	המפציץ
24.....	התותח
26.....	10. טקטיקות מתקדמות
29.....	11. תמיכה

1. הקדמה

קודגורו אקסטרים הוא מנוע משחק על בסיס מעבד 8086 של חברת אינטל®. חוברת זו מסבירה כיצד לשחק בדומה לדרך שבה ילדים לומדים לדבר – במקום ללמוד דקדוק, תחביר ולקרוא את כל המילון לפני אמירת המילה הראשונה, נתחיל בדוגמאות. איננו מניחים רקע מוקדם כלשהו, אם כי היכרות עם שפות תכנות יכולה לעזור.

2. על התחרות

תחרות "השורד הדיגיטלי" היא משחק המתרחש בזירה וירטואלית בין מספר תוכניות. כל תוכנית מייצגת מתחרה. התוכניות נכתבות בשפת אסמבלי 8086 סטנדרטי¹, נטענות כולן למרחב זיכרון משותף ("הזירה"), ומורצות במקביל על-ידי מנוע המשחק. מטרת כל מתחרה היא להיפטר מהמתחרים היריבים. התוכנית המנצחת היא זאת ששורדת אחרונה בזירה.

הפגיעה בתוכנית מתחרה נעשית על-ידי פגיעה בקוד שהתוכנית מריצה. אם אתם חושבים שתוכלו ותרצו להשתתף בתחרות, הירשמו לאתר בכתובת

<http://www.codeguru.co.il/Xtreme>

ותקבלו עדכונים לגבי התחרות.

3. התקנת תוכנות עזר להרצת מנוע המשחק

התקנת Java

מנוע המשחק נכתב בשפת Java, ולכן כדי להריץ את מנוע המשחק, נזדקק לתמיכה בשפת Java. כדי להתקין Java – או לבדוק אם הגרסה שלנו עדכנית – ניכנס לאתר

<http://www.java.com>



כדי לבדוק אם יש לנו גרסה עדכנית נקיש על "Do I have Java?"

¹ עם שינויים קלים שיתוארו בהמשך

יפתח המסך הבא:

Verify Java and Find Out-of-Date Versions

Check to ensure that you have the recommended version of Java installed on your Windows computer and identify any versions that are out of date and should be uninstalled.

Agree and Continue

By clicking Agree and Continue, you acknowledge that you have read and accepted the [license terms](#) for the verify and find old versions feature.



After clicking the button, a security prompt may appear. Click **Run** to allow the application to continue.

נבחר Agree and Continue.

אם קיימת גרסת Java עדכנית, נקבל את ההודעה הבאה:

Java Versions on Your Computer



Congratulations!

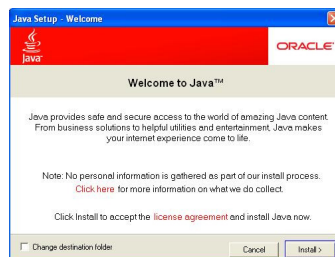
You have the recommended Java installed
Version 7 Update 67

No out-of-date versions of Java were found.

[Return to the Java.com home page](#)

אחרת, נחזור למסך הראשון ונחל בהתקנה.

לאחר שנוריד ונריץ את תוכנית ההתקנה של Java, ייטען המסך הראשון של תוכנית ההתקנה:



כדי להתחיל בהתקנה נלחץ על כפתור ה "Install". לפני כן מומלץ מאד לבחור באפשרות Change destination folder. בהמשך ההתקנה, כאשר נישאל על-ידי תוכנת ההתקנה לאיזו ספרייה להתקין, ניצור ספרייה בשם Java\c, אליה התוכנה תותקן. בסיום (מוצלח) של ההתקנה ייטען לפנינו המסך הבא:

Verify Java and Find Out-of-Date Versions

Check to ensure that you have the recommended version of Java installed on your Windows computer and identify any versions that are out of date and should be uninstalled.

Agree and Continue

By clicking Agree and Continue, you acknowledge that you have read and accepted the [license terms](#) for the verify and find old versions feature.



After clicking the button, a security prompt may appear. Click **Run** to allow the application to continue.

ולאחר שנבחר Agree and Continue אנחנו צפויים לקבל את ההודעה –

Java Versions on Your Computer



Congratulations!

You have the recommended Java installed
Version 7 Update 67

No out-of-date versions of Java were found.

[Return to the Java.com home page](#)

הערה: ייתכן שכדי לקבל את ההודעה הזו נצטרך לסגור את הדפדפן שלנו, לפתוח אותו מחדש ולגלוש שוב לאתר של Java.

הורדת מנוע המשחק

את המנוע של קודגורו אקסטרים נוכל להוריד מן האתר

<http://codeguru.co.il/Xtreme/tech1.htm>

כפי שניתן לראות בתמונה, נבחר להוריד את קובץ ה-Zip המכיל את קבצי מנוע המשחק:

corewars8086
CoreWars 8086 game engine, written in Java.

Project Home Downloads Wiki Issues Source

Search Current downloads for Search


★ Download: corewars8086 v3.0 (binary) - used in CodeGuru Extreme 2007-2009


Uploaded by: [dleshem](#)
Uploaded: Feb 21, 2009
Downloads: 622
Type: Executable
OpSys: All
Featured

corewars8086-3.0-bin.zip 62.2 KB

SHA1 Checksum: 1eebc44718cc92b513be2f468b52a6a9eeb123e
Tip: Use the SHA1 checksum shown to verify file integrity.

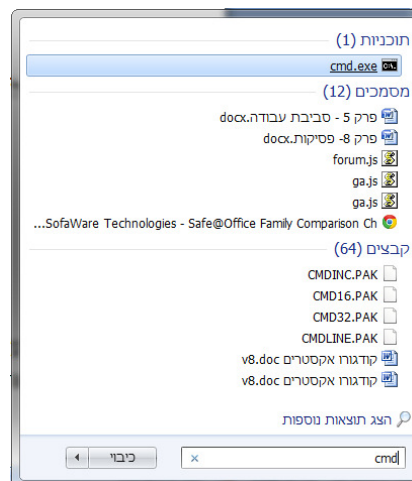
לחץ כאן להורדת קובץ ההתקנה של מנוע המשחק

 corewars8086-3.0-bin.zip	63 KB	WinRAR ZIP archive
--	-------	--------------------

 CoreWars8086

Name	Size	Type	Date Modified
survivors		File Folder	30/09/2006 01:15
zombies		File Folder	30/09/2006 01:15
cgx.bat	1 KB	MS-DOS Batch File	13/01/2006 20:09
corewars.jar	67 KB	Executable Jar File	30/09/2006 01:15

את מנוע המשחק מריצים דרך תוכנת cmd – מסך של מערכת ההפעלה DOS.
הריצו cmd על-ידי פתיחת Start > run > cmd, או בתפריט החיפוש של windows הקישו cmd ובחרו את התוכנה cmd, כמו בצילום המסך:



הסבר על הפקודות השונות של cmd ניתן למצוא בגוגל, או בספר האסמבלי של גבהים, בפרק על סביבת העבודה (www.cyber.org.il), הורידו את ספר האסמבלי).

בתוך cmd הריצו את הפקודות הבאות:

```
Set path=%path%;c:\java\bin
```

פקודה זו מאפשרת לנו להריץ תוכנות מספריית java גם אם אנחנו נמצאים בספרייה אחרת.

כדי להכניס את c:\java\bin באופן קבוע ל-path של המחשב, תוכלו להיעזר בסרטון ההדרכה שבקישור הבא (הסרטון מתייחס לתוכנה אחרת אולם הצעדים הם זהים):

https://www.youtube.com/watch?v=civ4hgoZynU&list=PLRvIHG-f_e8UJdv9O-hsxRZUzFkBBHpl8

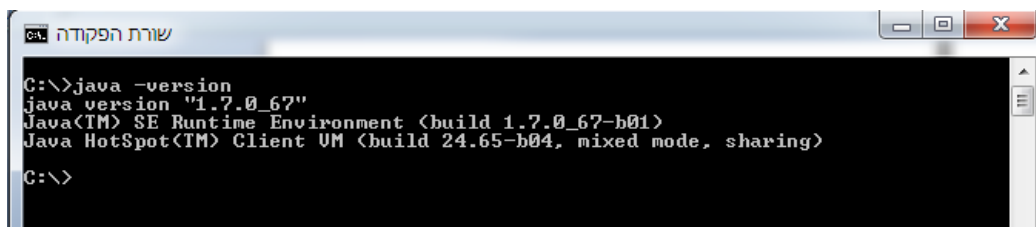
אם התקנו Java וסידרנו את ה-path, נוכל להפעיל את cgx פשוט על-ידי לחיצה כפולה על שם הקובץ cgx.bat שבספרייה.

