



ארגון המחשב ושפת סף

ברק גונן

מהדורות ניסוי



ארגון המחשב ומשפט סוף

גרסה 2.04

כתיבה:

ברק גונן

mbosso על מצגות וחומר לימוד מאות אנטולי פיימר

יעוץ פדגוגי:

אנטולי פיימר

עריכה:

עומר רוזנבוים

הגה ופתרון תרגילים:

אליזבת לנגרמן

עדן פרנקל

אריאל ווטפריד

ליאל מועלם

תודה מיוחדת: לעודד מרגלית וליהודה ויכטלפיש על ההערות המועלות, שתרמו רבות לגרסה העדכנית.

אין לשכפל, להעתיק, לצלם, להקליט, לתרגם, לאחסן במאגר מידע, לשדר או לקלוט בכל דרך או אמצעי אלקטרוני, אופטי או מכני או אחר- כל חלק שהוא מהחומר שבספר זה. שימוש מסחרי מכל סוג שהוא בחומר הכלול בספר זה אסור בהחלט, אלא ברשות מפורשת בכתב ממטה הסיבר הצה"ל.

© כל הזכויות על החומרים המקוריים ששובצו בספר זה שמורות לבעליין. פירוט בעלי הזכויות- בסוף הספר.

© כל הזכויות שמורות למיטה הסיבר הצה"ל. מהדורה שנייה תשע"ה 2015

<http://www.cyber.org.il>

תוכן עניינים

12	הקדמה
15	פרק 1 – מבוא ללימוד אסטטטי
18	פרק 2 – שיטות ספירה ויצוג מידע במחשב
18	מבוא – מהן שיטות ספירה?
19	משחק: Pearls Before Swine 3
20	שיטת העשרונית
21	חישוב ערך של מספר עשרוני בסיס אחר
21	שיטת הבינהaria
22	שיטת החקסדיימליה
25	פעולות החשבון
25	היבור
28	חיסור (הרחבה)
30	כפל (הרחבה)
32	חילוק (הרחבה)
33	יצוג מספרים על-ידי כמות מוגדרת של ביטים
34	יצוג מספרים שליליים
34	שיטת גודל וסימן
35	שיטת המשלימים לאחת
36	שיטת המשלימים לשתיים
39	או איך מנצחים את חואן?
41	יצוג מידע במחשב
41	סיבית – Bit
41	נגיסה – Nibble
42	בית – Byte
43	מילה – Word
43	מילה כפולה – Double Word
43	קוד ASCII
44	סיכום

45	פרק 3 – ארגון המחשב
45	מבוא
46	מכונת פון נוימן – Von Neumann Machine
48	פסי המערכת – SYSTEM BUSES
49	פס נתונים – DATA BUS
49	פס מענים – ADDRESS BUS
50	פס בקרה – CONTROL BUS
50	הזיכרון
53	טగמנטים
56	יחידת העבודה המרכזית – CPU
56	אוגרים – Registers
57	רגיסטרים כלליים – General Purpose Registers
61	רגיסטרי סגמנט – Segment Registers
62	רגיסטרים ייעודיים – Special Purpose Registers
62	היחידה האריתמטית – Arithmetic & Logical Unit
63	יחידת הבקרה – Control Unit
63	שעון המערכת (הרחבה)
65	סיכום
66	פרק 4 – סביבת עבודה לתוכנות באסמלבי
66	מבוא
66	Editor – Notepad++
67	קובץ Base.asm
69	_Command Line
74	TASM Assembler
75	Turbo Debugger – TD
83	סיכום
84	פרק 5 – IP, FLAGS – 5
84	מבוא
84	IP – Instruction Pointer

87	Processor Status Register – FLAGS
88	דגל האפס – Zero Flag
89	דגל הגלישה – Overflow Flag
90	דגל הנשא – Carry Flag
91	דגל הסימן – Sign Flag
91	דגל הכוון – Direction Flag
92	דגל הפסיקות – Interrupt Flag
92	דגל צעד ייחיד – Trace Flag
92	דגל זוגיות – Parity Flag
92	דגל נשא עזר – Auxiliary Flag
93	סיכום
94	פרק 6 – הגדרת משתנים ופקודת mov
94	מבוא
94	הגדרת משתנים
95	הקצאת מקום בזיכרון
97	משני – Signed, Unsigned
99	קביעת ערכים התחלתיים למשתנים
100	הגדרת מערכיים
102	פקודת MOV
104	העתקה מרגייסטר לרגייסטר
105	העתקה של קבוע לרגייסטר
105	העתקה של רגייסטר אל תא בזיכרון
106	העתקה של תא בזיכרון אל רגייסטר
108	העתקה של קבוע לזיכרון
108	региיסטרים חוקיים לגישה לזיכרון
109	תרגם אופרנד לכנתובת בזיכרון
111	Little Endian, Big Endian
111	העתקה ממערכיים ואל מערכיים

112	פקודת offset
113	פקודת LEA
113	הנחהה word ptr / byte ptr
114	ازהרת type override
115	פקודת mov - טעויות של מתחילים
116	שינוי קוד התוכנית בזמן ריצה (הרחבה)
117	סיכום
118	פרק 7 – פקודות אРИתמטיות, לוגיות ופקודות הזזה
118	מבוא
118	פקודות אРИתמטיות
119	פקודת ADD
120	פקודת SUB
121	פקודות INC / DEC
121	פקודות MUL / IMUL
124	פקודות DIV, IDIV
126	פקודת NEG
127	פקודות לוגיות
128	פקודת AND
130	פקודת OR
131	פקודת XOR
133	פקודת NOT
133	פקודות הזזה
133	פקודת SHL
134	פקודת SHR
135	שימושים של פקודות הזזה
136	סיכום
137	פרק 8 – פקודות בקרה
137	מבוא
137	פקודת JMP

138	קפיצות FAR ו-NEAR
139	תוויות LABELS
141	פקודת CMP
142	קפיצות מותנות
144	ティאור בדיקת הדגלים (הרחבה)
145	פקודת LOOP
146	זיהוי מקרי קטן
147	לולאה בתוך לולאה Nested Loops (הרחבה)
150	קפיצה מחוץ לתוחם
151	סיכום
152	פרק 9 – מהשנית ופרוצדורות
152	מבוא
154	המהשנית STACK
154	הגדרת מהשנית
156	פקודת PUSH
158	פקודת POP
160	פרוצדורות
160	הגדרה של פרוצדורה
162	פקודות CALL, RET
165	פרוצדורה NEAR, FAR
167	שימוש במהשנית לשימירת מצב התוכנית
170	העברה פרמטרים לפרוצדורה
173	העברה פרמטרים על המהשנית
173	Pass by Value
176	Pass by Reference
178	שימוש ברגיסטר BP
183	שימוש במהשנית להגדרת משתנים מקומיים בפרוצדורה (הרחבה)
186	שימוש במהשנית להעברת מערך לפרוצדורה
188	גילישת מהשנית - Stack Overflow (הרחבה)

194	(הרחבה) Calling Conventions
196	קונבנציות נפרצות
197	סיכום
198	פרק 10 (הרחבה) CodeGuru Extreme – 10
199	מבוא
201	פקודות אסמלבי שימושיות
202	Reverse Engineering
203	duck.com
205	coffee.com
207	codeguru.com
210	תרגיל: Make it – Break it – Fix it
211	סיכום
211	פרק 11 – פסיקות
213	מבוא
214	שלבי ביצוע פסיקה
216	ISR ו-IVT (הרחבה)
217	DOS
219	פסיקות תוו ממקלדת – AH=1h
222	הדפסת תוו למסך – AH=2h
223	הדפסת מהירות למסך – AH=9h
227	קליטת מהירות תווים – AH=0Ah
228	יציאה מהתוכנית – AH=4Ch
229	קריאת השעה / שינוי השעה – AH=2Ch ,AH=2Dh (הרחבה)
232	פסיקות חריגה – Exceptions
232	פסיקות תוכנה – Traps
233	כתיבת ISR (הרחבה)
239	סיכום
240	פרק 12 – פסיקות חומרה (הרחבה)
240	מבוא

240	פסקות חומרה – Interrupts
243	בקר האינטראפטים – PIC
244	אובדן אינטראפטים
245	זיכרון קלט / פלט – I/O Ports
247	המקלדת
247	הקדמה
248	יצירת Scan Codes ושליחתם למעבד
249	באפר המקלדת Type Ahead Buffer
250	שימוש בפורטים של המקלדת
255	שימוש בפסקת BIOS
256	שימוש בפסקת DOS
259	סיכום
260	פרק 13 – כלים לפרויקטים
260	מבוא לפרויקט סיום
260	בחירת פרויקט סיום
262	עבודה עם קבצים
262	פתחת קובץ
263	קריאה מקובץ
263	כתיבת קובץ
264	סגירות קובץ
265	פקודות נוספת של קבצים
265	תכנית לדוגמה – filewrt.txt
268	גרפיקה
269	גרפיקה ב-Text Mode
269	שימוש במחוזות ASCII
273	גרפיקה ב-Graphic Mode
274	הדפסת פיקסל למסך
276	קריאה ערך הצבע של פיקסל מהמסך
277	יצירת קווים ומלבנים על המסך

278	קריאה תמונה בפורמט BMP
285	קטעי קוד וטיפים בנושא גרפיקה
286	הש舅ת ציללים
290	שעון
290	מדידת זמן
293	יצירת מספרים אקראיים – Random Numbers
298	ממשק משתמש
298	קליטת פקודות מהמקלדת
298	קליטת פקודות מהעכבר
302	שיטות ניפוי Debug
302	תיעוד
302	תכנון מוקדם – ייצירת תרשימים ורימה
304	חלוקת לפרוצדורות
305	מעקב אחרי מונחים
305	העתיקות זיכרון
306	הודעות שגיאה של האסמלר
307	סיכום
308	נספח א' – רשימת פקודות חובה לבגרות בכתוב באסמלר
312	נספח ב' – מדריך לתלמידים: כיצד נганשים לפורום האסמלר הארץ
317	זכויות יוצרים – מקורות חיצוניים

הקדמה

ספר זה מיועד לתלמידי בתיכון ספר תיכון במגמות מחשבים והנדסת תוכנה, בדגש על תלמידים שמתמחים בסיביר. לא נדרש ידע מוקדם במחשבים ובשפות תכנות אחרות.

הספר כולל את הבסיס שנדרש לכתיבת תוכניות באסמבלי, בהתאם לתכנית הלימודים של משרד החינוך עבור יחידת המעבדה. תלמידים המעוניינים להעמק את הידע שלהם, ימצאו בספר את הרקע התיאורטי לתוכנים שהם מעבר לתוכנית הלימודים. כמו כן, הספר כולל הדרכה וכליים לביצוע פרויקטי סיום מורכבים, מעבר לדרישות משרד החינוך. הנושאים שהווים מעבר לתוכנית הלימודים מסוימים בספר כ"הרחבה".

לכתביה בשפת אסמבלי קיימות מספר סביבות פיתוח, שפותחו על-ידי חברות שונות. הקוד שבספר כתוב בסביבת העבודה TASM, והספר מליץ על אוסף תוכנות לשימוש בתור סביבת עבודה - גם כדי לחסוך לקוראים את הצורך לחפש סביבת עבודה באופן עצמאי, וגם כדי שייהי סטנדרט אחיד לעובדה וلتרגול בכתות. מומלץ להתקין את סביבת העבודה ולהשתמש בה כבר בתחילת הלימוד, וכן לפתור את התרגילים המופיעים בפרקיהם תוך כדי תהליכי הלימוד, ולא רק בסופו.

תוכנית הלימוד מאורגנת כך:

- פרק 1 יוקדש לרקע כללי על שפת אסמבלי ועל הסיבות שבזכותן היא נחשבת לשפה חשובה לומדי מחשבים בכלל וטיבר בפרט.
- בפרק 2 נלמד איך ליצג מספרים בשיטות ספירה שמקנות על עבודה עם מחשב.
- בפרק 3 נלמד על מבנה המעבד במחשב. כיוון שיש לא מעט סוגים מעבדים, נבחר משפחת מעבדים נפוצה ונתמך בה.
- בפרק 4 נלמד להתקין את סביבת העבודה, נכתב ונರין את התוכנית הראשונה שלנו באסמבלי.
- בפרק 5 נלמד על רכיבים במעבד שמאפשרים לנו לדעת מה מצב התוכנית - רגיסטרים ייעודיים.
- בפרק 6 נלמד איך מגדרים משתנים בזיכרון, כולל מהרווזות ומערכות, ואיך מבצעים העתקה של ערכיהם מהזיכרוןandal הזיכרון.
- בפרק 7 נלמד פקודות אריתמטיות (חיבור, חיסור וכו'), פקודות לוגיות ופקודות הווה.
- בפרק 8 נלמד איך להגדיר תנאים לוגיים ואני לכתוב פקודות בקרה. בסוף הפרק תוכלנו לכתוב אלגוריתמים שונים, לדוגמה תוכנית שמבצעת מינון של איברים במערך.
- בפרק 9 נלמד על אוצר זיכרון שנקרא מהנסנית, נלמד לכתוב פרוצדורות, להעיר אליה פרמטרים ולהגדיר משתנים מקומיים. בסיום הפרק נראה איך החומר שלמדו לנו להבין נושאי תוכנה מתקדים.
- בפרק 10 נקבל רקע וכליים בסיסיים הדרושים להשתתפות בתחרות קודגورو אקסטרים, בין היתר נלמד את העקרון הבסיסי של Reverse Engineering על ידי פיצוח "זומבים", היזות תוכנה שמספרם צוות התחרות.
- בפרק 11 נלמד על פסיקות. נלמד איך להשתמש במגוון פסיקות DOS כגון קריאה של תוו מהמקלדת או הדפסה של מהרווזות למסך.

- בפרק 12 נעמיק את הידע בנושא פסיקות חומרה, פורטים ועבודה עם התקני קלט פלט. נתמקד במקלדת ונדגים באמצעותה את אפשרויות העבודה השונות מול רכיבי חומרה. חומר זה הינו הרחבה, אך הוא נדרש לטובת הבנת הפרק הבא על פרויקטי הסיום.

- בפרק 13 ניקח נושאים שונים שקשורים לכתיבת פרויקט סיום - לדוגמה גרפיקה, עבודה עם קבצים ועבודה עם עבר – ונעמיק את ההבנה שלנו בהם בעזרת תוכניות דוגמה.

הערות לגרסה 2.0

עם סיום שנת הלימודים התשע"ה, בה שימש הספר בכ-25 כיתות גבאים, עודכן הספר על פי לключи ההוראה ולוקטו אליו הומרים שפותחו כהרחבות, הסברים או תרגילים. העדכונים העיקריים נוגעים לנושאים הבאים:

- חומר בנושא תחרות קודגورو אקסטרים כולל reverse engineering לדוגמאות תוכנה מהתחרות.
- נושאי הרחבה מתחומי התוכנה - calling conventions, stack overflow.
- תרגילים נוספים, בין היתר: שינוי קוד תוך כדי ריצה ותרגיל צופן הזזה.
- תלמידים שלומדים אסמבלי כיחידה בחירה במחשבים (ולא כיחידה מעבדה במסגרת מגמת סייבר) ימצאו בספר התיאיות לפקודות אסמבלי שנדרשות על ידי משרד החינוך בבחינות הבגרות.

פורום ארצי לעזרה ושאלות

לרשות התלמידים עומדת "אוניברסיטת" גבאים. זהו אתר שאלות ותשובות בדומה לאתר המפורסם stackoverflow, אך מקומי – ובעברית. הנכם מוזמנים להרשם אליו ולהעלות שאלות, שייענו הן על ידי תלמידים אחרים והן על ידי צוות תכנית גבאים, מורים ועוורי הוראה. הוראות הרשמה והתחברות לאתר ניתנת למצוא בנספח סוף ספר זה.

ספרים לימוד ותרגילים

הספר **Randall Hyde Art of assembly** הוא ככל הנראה המקיף ביותר ללימוד השפה. אמנם הקוד שם כתוב בהתאם לחוקי כתיבה מעט שונים מאשר בספר זה (NASM לעומת TASM, למען הדיווק), אך הספר הבינו מקור מועלה להבנת השפה ולהסבירים אודוט פקודות אסמבלי. ניתן להוריד אותו בחינם מהאינטרנט.

חוברת התרגילים "הכנה לבגרות 5 ייח"ל במדעי המחשב" מאת רונית גל או רמציאנו מכילה תרגילים רבים בנושאים שנדרשים על ידי משרד החינוך לפרק האסמבלי בבגרות במחשבים. התרגילים הם בהיקף ובסוגנון השאלות בבחינות הבגרות.

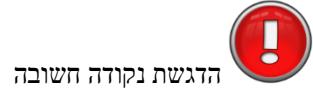
لتלמידים

בספר זה תלמדו את הבסיס להמשך לימודי הסייבר. ספר הלימוד יעזור לכם להגעה לעומקה של מבנה המחשב ולרמות תכניות שתאפשר לכם לכתוב תוכנות בהיקף של כמה אלפי שורות קוד, כדוגמת משחקי מחשב קטנים.

אתם יוצאים למסע לרכישת ידע. המשע לא צפוי להיות קל, אך במהלךו תלמדו לא מעט ותרכשו יכולת חדשה וחשובה להמשך. יחד עם לימודי האסמבלי כדאי שתפתחו את סט הכלים שיאפשר לכם ללמידה לפתור בעיות בלבד: חיפוש באינטרנט, סינון חומר, התמקדות בעיקר, ניסוי וטעיה. בסופה של דבר, אלו הן המומנויות שהכי יעוזו לכם להצליח, גם בסיביר וגם בחיים.

אייקונים

בספר, אנו משתמשים באיקונים הבאים בכדי להציג נושאים ולהקל על הקריאה:



פרק 1 – מבוא ללימוד אסמבלי

ברוכים הבאים! אם אתם קוראים את השורות האלה,-CNראה שהחליטתם ללימוד אסמבלי. או שאלות מישחו אחר החלטת במקומכם שאתם צריכים ללימוד אסמבלי? כך או כך, זו החלטה טובה. מיד ננסה להבין מה חשוב שתדעו אסמבלי.

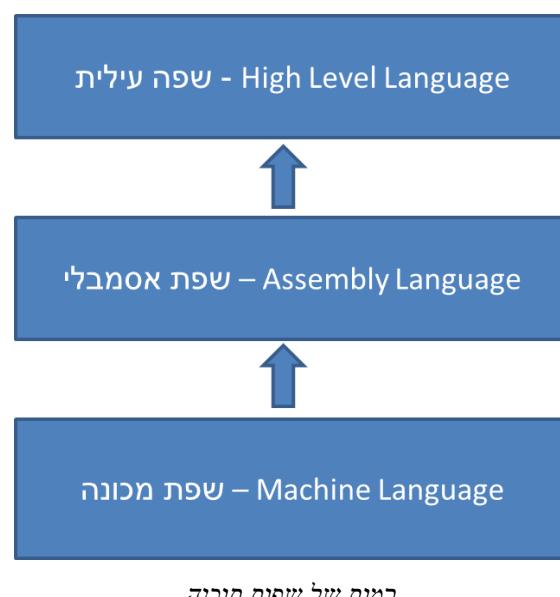
אסמבלי היא שפת התכנות הראשונה בעולם, שפותחה לפני עשרות שנים בתור תחילה פשוט יחסית לשפת מכונה, שבה כותבים באחדות ובאפסים. במילים אחרות, לפני שפותחה שפת אסמבלי, מי שרצה לתוכנת היה צריך להזין למחשב רצף של אחדות ואפסים. ומה קורה אם מתבלבלים? אם מחליפים באיזשהו מקום 0 ב-1? חבל מאוד.

שפה אסמבלי הביאה ליעול ניכר בכך שהיא מאפשרת פקודות שניננות להבנה בשפה אנושית. במקביל לפיתוח שפת אסמבלי פותח אסמלבר, שהוא כלי שמסוגל להמיר את פקודות האסמבלי לשפת מכונה.

אם קודם לכן, על מנת לבצע פקודה, היה צריך לרשום רצף של אחדות ואפסים, למשל "10111000", משוהמתה שפת אסמבלי אפשר לכתוב פקודה שקל לקרוא ולהבין, כגון mov, והאסמלבר יתרגם אותה אל הרצף "10111000". דבר זה מקל מאד על כתיבת הקוד וכן על קרייאתו.