קיימת אפשרות נוספת להפעלת cgx, שמומלץ להשתמש בה אם לחיצה כפולה על שם הקובץ איננה עובדת.

נפעיל את cmd ונכתוב:

```
Java -version
```

פקודה זו תאפשר לנו לוודא שאנחנו אכן יכולים להריץ תוכניות מספריית java. אם הכול תקין עד עכשיו נקבל:



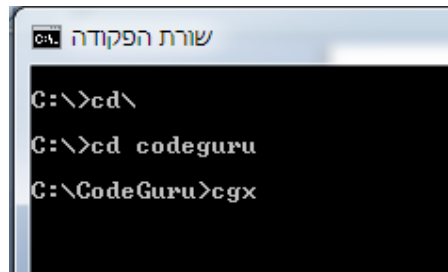
```
C:\>java -version
java version "1.7.0_67"
Java(TM) SE Runtime Environment (build 1.7.0_67-b01)
Java HotSpot(TM) Client VM (build 24.65-b04, mixed mode, sharing)
C:\>
```

לאחר מכן יהיה ניתן להריץ את מנוע המשחק ע"י גישה לספרייה codeguru:

```
Cd\
```

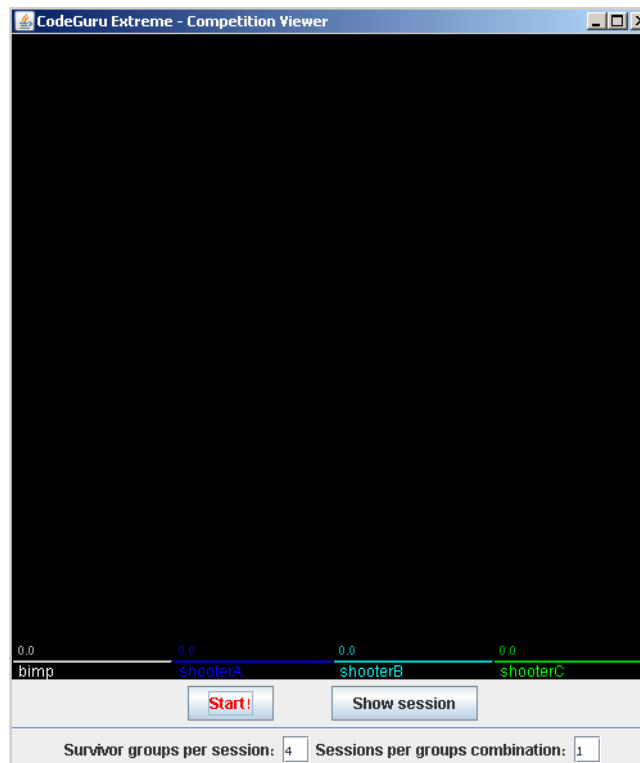
```
Cd codeguru
```

והפעלת התוכנית cgx.bat. יש לכתוב cgx וללחוץ enter:



```
C:\>cd\  
C:\>cd codeguru  
C:\CodeGuru>cgx
```

אם התקנו את הכול כשורה, נפתח מנוע המשחק:



בחלון הזה (ובחלון נוסף שאפשר לפתוח בשם Show Session שמראה לנו את ריצת התוכניות בזירה) קורה כל האקשן.

סביבת עבודה אסמבלי

התוכנות שלנו, שנקרא להן "שורדים", כתובות בשפת אסמבלי. סביבת העבודה של אסמבלי כוללת מספר תוכנות:

Editor – תוכנה שמאפשרת לנו לכתוב פקודות בשפת אסמבלי

Assembler + Linker – תוכנות שמאפשרות לנו לתרגם פקודות אסמבלי לשפת מכונה

Debugger – תוכנה שמאפשרת לנו הן לאתר שגיאות בקוד שלנו והן לתרגם שפת מכונה לפקודות אסמבלי. בהמשך נראה למה זה שימושי.

Emulator – תוכנה שמאפשרת לנו להריץ על המעבד החדש שיש לנו במחשב תוכנות אסמבלי שמיועדות לרוץ על מעבד "עתיק".

הסברים לגבי אוסף התוכנות, מאיפה מתקינים, איך מתקינים, איך מריצים – בספר האסמבלי של גבהים, בפרק על סביבת העבודה.

4. התחלה מחויכת

"יום ללא צחוק הוא יום מבוזבז" אמר פעם צ'רלי צ'פלין, ולכן נתחיל בכתיבת שורד פשוט שמצייר פרצופון מחייך.

התוכנית של השורד נראית כך:

```

IDEAL
model    tiny
CODESEG
org      100h
start:
    mov [2041h], al
    mov [2045h], al
    mov [2243h], al
    mov [2340h], al
    mov [2441h], al
    mov [2542h], ax
    mov [2444h], al
    mov [2345h], al
END      start

```

הפקודות שאתם רואים כאן כתובות בשפת [אסמבלי 8086](#).

את השורות המודגשות בצהוב, אין לשנות. ניתן לשנות רק את הפקודות שבין ה-start וה-end.

הסבר על השורות המודגשות:

1. IDEAL – מאפשר פקודות פשוטות של הצבה ישירה לזיכרון כמו `mov [2041h], al`.
2. Model tiny – קובע לאסמבלר שגודל סגמנט הקוד והנתונים יחד הוא 64K.
3. CODESEG – מציין לאסמבלר שמקטע הקוד מתחיל.
4. ORG – הקונבנציה בתוכניות COM היא שביצוע הפקודות מתחיל מההיסט 100h (בהתאם למקטע אליו נטענה התוכנית), לכן נדרשת ההגדרה ORG 100h בראש התוכנית, כדי לציין את ההיסט.
5. Start – תווית שאומרת לאסמבלר מה כתובת השורה הראשונה ב-CODESEG.
6. END start – קובע לאסמבלר שיש להפסיק את התרגום לשפת מכונה בשורה זו, ויש להתחיל את ההרצה מהתווית start.

נעתיק את הפקודות של השורד ונכתוב אותן ב-++notepad (או כל מעבד תמלילים/כתבן אחר):

```
1  IDEAL
2  model    tiny
3  CODESEG
4  org      100h
5  start:
6      mov [2041h], al
7      mov [2045h], al
8      mov [2243h], al
9      mov [2340h], al
10     mov [2441h], al
11     mov [2542h], ax
12     mov [2444h], al
13     mov [2345h], al
14  END start
```

ונשמור את הקובץ בתור smiley.asm. חשוב לזכור שבחלון ה"שמירה בשם.." נבחר בתיבה "סוג הקובץ" בערך "כל הקבצים *.*", וזאת כדי שהסיומת "txt" לא תתווסף לסיומת שכבר בחרנו (שהיא ".asm").
את הקובץ נשמור בספרייה

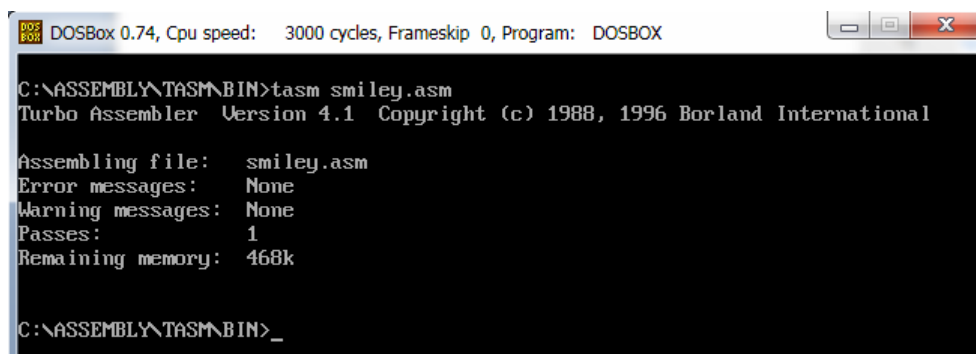
C:\tasm\bin

לאחר שכתבנו את התוכנית עלינו לתרגם אותה לשפה שהמחשב שלנו מבין – מכיוון שמחשבים (עדיין) אינם מבינים שפת אנוש, אנחנו צריכים לתרגם את הפקודות משפת אנוש לשפת מכונה.