אסמבלי נחשבת Low Level Language. הסיבה היא因为她 שמי שמתכוна באסמבלי כתוב פקודות שעובדות ישירות מול החומרה של המחשב – אי אפשר לתוכנת אסמבלי בלי להבין איך עובדים המעבד, הווירוון ורכיבי החומרה שהשורים אליהם. כל פקודה באסמבלי מתרגםת לפקודה אחת בשפת מכונה – דוגמה העתקה, חיבור, כפל וכו'.

עם השנים פותחו שפות תוכנה עיליות, או High Level Languages. בשפות אלו שפת התוכנה מסתירה מהמתכוна את ה"קרביים" של המחשב ומאפשרת לתוכנת בזרה יותר פשוטה וקללה. שפות תוכנות אלו כוללות בין היתר את C++, Java ו-Python. פקודה בשפה עילית עשויה להיות מתרגמת במספר פקודות בשפת מכונה.



או למה בעצם ללימוד אסמבלי?

ישנן דוגמאות לא מעט סיבות למה לא ללימוד אסמבלי. הנה כמה מהן:

- מאז שאסמבלי פותחה העולם התקדם ויש שפות תוכנה מודרניות.
- אסמבלי היא שפה מורכבת יחסית ללימוד.
- מסובך לכתוב קוד באסמבלי.
- קשה לדבג (למצוא שגיאות ולנפוח אותן) באסמבלי.
- קוד אסמבלי הוא תלוי מעבד. ככלומר, קוד שנכתב למשפחת מעבדים מסוימת לא יתאים למשפחת מעבדים אחרת.
- קוד המקור של תוכנית שכותבה באסמבלי כמעט תמיד יהיה ארוך יותר מאשר קוד של תוכנית שביצעת את אותן הפעולות וכותבה בשפת תכנות אחרת.

ואמנם, מי שרגיל לתכנות בשפות עיליות בדרך כלל לא מתלהב לתוכנת באסמבלי. מתקנים מנוסים שלומדים אסמבלי, חשים לעיתים קרובות שבשפות אחרות אפשר לעשות הרבה בפחות הנלמודות בשיעורי אסמבלי. זה באמת נכון, עד שמניגעים למוגבלות של שפות תכנות אחרות, מה שחייב אותנו לסייע למה כן ללימוד אסמבלי:

- שפת אסמבלי תורמת להבנה עמוקה של המחשב על חלקייו השונים. העובדה שatoms נחשפים לחלקים הפנימיים ביותר של המחשב ושותם דבר אינו מהוה "קופסה שחורה" בשביבכם, תאפשר לכם בעיה לחתמוד עם בעיות תכנות בלתי שגרתיות, שנדרשות בעולם הסיביר.
- שליטה בשפת אסמבלי היא כדי טכנולוגייתית חשובה בעולם הסיביר. לדוגמה, כדי לבצע מחקר קוד על ידי **Reverse Engineering** כדי להבין אבטחה כמו **Stack Overflow**. הבנת בעיות האבטחה וכל מחקר קוד אפשרית, בין היתר, כתיבת קוד מוגן יותר בפני בעיות אבטחה.
- אסמבלי עובדת בצורה צמודה עם החומרה של המחשב. אם נדרש לכתוב קוד שעבוד מול חומרה, או להפעיל חומרה בצורה לא שגרתית, אז שימוש באסמבלי הוא עדין בחירה נפוצה. מסיבה זו, חברת Apple לדוגמה, ממליצה לכותבי אפליקציות לשנות באסמבלי.
- המקום שתוכנית אסמבלי תופסת בזכרון הוא קטן למדי יחסית לתוכנות שכותבות בשפות עיליות.

יכול להיות שהשתכנעתם בחשיבות לימוד אסמבלי ויכול להיות שלא. בכלל אופן – שימוש לב לדרישות הדרישות שנוסחה על ידי חברת אבטחה מידע ישראלי, מי שמעוניין להציג מועמדות למשרה בתחום הסיביר:

Cyber Security Researcher

- Familiarity with programming languages (e.g. C++, Java, C#, PHP, Assembly, etc.)
- Knowledge of networking and internet protocols (e.g. TCP/IP, DNS, SMTP, HTTP)
- **Reverse engineering** experience – a must.
- Analysis of malicious code – Major advantage

לבסוף נזכיר, עוד מספר מטרות של לימוד האסמבלי ב大妈ת גבהים. ראשית, רכישת מיומנויות של סדר, ארגון וחשיבה מתודית. שפת אסמבלי היא מקום טוב במיוחד לרכוש בו יכולות תכנות מאורגן ומסודר, בಗל הדקדנות והירידה לפרטים שנדרשת כדי לכתוב קוד תקין. יכולות אלו הן אבני הבניין לעובדה בתחום הסיבר.

שנית, הكنيית יכולת לימוד עצמית והתמודדות עם אתגרים. במסגרת גבאים תידרש לכתוב קוד בהיקף שימושתי ולדיבג אותו. זהו אתגר שיכשיר אתכם לקראת הבאות.

שלישית, ידע מקדים באסמבלי נדרש לחומר הלימוד של גבאים במערכות הפעלה. אז קדימה, אנחנו מוכנים להתחילה במסע נפלא בו נלמד רבות על דרך הפעולה של המחשב, על כתיבת קוד ועל איך ללמוד בעצמנו. נתחיל מהבסיס – שיטות ספירה וייצוג מידע במחשב.



פרק 2 – שיטות ספירה ויצוג מידע במחשב

מבוא – מהן שיטות ספירה?

לבני האדם יש עשר אצבעות ולכן ספירה בשיטה העשרונית (בסיס עשר, Decimal) נראית לנו טבעיות מילודות, אך קיימות שיטות רבות לספור. לעומת זאת השיטה האחروفית דורשת מאיינו מאין – אדרבא כשמדבר בשיטה הקסדצימלית (בסיס ששה עשרה, Hexdecimal) שבה אותיות מייצגות חלק מהמספרות. יתכן שאלתו הינה חיזרים בעלי ששה עשרה אצבעות, היה לנו יותר נוח לספור בסיס הקסדצימלי. בכל מקרה, תכונות בשפת אסטטיל מצריך הבנה של ייצוג מספרים בשיטה הבינארית (בסיס שתיים, Binary) ובשיטה הקסדצימלית.

למרות חוסר הנוחות, כשהעובדים עם מחשבים, היתרונות שבשימוש בשיטה ספירה ביבנארית והקסדצימלית עולים בהרבה על החסרונות. שיטות אלו מפשטות את העבודה במגוון נושאים כגון מספרים שליליים, ייצוג של תווים, פועלות לוגיות, קריאת מידע שומר בזיכרון ועוד נושאים רבים שכרגע אינם מוכרים לנו אבל עוד נגיע אליהם.

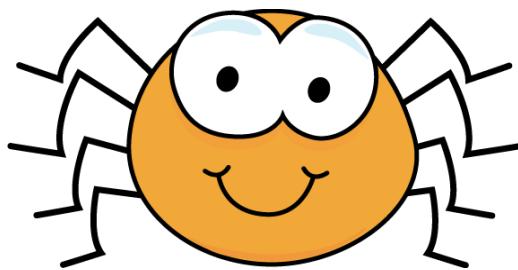
בפרק זה נלמד שיטות לייצג מספרים בסיסי ספירה שונים תוך התמקדות בשיטה הבינארית ובשיטה הקסדצימלית, נבצע את פעולות החשבון הבסיסיות בסיסים שאינם בסיס עשר, נראה איך מייצגים מספרים שליליים בשיטות שונות ונתמקד בשיטה שנקראת "המשלים לשתיים". בואו נתחיל.

שיטת ספירה זו בסיס הכל דרכיהם שונות לייצג כמות נתונה של פרטם שאפשר לספור אותם. נסתכל על הטבלה הבאה, שרשומים בה מספרים בשלושה בסיסים – בסיס 10, בסיס 8 ובסיס 3.

- בסיס 10, יש ספרות שמייצגות את המספרים מ-0 עד 9. קלומר לא ניתן לייצג מספרים מעל 9 באמצעות ספרה בודדת.
- בסיס 8 יש ספרות שמייצגות את המספרים מ-0 עד 7. קלומר לא ניתן לייצג מספרים מעל 7 באמצעות ספרה בודדת.
- בסיס 3, יש ספרות שמייצגות את המספרים מ-0 עד 2. קלומר לא ניתן לייצג מספרים מעל 2 באמצעות ספרה בודדת.

בסיס 3 0,1,2	בסיס 8 0,1,2,3,4,5,6,7	בסיס 10 0,1,2,3,4,5,6,7,8,9
0	0	0
1	1	1
2	2	2
10	3	3
11	4	4
12	5	5
20	6	6
21	7	7
22	10	8
100	11	9
101	12	10
102	13	11

ניקח לדוגמה עכבייש. כמה רגליים יש לו?



בבסיס 10, הספרה 8 מייצגת את כמות הרגליים שיש לעכבייש. בסיס 8, לעכבייש יש 10 רגליים ובבסיס 3 לעכבייש יש 22 רגליים. כמות הרגליים שיש לעכבייש לא השתנתה, רק הדרך שלנו לייצג את המידע זהה.

Pearls Before Swine 3: משחק

לפני שנתיים, בואו נshallק משחק קטן. המשחק הקשור להומר הלימוד בפרק זה, אך בשלב זה לא נגלה יותר מזה.

היכנסו ל קישור הבא: www.transience.com.au/pearl3.html

חוקי המשחק מוסברים על ידי "חואן", השדון בעל השינויים ה策horot. נסו לנצח אותו!



"חואן. נצחו אותו!"

אל תהיו מתוסכלים אם לא הצליחם להגיע רחוק... בקרוב, באמצעות הידע שתרכשו בפרק זה, תוכלו לנצח את חوان ללא קושי רב.

השיטה העשרונית

כעת נרחיב את ההסבר לגבי איך מייצגים מספרים בסיסים שאנו לא רגילים אליהם וכדי להקל, נתחיל דוקא מייצג מספרים בסיס عشر המוכר. בשיטה העשרונית קיימות עשר ספרות: 0,1,2,3,4,5,6,7,8,9. בעזרת הספרות הללו אנחנו מייצגים את כל המספרים. הערך של ספרה נקבע לפי המיקום שלה. כך, המספר 501 שונה מהמספר 105. במספר 501 הספרה 5 היא ספרת המאות, ואילו במספר 105 הספרה 5 היא ספרת היחידות.

שימוש לב – מעכשיו נכתב את בסיס הספרה בקטן, מתחתית המספר, כך:



$$47_{10}$$

נעשה זאת כדי להבדיל מספרים שנראים אותו דבר אך מייצגים ערכים שונים בסיסי ספרה שונים. לדוגמה:

$$47_{10} \neq 47_8$$

כדי להרגיש בינה עם נושא הערך לפי מיקום, נפרק כמה מספרים עשרוניים למרכיבים שלהם:

$$47_{10} = 7 \cdot 10^0 + 4 \cdot 10^1$$

$$375_{10} = 5 \cdot 10^0 + 7 \cdot 10^1 + 3 \cdot 10^2$$

$$1994_{10} = 4 \cdot 10^0 + 9 \cdot 10^1 + 9 \cdot 10^2 + 1 \cdot 10^3$$

הישוב ערך של מספר עשרוני בבסיס אחר

נחוק את המספר 199_{10} ליצוג שלו בסיס 5:



פעולה	שארית
$199:5 = 39$	4
$39:5 = 7$	4
$7:5 = 1$	2
$1:5 = 0$	1



נתחיל בחולקה 5. התוצאה היא 39 והשארית היא 4.

נחלק את התוצאה 5.39:5 הוצאה היא 7, השארית 4.

חלוקת של 7:5 היא 1, שארית 2.

חלוקת של 1:5 היא 0, שארית 1.

כעת נרשום את כל השאריות, מלמטה למעלה - 1244.

ומכאן ש- $199_{10} = 1244_5$.

נבדוק את החישוב שעשינו, על-ידי ביצוע הפעולה הפוכה- תרגום של 1244_5 למספר דצימלי:

$$1244_5 = 4*5^0 + 4*5^1 + 2*5^2 + 1*5^3 = 4*1 + 4*5 + 2*25 + 1*125 = 4+20+50+125 = 199_{10}$$

השיטה הבינארית

בසפירה לפי בסיס שניים, ספירה שנקרה גם השיטה הבינארית, קיימות שתי ספרות בלבד: 0,1. ככלומר הספרה 2 לא קיימת וצריך ליצג אותה על-ידי שתי ספרות. השיטה הבינארית חשובה במיוחד בהקשר של מערכות מחשב, כיון שככל המידע מוצג בזיכרון המחשב באמצעות רצפים של אחדות ואפסים. המחשב לא מכיר את הספרה 2.

בשיטת הבינארית, ערך המיקום של הספרות משתנה לפי חזוקות של 2. ערך המיקום של הספרה הימנית ביותר הוא 2^0 ובאופן כללי ערך המיקום של הספרה ה-n הוא 2^{n-1} , כפי שניתן לראות בדוגמה הבאה בה מוצגים ערכי המיקום של שמונת הספרות הראשונות:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

ניקח לדוגמה את המספר 10011_2 . נתרגם את המספר זהה למספר עשרוני. לטובת קלות הציגו נכnis את המספר בטבלה שלנו, כאשר כל ספרה נמצאת במקום בעל הערך המתאים לה:



2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
-	-	-	1	0	0	1	1

$$10011_2 = 1 + 2 + 16 = 19_{10}$$

הمرة מבסיס עשרוני לבינארי מתבצעת בדיקת אותה שיטה שהדגמנו בסעיף "חישוב ערכו של מספר עשרוני בסיס אחר", רק שהפעם הבסיס الآخر הוא 2. ניקח את המספר 19:



פעולה	שארית
$19:2=9$	1
$9:2=4$	1
$4:2=2$	0
$2:2=1$	0
$1:2=0$	1



וכשמעתיקים את השאריות מלמטה למעלה, מקבלים את 10011_2 .

השיטה ההקסדצימלית

במספרה לפי בסיס 16, ספירה שנקרأت גם בשיטה ההקסדצימלית, קיימות שש עשרה ספרות. כיוון שהכתב שלנו מכיל רק עשר ספרות (מ-0 עד 9), בשיטה זו לוקחים שש אותיות ונותנים להן ערך מסוימי. האות A מקבלת את הערך 10, האות B את הערך 11 וכן הלאה. בטבלה הבאה מ羅כזים הערכים של הספרות ההקסדצימליות:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

בשיטת ההקסדצימלית, מספר יכול להיות צירוף של אותיות וספרות. לדוגמה $1A_{16}$, $2B_{16}$, $4C4_{16}$, $C1A_{16}$, $F15_{16}$, $C0FFEE_{16}$, $DEAD_{16}$ או $C0DE_{16}$.

בשיטת זו אפשר להשתעשע עם מספרים בעלי ערכיים נחמדים כמו $C0DE_{16}$.

שימוש לב שאנו משתמשים בספרה 0, לאות האנגלית O אין משמעות בייצוג הקסדצימלי.



השיטה הבאות כולן שיטות תקופות לייצוג מספרים הקסדצימליים:

- רישום הבסיס 16 מתחת למספר – לדוגמה $C0DE_{16}$

- הוספת האות **h** בסוף המספר. בשיטת ייזוג זו, כאשר המספר מתחילה באות, מוסיפים '0' מצד שמאל. לדוגמה $0C0DEh$ (נדגיש כי ניתן להוסיף אפסים מצד שמאל בכל מקרה, אבל כאשר המספר מתחילה באות – חיברים להויסף)

- הוספת הצירוף **A** בתחילת המספר – לדוגמה $0xC0DE$

לאחר שהשתעשנו, נבעץ תרגום של מספר, $4F_{16}$, מהקסדצימלי לעשרוני:

$$4F_{16} = F \cdot 16^0 + 4 \cdot 16^1 = 15 + 64 = 79_{10}$$

نبצע את הפעולה הפוכה: תרגום מבסיס עשרוני להקסדצימלי, למספר 199_{10} :



פעולה	שארית
$199:16 = 12$	7
$12:16 = 0$	C (12)

נחלק $199:16$. התוצאה 12, השארית 7.

נחלק את התוצאה 12:16, התוצאה 0 השארית 12 (בבסיס 10), שמייצגת במסיס 16 על ידי 'C'.

נקרא את השאריות מלמטה למעלה ונקבל $199_{10} = C7_{16}$

כעת נבעץمرة של מספרים ביןaries להקסדצימליים. להמרה זו יש מאפיין מיוחד, מכיוון ש-16 הוא חזקה של 2. נראה מיד איך תcona זו באה לידי ביטוי:

הקסדצימלי	ביןארי
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

כל ספרה הקסדיימלית ניתנת לייצוג עליידי מספר בינארי בן ארבע ספרות. בעקבות זאת, ההמרה מהקסדיימלי לבינארי יכולה להתבצע פשוט עליידי מעבר של ספרה אחר ספרה, והחלפת הספרה הקסדיימלית באربع ספרות בינאריות. ניקח לדוגמה את המספר $9B_{16}$:

$$9_{16} = 1001_2$$

$$B_{16} = 1011_2$$

$$9B_{16} = 10011011_2$$

התכוונה זו, המעבר הפשט מביבנארי להקסדיימלי, הוא מה שהופך את השיטה הקסדיימלית לשימושית בעולם המחשבים. במקום לזכור רצפים של אחדות ואפסים שנמצאים בזיכרון המחשב, יותר קל להכין ערכיהם הקסדיימליים והסיכוי לטעות נעשה נמוך יותר. חישבו על מספר כמו $9B2C_{16}$, איך יותר קל כתוב אותו – כמספר הקסדיימלי או כמספר ביבנארי, שהוא 1001101100101100 ? איפה יש פחות סיכון לטעות?

תרגיל 2.1: מעבר בין בסיסי ספירה



השלימו את הטבלה הבאה – בטור השמאלי מספרים בסיס 10, דצימלי. בטור האמצעי מספרים בסיס 2, ביבנארי. בטור הימני מספרים בסיס 16, הקסדיימלי.

בסיס 16	בסיס 2	בסיס 10
35		
	1011	
		12
	1100011	
59		
		63
		5C
	11110	
21		
		31

פעולות חשבון

NELMED את ארבע פעולות החשבון הבסיסיות – חיבור, חיסור, כפל וחילוק – בסיסים שאינם בסיס עשר.

חיבור

נתחילה מתרגול בסיס עשר. נחבר את המספרים 133 ו-70.

$$\begin{array}{r}
 & 1 \\
 & 133 \\
 + & 70 \\
 \hline
 203
 \end{array}$$

אנחנו מתחילה את החיבור מצד ימין ומהברים כל ספרה עם הספרה שמתחתייה. כאשר מקבלים תוצאה שהיא שווה או גדולה מ-10, אנחנו כותבים למטה רק את ספרת האחדות וailו את ספרת העשורה אנו מוסיפים ("נושאים" או באנגלית "carry" נשא) לזוג הספרות שמשמאל.

חיבור מספרים בינהירים אינם שונה ברמת העיקרון, פרט לכך ששארית מתקבלת עם כל תוצאה חיבור שהיא שווה או גדולה מ-2. נתבונן באפשרויות החיבור השונות, המרוכזות בטבלה למטה.

חיבור של 0 ועוד 0 שווה 0.

חיבור של 0 ועוד 1 שווה 1.

חיבור של 1 ועוד 0 שווה 1.

חיבור של 1 ועוד 1 שווה 10 (אפס עם נשא אחד).

+	0	1
0	0	1
1	1	10

נתרגל עם הדוגמה הבאה – 1010 ועוד 11 (או בסיס עשרוני – 10 ועוד 3):



$$\begin{array}{r}
 & ^1 \\
 & 1010 \\
 + & 11 \\
 \hline
 1101
 \end{array}$$

התחלנו מצד ימין. החיבור של 0 ועוד 1 שווה 1, ללא נשא.

החיבור של 1 ועוד 1 שווה 0, עם נשא.

בטור השלישי, החיבור של הנשא 1, עם 0 ועוד 0, שווה 1, ללא נשא.

לבסוף בטור הרביעי, החיבור של 1 ועוד 0 שווה 1, ללא נשא.

חיבור של מספרים הקסדצימליים מתבצע באופן זהה, פרט לכך שבכל פעם שנגיעה לתוצאה שווה או גדולה מ-16



נעביר נשא שמאליה. לדוגמה, חיבור של $ABCD_{16}$ עם 123_{16} :

$$\begin{array}{r}
 & ^1 \\
 & ABCD \\
 + & 0123 \\
 \hline
 ACF0
 \end{array}$$

התחלנו מימין. החיבור של D (13 בסיס עשר) ועוד 3 הוא 16, لكن התוצאה היא 0 עם נשא 1.

חיבור הנשא ועוד C ועוד 2 הוא F, ללא נשא.

וועוד 1 שווה C, ללא נשא.

וועוד 0 שווה ל-A.

תרגיל 2.2: חיבור



היבור בסיס בינארי:

A	B	A+B
100111	10100	
11001	11000	
110000	100101	
111001	11001	
110110	101011	
111001	10011	
111001	101010	

היבור בסיס הקסדצימלי:

A	B	A+B
25	F	
B	29	
26	2C	
34	34	
1F	2A	
1E	12	
22	2F	



חיסור (הרહבה)

כאשר מחסרים שתי ספרות, ישן שלוש אפשרויות:

1. הספרה הראשונה גדולה מהשנייה. לדוגמה 9 פחות 6. במקרה זה פועלות החיסור פשוטה.
2. הספרות שוות, لكن תוצאה החיסור היא אפס.
3. הספרה השנייה גדולה מהראשונה. לדוגמה 6 פחות 9. במקרה זה משתמש ב"נשא שלילי". בשפה פשוטה, ניקח יחידה אחת מהספרה שמצד שמאל.

כרגיל, לפני הכל נבצע את הפעולות בסיס עשר. 619 פחות 21:



$$\begin{array}{r}
 -1\ 10 \\
 619 \\
 - 21 \\
 \hline
 598
 \end{array}$$

החסור של 9 פחות 1 שווה 8.

בשביל החיסור של 1 הינו צריכים להשתמש בנשא שלילי, כלומר הורדנו יחידה מהספרה הבאה והוספנו עשר יחידות לספרה הנוכחית. קיבלנו 11 פחות 2, שווה 9. לאחר שהורדנו יחידה מ-6, בטור השמאלי נשאר 5.

נבצע עכשו בסיס 2 את הפעולה 1010 פחות 1 (10 פחות 1).



$$\begin{array}{r}
 -1^{10} \\
 1010 \\
 - 1 \\
 \hline
 1001
 \end{array}$$

נתחיל מימין. חיסור של 0 מצריך נשא שלילי. אנחנו מורידים את הנשא השלילי מהספרה הבאה, ומוסיפים את הנשא השלילי (10 בסיס 2) לספרה הימנית. 2 פחות 1, שווה 1.

בטור השני, המצב הוא שיש לנו נשא שלילי. 1 פחות הנשא השלילי שווה 0.

בטור השלישי והרביעי, אין שינוי בתוצאה מפעולת החיסור.

נבדוק את התוצאה שלנו בסיס עשר: 10 פחות 1 שווה 9, שבבסיס 2 הינו 1001 – התוצאה שקיבלנו.

כעת נבצע תרגיל חישור בסיס 16:



-1 16
 -1 16
- DEAD
CODE
 1DCF

הסביר:

חישור של D יהיה E מזכיר נשא שלילי. נוריד יחידה מ-A ונפרוט אותה לשש עשרה יחידות שמתווספות לד-D. תוצאה החישור היא לנ- F (בהמרה לעשרוני – 13 ועוד 16 פחות 14, שווה 15).

בטור השני, הורדנו יחידה מ-A וצריך לחסר גם את D. שוב נצרך להשתמש בנשא שלילי, ניקח יחידה מהטור השלישי. תוצאה החישור היא לנ- C (בהמרה לעשרוני – 10 פחות 1, ועוד 16 פחות 13, שווה 12).

בטור השלישי, הורדנו יחידה מ-E, נשארנו עם D.

לבסוף בטור הרביעי החישור של D יהיה C נתן 1.

תרגיל 2.3: חישור



השלימו את הטבלאות הבאות.

בסיס בינארי:

A	B	A-B
101010	10100	
11001	10110	
10100	1101	
100101	10011	
111000	1111	
101011	11010	
10101	1010	

בסיס הקסדיימלי:

A	B	A-B
93	3D	
130	22	
E7	60	
C3	19	
CF	56	
47	12	
54	D	

כפל (הרחבה)

פעולות הכפל של מספרים בינאריים היא כנראה פשוטה אפילו יותר מאשר כפל מספרים עשרוניים. אחרי הכל, בזמן שלוחה המספרים בינאריים הם בגודל 10×10 , בעוד המספרים העשרוניים הם בגודל 2×2 ואפשר לסכם אותו בטבלה הבאה:

X	0	1
0	0	0
1	0	1

נוקח לדוגמה את תרגיל הכפל 1010×11 (10 כפול 3 בעשרוני):

$$\begin{array}{r}
 1010 \\
 \times 11 \\
 \hline
 1010 \\
 1010 - \\
 \hline
 11110
 \end{array}$$

הסבר:

1 כפול 1010 שווה 1010, אותו אנחנו רושמים בשורה הראשונה.

גם בשורה השנייה – 1 כפול 1010 שווה 1010, את התוצאה אנחנו רושמים בשורה השנייה עם הזזה של ספרה אחת שמאליה.

לאחר מכן מוחברים את התוצאות חיבור פשוט ומת�בלת התוצאה 11110 (או בסיס עשר – $2+4+8+16=30$).

תופעה מעניינת – כפל בחזקות של 2: כאשר לוקחים מספר בסיס 10, וכופלים אותו בעשר, התוצאה היא המספר המקורי מוזז בספרה אחת שמאליה, כאשר במקומות היכי ימוי נכנסת הספרה 0. לדוגמה, $52 \times 10 = 520$.



דוגמה נוספת, עם חזקה יותר גבוהה של 10: $52 \times 100 = 5200$.

אותה התופעה מתרכשת בסיס 2, כאשר כופלים במספר שהוא שתיים או חזקה של שתיים. על כל חזקה של שתיים, מזינים ספרה אחת שמאליה ומוסיפים את הספרה 0.

לדוגמה, כפל בשתיים: $11 \times 10 = 110$.

כפל באربע: $11 \times 100 = 1100$.

כפל בשמונה: $11 \times 1000 = 11000$.

וכן הלאה.

ביצוע פעולות כפל במסיס הקסדצימלי: לוח הכפל במסיס 16 כולל $16 \times 16 = 256$ איברים, כלומר 256 איברים. אך אל דאגה – אין צורך לזכור אותו בעל פה. הזרך הפשטת יותר היא להמיר את המספרים במסיס בינארי ולהמשיך משם. לדוגמה, כפל של C כפול 5, לפי טבלת ההמרה מתתקבל בקלות 1100 כפול 101 והזרך משם ברורה.

תרגיל 2.4: כפל



A	B	$A \times B$
111	1011	
1001	1001	
100	1010	
100	111	
1110	1100	
1001	11	
111	1110	



חילוק (הרחבה)

בחילוק של מספרים בינאריים יש שני חוקים:

$$1/1 = 1$$

$$0/1 = 0$$

כשמחלקים מספר במספר BINARIYI אחר, התהליך זהה לחילוק מספרים עשרוניים. בכל פעם רושמים במקום אחד את המנה ובמקום אחר את השארית, ומוסיפים את הספרות הבאות לשארית עד לקבלת מנה חדשה. התהליך נגמר כאשר ישן יותר ספרות להוסיף.

nbetz' dogma - 10110₂/101₂:



$$\begin{array}{r}
 100 \\
 \overline{10110} \quad | \quad 101 \\
 -101 \\
 \hline
 010
 \end{array}$$

התוצאה היא 100, שארית 10.

תרגום התרגיל לבסיס עשר – 22 לחלק ל-5, התוצאה היא 4, שארית 2.

תרגיל 2.5: חילוק מספרים בינאריים



A	B	A/B	שארית
100000	110		
10101	110		
100110	1001		
10101	100		
101010	10		
111001	110		
101000	1001		

יצוג מספרים על-ידי כמות מוגדרת של ביטים

זכרון המחשב בניו מתאים, שבתוכם שמורים מספרים בינאריים. כל ספרה בינארית (0 או 1) נקראת **סיבית (Bit)**. זה קיצור של המילים ספרה בינארית (Binary digit). נניח עכשו הנחת יסוד, שאומרת שיש לנו מגבלה על כמות הביטים שאנו יכולים להשתמש בה בשבייל ליצג מספר. במחשבים זהה מגבלה מאוד מעשית, כיון שהמחשבים שמורים במספרים בהתאם בגודל קבוע. הגודלים המקובלים הם 8, 16, 32 או 64 ביטים. ככלומר המגבלה על כמות הביטים ליצוג מספר היא חלק בסיסי באופן שבו מחשבים פועלים.

נניח שעומד לרשותנו תא בעל N ביטים. מהו המספר הכי גדול שאנו יכולים לשמור בתוכו? נסתכל על הטבלה הבאה, שמרכזת את המספר הכי גדול שאפשר ליצג על-ידי כמות ביטים שבין 1 ל-8:

כמות ביטים שיש ב-N	המספר הכי גדול שאפשר ליצג	תרגום המספר לעשרוני
1	1	1
2	11	3
3	111	7
4	1111	15
5	11111	31
6	111111	63
7	1111111	127
8	11111111	255

אנו יכולים להמשיך מעבר לשמונה ביטים, ועדיין תישמר החוקיות הבאה: באופן כללי, על-ידי N ביטים אפשר ליצג מספר שערכו הוא לכל היותר $2^N - 1$.

מה קורה אם אנחנו מנסים ליצג מספר שהוא יותר גדול מהמספר הכי גדול שאפשר ליצג על-ידי N ביטים? נניח, לבצע את פעולה החשבון $255 + 1$, אבל עם תא זיכרון בגודל 8 ביטים?

$$\begin{array}{r}
 & 11111111 \\
 + & \underline{00000001} \\
 (1)00000000
 \end{array}$$

את התוצאה אי אפשר לשמור על-ידי 8 ביטים! כל שמונה הביטים הראשונים מתאפסים, ומתקבל נשא. הדרך שבה המחשב מטפל במקרים כאלה היא כזו: הזיכרון שלנו, שלפני כן הכיל את הערך 11111111, מכיל עכשו סדרה של שמונה אפסים. המחשב מדליק במקום אחר ביט, שאומר שהוא נשא בפעולה האחרונה (נלמד עליו בהמשך). זהו. מעכשו בזכרון שלנו יש 00000000. ככלומר, מבחינת המחשב, כמשמעותו ליצוג של 8 ביטים, אז המשווה $255 + 1 = 0$ היא נכונה. עליינו להבין כי כך המחשב עובד, ונctrיך להתאים את עצמנו כדי שלא יקרו לנו טעויות כאלה.

יצוג מספרים שליליים

עד כה חשבנו על מספרים בינאריים רק בתור מספרים עם ערכים חיוביים. המספר הבינארי 0000 מייצג אפס, 0001 מייצג שתים ועוד אלה עד אינסוף. ליצוג מספרים בדרך זו קוראים **unsigned** – קלומר חסרי סימן. אבל, איך, 0010 מייצג שתים ועוד אלה עד אינסוף?

מה אם נרצה ליצוג מספרים שליליים?

בחלק זה נלמד את השיטות הנפוצות ליצוג מספרים עם סימן – קלומר **signed**.

שיטת גודל וסימן

נניח שעומדים לרשותנו N ביטים בכל תא, ואנחנו רוצים ליצג גם מספרים שליליים. בשיטת גודל וסימן, הבית השמאלי ביותר מייצג את הסימן ויתר הביטים את הגודל. נפרט:

אם הבית השמאלי ביותר הוא 0 – המספר הוא חיובי. אם הבית השמאלי הוא 1 – המספר הוא שלילי. יתר הביטים מייצגים את הגודל, קלומר מתרגם אותם למספר כמו שעשינו עד עכשיו עם מספרים **unsigned**, ואז לפי הבית השמאלי כותבים את הסימן לפני המספר.

נראה לדוגמה את המספר: 0011. חשבו, האם הוא שלילי או חיובי?

הבית השמאלי ביותר הוא 0, ולכן המספר חיובי. יתר הביטים הם 011, לכן הגודל הוא 3. בשיטת גודל וסימן, 0011 מייצג את הערך החיובי 3.

לעומת זאת במספר 1011, הבית השמאלי ביותר הוא 1, ולכן המספר שלילי. יתר הביטים הם 011, ומכאן שהגודל הוא 3. לכן, הרץ 1011 מייצג את הערך מינוס 3.

בשיטת זו, המספר היכי גבוה שאנחנו יכולים ליצג עלי-ידי 4 ביטים הוא 0111, קלומר 7, ואילו המספר היכי נמוך הוא 1111, קלומר מינוס 7.

שימוש לב שיש מספר שיש לו שני ייצוגים - גם 0000 וגם 1000 שווים אפס!



מספר עשרוני	ייצוג בשיטת גודל וסימן	מספר עשרוני	ייצוג בשיטת גודל וסימן
7	0111	-7	1111
6	0110	-6	1110
5	0101	-5	1101
4	0100	-4	1100
3	0011	-3	1011
2	0010	-2	1010
1	0001	-1	1001
0	0000	-0	1000

היתרון המרכזי של שיטת גודל וסימן הוא הפשטות שללה. קל לחסית להסתכל על מספר ולקבוע מה הערך שלו, וכן קל להשוות בין שני מספרים.

החיסרון המרכזי של השיטה והוא שמסובך לבצע תרגילי חישוב. נסתכל לדוגמה על 3 ועוד (-3):

$$\begin{array}{r} 0011 \\ + \underline{1011} \\ 1110 \end{array}$$

למרות שהתרגיל צריך לחתת תוצאה של אפס, תוצאה החיבור היא הייצוג של (-6). עקב בעיה זו, שיטת הסימן והגודל לא נפוצה בשימוש.

שיטת המשלימים לאחת

שיטת המשלימים לאחת (One's complement) אמורה להתגבר על החיסרון העיקרי של שיטת הגודל והסימן, ביצוע פעולות חישוב נכונות.

בשיטת זו, מוצאים את הנגדי של מספר בינארי על-ידי הפיכת כל בית. בית שערכו 0 הופך ל-1, ואילו בית שערכו 1 הופך ל-0. לדוגמה, 0001 מייצג את הערך "1". הערך "-1" מייצג על-ידי הפיכת כל הביטים – 1110.

ייצוג המספרים מ-0 עד 7 והנגדיים שלהם:

מספר עשרוני	ייצוג בשיטת המשלימים ל-1	מספר עשרוני	ייצוג בשיטת המשלימים ל-1
7	0111	-7	1000
6	0110	-6	1001
5	0101	-5	1010
4	0100	-4	1011
3	0011	-3	1100
2	0010	-2	1101
1	0001	-1	1110
0	0000	-0	1111

שימוש לב לתופעה שהתרחשה מליליה - הבית השמאלי הוא בית הסימן. כשהוא 0 המספר חיובי וכשהוא 1 המספר שלילי. כפי שאפשר לראות, חיבור של כל מספר עם הנגדי שלו נותן 1111, שבשיטה זו שקול לערך 0. ככלمر התגברנו על מכשול אחד.



בשיטת המשלימים לאחת, כדי שגם יתר פעולות החישוב יתנו תוצאות נכונות, משתמשים בחוק הבא: אם לתוצאה הפעולה יש נשא, מחברים אותו לבית הימני ביותר.

נסתכל על דוגמה, התרגיל 5 פחות 2. ניתן לכתוב את התרגיל זה גם בתור 5 ועוד (-2), כך:



$$\begin{array}{r}
 & 1 & 1 \\
 & 0101 \\
 + & \underline{1101} \\
 (1)0010 \\
 0011
 \end{array}$$

ניתן לראות, שאחרי שהעבכנו את הנשא אל הבית הימני ביותר, קיבלנו תשובה נכונה – פלוס 3.

היתרונות של שיטת המשלים לאחת, הם פשוט מואוד למצוא את המספר הנגדי (רק צריך להפוך ביטים) ופעולות החשבון יוצאות נכון. החסרונות של השיטה הם הסיכון שבתוספת הנשא לביט הימני, אבל בעיקר – העובדה שיש שתי דרכים לייצג את הערך אפס. כדי ללמד את המחשב ש- $1111 - 0000 = 1111$ צריך להשיק עוד זמן ומאמץ, וудיף פשוט לעבור לשיטת המשלים לשתיים.

שיטת המשלים לשתיים

שיטת המשלים לשתיים דומה לשיטת המשלים לאחת בכך שהוא כל בית 0 ל-1 ולהיפך. אלא שבמשלים לשתיים מבצעים בסוף תהליך הפיכת הביטים עוד פעולה – מוסיפים 1 לתוצאה (לכן השיטה נקראת המשלים לשתיים, כי אם מסתכלים על הבית הימני אז כביכול הוספנו לו שתים).

נראה כיצד השיטה עובדת, ואו נבין את ההשלכות המענטיות של התהליך.

כדוגמה, נמצא את הייצוג של מינוס 6, בייצוג שלו עליידי 8 ביטים:



הייצוג של 6 הוא 00000110.

לאחר הפיכת הביטים תוצאה הביניים היא 11111001.

לאחר הוספת 1, התוצאה היא 11111010. זהו הייצוג של מינוס 6 בשיטת המשלים לשתיים, בייצוג עליידי 8 ביטים. אם היינו רוצים לייצג מינוס 6 עליידי 16 ביטים, היינו צריכים להוסיף להוסף אחדות בהתחלה – 1010 1111 1111 1111 (מעכשיו, כשנרצה לרשום מספרים ארוכים בבינארי, נפרק אותם לקבוצות של ארבע ספרות מסוימות של נוחות קריאה). כדי לייצג מינוס 6 עליידי 32 ביטים נוסיף עוד שישה عشر אחדות בהתחלה.

נבדוק את התוצאה שקיבלנו. 6 ועוד מינוס 6, התוצאה צריכה להיות אפס.

$$\begin{array}{r}
 + 00000110 \\
 \underline{11111010} \\
 (1)00000000
 \end{array}$$

וכפי שרואים, כל שМОנת הביטים אכן התאפסו (זיכרו – הד' שמאך הוא נשא והוא אינו נכנס בשМОנת הביטים).

נסתכל על הייצוג של כמה מספרים בינהירים בשיטת המשלים לשתיים. לטוּבת המשך, אנחנו נסתכל על הייצוג ב-8 ביטים, למروת שבשביל לייצג את המספרים שבטבלה אפשר היה להסתפק גם ב-4 ביטים.

מספר עשרוני	ייצוג בשיטת המשלים ל-2	מספר עשרוני	ייצוג בשיטת המשלים ל-2
7	0000 0111	-1	1111 1111
6	0000 0110	-2	1111 1110
5	0000 0101	-3	1111 1101
4	0000 0100	-4	1111 1100
3	0000 0011	-5	1111 1011
2	0000 0010	-6	1111 1010
1	0000 0001	-7	1111 1001
0	0000 0000	-8	1111 1000

כפי שאנו רואים, בשיטה זו יש רק ייצוג אחד לאפס – 0000 0000. זהו יתרונו ממשמעותי על שיטת המשלים לאחד. בנוסף, אנחנו יכולים לייצג גם את מינוס 8!

באופן כללי, באמצעות N ביטים ניתן לייצג בשיטת המשלים לשתיים את טווח המספרים שבין (-2^{N-1}) לבין (2^{N-1}) . כך לדוגמה, בעזרת 8 ביטים אפשר לייצג את המספרים שבין 127 ל-128. בעזרת 16 ביטים אפשר לייצג מספרים בתחום שבין 32,767 ל-32,768.

לטוּוח הערכים הזה יש חשיבות מעשית גדולה – כל פעולה חשבונית שההוצאה של הורגת מתחום הערכים שניתן לייצוג, תגרום להחזרת תשובה שגויה.

כדי לסכם את הדיון על שיטת המשלים לשתיים, נותר לנו רק לראות איך מבצעים המרה של מספרים בינאריים (בשיטת המשלים לשתיים) למספרים עשרוניים. השיטה היא פשוטה:

- ראשית, אם המספר לא מכיל בדיק 8, 16, 32, 64 וכו' ביטים, אז מוסיפים אפסים מצד שמאל של המספר עד שmagiim לכמה זו של ביטים.
- אם המספר חיובי (הבית השמאלי ביותר שלו הוא 0) אז ההמרה לעשרוני היא בדיק כמו שלמדנו בעבר - לכל בית יש מקום, וכל מקום קובע חזקה אחרת של 2.
- אם המספר שלילי (הבית השמאלי ביותר הוא 1), אז ממיר את המספר למספר חיובי בשיטת המשלים ל-2, מחשבים את הערך שלו כמו שעשווים למספרים חיוביים, ושים מינוס לפני התוצאה.