ראשית נכתוב:

tasm smiley.asm

התוצאה תהיה:



```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX
C:\ASSEMBLY\TASM\BIN>tasm smiley.asm
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

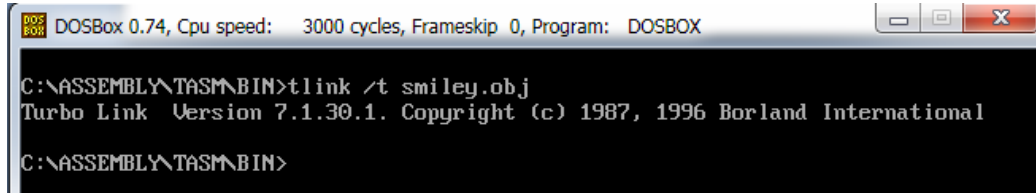
Assembling file:  smiley.asm
Error messages:   None
Warning messages: None
Passes:          1
Remaining memory: 468k

C:\ASSEMBLY\TASM\BIN>_
```

לאחר מכן ניצור קובץ הרצה מסוג com באמצעות הפקודות:

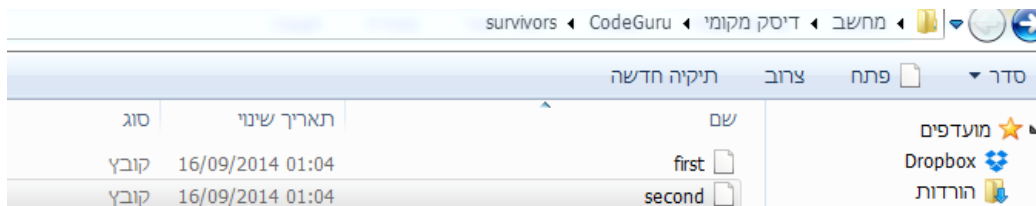
```
tlink /t smiley.obj
```

התוצאה תהיה:

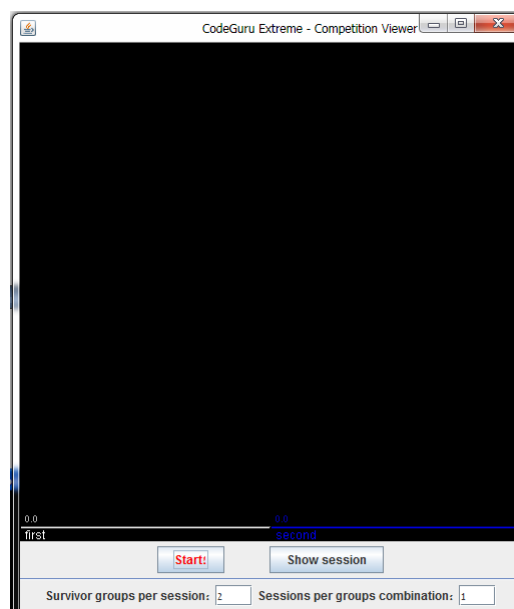


```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX  
C:\ASSEMBLY\TASM\BIN>tlink /t smiley.obj  
Turbo Link Version 7.1.30.1. Copyright (c) 1987, 1996 Borland International  
C:\ASSEMBLY\TASM\BIN>
```

כמו כן בספרייה שלנו נוצר קובץ בשם smiley.com. קובץ זה כולל פקודות בשפת מכונה. נעתיק את הקובץ הזה לתוך הספרייה c:\codeguru\survivors באמצעות לחיצה על הקובץ ניתן לשנות את שמו. נמחק את הסיומת com (נתעלם מהאזהרה של מערכת ההפעלה) ונשנה את שם הקובץ ל-first. נעתיק את הקובץ ונשמור אותו שנית, תחת השם second:



כעת נחזור לתוכנת ה-cmd. נריץ את קודגורו אקסטרים על ידי הקשת cgx ולאחר מכן Enter. ייפתח המסך הבא:

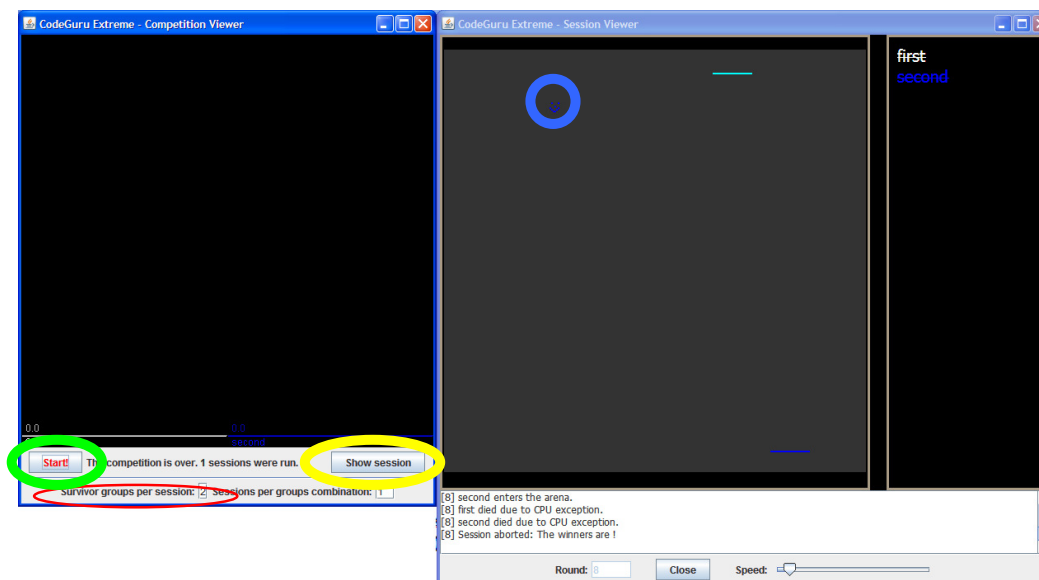


1. כווננו את מספר השורדים (survivors group per session) ל-2 במקום 4, אם יש צורך בכך. ראו אליפסה אדומה באיור להלן;

2. בקשו להראות את הזירה (Show session) – ראו אליפסה צהובה באיור להלן;

3. והריצו (Start) – ראו אליפסה ירוקה באיור להלן.

חדי העין יוכלו להבחין מייד בפרצופון מחייך – המוקף באיור באליפסה כחולה:



כל הכבוד – כרגע הרצתם את תוכניתכם הראשונה!

5. חזרה לחוקי המשחק

בעצם מה שעשינו בפרק הקודם היה לכתוב תוכנית פשוטה בשפת סף (אסמבלי). מנוע המשחק צובע נקודות (פיקסלים) על המסך בהתאם לפקודות התוכנית. בתוכנית החייך שלעיל, כל פקודה מציירת נקודה (פרט לאחת הפקודות שמציירת שתי נקודות, נגיע לכך בהמשך). מנוע המשחק מריץ מספר תוכניות במקביל. כדי לפשט את הדוגמא הרצנו את ה"שורד" (תוכנת החייך) עם תוכנה נוספת שלא עושה דבר.

אפשר לראות כיצד הופיע החייך (סמיילי):

	40	41	42	43	44	45	46
20	2041					2045	
21							
22				2243			
23	2340					2345	
24		2441			2444		
25			2542	2542			

לוח המשחק מכיל 256 שורות (00 עד FF בבסיס הקסה־דצימלי), ובכל שורה 256 פיקסלים. בסך־הכול יש 65,536 נקודות על המסך, לכל אחת מהן יש כתובת של ארבע ספרות הקסה־דצימליות.