לדוגמה, המספר 10111111. הבית השמאלי הוא 1, לכן נמיר את המספר בשיטת המשלים ל-2:



מספר מקורי:	10111111
משלים ל-1:	01000000
משלים ל-2:	01000001

המרת התוצאה למספר עשרוני:

$$2^0 + 2^6 = 65$$

אסור לנו לשכח שהמספר הוא שלילי – נוסף סימן מינוס ונגיע לתוצאה: (65)-

תרגיל 2.6: המשלים לשתיים



שיטת המשלים ל-2 משמשת לייצוג מספרים בזיכרון המחשב. תרגמו את המספרים הבאים לייצוג הבינארי שלהם על ידי 8 ביטים בשיטת המשלים ל-2. הדרכה: הוסיפו במידת הצורך אפסים לפני המספר כדי להראות את הייצוג בזיכרון המחשב. לדוגמה המספר 12 יוזג 00001100 בבסיס 2

בינארי	דצימלי	בינארי	דצימלי
-9			247
-128			128
-94			162
-102			154
-1			255

מה ניתן להסיק מתרגיל זה, לגבי הקשר בין הייצוג הבינארי של מספרים חיוביים ושליליים?

או איך מנצחים את חואן?

... ומה הקשר של המשחק Pearls3 לבסיסי ספירה?

המשחק Pearls3 הוא וריאציה של משחק Nim, עליהם אפשר לקרוא באינטרנט (לדוגמה ויקיפדיה והשו Nim-Sum בגוגל). (<https://en.wikipedia.org/wiki/Nim>)

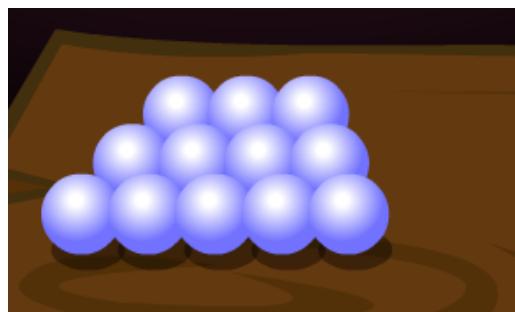
קיימת טכניקה שמבטיחה את הנצחון במשחק, אנחנו נפרט עליה כאן. אם אתם סקרנים מדוע הטכניקה זו עובדת, ישנו הסבר מתמטי, שהינו מחוץ להיקף של ספר זה, אך הוא מופיע בוויקיפדיה ובמקומות אחרים באינטרנט.

כדי להבטיח ניצחון במשחק, יש לבצע את הצעדים הבאים:

1. יש לתרגם כל מספר של "פנינים" לייצוג הבינארי שלו (לדוגמה 7 פנינים - 111 ביבנاري).
2. לרשום את כל הייצוגים הבינאריים אחד מעל השני ולבצע חיבור. אך בלי להעביר נשא (החיבור של $1+1$ הוא 0 ולא 10)
3. אם הסכום (המכונה Nim Sum) הוא אפס – אין צורך לעשות דבר, רק להעביר את התור לשחקן השני. אחרת, יש להחסיר מספר כלשהו של פנינים מהחתה השורורה כך שהסכום יהיה אפס.

דוגמאות:

חואן מראה לנו 3 שורות פנינים: 3, 4 ו-5 פנינים בהתאם.



ייצוג בינארי (נשתמש באربע ספרות ביבנאריות, המתאימות לייצוג מספרים עד 15, כך שסביר שהמספר יתחל באפסים):

(3) 0011

(4) 0100

(5) 0101

נחשב את הסכום. כזכור מסכמים כל טור בנפרד, ולא מעבירים נשא לטור הבא:

0010

כלומר הסכום כרגע אינו אפס. כמה פנינים אנחנו צריכים להסיר – ומהיכן – כדי שהסכום יהיה אפס?

... אם נסיר 2 פנינים מהשורה הראשונה תשאר בה פנינה אחת, ומצב הפנינים הבינאריות יהיה:

(1) 0001

(4) 0100

(5) 0101

הסכום:

0000

נמשיך בתהליך עד שנגיע למספר בו אנו יכולים להשאיר רק פניה אחת ולנזהה.

תרגיל 2.7: נצחו עשרה משחקים

נצחו את החואן בעשרת השלבים הראשונים! אתגר - נסו לא להפסיק באתגר משחק ולשמור על מזון נצחונות מושלים.



ייצוג מידע במחשב

בתחילת הפרק, כשניסינו להסביר למה כדאי ללמידה בסיסי ספירה, נתנו הבטחה מעורפלת שבזורה בסיס בינהרי והקסדצימלי קל יותר לעבוד עם מחשבים. עכשו הגיע הזמן לפרק את השטר – נעבר לדבר על מה קורה בתוך המחשב, על הדרך בה מידע מיוצג במחשב. כך, נוכל לראות את השימושות של החומר שלמדנו.

סיבית – Bit

היחידה הקטנה ביותר של מידע במחשב היא בית בודד. בית מקבל ערך אפס או אחד. באמצעות בית בודד ניתן לייצג כל ערכיהם שונים זה מזה. לדוגמה, אפס או אחד,אמת או שקר, למעלה או למטה, יירוק או אדום, 340 או 519. כל שני ערכיהם השונים זה זהה ניתנים לייצוג על-ידי בית בודד, אך ניתן לייצג באמצעות רק שני ערכים שונים.



אם כך, כשהסתכלים על בית, איך יודעים אילו שני ערכיהם הוא מייצג? התשובה היא, שאין דרך לדעת. למשל, ניתן לפחות את המידע השמור על-ידי בית במגוון דרכיהם, וכל דרך תיתן פרשנות אחרת. העיקר, בכתב קוד, הוא להיות עקיبي לגבי הפרשנות שאנו נותנים לביטים השונים.

כדי להדגים את הנושא, נזכיר בחלק שבו דנו בייצוג מספרים בשיטה הבינארית. המספר 1100 בבסיס 2 יכול להיות מתורגם לבסיס עשרוני כ-12, אבל גם כ-5inos (היווצרו בשיטת המשלים לשתיים). הערך אותו רצף הביטים הזה מייצג תלוי בפרשנות שלנו – האם מדובר במספר `signed` או `unsigned`.

כל המידע שיש בזכרון מחשב, יהיה משמשו אשר תהיה, אוור תוק שימוש באחדות ואפסים בלבד. בזכרון המחשב לא קיימת הספהה 2, לא קיים מינוס, לא קיים גדול או קטן. כל מה שקיים בזכרון המחשב הוא רק רצף של אחדות ואפסים, שעלה פי הפרשנות שהמחשב נותן להם אפשרות ייצוג של כל דבר – החל מחישובים מתמטיים וכלה ברטוי וידואן.

כיוון שבאמצעות בית בודד ניתן לייצג רק שני ערכים, השימושocabits בוודדים הוא בעל שימושים מוגבלים. במקרה זה, לרוב השימושים משתמשים באוספים של ביטים.

Nibble – Nibble

Nibble הוא אוסף של ארבעה ביטים. Nibble יכול לייצג 2 בחזקת 4 ערכים שונים, כלומר 16 ערכים. במקרה של מספר הקסדצימלי, Nibble יכול לייצג את הערכים 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F חשיבותו – כאשר אנחנו מסתכלים בזכרון המחשב, לא נוח לקרוא אותו בbasis 2, אחדות ואפסים בלבד. ייצוג בbasis 16 ייחוס כל ארבעה ביטים לספהה אחת ויקל על הקראיה והזכירה. נראה דוגמה.



נניח שזכרון המחשב כולל את הרצף הבא – נסו לזכור אותו בעל פה:



1101 1110 1010 1101 1100 0000 1101 1110

נתרגם את הרצף למספרים הקסדצימליים לפי טבלת ההמרה המוכרת:

בינארי	הקסדצימלי
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

ונקבל:

1101 1110 1010 1101 1100 0000 1101 1110

D E A D C 0 D E

...האם זכירת המידע קלה יותר עכשו?

בית – Byte

בית (Byte) הוא אוסף של שמונה ביטים, והוא גם היחידה הקטנה ביותר שיש לה כתובות מסוימת – הזיכרון מחולק לכתובות, וכל כתובות היא של בית יחיד. משמעו הדבר היא שהיחיד הזיכרון הקטנה ביותר שניתן לגשת אליה היא בית. כדי לגשת ליחידה קטנה יותר, נניח ביט, נדרש לקרוא את כל הבית שמכיל את הבית הרצוי, ואז על-ידי פעולה שנראית מיסוך ביטים (NELMD עלייה בהמשך, בחלק שמסביר פקודות לוגיות) אפשר לבדוק את הביט.



הBITS בתוך כל בית ממושפרים לפי הסדר הבא:



כאשר ניתן להתייחס אליהם גם כצירוף של שני Nibbles:



מילה – Word

מילה (Word) היא אוסף של 16 ביטים, או בדיק שני בתים. באמצעות מילה ניתן לייצג שתיים בחזקת 16, או

, 65,536 ערכים שונים. באסםבי 16 ביט, אליו מתיחס ספר זה, נשתמש במסתנים בגודל מילה בעיקר כדי

לשמר ערכי מספרים או כתובות בזיכרון.



מספר של ביטים בתוך מילה וחלוקתם לבתים:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

High Order Byte

Low Order Byte

מילה כפולה – Double Word

מילה כפולה (Double Word) או בקיצור **DWORD** היא אוסף של 32 ביטים, או שני מילים, או ארבעה

בתים. מספר הערכים ש-Word יכול לקבל הוא 2 בחזקת 32, מה שמאפשר להחזיק מספרים

בתחום 0 עד 4,294,967,295 או מספרים Signed בתחום -2,147,482,648 עד

.2,147,482,647



קוד ASCII

קוד ASCII, או בקיצור American Standard Code for Information Interchange, הוא הקוד

הנפוץ לייצוג אותיות ותווים. הקוד משתמש ב-7 ביטים כדי לייצג 128 תווים, מכאן שלכל תוו מותאם מספר בין

.0 ל-127



		Regular ASCII Chart	character	codes	0 – 127			
000	<nul>	016 ► <dle>	032 sp	048 0	064 ¶	080 P	096 `	112 p
001 ☺	<soh>	017 ▲ <dc1>	033 !	049 1	065 A	081 Q	097 a	113 q
002 ☻	<stx>	018 ♫ <dc2>	034 "	050 2	066 B	082 R	098 b	114 r
003 ☻	<etx>	019 !! <dc3>	035 #	051 3	067 C	083 S	099 c	115 s
004 ☹	<eot>	020 ¶ <dc4>	036 §	052 4	068 D	084 T	100 d	116 t
005 ☹	<eng>	021 § <nak>	037 %	053 5	069 E	085 U	101 e	117 u
006 ☹	<ack>	022 = <syn>	038 &	054 6	070 F	086 V	102 f	118 v
007 ☺	<bel>	023 § <etb>	039 ,	055 7	071 G	087 W	103 g	119 w
008 ☻	<bs>	024 ↑ <can>	040 <	056 8	072 H	088 X	104 h	120 x
009	<tab>	025 ↓ 	041 >	057 9	073 I	089 Y	105 i	121 y
010	<lf>	026 <eof>	042 *	058 :	074 J	090 Z	106 j	122 z
011 ☺	<vt>	027 ← <esc>	043 +	059 ;	075 K	091 [107 k	123 {
012 ☻	<np>	028 ← <fs>	044 ,	060 <	076 L	092 \	108 l	124 ;
013	<cr>	029 → <gs>	045 -	061 =	077 M	093]	109 m	125 }
014 ☻	<so>	030 ^ <rs>	046 .	062 >	078 N	094 ^	110 n	126 ~
015 ☹	<si>	031 ▼ <us>	047 /	063 ?	079 O	095 _	111 o	127 △

לדוגמה, הטקסט "HELLO WORLD!" מיוצג קוד ASCII על-ידי רצף המספרים:



72 69 76 76 79 32 87 79 82 76 68 33

H E L L O W O R L D !

או בייצוג הקסדצימלי:

48 45 4C 4C 4F 20 57 4F 52 4C 44 21

אם נעתיק את הטקסט הנ"ל אל זיכרון המחשב (במהשך נלמד איך עושים זאת) נקבל את הייצוג הבא בזיכרון:

```
ds:0000 48 45 4C 4C 4F 20 57 4F HELLO WO
ds:0008 52 4C 44 21 00 00 00 00 RLD!
ds:0010 00 00 00 00 00 00 00 00
ds:0018 00 00 00 00 00 00 00 00
ds:0020 00 00 00 00 00 00 00 00
```

סיכום

בפרק זה למדנו לעבוד עם מספרים בשיטות ספירה שונות מאשר השיטה העשרונית שאנחנו רגילים אליה. התמקדנו בשתי שיטות: השיטה הבינארית, שמשמשת לייצוג של ערכים בזכרון המחשב, והשיטה הקסדצימלית, שמקלה علينا לקרוא ערכים בינאריים.

תרגלנו המרות מבסיס לבסיס וראינו שיש קיצור דרך להמרה בין בסיס שתים לבסיס שש עשרה.

לאחר מכן עברנו לביצוע של פעולות חיבור-חיסור, כפל וחילוק-בבסיסים שאינם בסיס עשר.

משם עברנו לייצוג של מספרים שליליים. סקרנו מספר שיטות: שיטת גודל וסימן, שיטת המשלים לאחד ושיטת המשלים לשתיים, שפותחה כדי לענות על החסרונות של השיטות הקודמות.

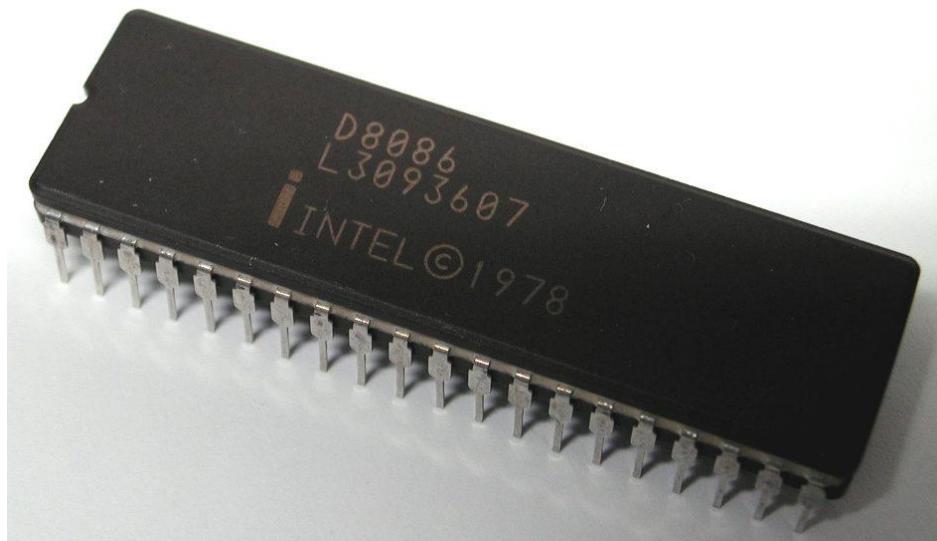
בסוף פרק זה, סקרנו איך מחשב שומר ערכים שונים בזכרון וראינו את היתרונות של שימוש בבסיס בינארי והקסדצימלי. כתע אנחנו מבין איך מספרים מסוגים שונים מיוצגים במחשב, ואנחנו בשלים לעבור לפרק הבא בו נרחיב ונתעמקמבנה המחשב.

פרק 3 – ארגון המחשב

מבוא

אסambilי היא שפת התוכנה שעובדת בזורה הקרובה ביותר מול החומרה של המחשב. רק אסambilר קטן מפheid בין הקוד שאתם כתובים באסambilי לבין שפת מכונה, ובמהשך גם מתנשו בתרגום מאסambilי לשפת מכונה. לכן כדי לכתוב תוכנית, אפילו **בסיית**, בשפת אסambilי, נדרש הבנה של ארגון המחשב והחומרה. כדי לעסוק בסיביר באופן מקצועי צריך כישורים לביצוע פעולות מתקדמות כמו מציאת אגיים בתוכנה, הקטנת גודל הזיכרון שבודק תופס או העלאת מהירות הריצה של תוכנה. לטובות פעולות מתקדמות كالה נדרש הבנה **טובה** של אופן פעולה המחשב והקשר בין החומרה לתוכנה.

אנחנו נלמד על ארגון המחשב באמצעות ניתוח הדרך שבה מאורגן מעבד משפחתי 80x86 של אינטל. משפחת הד-86x80 כוללת מספר מעבדים, כאשר במקום הד-X ישנה ספרה שמצוינת את דור המעבד. המעבד הראשון למשפחה זו, שנקרא 8086, יוצר לראשונה בשנת 1978. אם אתם קוראים את הספר הזה, סביר להניח שהמעבד יצא לפני שנולדתם.



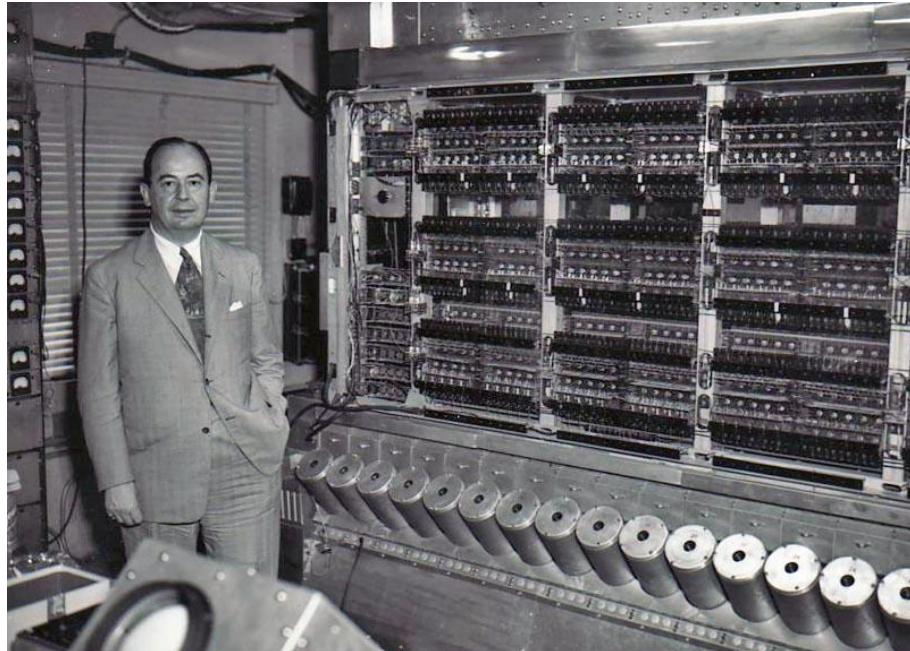
ראשית, ראוי שנשאול את עצמנו – איזו תועלת יכולה להיות ללימוד מעבד כל כך מושן? הרי חומרת המחשבים משתנה כל מספר שנים בודדות. איך אפשר ללמוד משהו מעשי לעולם הסיביר באמצעות מעבד שמקורו בדף ההיסטוריה?

ובכן, ישנן כמה תשובות לשאלת זו:

- מעבד הד-86x80 מאורגן על פי ארכיטקטורת פון נוימן, עליה נפרט בהמשך. בהיבט זה אין הוא שונה מהמעבדים המתקדמים ביותר (נכון לזמן כתיבת הספר).
- כל משפחת המעבדים של אינטל שומרת על תאימות לאחר עם מעבד הד-80x80. כמובן, אתם יכולים לכתוב פקודת אסambilי שתרוץ על מעבד 8086, והמעבד החדש ביותר של אינטל יוכל להריץ אותה.
- בשוביל ללמידה את ארגון המחשב צריך להתחיל ממקום כלשהו, ומעבד הד-86x80 הוא מקום טוב להתחיל ממנו.