ניתן להתייחס לפי נתונים אלו אל הכתובות בלוח המשחק בצורה הבאה:

	00	01	FE	FF
00	0000	0001	00FE	00FF
01	0100	0101	01FE	01FF
:	:	:			:	:
:	:	:			:	:
FE	FE00	FF01	FEFE	FEFF
FF	FF00	FF01	FFFE	FFFF

עכשיו ציירו על נייר משבצות תמונה אחרת (עץ, בית, שמש, השם שלכם, קבוצת כדורגל אהובה – כל העולה על דעתכם) וכתבו את התוכנית המתאימה שתצייר את אותו ציור על המסך.

6. מרוב עצים רואים יער

חייקן בודד קל יותר לצייר ידנית. אבל הכוח של שפת תכנות הוא האפשרות לשכפל.

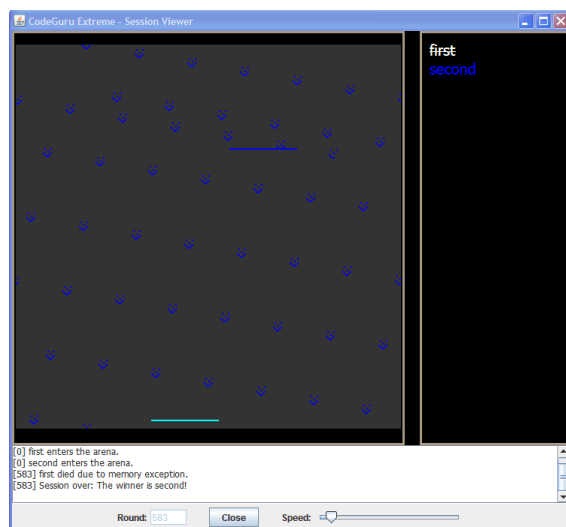
ראו כיצד מתמלא הלוח כולו בחייכנים על ידי לולאה:

```

mov     cx, 100
l:
    call smile
    add     bx, 623h
    loop    l
smile:
    mov     [bx+2041h], al
    mov     [bx+2045h], al
    mov     [bx+2243h], al
    mov     [bx+2340h], al
    mov     [bx+2441h], al
    mov     [bx+2542h], ax
    mov     [bx+2444h], al
    mov     [bx+2345h], al
    ret

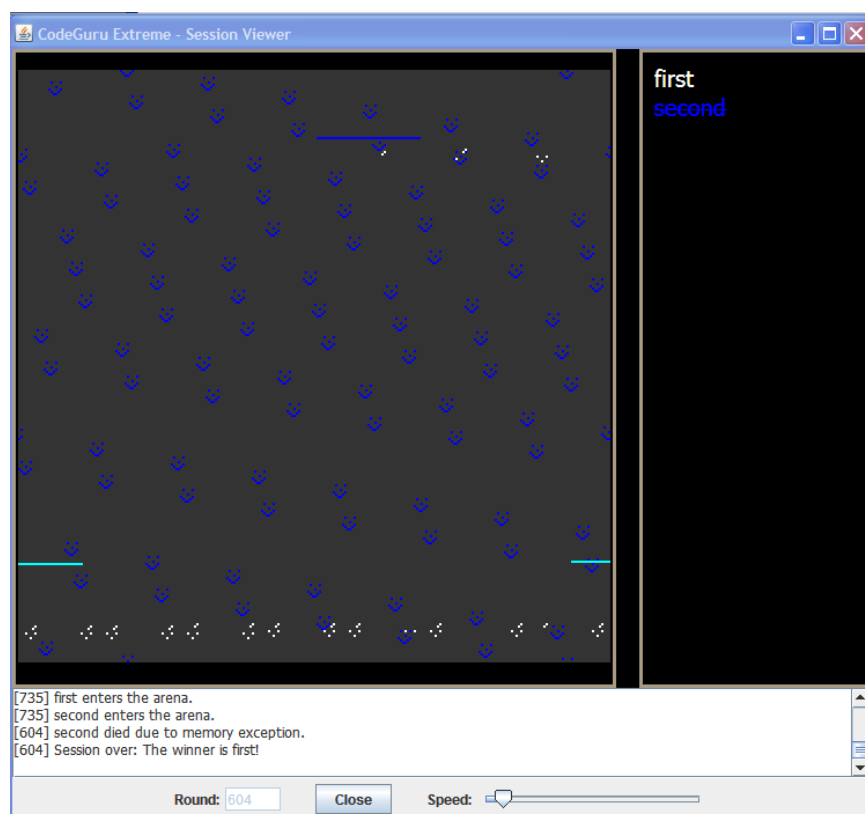
```

והרי לנו יער של חייכנים:



שחקו עם הפרמטרים, וראו מה יוצא.

שימו לב שמדי פעם נופלות שגיאות שגורמות לכל מיני צורות שונות ומשונות להופיע. למשל:



עוד נגיע לכך בהמשך; בינתיים רק דעו שאם לא קיבלתם את התוצאה המבוקשת, אתם יכולים לנסות שוב.

7. קצת תיאוריה

מומלץ לקרוא את ספר הלימוד "ארגון המחשב ושפת סף" של גבהים, שם תמצאו דיון בהרחבה אודות פקודות האסמבלי. בסעיף זה נתוודע לפקודות האסמבלי בקצרה.

רגיסטרים

ועכשיו, כמובטח מפרק אי-זוגי, ניתן קצת הסבר מסודר מה בעצם אנחנו עושים. המרכיב הבסיסי של המידע במחשב הוא **סיבית** (bit בלעז). זוהי **ספרה בינארית** (יכולה לקבל שני ערכים של אפס או אחד; הסיבה לכך מגיעה מתחום החשמל, אך לא נרחיב עליה כעת). שמונה סיביות מרכיבים בית (byte), דוגמאות להמחשה:

- [11111111]
- [10101011]
- [10001001]
- [10001]

ניתן לראות כי כמעט כל הדוגמאות מכילות שמונה סיביות כל אחת, הסיביות הללו יכולות להיות כביות (כלומר להכיל את הספרה 0) או דולקות (להכיל את הספרה 1), הדוגמה האחרונה לא מכילה 8 סיביות, אך עדיין יכולה לייצג בית, כיוון ששלוש הסיביות האחרונות החסרות, אשר משלימות את הדוגמה לבית (8 סיביות), מכילות אפסים, ולכן אין לכתוב אותם. למעשה הדוגמה האחרונה מקבילה לדוגמה הזו:

- [00010001] = [10001]

חשוב מאוד לזכור זאת כדי למנוע בלבול מיותר.

מרחב הזיכרון של קודגורו אקסטרים הוא 64 קילובייט (64 כפול 1024, שהם 65,536 בתים) המיוצג על המסך כ-65,536 פיקסלים (נקודות) המסודרים בריבוע של 256 על 256.

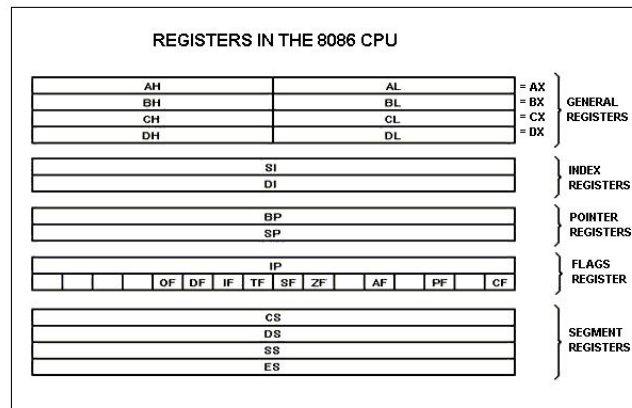
בנוסף, שיטה הספירה שנשתמש בה עלולה להיות שונה משיטת הספירה המוכרת לך. עד היום נהגנו לספור **בשיטה העשרונית**, הקרויה גם ספירה בבסיס 10 (האם תוכלו לשער למה?). בתכנות (ולאו דווקא באסמבלי) נהוג לספור בשיטה ההקסה-דצימלית. שיטה זו נקראת גם ספירה בבסיס 16, מכיוון שלהבדיל מהשיטה העשרונית, לאחר הסיפרה 9 תגיע הספרה a, ולאחר מכן b, c, d, e, f. ניתן לקרוא על השיטה הזו בקישור הבא:

<http://he.wikipedia.org/wiki/הקסדצימלי>

שפת המכונה עצמה כאמור מורכבת מאופ־קודים (באנגלית: opcodes), שהם תרגום של תוכנית האסמבלי שלנו בשפה שהמחשב יבין. כל פקודה שכתבנו בתוכנית שלנו מתורגמת לפקודה שהמעבד יבין.