מכונת פון נוימן – Von Neumann Machine

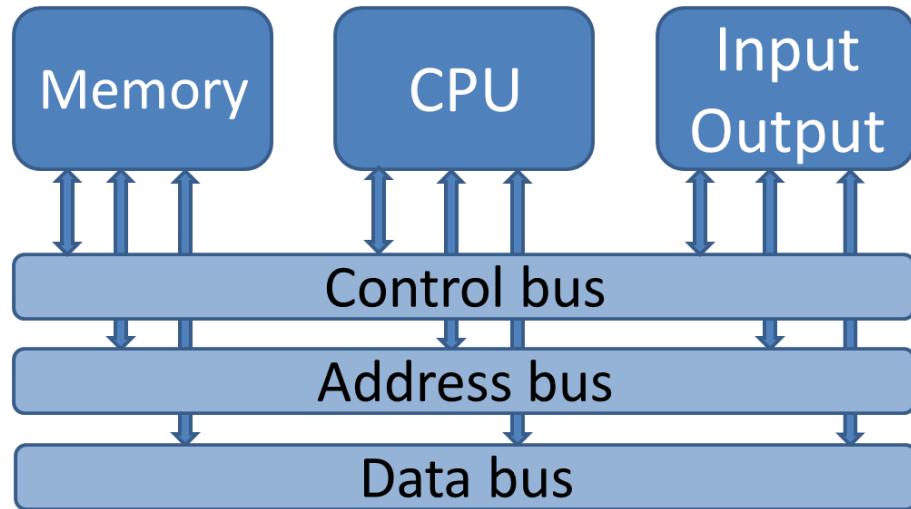
התכנון הבסיסי של מחשב נקרא ארכיטקטורה. ג'ון פון נוימן היה חלוץ בתכנון מחשבים, וניתן לו הเครดיט עבור הארכיטקטורה של רוב המחשבים שאנחנו משתמשים בהם היום. **ארכיטקטורת פון נוימן (או באנגלית Von Neumann Architecture – VNA)** כוללת שלוש אבני בניין מרכזיות: יחידת העיבוד המרכזית (Memory) (זיכרון), (Central Processing Unit – CPU) וקלט/פלט (I/O).



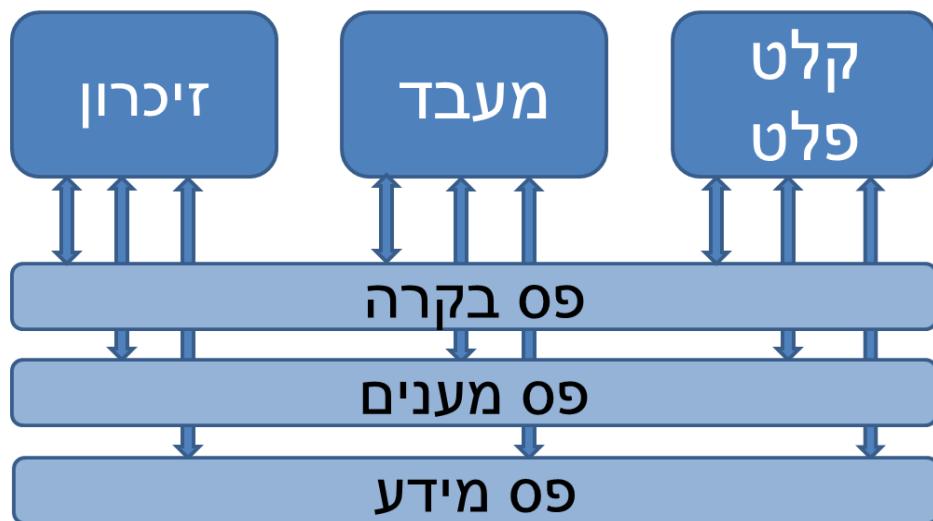
ג'ון פון נוימן (1903–1957) והמחשב הדיגיטלי הראשון

במעבדים מבוססי ארכיטקטורת VNA, כמו משפחת ה-i86x80, ייחידת העיבוד המרכזית מבצעת את כל החישובים. מידע והוראות למעבד (מה שנקרא קוד התוכנית) נמצאים בזיכרון עד שהם נדרשים על-ידי המעבד. מבחינות המעבד, ייחידות הקלט / פלט נראות כמו זיכרון, כיון שהמעבד יכול לשולח אליו מידע ולקרוא מהן מידע. ההבדל העיקרי בין מקום בזיכרון לבין מקום קלט / פלט, הוא שיחידות הקלט / פלט בדרך כלל משוויכות להתקנים שנמצאים בעולם החיצוני למעבד (כגון מקלדת).

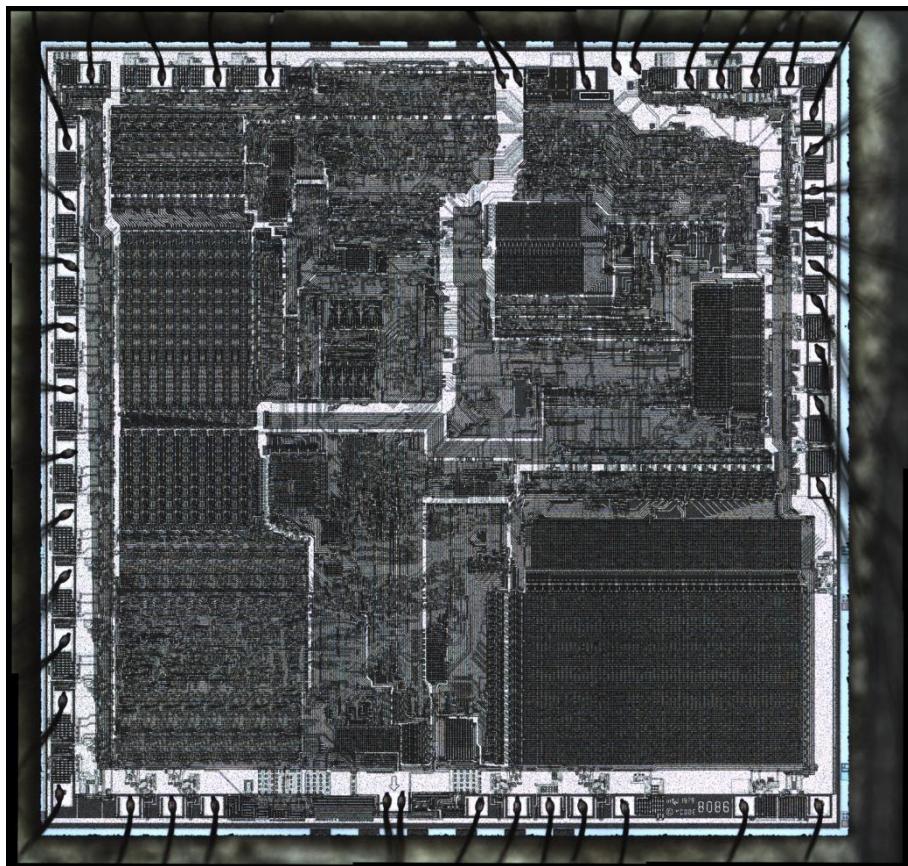
יחידת העיבוד המרכזית של המעבד מתחברת עם הזיכרון ועם ייחדות הקלט / פלט באמצעות קווי תקשורת שנקראים **Bus**, או בעברית **פסים**. יש סוגים שונים של קווי תקשורת, שלכל אחד יש תפקיד ייחודי.



שמות הרכיבים ב-**VNA**, במושגים המקובלים באנגלית



המושגים המקובלים בעברית



תמונה רנטגן של מעבד הדגם 8086 (התמונה בהגדלה, הגודל האמתי הוא 33 מ"מ מרובעים). הקווים הבוהרים הם הפסים השוניים. המרובע מצד ימין למטה הוא זיכרון פנימי של המעבד. החוטים הכהים המוחברים להיקף הם קווי התקשרות לרכיבי ה-I/O שמחוץ למעבד.

פסי המערכת – SYSTEM BUSES

פסי המערכת (System Buses) מחברים את הרכיבים השונים של מכונת A. במשפחת ה-80x86 ישנו שלושה פסים עיקריים: פס הנתונים – DATA BUS, פס מענים – ADDRESS BUS ופס הבקרה – CONTROL BUS. כל פס הוא למעשה אוסף קווי חשמל שימושיים בין הרכיבים השונים. האותות המועברים הם רמות מתח שונים – יש מתח שמייצג 0 ויש מתח שמייצג 1. כיוון שរמות המתח שונות בין מעבד למעבד, נתייחס אליו רק כ-0 או כ-1.

בשביל מה צריך אוסף של פסי מערכת? למה לא מספיק פס אחד, נניח, לתקשרות בין המעבד לזיכרון?

נניח שהמעבד שלח לזכרון "1000h". האם מדובר בכתבota בזיכרון, או במידע שיש לשמר אותו בזיכרון? תפקודם של הפסים השונים הוא לאפשר לתת פרשנות ברורה לכל נתון והוראה שעוברבים.

פס נתונים – DATA BUS

פס הנתונים משמש להעברת נתונים בין המרכיבים השונים של המחשב. גודל פס הנתונים משתנה בין משפחות שונות של מעבדים. במחשבים סטנדרטיים פס הנתונים מכיל 16, 32 או 64 קווים.

מחשב שיש לו פס נתונים של 16 קווים מסוגל להעביר 16 ביטים בבה אחת. אין זה אומר שמעבד שמהוצר לפס של 16 ביטים מוגבל לעיבוד נתונים של 16 ביטים. זה רק אומר שהמעבד יכול לגשת ל-16 ביטים של זיכרון בכל פעולת קריאה או כתיבה של מידע.

ראוי לציין שמעבד שיכל לגשת ל-16, 32 או 64 ביטים בבה אחת, מסוגל לגשת גם לכמות קטנה יותר של מידע – לדוגמה, בית אחד. ככלומר כל מה שניתן לעשות עם פס נתונים צר, ניתן לעשות גם עם פס נתונים רחב יותר, אך פס נתונים רחב יותר מאפשר גישה מהירה יותר לזכרון. כיוון שככל פניה לזכרון לוקחת זמן, מעבד שיש לו פס נתונים רחב יותר יוכל לכתוב ולקראן מהזיכרון בקצב מהיר יותר ולכנן לעבוד מהר יותר מעבד זהה שיש לו פס נתונים עם פחות קווים.

פס מענים – ADDRESS BUS

פס הנתונים מעביר מידע בין אוזור מוגדר בזכרון (או ביחידה O/I) לבין המעבד. השאלה היא – מרחב הזיכרון הוא גדול, איך יודעים לאן לבדוק לגשת בזכרון? פס המענים עונה על שאלה זו. כדי להפריד בין מקומות שונים בזכרון, לכל בית בזכרון יש כתובת נפרדת (כזכור, בית הוא היחידה הקטנה ביותר של זיכרון שיש לה כתובת משלה).

כאשר התוכנה רוצה לפנות מקום כלשהו בזכרון, או ביחידה O/I כלשהי, היא מכניסה את כתובת הזיכרון המבוקש לתוכה פס המענים. רכיבים אלקטרוניים ששולטים על הזיכרון מזוהים את הכתובת שבס המענים ודואגים לשולח למעבד את המידע בכתובת המבוקש, או לכתוב לזכרון בכתובת הנ"ל את המידע ששלח המעבד.

כמוות הקווים בפס המענים קובעת את גודל מרחב הכתובות שהמעבד יכול לפנות אליו. לדוגמה, מעבד שיש לו שני קווים בפס המענים יכול לפנות ארבע כתובות בלבד: 00,01,10,11. מעבד שיש לו ח קווים בפס המענים, יכול לפנות לשתיים בחזקת שתיים שונות. לדוגמה, מעבד ה-8086, יש 20 קווים בפס המענים. ככלומר מרחב הכתובות שלו הוא 1,048,576 (או שתים בחזקת עשרים). כשתכננו את המעבד הזה, הערכו שמגבית אחד של זיכרון הוא מעլ ומ עבר. מעבדים מתקנים כוללים פס מענים של 32 בית, ככלומר הם מוגבלים ב-4,294,976,296 בתיים – ארבעה ג'יגabit. בשלב מסוים, גם ארבעה ג'יגה של מרחב כתובות זיכרון הפכו להיות מגבלה, וכיוום מעבדים ומערכות הפעלה כגון Windows 7 תומכים במרחב כתובות בגודל 64 בית.

פס בקרה – CONTROL BUS

פס הבקרה מכיל קווים חשמליים שתפקידם לעשوت סדר בדרך שבה המעבד מתקשר עם יתר הרכיבים. הסבירנו את המנגנון שמאפשר גישה של המעבד לזכרון באמצעות פס המענים, אך לא הסבירנו איך יודעים האם המעבד מבקש לכתוב לזכרון, או שמא לקרוא מהזכרון?

פס הבקרה מכיל שני קווים, קו קריאה (read) וקו כתיבה (write), אשרקובעים את כיוון העברת המידע. כאשר קווים ה-`read` וה-`write` מכילים שניהם ערך 1, המעבד והזיכרון לא מתקשרים זה עם זה. אם קו ה-`read` מכיל אף, המעבד קורא מהזיכרון. אם קו ה-`write` מכיל אף, המעבד כותב לזכרון.

בסעיף הקודם הזכרנו שמעבד בעל פס מידע של 16, 32 או 64 ביטים, יכול לקרוא מידע של בית בודד. קווי בקרה שננקרים byte enable מאפשרים פעולה זו.

במשפחת ה-`i86` יש הפרדה בין מרחב הכתובות שמיועד ל-`I/O`. בעוד גודל מרחב הכתובות של הזיכרון משתנה בין דורות שונים במשפחת ה-`i86`, גודל מרחב הכתובות של ה-`I/O` הוא בעל ערך קבוע של 16 ביט. דבר זה מאפשר פניה ל-65,536 כתובות שונות של התקנים חיצוניים. כאשר מזנתה כתובה כלשהי לפס המענים, פס הבקרה קובע האם היא מיועדת לאזרז זיכרון או לאזרז ב-`I/O`.

הזיכרון

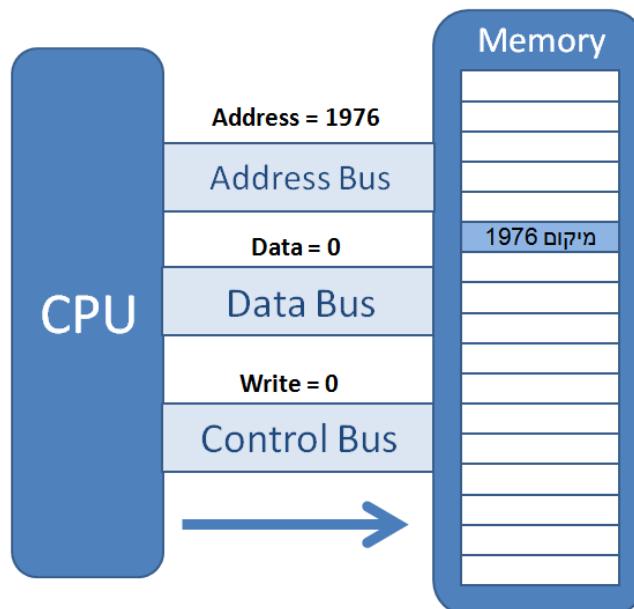
בסעיפים הקודמים הזכרנו, שכמויות המיקומות בזכרון היא 2^n , כאשר n כמות הביטים בפס המענים. בפרק זה נדון בהרחבה בנושא הגישה לכתובות שונות בזכרון.

אפשר לחשב על הזיכרון בתור מערך של ביטים. כתובתו של הבית הראשון היא 0, כתובתו של הבית האחרון היא $(2^n - 1)$. לכן, עבור מעבד בעל 20 ביטים בפס המענים, הזיכרון הוא מערך בגודל 1,048,576 ביטים.

לדוגמה, כדי להציב במקומות ה-1976 בערך "0", מתבצעות הפעולות הבאות:



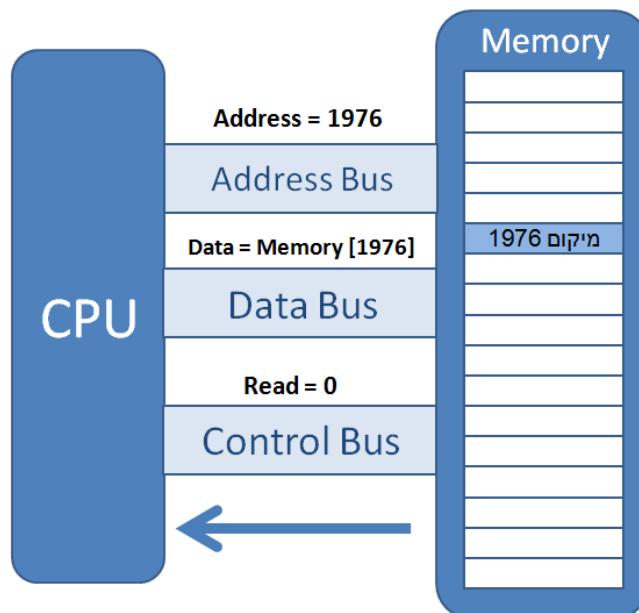
1. המעבד שם את הערך "0" בפס המידע.
2. המעבד שם את הכתובת 1976 בפס המידע.
3. המעבד משנה את קו ה-`write` בפס הבקרה וקובע את ערכו ל-0 (הקו "פועל").



פעולות כתיבה לזכרון

כדי לקרוא את מה שנמצא בזיכרון במיקום ה-1976 בזיכרון, מתבצעות הפעולות הבאות:

1. המעבד שם את הכתובת 1976 בפס המיעון.
2. המעבד קורא את הערך שבספס המידע.
3. המעבד משנה את קו הד-**read** בפס הקריאה וקובע את ערכו ל-0 (הכו "פועל").



פעולות קריאה מזיכרון

כעת נבחן איך ערכאים בגודלים שונים שמוראים בזיכרון.

נניח שהגדרנו בזיכרון שלושה ערכים:

1. במקום 1970 בזיכרון שמו ערך בגודל 8 ביטים, או byte, שערכו הוא 0ABh.
2. במקום 1974 בזיכרון שמו ערך בגודל 16 ביט, או word, שערכו הוא 0EEFFh.
3. במקום 1976 בזיכרון שמו ערך בגודל 32 ביט, או double word, שערכו הוא h12345678.

שימוש לב לאופן הכתיבה של ערך הקסדצימלי – אם הערך מתחילה באות, נקדמים אותו עם הספרה אפס. בצד ימין נשים ה כדי לציין שמדובר בערך בסיס הקסדצימלי. הסיבה לכך היא צורך להפריד בין צירופים כגון עשר בכתב הקסדצימלי, שנitin לרשום כ-ah, אך כפי שנלמד בהמשך יש לה-ah גם משמעות אחרת.

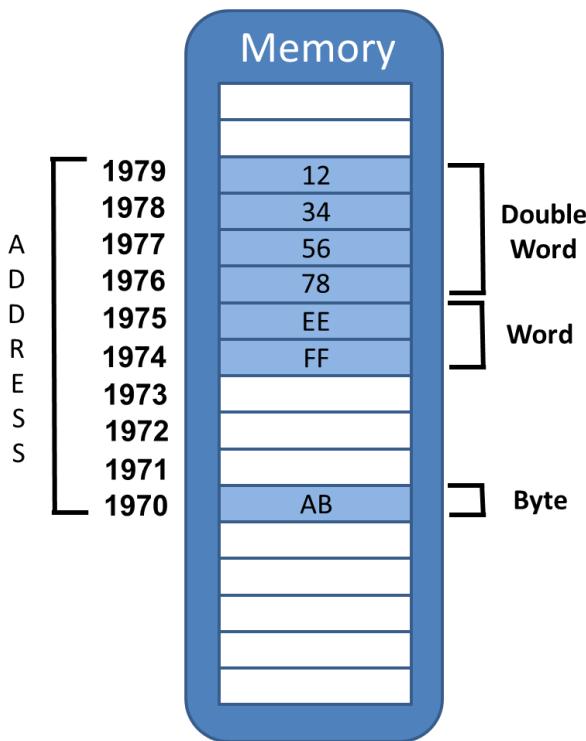


לאחר שמירת הערכים, הזיכרון שלנו ייראה כך:

1. במקום 1970 בזיכרון שמור הערך 0ABh.
2. במקום 1974 בזיכרון שמור הערך 0FFh, במקום 1975 בזיכרון שמור הערך 0EEh.
3. במקום 1976 בזיכרון שמור הערך h78, במקום 1977 שמור h56, במקום 1978 שמור h34 ובקום 1979 שמור h12.



L.O. Byte, H.O. Byte – תמצורת –



אפקן השמירה של גדים שונים בזיכרון

הקריאה מהזיכרון יכולה להתבצע בכל צורה שאנחנו מבקשים. לדוגמה, אנחנו יכולים לבקש לקרוא:

1. Byte ממיקום 1975, ולקבל `0EEh`.
2. Word ממיקום 1978, ולקבל `1234h`.
3. Double word ממיקום 1974, ולקבל `05678EFFh`.
4. Word ממיקום 1970, ולקבל את הדיבר `byte` ששמרנו במיקום 1970 בתוספת ערך כלשהו שקיים במיקום 1971
– בזיכרון תמיד קיים ערך כלשהו, זיכרון אף פעם אינו ריק!

סגמנטים

כזכור, למעבד הד-86 יש 2^{20} כתובות זיכרון. מצד שני, כפי שנראה בהמשך, כדי לגשת לזכרון המעבד עוזה שימוש ביחידת בגודל 16 ביטים, שנקראות **רגיסטרים (registers)**. כל רגיסטר של 16 ביטים מסוגל להזיז מרחב כתובות מ-0 עד 65,535 (או `0FFFFh`). כמובן, צריך למצוא שיטה לחבר על הפער שבין מרחב הזיכרון הקיים לרשות המעבד לבין מרחב הזיכרון, המצויץ יותר, שרגיסטר מסוגל לפנות אליו. השיטה שנבחרה היא לחלק את הזיכרון למקטעים קטנים יותר – **סגמנטים (segments)**. כל כתובה בזיכרון ניתנת לביטוי על-ידי מספר ה-`segment` שלו, והיחס (offset) מתחילה הסגמנט. הזרה המקובלת לרשום של כתובה בזיכרון היא:

Segment:offset

במשפחת ה-80x86, פניה לזכרון מטבחה על-ידי שילוב של שני רגיסטרים בני 16 ביטים: הרגיסטר הראשון מוחזק את מקום תחילת הסגמנט בזיכרון. הרגיסטר השני מוחזק את האופסט של הזיכרון מתחילה הסגמנט.

הגודל המקסימלי של האופסט קובע את הגודל המקסימלי של הסגמנט. במערכות בהם אנו נדונן, בעלי אופסט של 16 ביטים, גודל של סגמנט לא יכול להיות יותר מאשר 2^{16} כתובות, כאשר הכתובות בכל סגמנט הן בין 0000h לביןFFFFh. מעבד ה-8086 גודל כל הסגמנטים הוא בדיק K, 64K, במערכות מתקדמות יותר ניתן להגדיר גם סגמנטים קטנים יותר.

כדי להגיע לכתובת של מקום תחילת סגמנט, כופלים את הסגמנט ב-16. כתוצאה לכך יוצא שסגמנטים תמיד מתחילה בכתובת שהיא כפולה של 16 בתים. לדוגמה, סגמנט מס' 0002h מתחיל $(2 * 16) = 32$ בתים לאחר תחילת הזיכרון. סגמנט מס' 0011h מתחיל 272 בתים לאחר תחילת הזיכרון (11 בסיס 16 הוא 17, 17 כפול 16 בתים שווה 272 בתים).

כדי להגיע לכתובת כלשהו בזכרון, מוסיפים לכתובת תחילת הסגמנט את האופסט. לדוגמה, נניח שהסגמנט הוא 3DD6h והאופסט הוא h12. הכתובת בזכרון תרשם כך – 3DD6h:0012h.

הכתובת בזכרון תחוسب כך:

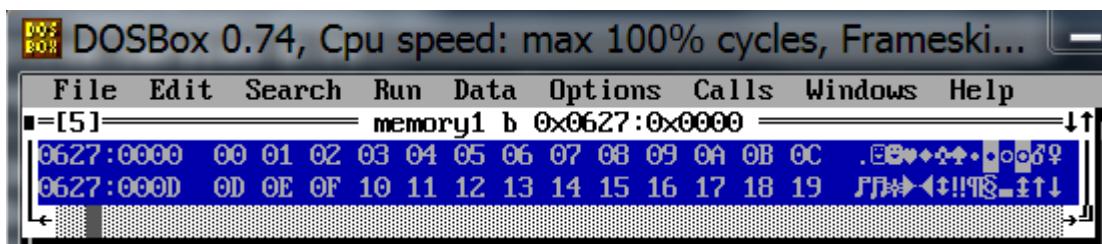
$$3DD60h + 0012h = 3DD72h$$

שימוש לבן לכך שתוספת ה-0 מצד ימין באבר הראשון נועדה לכפול את כתובת הסגמנט ב-16, כדי להגיע למקום תחילת הסגמנט. 

נבחן מספר דוגמאות לשילוב סגמנט ואופסט, היישר מתוך זיכרון המחשב. הדוגמאות הבאותLKות מתוכן תוכנת dosbox שרצה בסביבת codeview. בהמשך הספר, נעבד בתוכנה אחרת. השתמש ב-wview רק בסעיף זהה כיוון שהוא ממחישה היטב את נושא הסגמנט והאופסט.

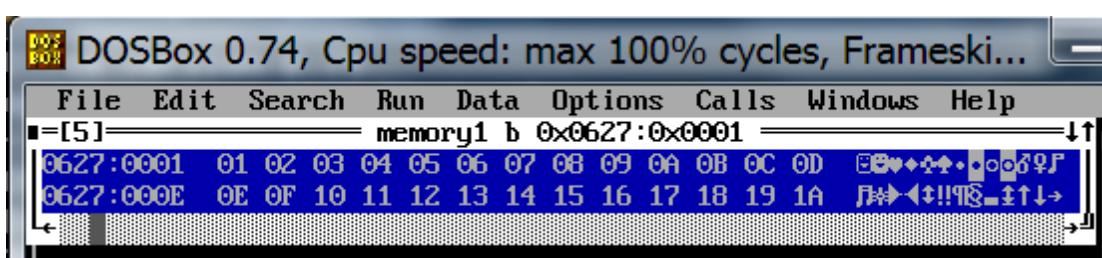
מצד שמאל, רשום מרחב הכתובות בזכרון שאנו מסתכלים עליו. בדוגמה שלנו, השורה הראשונה מתחילה במיקום בזכרון שכתובתו היא 0000:0627. בשורה זו יש 13 בתים, ולכן הכתובת الأخيرة בשורה הראשונה היא C000:0627. השורה השנייה מתחילה בכתובת של הבית הבא אחריו: D000:0627.

במרכזו רשומים הערכים השמורים בזיכרון. בדוגמה זו, יש בזיכרון ערכים עליים – 0, 1, 2 וכו'. באוטה מידת הינו יכול לשמר בזיכרון אוסף אחר של ערכים בגודל בית אחד. מצד ימין מוצג התרגום של הערכים בזיכרון לתווים טקסטואליים לפי קוד ASCII. אתם מוזמנים להזור לסעיף שմסביר על קוד ASCII ולהיווכח שהතווים הרשומים הם אכן התרגום של הערכים שבזיכרון.

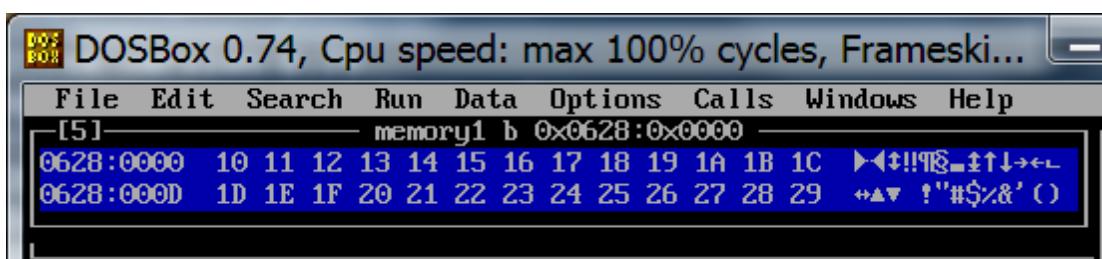


הראשונה מחייבת בכתובות 0001:6279.

שימושו לברשותם של שמות הערבית ותעתיקם ערבית-ASCII מופעים בזאת רבייה רוגע



להמחשת מושג הסגמנט, נאפס את האופטט ובמקומו נקדם את הסגמנט ביחידה אחת. כעת השורה הראשונה מתחילה בכתובת 0628:0000. שימו לב לכך שפעולות זו גורמת להסתה של 16 בתים בזיכרנו:



לפיו. בוגמה אחרת מוחשית שסגן תמי' מחייב שווה כפולה של 16 גמ'.

יחידת העיבוד המרכזית – CPU

הגיע הזמן לדון ביחידת העיבוד המרכזית עצמה ובאופן הפעולה שלה. בחלק זה נעמיק את ההבנה על החלקים במעבד שמאפשרים לו לבצע את הפעולה שלו – עיבוד של ביטים. נתחילה מסקירה של הרגיסטרים (Registers), לאחר מכן נמשך ליחידה האריתמטית לוגית (Arithmetic & Logical Unit), שאחריות על ביצוע פעולות חישוב כגון חיבור, חיסור, השוואה ופעולות לוגיות שונות, ונסיים ביחידה הבקרה (Control Unit), שאחריות על פענוח פקודות המכונה, ניהול רצף התוכנית, הוצאה לפועל של פקודות ואחסון התוצאות המתקבלות.

אגרים – Registers

רגיסטרים, או במינוח העברי אוגרים, הם סוג מיוחד מאוד של זיכרון. הם אינם חלק מהזיכרון המרכזי, אלא נמצאים ממש בתוך המעבד. כיוון שהם מטופלים ישירות על ידי המעבד הם גם מהירים ביותר – במקום לפנות לזכרון ולהחזיר ל渴בלת הנתונים, המעבד אינו מחייב כלל. מצב זה נקרא *zero wait*. יש כמה מצומצמת של רגיסטרים ולחלקו יש תפקידים מיוחדים, שמנגדים את השימוש בהם, אבל באופן כללי רגיסטרים הם מקום מיוחד לשימור בו מידע באופן זמני.



בספר זה נעדיף את השימוש במונח הלועזי "רגיסטר" על פני "אגר"

קייםים דורות שונים של מעבדים במשפחה ה- $80x86$. כמוות הרגיסטרים וגם הגודל שלהם בביטים עשוי להיות שונה בין דור אחד למשנהו. אנחנו נדון בדרך הבסיסי הכלול רגיסטרים של 16 ביט.

רегистרים כלליים – General Purpose Registers

קיימים שמונה Registers כלליים, אשר מרכיבים בטבלה הבאה:

הרגיסטר	שם לועזי	שם עברית	תיאור ושימוש עיקרי
ax	Accumulator register	צובר	משמש לרוב הפעולות האריתמטיות והלוגיות. למرات שניית לבצע פעולות חישוב גם בעזרת Registers אחרים, השימוש ב-ax הוא בדרך כלל יעיל יותר.
bx	Base address register	בסיס	בעל חשיבות מיוחדת בגין זיכרון. בדרך כלל משמש לשימירת כתובות בזכרון.
cx	Count register	מנונה	מנונה דברים. בדרך כלל נשתמש בו למספר כמות הפעמים שהרצינו לו לא, לכמה התווים בקובץ או במחוזת.
dx	Data register	מידע	משמש לשתי הפעולות מיוחדות: ראשית, ישן פעולה אריתמטית שדורשות מקום נוספת נסוף לשימירת התוצאה. שנייה, כפונים להתקני I/O, Register ax שומר את הכתובת אליה צריך לפנות.
si	Source Index	מצבייע מקור	ניתן לשימוש בהם בתור מצביעים כדי לפנות זיכרון (כמו שהראינו שניית לעשوت עם ax). כמו כן הם שימושים בטיפול במחוזות.
	Destination Index	מצבייע יעד	
bp	Base Pointer	בסיס	משמש לגישה זיכרון בסגמנט שקרי "מחסנית" Stack.
sp	Stack Pointer	מצבייע מחסנית	קס מצביע על כתובות בזכרון בה נמצא ראש המחסנית. באופן נורמלי, לעולם לא ניגע ב-ks ולא נעשה בו שימוש לטובות ביצוע פעולות אריתמטיות. התפקיד התקין של התכנית שלנו תלו依 בכר שערכו של ks תמיד יצביע למיקום הנכון בתוך המחסנית.

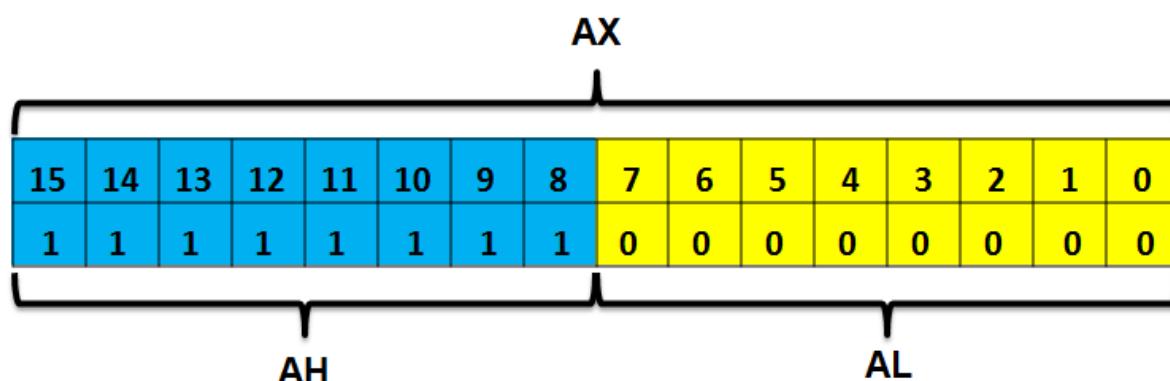
כאמור כל אחד מהרגיסטרים הנ"ל הוא בעל 16 ביטים, כולל שני בתים. לעיתים אנחנו צריכים לעסוק רק ב-8 ביטים – לדוגמה, כשאנו רוצים להעתיק מידע לבית כלשהו זיכרון. אם נעתיק לזכרון את אמ' כולם, הזיכרון ישתנה בשני בתים – שינינו את הבית שרצינו לשנות בזיכרון, אבל כתוצאת לוואי לא רצואה שיניינו גם את ערכו של הזיכרון בכתובת הבאה ויתכן שדרשו ערך חשוב.

כדי לאפשר גמישות מקסימלית לטיפול במשתנים בגודל בית אחד, ארבעה מהרגיסטרים – ax , bx , cx ו- dx – מחולקים לשני חלקיים בני שמונה ביטים כל אחד. ax לדוגמה, מחולק ל- ah ו- al . H מסמן **high**, l מסמן **low**, כלומר 8 הביטים העליונים של ax , ואילו al מסמן l מסמן **low**, כלומר 8 הביטים התחתונים של ax .

נלמד כעת פקודה אסמבלי ראשונה – הפקודה **move** (קייזור של move) מבצעת העתקה של הערך שנמצא בצד ימינו אל הצד השמאלי. לדוגמה:

mov ax, 0FF00h

לאחר ביצוע הפקודה, רегистר ax ייראה כך:



ניתן להגיעה לאותה תוצאה גם על-ידי הקוד הבא:

```
mov ah, 0FFh  
mov al, 0
```

שימוש לב שרגיסטרים של 8 ביטים אינם רגיסטרים עצמאיים. שינוי של a ישנה גם את ax , ולהיפך.

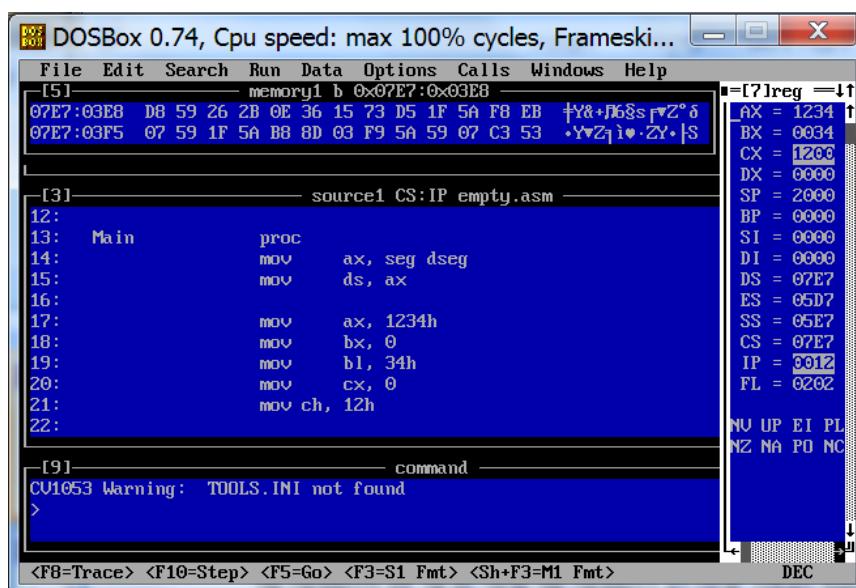


. dl, dh ,cl ,ch ,bl ,bh בואפן דומה, ניתן לעשות שימוש ברגיסטרים

להלן טבלה המסכםת את הרגיסטרים הכלליים שגודלם 8 בית:

16 bit	8 bit	8 bit
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

התוכנית הבאה כוללת מספר פעולות פשוטות עם רגיסטרים. אנחנו נתמקד בשורות 17 עד 21:



```
mov ax, 1234h
mov bx, 0
mov bl, 34h
mov cx, 0
mov ch, 12h
```

מצד ימין של המסך אפשר לראות את מצב הרגיסטרים לאחר סיום הרצת השורות הנ"ל ולפני הרצת שורת הקוד הבאה.

שימוש לב:



- להשפעה של שינוי **bx** על **ax**: שינוי של **bx** משנה רק את 8 הביטים התחתיונים של **ax**.
- להשפעה של שינוי **ch** על **cx**: שינוי של **ch** משנה רק את 8 הביטים העליונים של **cx**.

יתר הרегистרים הכלליים,osi, sp, bp, si ו-ds, הם בעלי 16 ביטים ואין כל דרך לפנוט באופן לשר אל הבית העליון או התחתיו שלהם כמו שעשינו עם **ax**, **bx**, **cx** ו-**ds**.

לתרגול נושא הרегистרים הכלליים, להלן טבלה הכוללת פקודות העתקה ורегистרים. כל פקודה העתקה היא של ערך או רגיסטר לתוכן רגיסטר כלשהו. יש להשלים את ערכי הרегистרים לאחר כל פקודה העתקה. הניחו שהמצב ההתחלתי הוא שכל הרегистרים מאופסים.



DX		CX		BX		AX		הפעולה	
DH	DL	CH	CL	BH	BL	AH	AL	העתק את	לregsiter
								AX	הערך h1234
								AL	0ABh
								BX	0ABCDh
								CH	0EEh
								DL	0BBh
								DH	הregsiter CH
								AH	DL
								CX	הregsiter BX

הפתרו בעמוד הבא.

פתרונות:

DX		CX		BX		AX		הפעולה		העתק את
DH	DL	CH	CL	BH	BL	AH	AL	לרגיסטר		
00	00	00	00	00	00	12	34	AX	הערך h 1234	
00	00	00	00	00	00	12	AB	AL	הערך h 0ABh	
00	00	00	00	AB	CD	12	AB	BX	הערך h 0ABCDh	
00	00	EE	00	AB	CD	12	AB	CH	הערך h 0EEEh	
00	BB	EE	00	AB	CD	12	AB	DL	הערך h 0BBBh	
EE	BB	EE	00	AB	CD	12	AB	DH	הרגיסטר CH	
EE	BB	EE	00	AB	CD	BB	AB	AH	הרגיסטר DL	
EE	BB	AB	CD	AB	CD	BB	AB	CX	הרגיסטר BX	

ריגיסטרי סגמנט – Segment Registers

במעבד ה-8086 יש ארבעה ריגיסטרים מיוחדים (בדגמים מאוחרים יותר במשפחה ה-80x86 נוסף ריגיסטר חמישי - FS):

CS - Code Segment

DS - Data Segment

SS - Stack Segment

ES - Extra Segment

כל הריגיסטרים הללו הם בעלי 16 ביטים. תפקידם הוא לאפשר בחירה של סגמנטים בזיכרון. כל Segment Register שומר את הערך של סגמנט בזיכרון.