פקודות אלו ייטענו לזיכרון וירוצו אחת אחרי השנייה עד שהתוכנית תופסק מסיבות כלשהן (לדוגמה, סיום התוכנית).

במעבד 8086 יש מספר רגיסטרים (בלעז רגיסטרים):



1. הרגיסטרים הכלליים:

- ax – רגיסטר כללי לחישובים (פקודות הכפל, למשל, עובדת רק עליו)
- bx – הרגיסטר הכללי היחיד היכול לשמש כהזחה (offset) לכתובות
- cx – רגיסטר כללי בעל פקודות מיוחדות ללולאות
- dx – רגיסטר כללי היכול להתחבר לרגיסטר ax לקבלת ערך של 32 ביט

2. רגיסטרי מחרוזות:

- si – משמש כ־Source Index לפקודות הזזה
- di – משמש כ־Destination Index לפקודת הזזה

3. רגיסטרי מחסנית:

- bp – Base Pointer למחסנית ועוד
- sp – Stack Pointer, מצביע למחסנית

4. רגיסטר הפקודות והדגלים:

- ip – Instruction Pointer, מצביע לפקודה המתבצעת ברגע זה
- flags – תשעה ביטי דגל (נשא, זוגיות, נשא משני, אפס, סימן, דיבאג, פסיקה, כיוון וגלישה)

5. רגיסטרי מקטע:

- cs – רגיסטר מקטע המצביע לקוד
- ds – רגיסטר מקטע המצביע למידע
- es – רגיסטר מקטע נוסף המצביע למידע
- ss – רגיסטר מקטע המצביע למחסנית

גודלם של רגיסטרים אלו הוא 16 סיביות, או 2 בתים, והם משמשים לאחסון מידע פרטי של התוכנית שלנו.

חשוב להדגיש כי המידע הזה אינו משותף לשאר השורדים ואינו ניתן לשינוי על ידם, להבדיל מאזור הזיכרון המשותף שהשורדים נמצאים בו, אשר נגיש לכולם. רגיסטרים אלו ישמשו אותנו בעת ריצת התוכנית.

לארבעת הרגיסטרים הראשונים (לשימוש כללי) ניתן להתייחס כאל רגיסטרים בגודל של 8 סיביות, או בית אחד.

לדוגמה, לשמונה הסיביות הראשונות (הנקראות גם הסיביות הנמוכות) שברגיסטר ax, נתייחס כאל ah, ואל שמונה הסיביות שיבואו אחריהן (הנקראות הסיביות הגבוהות) נתייחס כאל ah. אותו הדבר חל על שאר הרגיסטרים, כלומר אפשר להתייחס ל-bx כאל bh/bl, לרגיסטר cx כאל cl/ch, ולרגיסטר dx כאל dl/dh.

חשוב להבין כי אם נכתוב את הספירה 0fh לתוך הרגיסטר al, אזי תוכן הרגיסטר ax ישתנה כך ששמונה הסיביות התחתונות שלו יכילו את הספירה 0fh.

פקודות

פקודות שהשתמשנו בהן בתוכנית:

label

Label, טוב, זו אינה ממש פקודה, אלא הגדרת אזור בקוד (סימנייה). נגדיר אותו בעזרת מילה שנבחר (במקרה שלנו, smiley או l) ונצמיד לה את התו ":" (כמובן, ללא המרכאות), נשתמש ב-label זה בהמשך.

mov

הפקודה mov (קיצור של המילה האנגלית move) תשמש אותנו כאשר נרצה להעתיק ערך מסוים ממקום א' למקום ב'. מקום ב' יכול להיות רגיסטר או כתובת זיכרון, ומקום א' יכול להיות גם הוא רגיסטר או כתובת זיכרון, אבל יכול גם להיות ערך קבוע (ראו שורה ראשונה בתוכנית). כדי לציין שאנחנו משתמשים בכתובת זיכרון ולא בערך קבוע, נקיף את הכתובת בסוגריים מרובעים (ראו שימוש ב-mov בתוך אזור ה-smiley)

call

תפקיד הפקודה call הוא לקפוץ אל ה-label שהגדרנו, והיא מקבלת כפרמטר את שם ה-label (במקרה שלנו smiley). כדי לחזור מתוך label שנקרא על ידי call, נשתמש בפקודה ret.

add

תפקיד הפקודה add הוא לבצע חיבור של ערך מסוים עם ערך אחר. במקרה שלנו, הערך 623h (זכרים מה מייצג h? סימון הקסה-דימלי) יתווסף לערך שנמצא ברגיסטר ax.

loop

הפקודה loop מקבלת כפרמטר את ה-label אליו נקפוץ בעת ביצוע הלולאה. בנוסף, מונה הלולאה (מספר הפעמים שהיא תבוצע) נשמר ב-cx, ובמקרה שלנו ערכו הוא 100.

בטבלה שלהלן תמצאו רשימה מלאה של כל פקודות המעבד 8086. לכל אחת מהפקודות בכול קיים קישור כדוגמת http://home.comcast.net/~fbui/intel_m.html#mul לקבלת תיאור.

STC	CMPSW	JAE	JNC	JZ	MUL	REPNZ	STD
MOVSW	CWD	JB	JNE	LAHF	NEG	REPZ	STI
JS	JNBE	JBE	JNG	LDS	NOP	RET	STOSB
CMPSB	JA	JC	JNGE	LEA	NOT	RETF	STOSW
ADC	DEC	JCXZ	JNL	LES	OR	ROL	SUB
ADD	DIV	JE	JNLE	LODSB	SHR	ROR	TEST
AND	CMP	JG	JNO	LODSW	POP	MOVS	XCHG
CALL	JNB	JGE	JNP	LOOP	POPF	JPO	XLAT
CBW	IRET	JL	JNS	LOOPE	PUSH	SAR	XOR
CLC	JPE	JLE	JNZ	LOOPNE	PUSHF	SBB	SHL
CLD	INC	JMP	JO	LOOPNZ	RCL	SCASB	REP
CLI	* INT	JNA	JP	LOOPZ	RCR	SCASW	MOV
CMC	JNAE						

*הפקודה INT מקבלת פרמטרים מסוימים שיצוינו תחת הקטגוריה "טכניקות מתקדמות".

הסיבה שביעיר החייכנים בחרנו ברגיסטר ax היא שהוא הרגיסטר הכללי היחיד שמסוגל למען כתובת בצורה שרצינו. אפשר להעביר קבוע לרגיסטר; אפשר להעביר רגיסטר לכתובת קבועה בזיכרון; אפשר להעביר קבוע לכתובת שרגיסטר מצביע עליה; אבל אי אפשר להעביר קבוע לכתובת קבועה בזיכרון.

כדי להבין טוב יותר מה אפשר ומה לא, להלן טבלה של כל צורות המיעון האפשריות.

8086 ADDRESS MODES

TYPE	INSTRUCTION	SOURCE	ADDRESS GENERATION	DESTINATION
1) REGISTER	MOV AX, BX	REGISTER BX		REGISTER AX
2) IMMEDIATE	MOV CH, 3AH	DATA 3AH		REGISTER CH
3) DIRECT	MOV [1234], AX	REGISTER AX	(DS x 10H) + DISPLACEMENT 10000H + 1234	MEMORY 11234H
4) REGISTER INDIRECT	MOV [BX], CL	REGISTER CL	(DS x 10H) + BX 10000H + 0300H	MEMORY 10300H
5) BASE PLUS INDEX	MOV [BX + SI], BP	REGISTER BP	(DS x 10H) + BX + SI 10000H + 0300H + 0200H	MEMORY 10500H
6) REGISTER RELATIVE	MOV CL, [BX + 4]	MEMORY 10304H	(DS x 10H) + BX + 4 10000H + 0300H + 4	REGISTER CL
7) BASE RELATIVE PLUS INDEX	MOV ARRAY [BX + SI], DX	REGISTER DX	(DS x 10H) + ARRAY + BX + SI 10000H + 1000H + 0300H + 0200H	MEMORY 11500H

ASSUME BX = 0300H, SI = 0200H, ARRAY = 1000H, DS = 1000H

המחסנית

המחסנית היא אזור זיכרון פרטי (בצורת ברירת המחדל) המשמש לאחסון נתונים החיוניים לריצת התוכנית שלנו.