הריגיסטר CS שומר את מיקום תחילת סגמנט הקוד. סגמנט זה מכיל את פקודות המคำה שבוצעות כרגע על ידי המעבד. כזכור, הגודל המקסימלי של סגמנט הוא 64K. האם זה אומר שהגודל המקסימלי של תכנית הוא לא יותר מאשר K? לא, משומש שאפשר להגדיר כמה סגמנטים של קוד, ולהעתיק ל- CS ערכיהם שונים, וכך שכל פעם יצביע על הסגמנט הנכון (כאשר מודול הזיכרון מאפשר זאת. בתוכנית `base.asm` אנחנו מגדירים בתחלת התוכנית small model וכאן ישנו רק סגמנט קוד אחד). באופן מעשי לא סביר שתכתבו תוכנית אסמלבי שתדרוש יותר מסגמנט קוד אחד.

לגביו הריגיסטר DS, בדרך כלל בתחלת התוכנית מתבצעות פעולות העתקה כך ש-DS מצביע על המיקום של סגמנט DATA, בו נשמרים המשתנים הגלובליים בתכנית. גם כאן, במודול זיכרון small בו אנחנו משתמשים אנחנו מוגבלים ל- 64K של בתים, אבל במקרים אחרים של זיכרון ניתן להגדיר מספר סגמנטים מסוג DATA ולשנות את DS כדי לפנות אליהם.

הרגיסטר SS מציין על מקום בזיכרון שמכיל את המחסנית (STACK) של התוכנית. נושא המחסנית יוסבר בפירוט בהמשך. אותה האזהרה שנותנו לגבי שינוי הרגיסטר DS תקפה גם כאן – אין לשנות את SS אם אתם לא יודעים לבדוק מה אתם עושים.

הרגיסטר **ES**, Extra Segment, נתן לנו גמישות נוספת – אם אנחנו רוצים לפנות לSEGMENT נוסף, זה בדיקת מה שהוא אפשר לנו. רגיסטר זה שימושי, לדוגמה, בהעתקות של רצפים נתונים אל הזיכרון.

רגיסטרים ייעודיים – Special Purpose Registers

ישנם שני רגיסטרים ייעודיים:

IP - Instruction Pointer- מצביע ההוראה

FLAGS - דגליים

רגיסטר ה-IP מחזיק את הכתובת של הפקודה הבאה לביצוע. והוא רגיסטר של 16 ביט שלמעשה משתמש במצב (pointer) לתוך ה-.code segment

רגיסטר הדגליים שונה מיתר הרגיסטרים. הוא אינו מחזיק ערך באורך 8 או 16 ביטים, אלא הוא אוסף של אותות חשמליים בני בית אחד. רגיסטר זה משתנה בעקבות הרצת פעולות אריתמטיות שונות וכאן הוא שימושי לפעולות בקרה על התוכנה. כמו כן כולל רגיסטר הדגליים אותן את מצב הריצה של המעבד, לדוגמה לצורך עבודה עם דיבאגר.

בניגוד ליתר הרגיסטרים שנתקלנו בהם עד כה, לא נבצע שינויים ישירות ברגיסטרים אלו. תחת זאת, הערךם של רגיסטר ה-IP והדגליים השתנו על-ידי המעבד בזמן ריצת התוכנית. נציין, כי קיימות פקודות שמאפשרות לתוכנת לשנות את רגיסטר הדגליים, אולם הן מחויז להיקף של ספר לימוד זה.

אנחנו נקדים לרוגיסטרים הייעודיים פרק נפרד, מיד לאחר שנלמד את סביבת העבודה שלנו. כך, נוכל להתנסות בעבודה עם הרגיסטרים הללו תוך כדי הלימוד.

היחידה האריתמטית – Arithmetic & Logical Unit

היחידה האריתמטית והלוגית (Arithmetic & Logical Unit) או בקיצור – ALU (Arithmetic & Logical Unit) היא המקום במעבד שתרחשת בו רוב הפעולות. ה-ALU טוען את המידע שהוא צריך מהרגיסטרים, לאחר מכן יחידת הבקרה קובעת ל-ALU איזו פעולה צריך לבצע עם המידע, ובסיום ה-ALU מבצע את הפעולה ושומר את התוצאה ברגיסטר שנקבע כרגיסטר יעד.



לדוגמה, נניח שאנו רוצים להוסיף את הערך 3 לרגיסטר ax:

- המעבד יעתיק את הערך שב-ax לתוך ה-ALU.
- המעבד ישלח ל-ALU את הערך 3.
- המעבד יתן ל-ALU הוראה לחבר את שני הערךים הנ"ל.
- המעבד יחזיר את תוצאה החישוב מה-ALU לתוך הרגיסטר ax.

יחידת הבקרה – Control Unit

בחלק זה נענה על השאלה – איך בדיקת המעבד יודע איזו פקודה לבצע?

למחשבים הראשונים היה פאנל עם שורות של מוגלים魑魅魍魎, שנitin היה לחוטם אותם החשמליים אפשר היה לקבע איזו פקודה המעבד ייריצ'. בשיטה זו, כמות הפקודות שנitin היה להריצ' בתוכנית אחת הייתה שווה למספר השורות בפאנל. אחת הפתוחיות המשמעותית במדעי המחשב הייתה המצאת הרעיון לשומר את התוכנה בזיכרון המחשב ולהביא אותה ליחידת העיבוד פקודה אחרי פקודה. בשיטה זו כל פקודה מותרגמת לרצף של אחדות ואפסים ונשמרת בזיכרון.

Operational – יחידת הבקרה, Control Unit, מביאה מהזיכרון את פקודות המכונה, הידועות גם בשם

OpCodes או Codes. 

כדי ליעיל את הבאת ה-OpCode למעבד, גודל של OpCode הוא בדרך כלל כפולת של 8 ביטים. יחידת הבקרה מכילה רגיסטר הוראות, או IP, או IP, אשר זכרנו בסעיפים הקודמים. ה-IP מחזק את הכתובת בזיכרון של הפקודה הבאה שיש להעתיק. לאחר שייחידת הבקרה מעתקה את ה-OpCode לתוך רגיסטר הפונCTION, היא מקדמת את ה-IP אל כתובת הפקודה הבאה וכך הלאה.

שעון המערכת (הרחבה)



מכונות פון נוימן מעבדות פקודה אחרי פקודה, באופן טורי.

נתבונן בפקודות הבאות, הטוענות לתוך ax את bx+5:

mov ax, bx

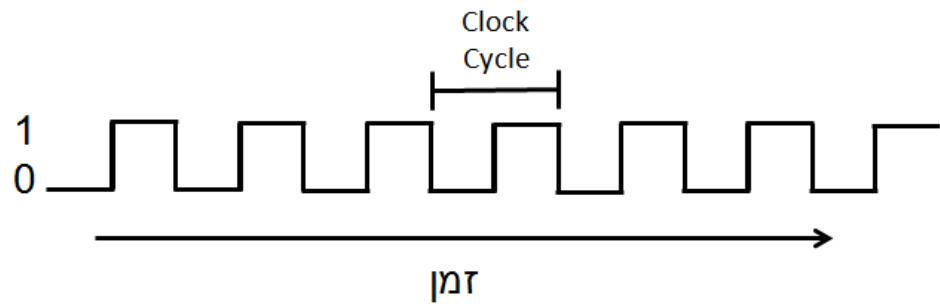
add ax, 5

ברור שפקודת החיבור צריכה להתרחש אחרי פקודה הטעונה של bx לתוך ax, אחרת נקבל תוצאה לא נכוןה. אם שתי הפקודות תבוצענה בו זמנית, ולא טורית, לאחר הרצת הפקודות bx יכול את ax או יכול 5, אבל לא bx+5.

כל פעולה דורשת זמן לביצוע. זמן זה כולל פונCTION ה-OpCode, פניה לזכרון (בדרכ' כללו), טיעות ערך כלשהו לריגיסטר, ביצוע פעולה אריתמטית ועוד. כדי שפעולות יתבצעו לפי סדר, המעבד צריך מגננון שיתזמין את הפעולות – שעון המערכת.

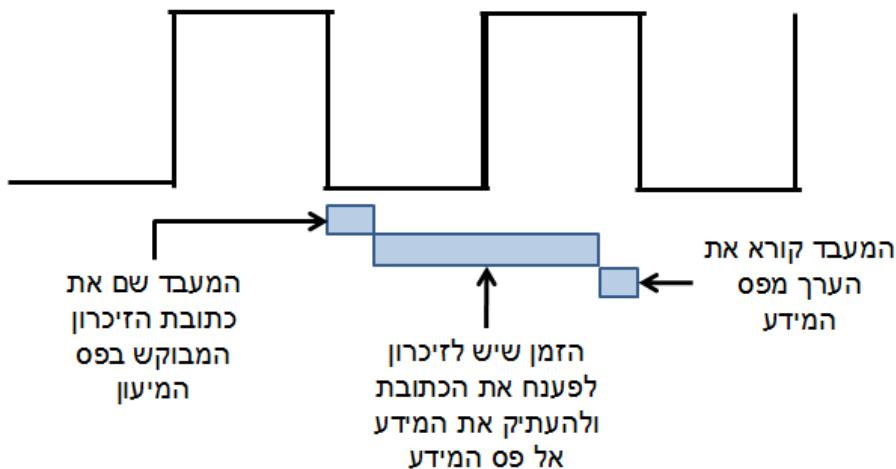
שעון המערכת הוא אות החסמי על פס הבקרה Control Bus, אשר משנה את ערכו כל פרק זמן קבוע בין 0 ל-1. הדרירות שבזהו החסמי משתנה בין 0 ל-1 היא תדריות שעון המערכת. הזמן שלוקח לאות להשתנות מ-0 ל-1 וחזרה ל-0 נקרא **clock cycle**. הזמן של clock cycle הוא אחד חלק תדריות השעון. לדוגמה, מעבד בעל תדר שעון של 1MHz הוא בעלי clock cycle של 1 מיקרו שניה (1/1,000,000 של שניה).



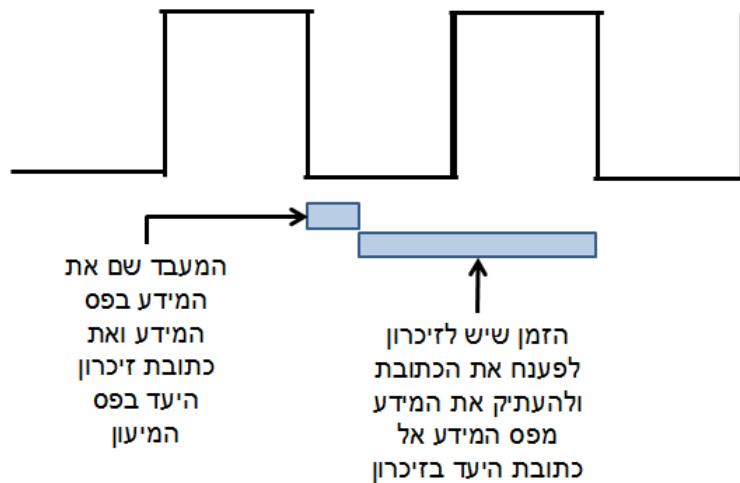


כדי להבטיח סync'רוני, המעבד יתחיל לבצע פעולה או בנקודות הזמן בה מתרחשת עלייה או הירידה (זמן שהאות משתנה מ-0 ל-1) או עם ירידת האות השעון (זמן בו האות משתנה מ-1 ל-0). כיוון שככל המשימות של המעבד מסונכרנות על ידי שעון המערכת, משימה לא יכולה להימשך פחות מ-1 *clock cycle* ייחד, כלומר תדריות שעון המעבד מגבילת למעשה את יכולות הפעולות המקסימליות שהמעבד יכול לבצע בזמן מסוים. עם זאת, אין משמעות הדבר שהמעבד תמיד יבצע את כל הפעולות המקסימליות, משום שפעולות רבות אורכו יותר מ-1 *clock cycle* ייחד.

אחד הפעולות הנפוצות שהמעבד מבצע היא גישה לזיכרון. גישה לזיכרון חייבת להתרחש באופן מסונכרן עם שעון המערכת. להלן סכימה של הפעולות המתבצעות על ידי המעבד בזמן גישה לזיכרון. חשוב לציין שבדיקה זו הגישה אורכת *clock cycles* ייחד, כמו במעבדים מתוקדים, אך תיתכן גישה לזכרונות שאורכת מספר *clock cycles*.



סיכום של פעולה קריאה מהזיכרון על ציר הזמן של שעון המערכת



סכמה של פעולה כתיבה לזכרון על ציר הזמן של שעון המערכת

סיכום

בפרק זה למדנו אודות ארגון המחשב, בדגש על מעבדי משפחת ה-i86x80: התחלנו מפירוט הסיבות שבגלאן רלבנטי ללמידה על משפחת מעבדי ה-i86x80 למטרות שקיימים מעבדים חדשים יותר, בין היתר מכיוון שככל המעבדים חולקים ארכיטקטורה משותפת וכולם מכונوت VNA. הסבכנו מהי ארכיטקטורת VNA ונכנסנו להסביר מפורט אודות מרכיביה השונים. ראיינו איך פסי המידע, הנתונים והבקרה משתמשים את המעבד לתקשורת עם רכיבים שונים, תוך דגש על קריאה וכתיבה לזכרון.

בהמשך, סקרנו את מבנה הזיכרון של משפחת ה-i86x80 והכרנו את החלוקה לsegments ואופסטים, שמשמשת להגעה לכל כתובת בזכרון. לאחר מכן, עברנו לדון ביחידת העיבוד המרכזית על חלקיה השונים. התחלנו מהרגיסטרים השונים ועבור מה משמש בדרך כלל כל רגיסטר. ממש עברנו ליחידה האריתמטית, האחראית על ביצוע הפעולות, וליחידה הבדיקה, שהחראית להביא את פקודות המכונה לפענה. סיימנו בהסביר על שעון המערכת – הבנו שכדי לבצע פקודות באופן טורי יש צורך בשעון המערכת, והראינו את תהליך הפניה לזכרון על ציר הזמן.

פרק זה היה תיאורטי בעיקרו, והקנה את הכלים הבסיסיים להבנת הפרקים הבאים. חשובה במיוחד ההבנה של נושא הכתובות בזכרון ומהם הרגיסטרים השונים, או אם איןכם בטוחם שאתם שולטים בנושאים אלו – חיזרו וודאו שגם מבינים את התיאוריה. בפרק הבא נתקין את סכיבת העבודה שתאפשר לנו להתחילה לתוכנה.

פרק 4 – סביבת עבודה לתוכנות באסמבלי

מבוא

בפרק זה נלמד על כל הפעולות בהם השתמש בשביל לתוכנת בשפת אסמבלי. ראשית, נציין כי יש מגוון של כלים לעובדה באסמבלי, ופרק זה אינו מתיימר להציג את כולם. קיים אף של אסמבלירים (תוכנות הממירות אסמבלי לשפת מוכנה) – הבודלים ביניהם קטנים, אבל קוד אסמבלי שנכתב לאסמבילר כלשהו לא יתאים בהכרח לאסמבילר אחר. למעשה, אפשר להחליף כל אחת מהתוכנות שנציג כאן בתוכנה אחרת, אבל משיקולים של נוחות ורצון להשיג בסיס משותף בין כלל לומדי האסמבלי, בחרנו להציג אפשרויות אחת בלבד לכל תוכנה. היעד אליו אנו שואפים להגיע בסיום פרק זה הוא ליצור קובץ ראשון בשפת אסמבלי, קובץ שנקרא לו `base.asm`, להפוך אותו לקובץ בשפת מוכנה ולהיות מסוגלים להריץ את התוצאה באמצעות כלי שנקרא **דיבאג'ר (Debugger)**. בסיום הפרק יש תרגיל מחקר קטן, לתלמידים המעניינים להריץ את הידע שלהם. בפרק הקודם למדנו באופן תיאורטי שהאסמבילר מתרגם פקודות בשפת אסמבלי לשפת מוכנה – `Opcodes`. תרגיל המחקר משלב את הכלים שנלמדו בפרק זה על מנת להבין יותר לעומק את הדרך בה מתורגמות פקודות האסמבלי `to Opcodes`.

Editor – Notepad++

התוכנה הראשונה שנשתמש בה היא פשוט – כתבן. ישנו מגוון של תוכנות לעריכת קבצי טקסט שיכולים לשמש אותנו. חשוב לציין שישנן תוכנות "חכמות" מדי, כגון `Word`, ששמורות תווים מיוחדים ולא רק את הטקסט. שימוש בתוכנות אלו הינו בעייתי. אנחנו יכולים לכתוב ב-`Notepad++`, `Visual Studio` או כל `Editor` אחר. השימוש ב-`Notepad++` מומלץ במספר סיבות:

- התוכנה חינמית
- פשוטות התקנה ושימוש
- טקסט מואר באבעים שונים – פקודות בכחול, הערות בירוק, רגיסטרים בשחור וכו'. דבר זה מקל על ההתקפות בקוד.

התקנה: ממחפשים `Notepad++` בגוגל, ממשיכים לאתר [Notepad++ installer](http://notepad-plus-plus.org/download/v6.5.5.html) ובוחרם באפשרות <http://notepad-plus-plus.org/download/v6.5.5.html>



- לכל קובץ יש שם ו הסיומת. לדוגמה hello.doc, doc היא הסיומת של הקובץ שנקרא hello.h. בשם הקובץ hello יש חמישה תווים.
- הקפידו לשמר את הקבצים שלכם עם הסיומת .asm. אם תעשו זאת, תוכנת ה-Notepad++ תציג לכם את הקובץ בצורה צבעונית וקלה לקרוא.
- תנו לקבצי ה-.asm שמות בני 8 תווים לכל היותר. זה יקל עליכם לעבודה בסביבת DOS.

קובץ Base.asm

ניצור כעת את הקובץ הראשון בשפת אסמבלי. קובץ זה ישמש כבסיס להמשך העבודה בפרק זה ובפרקים הבאים. בשלב זה, לא כל הפקודות וההגדרות צריכות להיות ברורות לכם – הן יתבררו בהמשך. לטובת המשך העבודה, פיתחו קובץ חדש ב-Notepad++ והעתיקו לתוכו את הקוד הבא (האזורים הרוקים הם הערות ואין צורך להעתיק אותם, כמו כן מספרי השורות נוצרמים אוטומטית ואין להעתיק אותם). לאחר מכן, שימרו את הקובץ בשם base.asm.

[הקובץ נמצא גם בקישור](http://www.cyber.org.il/assembly/TASM/BIN/base.asm)

```

1 IDEAL
2 MODEL small
3 STACK 100h
4 DATASEG
5 ; -----
6 ; Your variables here
7 ; -----
8 CODESEG
9 start:
10    mov ax, @data
11    mov ds, ax
12 ; -----
13 ; Your code here
14 ; -----
15
16 exit:
17    mov ax, 4c00h
18    int 21h
19 END start

```

הסבר על base.asm

ניתן כעת שני הסברים לקובץ base.asm. הסבר אחד בסיסי, שכולל את מה שצריך לדעת כדי להתחיל לתוכנת בשפת אסמבלי, והסביר שני מתקדם. בשלב זה עדיין אין לנו את הכלים להבין את ההסביר המתקדם, אבל אין סיבה להיות מתחוסכים – עיברו הלאה וחזרו אליו בסיום הפרק על שפת אסמבלי – הידע על מבנה השפה והזיכרונו יאפשר לכם להבין את ההסביר המתקדם.

הסבר בסיסי:

נתמך רק באזורי שאותם נרצה לעורך בקובץ: האזור הראשון נמצא תחת הכתובת **DATASEG**. את המושג סגמנט אנו מכירם מהפרק על ארגון המחשב. הסגמנט שקוראים לו **DATASEG** הוא הסגמנט שבו מגדירים שמות למשתנים בזיכרון המחשב. נזכיר כי משתנים הם שמות שאנו נוהגים לכתובות בזיכרון של המחשב. בtower **DATASEG** לא נהוג לשים קוד.

אנו יכולים לכתוב ב-**DATASEG**, לדוגמה:

```
var1 db 5
```

ובכך הגדרנו משתנה בשם **var1** שמקבל את הערך 5. האסמלבר (געיו אליו בהמשך) כבר יdag להקצת אזור בזיכרון, לקרוא לאזור זהה **var1** ולשים בזיכרון את הערך 5.