המחסנית היא מבנה נתונים אשר עובד צורת Last-In-First-Out (בקיצור LIFO), ואפשר להבין זאת בעזרת מחשבה על מחסנית אמיתית, כאשר הכדור הראשון אשר יידחק למחסנית יהיה הכדור האחרון שיצא ממנה, לעומת הכדור האחרון שיידחק למחסנית ויהיה הכדור הראשון שיצא ממנה.

המחסנית תאחסן את הנתונים אשר נשמרו בה בזיכרון המחשב, וכמו שלמדנו עד כה, אנחנו נתבונן בזיכרון המחשב בצורת מיעון שונה במקצת ממה שהכרנו עד כה – אנחנו נעבוד בצורת segment:offset, כלומר, יהיה מקטע (segment) למחסנית, והיסט (offset) בו המחסנית תתחיל.

סגמנט המחסנית נשמר ברגיסטר ss (קיצור ל-stack segment), חשוב לציין כי מנוע המשחק של קודגורו אקסטרים יאתחל לנו את הרגיסטר הזה לסגמנט פרטי אשר שורדים אחרים (שאינם מקבוצתנו, ועל קבוצות נדבר בהמשך) לא יוכלו לכתוב.

נקודת ההתחלה של המחסנית בתוך הסגמנט תקבע על ידי רגיסטר ההיסט, שם הרגיסטר הזה הוא bp (קיצור ל-base pointer), גם רגיסטר זה יאותחל אוטומטית על ידי מנוע המשחק.

בשפת אסמבלי, כדי לדחוף נתונים למחסנים נשתמש בפקודה `push`. האופרנדים של פקודה זו יכולים להיות קבוע (לדוגמה, `ABCDh`) רגיסטר (כולם, לדוגמה, `ax`) וערך של כתובת. חשוב לציין שגודל הערך חייב להיות `word` (כלומר, 16 סיביות), לפעמים האסמבלר יחייב אותנו להוסיף את מאפיין `word` לפני האופרנד, לדוגמה:

```
push ax
push word [bx]
```

בפקודה הראשונה הקומפיילר יודע שמדובר במילה אך בפקודה השנייה הוא צריך את עזרתנו כדי להבין האם נרצה לדחוף למחסנית מילה או בית בודד; אחרת ניתקל בשגיאת קימפול.

כדי לחלץ נתונים מהמחסנית נשתמש בפקודה `pop`. האופרנדים המתאימים לפקודה זו יכולים להיות רגיסטר או כתובת.

פקודה זו תמחק מהמחסנית את הערך האחרון שנדחף אליה ותכתוב אותו ברגיסטר או הכתובת שנבחרה.

כמובן שלאחר הרצת פקודה זו, הערך הקודם שהיה ברגיסטר או בכתובת שנבחרה, יימחק.

דוגמה כללית לאפשרויות שימוש במחסנית.

```
mov ax, 01100h
mov word [0ABCDh], 0ABCDh

push ax           ; pushing the value 01100h into the stack.
push word [0ABCDh] ; pushing the value 0ABCDh into the stack.
push 0DEADh      ; pushing the value 0DEADh into the stack.

pop word [0ABCDh] ; address 0ABCDh value is now 0DEADh.
pop ax           ; register ax value is now 0ABCDh.
pop cx           ; register cx value is now 01100h.
```

שימו לב שהדוגמא המחוקה בשורה השישית איננה חוקית במעבד 8086 (היא הוכנסה לשימוש רק במעבדי 80186 ואילך) ולכן איננה חוקית בקודגורו אקסטרים.

כאשר נדחוף למחסנית ערך בן 16 סיביות, היא תתחיל בכתובת `[ss:bp]` ותיגמר בכתובת `[ss:bp+2]`, או במילים אחרות `[ss:sp]`, הרגיסטר `sp` (קיצור ל-`stack pointer`) יצביע תמיד לראש המחסנית. חיסור של `sp` מ-`bp` ייתן לנו את גודל המחסנית הנתונה. לצורך הדגמה:

```
push ax
push ax
; sp = bp + 4.

pop ax
pop ax
; sp = bp.
```

8. תרגילי כספות

אחר שציירנו קצת, נעבור לתרגילי כספות.

מקור השם הוא בתחרות הכספות של מכון ויצמן – בתחרות זו כל צוות מתכנן כספת המתבססת על עקרון פיזיקלי שמאפשר לפרוץ אותה (קל ולא מעניין לתכנן כספת שקשה לפתוח).

הבעיה המעניינת היא כיצד לתכנן כספת שניתן לפרוץ במהירות אך קשה לגלות זאת.

מידע על התחרות ניתן למצוא בקישור הבא:

<http://www.weizmann.ac.il/zemed/activities.php?cat=0&id=571&act=large>

נתחיל בדוגמא – כיצד מנצחים את השורד הבא:

```
loop:
  mov     ax, [1234h]
  cmp     ax, 5678h
  jnz     loop
```

שורד זה קורא את תא הזיכרון 1234h ומשווה (CoMPare) את ערכו ל-5678h; אם הוא אינו שווה (Jump Not Zero) – הוא קופץ לתחילת התוכנית בלולאה אינסופית.

ברגע שבכתובת הזיכרון 1234h יופיע הערך 5678h, השורד בעצם "יתאבד" (ימשיך לבצע את הפקודה הבאה שלא כתבנו) – ומנוע המשחק מאתחל את כל הזיכרון לפקודות לא חוקיות.

ומה עם השורד הבא:

```
loop:
  mov     ax, [1234h]
  add     ax, 6789h
  cmp     ax, 5678h
  jnz     loop
```

כאמור – אין בעיה לכתוב שורד שקשה לפגוע בו, למשל:

```
loop:
  jmp     loop
```

התוכלו לכתוב שורד דומה שאפשר לנצח? הביאו את השורדים שכתבתם לשיעור הבא ונבחן אותם יחדיו.

9. קריאת שורדים מתחרים

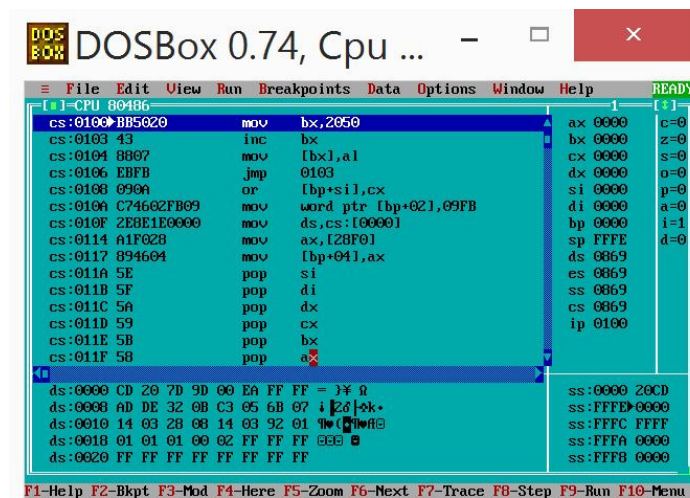
בתחרות עצמה לא תקבלו את קוד המקור של השורדים המתחרים, אלא את הגרסה המקומפלת שרצה במנוע. כדי להבין מול מה אתם מתמודדים עליכם לבצע תהליך של הנדסה הפוכה (Reverse engineering) של השורדים. אחת הדרכים לעשות זאת היא שימוש בפקודה td, אשר מריצה את turbo debugger. לפני ההרצה יש לשנות את שם הקובץ על-ידי הוספת הסימנת com. יש להעתיק את הקובץ com אל ספריית tasm/bin ואז ניתן להפעיל את הדיבאגר באמצעות הפקודה td. לדוגמה:

td soredA

להלן הקוד של soredA – פעם בתור קוד מקורי ב-++notepad ופעם בתור תוצר של תהליך RE בדיבאגר:

```
1  IDEAL
2  model tiny
3  CODESEG
4  org 100h
5  start:
6      mov bx, 2050h
7  myloop:
8      inc bx
9      mov [bx], al
10     jmp myloop
11  END start
```

קוד מקור



קוד לאחר ביצוע RE בעזרת תוכנת Turbo Debugger

אפשר ורצוי להריץ את השורדים (גם שורדים שכתבתם) ב־TD לפני הרצתם במנוע המשחק. חשוב להבין מה הם הבדלים בין המנוע ל־TD:

1. רגיסטרים

- א. במנוע ax מכיל את כתובת הטעינה – ושאר הרגיסטרים מאופסים.
- ב. ב־cx debug מכיל את אורך השורד – ושאר הרגיסטרים מאופסים.

2. רגיסטרי מקטע

- א. במנוע ds ו־cs מצביעים על הזירה; es מצביע על המקטע המשותף, ו־ss מצביע על המחסנית האישית.
- ב. ב־debug כל הרגיסטרים מצביעים לאותה הכתובת.

3. פקודות

- א. המנוע מריץ רק את הפקודות החוקיות (8086), לא כולל פסיקות וגישה מחוץ לזירה).
- ב. ב־debug אפשר לראות (Unassemble) רק פקודות 8086, אבל הרצה (sStep, Go) תריץ גם פקודות אחרות בלי להתלונן.

ועכשיו נחקור שני שורדים לדוגמא.

המפציץ



מפציץ הוא מטוס קרב המטיל פצצות על מטרות קרקע (נעשה בהם שימוש בעיקר במלחמת העולם השנייה).

במקרה שלנו זהו שורד שמפציץ את זירת המשחק בבית (byte) עם ערך 0xCC החל מתחילת הזיכרון (כתובת 0000).

מכיוון שהשורד מתחיל לכתוב בכתובת אפס, והוא נטען בכתובת מקרית, הרי שבסיכוי סביר הוא ייפגע בעצמו לפני שיפגע באחרים. מה יקרה במקרה זה?

מדוע בחרנו להפציץ בערך 0xCC?

בחרנו ב־byte של 0xCC מאחר והוא מסמל אופ־קוד "רע" (INT3), ולכן אם נצליח לפגוע בקוד של שורד אחר בנקודה מתאימה, כנראה שנצליח להרוג אותו.

בקוד גורו אקסטרים אין "אי המתים" – כאן אם מועצת השבט אמרה את דברה זה סופי ומוחלט... (לפחות עד הסיבוב הבא...)

נבחן את קוד האסמבלי של שורד זה:

```
; 1. initialization
    mov  al, 0CCh
    mov  bx, 0

; 2. bombing loop
bomb:
    mov  [bx], al
    inc  bx
    jmp  bomb
```

חלק האתחול מורכב משתי פקודות:

`mov al, 0CCh` – אנו מכניסים לרגיסטר `al` את המידע בו נרצה להפציץ ובמקרה זה האופי-קוד הזדוני שלנו `0xCCh`. בכל פעם שנבצע הפצצה נשתמש בתוכנו של רגיסטר זה.
`mov bx, 0` – אנו מכניסים לרגיסטר `bx` את הכתובת ההתחלתית שלנו, ובמקרה זה `0000`.
בכל פעם שנבצע הפצצה אנו נקדם את ערכו של רגיסטר הכתובת `bx`, כדי להתקדם לתא הבא בזירה.

חלק הלולאה מורכב מהפקודות הבאות:

`bomb:` – פקודת `label` שמציינת שכאן מתחילה הלולאה. בכל חזרה על הלולאה נקפוץ למיקום זה ונבצע את הפקודות שאחריה.
`mov [bx], al` – אנו מעבירים את ערכו של `al` לכתובת ש-`bx` מצביע עליה.
`inc bx` – אנו מקדמים את `bx` לכתובת הבאה וזאת כדי שנוכל להפציץ מקום אחר בלוח.
`jmp bomb` – אנו מבצעים קפיצה ל-`label` בשם `bomb` כדי לבצע חזרה נוספת על הפקודות.

התותח



תותח הוא כלי נשק חם שנועד להמטיר פגזים על אויב הנמצא במרחק ממנו.
במקרה שלנו זהו שורד מעט מתוחכם יותר משורד ה"מפציץ".
הוא מפציץ את זירת המשחק, במעגלים, בבית (byte) בעל ערך `0xCCh`. השורד אינו מפציץ את כל לוח המשחק ברצף כמו המפציץ שסקרנו קודם, אלא מפציץ בקפיצות – כלומר יש רווח בין מקומות שאנו מפציצים אותם.

אם אנחנו מדלגים על תאים בדרך, האם אין מצב שנפספס קוד של שורד יריב?

התשובה היא לא. אנו מניחים שהחלק הפגיע בקוד שורד היריב יתפרס על יותר מ-byte אחד ולכן עדיין נפגע בו; ואם לא בסיבוב הראשון, הרי שבסיבובים הבאים.

האם לא נפגע בקוד של השורד שלנו כמו שקרה בשורד ה"מפציץ"?

התשובה היא לא. השורד הזה יותר מתוחכם משורד המפציץ בתכונה נוספת – הוא נמנע מלהפציץ את הקוד שלו.

הוא עושה זאת על-ידי כך שערכו של ה-offset הראשון יהיה המקום לאחר הקוד שלו עצמו, ואם לאחר סיבוב הוא חוזר לאזור שהקוד שלו נמצא בו, הוא קופץ ומדלג על הקוד שלו עצמו.

נבחן את קוד האסמבלי של שורד זה:

```
; 1. initialization
start:
    mov  bx, ax
    add  bx, (end - start)
    mov  al, 0CCh

; 2. bombing loop
bomb:
    mov  [bx], al
    add  bx, 8
    jmp  bomb
end:
```

חלק האתחול מורכב מן הפקודות הבאות:

start – פקודת label שמציינת את תחילת קטע האתחול. אנו נשתמש ב-label זה גם כדי לציין את מקום תחילת השורד שלנו בזיכרון.

mov bx, ax – בעת טעינת השורד לזירה, הרגיסטר ax מכיל את הכתובת שהשורד נטען אליה. אנו מעבירים ערך זה לרגיסטר bx.

add bx, (end - start) – אנו מוסיפים לרגיסטר bx את אורך קטע הקוד שתופס השורד; לאחר הוספה זו רגיסטר bx יכיל את חישוב ה-offset הראשון אחרי הקוד של השורד – לפני שעדכנו את bx היה בו המיקום של השורד והוספנו את גודל השורד, מכאן שקבלנו עכשיו את המיקום לאחר קוד השורד. שימו לב שלמרות שהפקודה נראית מסובכת, הרי שהאסמבלר מחשב את ההפרש בזמן האסמבלי ובשפת המכונה זו תוספת קבוע בלבד.

mov al, 0CCh – אנו מכניסים לרגיסטר al את המידע בו נרצה להפציץ, ובמקרה זה האופ-קוד הזדוני שלנו 0xCC. בכל פעם שנבצע הפצצה נשתמש בתוכנו של רגיסטר זה.

חלק הלולאה מורכב מהפקודות הבאות:

bomb – פקודת label שמציינת שכאן מתחילה הלולאה; בכל חזרה של הלולאה נקפוץ למיקום זה ונבצע את הפקודות שאחריה.

mov [bx], al – אנו מעבירים את ערכו של al לכתובת עליה הרגיסטר bx מצביע.

add bx, 8 – אנו מעדכנים את ערכו של רגיסטר bx המחזיק את כתובת ההפצה על-ידי כך שאנו מוסיפים לו 8.

מדוע שמונה בתים? מכיוון שזה יותר גדול מגודל החלק הרלוונטי של הקוד של השורה שלנו (שבעה בתים, איך לחשב זאת נראה מאוחר יותר) וגם מתחלק בגודל לוח המשחק, כך שאחרי לולאה אחת סביב כל הכתובות, נחזור בדיוק לאותה הכתובת וכך לא נדרוך על עצמנו גם בלולאות הבאות.

jmp bomb – אנו מבצעים קפיצה ל-label בשם bomb כדי לבצע חזרה נוספת על הפקודות.

end – פקודת label שנשתמש בה כדי לציין את סוף קוד השורה בזיכרון.

10. טקטיקות מתקדמות

הפצה כבדה

סוג זה של הפצה יכתוב לכתובת הזיכרון המתחילה בערכם של es:di ול-255 הכתובות שלאחר מכן, את הערכים הנמצאים ברגיסטרים dx:ax (בסך-הכול ייכתבו 256 בתים).

סדר הכתיבה הוא al ראשון, ah שני, dl שלישי ו-dh רביעי ואחרון, כיוון הכתיבה יקבע על ידי דגל ה"כיוון" (direction flag) ברגיסטר הדגלים. ערכם של es:di משתנה בהתאם לאחר הקריאה.

הקריאה לסוג זה של הפצה תעשה על ידי האופקוד הבא (נקרא גם פסיקה):

```
int 86h
```

חשוב: יש לעדכן את הרגיסטר es לאותו ערך הנמצא ברגיסטר cs, אחרת השורה ייפסל בעקבות שגיאת memory exception לאחר הקריאה לפסיקה.

דוגמא לשימוש בקוד:

```
; heavy bombing starts at 0:0 to 0:255.
push cs
pop es                ; set es as the code segment (main memory map)

xor di,di            ; di is our offset, xoring it with himself
                    ; will set him to zero

mov ax, 0ccccch      ; set invalid opcodes in ax.
mov dx, ax            ; ... and copy it to dx.

int 86h              ; execute interrupt 0x86.

jmp $                ; will keep us doing nothing & alive ;-)
```

שורה זה "יפציץ" את השורה הראשונה של מוניטור התחרות. ניתן לשחק בכיוון הכתיבה על-ידי שינוי דגל ה"כיוון" באמצעות הפקודות CLD ו-STD.

הפצצה חכמה

הפצצה חכמה היא סוג נוסף של טקטיקת לחימה שהתווספה למשחק. הפצצה זו תחפש רצף של ארבעה בתים (אשר יכולים להיות ארבעה בתים של השורד המותקף) על גבי הסגמנט (64 קילו בית) הראשון ותחליף אותו ברצף אחר של ארבע בתים, אשר יכול להיות רצף של פקודות לא חוקיות (0CCh) או הוראת קפיצה (jmp) לאזור הקוד שלנו, כדי להרוויח את זמן המעבד של השורד המותקף. חשוב לזכור כי ההתקפה תחול על כל שורד, גם על שלנו, בהנחה שארבעת הבתים שחיפשנו ימצאו בקוד השורד, לכן יש לבחור רצף בתים ייחודי לשורד שנרצה לתקוף.

כדי להפעיל את סוג ההפצצה הזה נשתמש בפסיקה מספר 0x87, על ידי קריאה לפקודה:

```
int 87h
```

לפני הקריאה לפסיקה נאתחל את הרגיסטר es לאותו הערך המופיע ברגיסטר cs.

יש גם לאתחל את di למספר (offset) שנרצה להתחיל ממנו לחפש מתוך אותו סגמנט.

לאחר מכן יש לכתוב לתוך הרגיסטרים ax:dx את ארבעת הבתים שנרצה לחפש בסגמנט, וברגיסטרים bx:cx את ארבעת הבתים אשר יוחלפו לאחר מציאה.

חשוב לדעת כי לאחר הקריאה ערך es:di לא ישתנה, כך שלא ניתן לדעת את מיקום המופע של ארבעת הבתים שנמצאו, אם נמצאו בכלל בסגמנט

דוגמא לשימוש בקוד:

```
push cs
pop es          ; set es as the code segment (main memory map)

xor di,di       ; di is our offset, xoring it with himself
                ; will set him to zero

mov ax, 04187h  ; ax:dx is the byte string to find.
mov dx, 08820h

mov cx, 0ccccch ; cx:bx is the byte string to replace with.
mov bx, cx

int 87h         ; execute interrupt 0x87.

jmp $           ; will keep us doing nothing & alive ;-)
```

הערות: ארבעת הבתים הנמצאים ברגיסטרים ax:dx (הפקודה הרביעית והחמישית) הם אופ־קודים הנמצאים בדוגמת הקוד שהצגנו בפרק 6 "[מרב עצים רואים יער](#)", הריצו שורד זה יחד עם דוגמת הקוד ותראו כיצד השורד השני, המצוי בפרק 6, מפסיד מהר מאוד.

האצת שורד

במצב ריצה נורמלי של מנוע המשחק, לכל שורד מוקצה פרק זמן ריצה זהה לכל שאר השורדים. לצורך העניין, פרק הזמן הזה הזה לזמן הרצת אופ־קוד יחיד בקוד השורד. חשוב לזכור כי במנוע המשחק כל האופ־קודים הם בעלי אותו זמן ריצה. האצת שורד היא שיטה מיוחדת שנוספה למנוע המשחק, המאפשר לשורד המשתמש בה להריץ יותר מאופ־קוד אחד בפרק הזמן שהוקצה לו, ובכך להשיג עליונות על שאר השורדים שאינם משתמשים בשיטה. הוא יכול להשתמש בפרק הזמן המוגדל כדי לתקוף שטחים גדולים יותר באזור המשחק, וכן כדי לנסות טקטיקות לחימה אחרות. קביעת רמת האצת השורד נעשית על־ידי רגיסטר ווירטואלי בשם Energy. ערכו ההתחלתי הוא אפס (כלומר אין האצה) וכל 5 סיבובים במשחק (כלומר, 5 אופ־קודים שהשורד שלנו מריץ), אם הערך חיובי, מחוסר 1 מערכו של רגיסטר זה, אם ערכו הוא 0, לא יחוסר דבר (כלומר יישאר 0).

הגדלה של רגיסטר ה-Energy נעשית על־ידי קריאה לאופ־קוד WAIT בדיוק ארבע פעמים, בצורה רציפה. מנוע המשחק יריץ את ארבעת האופ־קודים הללו בזמן של אופ־קוד אחד. חשוב לדעת כי קריאה לאופ־קוד WAIT פחות מארבע פעמים תגרום לשגיאה בקוד השורד, והוא ייפסל.

כל רביעיית WAIT תגדיל את ערך הרגיסטר Energy ב-1, ומהירות השורד תחושב על ידי הנוסחה הבאה:

$$\text{Speed} = \log_2(\text{energy}) + 1$$

שיתוף פעולה בין שורדים

מנוע המשחק מאפשר שיתוף פעולה בין שני שורדים, ולכן כל קבוצה תוכל לשלוח לתחרות שני שורדים אשר יוכלו לשתף פעולה כדי לנצח את שאר השורדים בתחרות.

כדי שמנוע המשחק יידע שמדובר באותה קבוצה של שורדים, יש להוסיף את הספרה 1 לסוף שמו של השורד הראשון, ואת הספרה 2 לסוף שמו של השורד השני, לדוגמה:

Rocky1

Rocky2

לכל שורד בקבוצה יש רגיסטרים, מחסנית אישית וזמן ריצה פרטיים, אך מובטח כי השורדים ייטענו תמיד יחדיו לזירה (במקומות אקראיים), וכמו כן, ניקוד השורדים מאותה קבוצה יסוכם יחד בגרף הניקוד.

מסיבה זו אין לקרוא לשורד יחיד בשם המסתיים בספרה – מנוע המשחק יחפש את השורד השני של הקבוצה ו"יעוף" (אתם מוזמנים לתקן זאת בקוד המקור של המנוע).

לכל קבוצה מוגדר בלוק זיכרון משותף בגודל 1024 בתים, שחסום מפני שאר השורדים שמחוץ לקבוצה ונועד להעברת הודעת בין השורדים. כתובת הבלוק הזה היא es:0000.

11. תמיכה

חשוב לציין כי בכל מקרה של שאלה או בעיה אתם מוזמנים לפנות ל-support@codeguru.co.il.

כמו כן אפשר להיעזר בפורומים שלנו בקישור

<http://codeguru.co.il/wp/?forum=%D7%A7%D7%A1%D7%98%D7%A8%D7%99%D7%9D>