האזור השני שמעניין אותנו הוא מה שנמצא תחת הכתובת **CODESEG**. זהו הסגמנט שבו כותבים את הפקודות שאנו רוצחים שהמעבד יריצן. בתחילת הסגמנט יש כמה פקודות עוזר, ואז – מיד אחרי הערה – יש אזור שבו אפשר להקליד את הפקודות שלנו.

אנו יכולים לכתוב ב-**CODESEG**, לדוגמה:

```
mov al, [var1]
```

ופקודה זו תגרום למעבד להעתיק את הערך שהכנסנו ב-**var1**, 5, אל תוך הרגיסטר **al**.
זה כל מה שצרכי לדעת בשביל להתחיל לעורך שינויים בקובץ **base.asm**.

הסבר מפורט:

מעבר שורה שורה על הקוביץ:

- **IDEAL** – לתוכנת Turbo Assembler שאנו עובדים איתה, יש כמה צורות כתיבה. **IDEAL** היא צורת כתיבה פשוטה שמתאימה לתוכננים מתחילה.

- **MODEL small** – מודל זיכרון **small**, קבוע לאסמלבר שהתכוונית מכילה שלושה סגמנטים – **Data**, **Code**, **Stack** ושגודל סגמנטי הקוד והנתונים הוא 64K כל אחד.

- **STACK 100h** – גודל המחסנית. הסבר מפורט תמצאו בחלק על המחסנית.
-

- **DATASEG** – סגמנט הנתונים.

- **CODESEG** – סגמנט הקוד.

- **start** – תווית שמסמנת למעבד מאיפה להתחיל את ריצת התוכנית. כל שם אחר יכול לבוא במקומה – לדוגמה – “**main**” – העיקר להיות עקביים עם הוראת **hdsend** שבשורה האחרונה.

- **mov ax, @data** – מטרת הוראה זו וההוראה הבאה, קבוע ש-**ds** יציבע על מקטע הנתונים. ההוראה **mov ds, ax** מוחירה את הכתובת של סגמנט **data**. בסיום ההוראה הבאה **ax** מועתק לתוכה **ds** כתובת סגמנט ה-**data**.

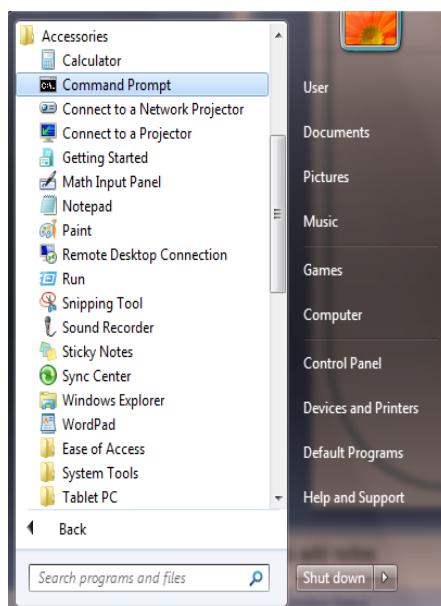
- exit – יציאה מסודרת מהתוכנית. שתי השרוות שמתוחת למילה `exit` הן קוד שגורם ליציאה מסודרת של התוכנית ושחרור הזיכרון שהיא תפסה, כך שמערכת הפעלה תוכל להשתמש בו לצרכים אחרים. הסבר מפורט תמצאו בחלק על פסיקות DOS.

- end start – המילה `end` אומרת לאסמלבר להפסיק את הקומpileציה, ככלומר פקודות שיופיעו אחרי מילה זו לא יתורגם לשפת מכונה. לאחר `end` מופיעה תווית שאומרת לאסמלברマイפה המעבד צריך להתחיל את הריצה על התוכנית. אם אין תווית, הריצה תתחיל מתחילה קובץ `exe`. במקרה שלנו, אנחנו רוצים שהריצה תתחיל מהתוויות `.start`.

Command Line

בעבר, טרם מערכות הפעלה הגרפיות, ניהול הקבצים והספריות הטענו התבצע עלי ידי Command Line. ב- Command Line אין אפשרות להשתמש בעבר, כגון כדי להריץ קובץ, לדוגמה, במקרה לעשות עליו דאבל קליק עם העכבר צריך לכתוב את שם הקובץ.

אנחנו נלמד פקודות בסיסיות של Command Line משתי סיבות. הראשונה היא שהמשק לאסמלבר שלנו הוא טקסטואלי. ככלומר, כדי להורות לאסמלבר איזה קובץ `asm` לתרגם לשפת מכונה, הדרך היחידה היא באמצעות הקשת הפוקודות ב-`Windows`. הסיבה השנייה, היא שמערכות הפעלה מודרניות רבות כבר לא מסוגלות להריץ את התוכנות הנדרשות לעבודה בסביבת אסמלבי. מערכות אלו עובדות ב-64Bit (כלומר, מייצגות כתובות בזיכרון עליידי 64Bitים) ואין להן תאיות לאחר ליצוג עליידי 20Bit שאנחנו נזדקק לו כדי לדמות עבודה עם מעבד `80x86`. הפיתרון המקביל, כפי שנם אנחנו עושים, הוא לעבוד עם תוכנה שנקראת אמולטור – מדמה מערכת הפעלה ומעבד ישנים יותר. כדי לעבוד עם אמולטור, נזדקק לתוכנות שעובדות בממשק טקסטואלי בדומה ל- Command Line. ההגעה ל- Command Line שונה במערכות הפעלה שונות. בחלונות 7, תמצאו אותה בתוך רשימת התוכניות -----> עזרים .Start Prompt .cmd. בדרך כלל ניתן יהיה להציג אליה על ידי כתיבת cmd בחילון חיפוש התוכניות ב- menu Start



לאחר מכן יפתח חלון הדוגמת החלון הבא:



קיימים אטרים שונים שמסבירים את הפקודות השונות שניתן להריצ' ב-**Command Line**. אתר מומלץ הוא <http://www.computerhope.com/msdos.htm>

אנחנו נשתמש בעיקר בפקודות הבאות:

❖ **Change Directory – קיזור של CD**

כדי לעבור לספריה כלשיה יש לרשום:

CDDirectoryName

כאשר שם הספריה בא במקום "DirectoryName". לדוגמה:

CD Games

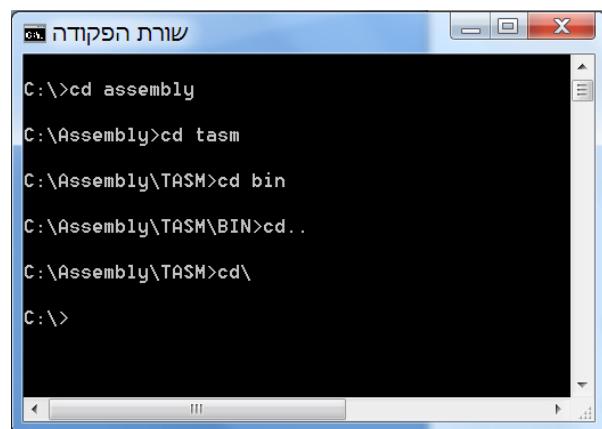
כדי לחזור בספריה אחת אחורה בעז הספריות יש לרשום:

CD ..

וכדי לחזור לתחילה העז יש לרשום:

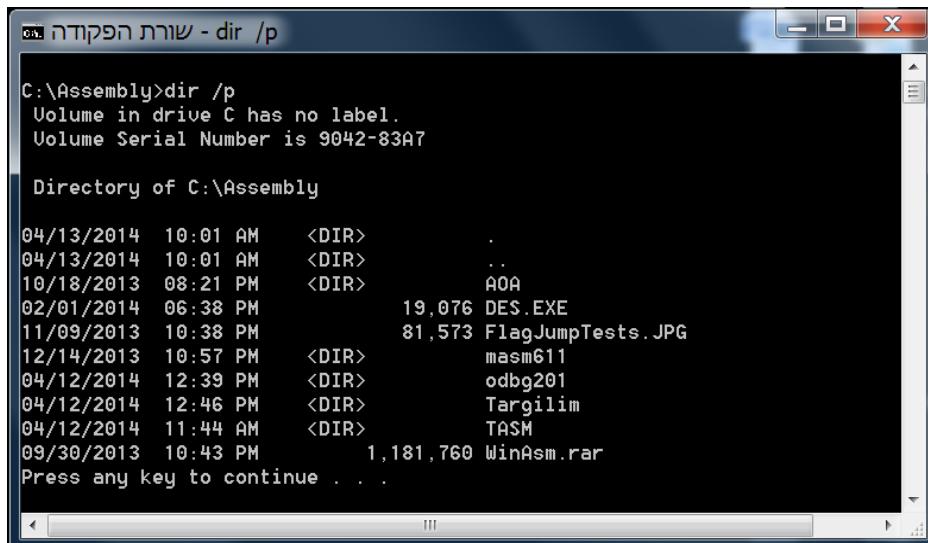
**CD **

לדוגמה:



DIR ♦

פירוט כל הקבצים ונתוני הספריות הקיימים בספריה כלשהי. לעיתים יש קבצים וספריות רבים מדי בשבייל הצגה על מסך אחד, ולכן רצוי לכתוב `/dir`. הסימון `k` הוא קישור ל-`page` והתוכן יוצג על המסך עמוד אחריו:



```
C:\Assembly>dir /p
Volume in drive C has no label.
Volume Serial Number is 9042-83A7

Directory of C:\Assembly

04/13/2014 10:01 AM    <DIR>      .
04/13/2014 10:01 AM    <DIR>      ..
10/18/2013  08:21 PM    <DIR>      AOA
02/01/2014  06:38 PM           19,076 DES.EXE
11/09/2013  10:38 PM          81,573 FlagJumpTests.JPG
12/14/2013  10:57 PM    <DIR>      masm611
04/12/2014  12:39 PM    <DIR>      odbg201
04/12/2014  12:46 PM    <DIR>      Targilim
04/12/2014  11:44 AM    <DIR>      TASMS
09/30/2013  10:43 PM          1,181,760 WinAsm.rar
Press any key to continue . . .
```

EXIT ♦

כדי לצאת מה-`Command Line` ולהזoor למערכת הפעלה, מקישים `exit` ואחר כך `enter`.

DOSBOX ♦

כיוון שלמערכות הפעלה מודרניות אין תאיות לאחור עם מעבדי ה-`80x86` ומרחיב הכתובות שלהם. נדרשת תוכנה שנקראת בשם **כלי אמולטור (Emulator)**. אמולטור היא תוכנה שمدמה מחשב או מערכת הפעלה כלשהי. לדוגמה, אנחנו יכולים להוריד מהאינטרנט תוכנת אמולטור שתגרום למחשב שלנו, ואפילו לסמארטפון שלנו, להתנהג כמו מחשב מיושן.

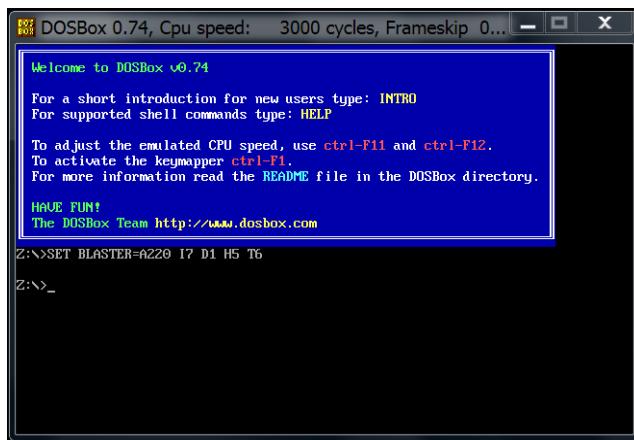
אמולטור של מחשב קלטי, **Commodore 64**, שיצא בשנת 1982, מאפשר לאייפון להרין משחקים שנכתבו ל-**Commodore 64**



כדי להריץ את סביבת העבודה של אסמבלי, נדרש מערכת הפעלה מוקדמת של מיקרוסופט. מערכת הפעלה זו קרוייה DOS KiTzor של Disk Operating System, והגרסה האחרונה שלה יוצאה בשנת 1994. האמולטור של DOS נקרא Dosbox ונקון לזמן כתיבת שורות אלו הגרסה העדכנית היא 0.74. הפשו בגוגל "Dosbox download" והתקינו את התוכנה.

ניתן להוריד את גרסה 0.74 גם מהלינק:

הקלקה על דוסבוקס תפוח מסך כדוגמת המסך הבא:



הפעולה הראשונה שאנו רוצים לעשות היא להגיע לכונן בו נמצאים הפרויקטמים שלנו ויתר התוכנות. אנחנו כרגע בכונן Z. דוסבוקס מציע לנו לכתוב Intro כדי לקבל עזרה למשתמשים חדשים. לפני שנוכל להשתמש בקבצים שעל המחשב שלנו, אנחנו צריכים לעשות לדוסבוקס הגדרה שנקראת mount. במקרה שהקבצים שלנו נמצאים על כונן C, בספריה Assembly, נכתבו:

Mount c: c:\

על המסך יודפס - Drive c:\ is mounted as local directory c. כתוב:

C:

ועברנו לכונן הקבצים C.

כל הפקודות של ה-Command Line תקפות גם כאן.

שימוש לב שלחיצה על מקש החץ למעלה, אפשר לנו לדפדף בין הפקודות הקודמות שכתבנו - דבר זה יכול לחסוך זמן.

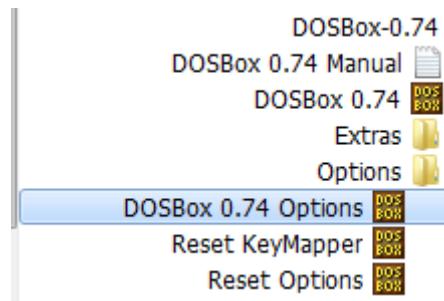
כיוון שאנו מביצים אמולציה של מעבד ישן, מהירות הריצה של הדוסבוקס הופחתה בהתאם וברירת המחדל היא ריצה בקצב של 3000 cycles לשניה בלבד (שימוש לב כתוב בכוורת למעלה). ניתן לשנות את הקצב הזה באמצעות כתיבת הפקודה הבאה במסך ה-DosBox:

Cycles = max

וכעת מהירות השעון של האמולטור עלתה למאהירות המקסימלית אותה הוא מסוגל להציג (עדין משמעותית פחות ממהירות השעון של המעבד במחשב שלכם, אבל ככל הנראה טוב בהרבה מ-3000 cycles). בכותרת מעלה כתוב כעת CPU .speed: max 100% cycles

טיפ לעובודה קלה עם Dosbox: אפשר לקבוע לדוסבוקס אוסף של פקודות שיורצו באופן אוטומטי עם עליית התוכנה. קביעת פקודות אלו יכולה לחסוך לנו הקלדה רבה בעתיד. לדוגמה, נוכל לנגן את דוסבוקס כך ש בכל פעם שהוא עולה, הוא יגיע מיד לספרייה הנקונה.

הקובץ בו ניתן להגדיר את הפקודות נקרא dosbox-0.74.conf ונitinן להגיאו אליו דרך תפריט הפתיחה של חלונות- יש לבחור ב-DOSBox 0.74 Options בתוך תפריט הדוסבוקס:



לאחר מכן הקלקה על לחץ שמאלי של העכבר פתחו לנו את הקובץ. נגלי עד הסוף, ונמצא את הכיתוב הבא:

[autoexec]

Lines in this section will be run at startup.

You can put your MOUNT lines here.

את הפקודות שלנו ניתן לכתוב מיד לאחר מכן. לדוגמה:

mount c: c:\

c:

cd tasm

cd bin

cycles = max

TASM Assembler

אסמבלי Assembler היא כל תוכנה שמסוגלת להפוך קוד בשפה כלשהי לשפת מוכנה. קיימים מגוון אסמבליים לשפת אסמבלי, אנחנו בחרנו לעבוד עם **TASM**, קיצור של **Turbo Assembler**. הגרסת האחרונה של TASM היא 5.0 והוא יצאה בשנת 1996.



לינקר Linker היא תוכנה אשר מבצעת המרת מושגתו מוכנה לקובץ הרצה. הלינקר יכול לקבל קובץ אחד או מספר קבצים בשפת מוכנה, ולהמיר אותם לקובץ הרצה היחיד. הלינקר שימושי במקרים שבהם התוכנה מחולקת בין מספר קבצים. לדוגמה, קובץ אחד כולל את התוכנית הראשית, SMBצעת קריאה קטיעי קוד שנמצאים בקבצים אחרים. הלינקר יודע לחבר בין הקריאה קטיעי קוד לבין קטיעי הקוד שנמצאים בקבצים האחרים.



לנוחותכם העלינו את **TASM** לאתר האינטרנט של התוכנית. הקובץ כולל בתוכו גם את האסמבלי, גם את הלינקר וגם את הדיבאגר.

ניתן להוריד את הקובץ **tasm.rar** בקישור:

לאחר ההורדה:

- צרו ספריה בשם **c:\tasm\bin**.

- פיתחו את קובץ **base.rar** והעתיקו לספריה את הקבצים (אם אין לכם תוכנה לפיתוח **rar**, חפשו "rar download" והורידו תוכנה חינמית).

- וודאו שהקובץ **base.asm** נמצא בספריה **bin**.

- כעת היכנסו לדיסק **Dosbox** והגיעו אל ספריית הדוח **cd** (על ידי פקודה **cd**, דוגמה ביצולו המסך שלמטה).

- הפקודה הבאה מmirah את **base.asm** לשפת מוכנה:

tasm /zi base.asm

- האופציה **/zi** שומרת את המידע הנדרש לטובת **debug**. נוצר לנו קובץ בשם **base.obj**.

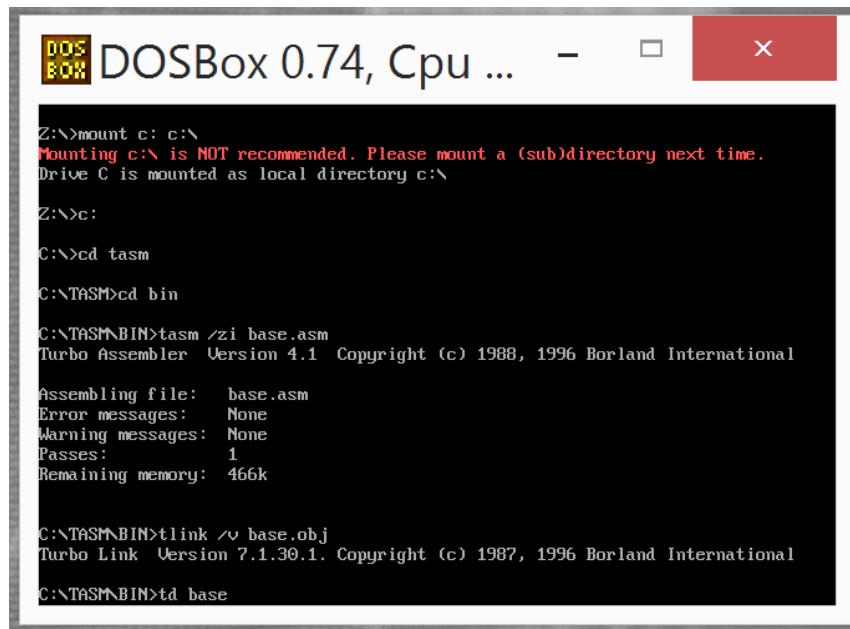
- הפקודה הבאה – שמובוצעת על ידי תוכנת הלינקר **tlink** –mirah את הקובץ בשפת המוכנה לקובץ הרצה:

tlink /v base.obj

האופציה **/v** שומרת את המידע החדש לטובת **debug**. נוצר קובץ בשם **base.exe**.

סיימנו!

כדי להרין את הקובץ שנוצר, כתבו base בשורת הפקודה והקישו enter, או כתבו td base כדי להרין אותו מהדיבאגר.



```
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 tasm  
  
C:\TASM>cd bin  
  
C:\TASM\BIN>tasm /zi base.asm  
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International  
  
Assembling file: base.asm  
Error messages: None  
Warning messages: None  
Passes: 1  
Remaining memory: 466k  
  
C:\TASM\BIN>tlink /v base.obj  
Turbo Link Version 7.1.30.1. Copyright (c) 1987, 1996 Borland International  
  
C:\TASM\BIN>td base
```

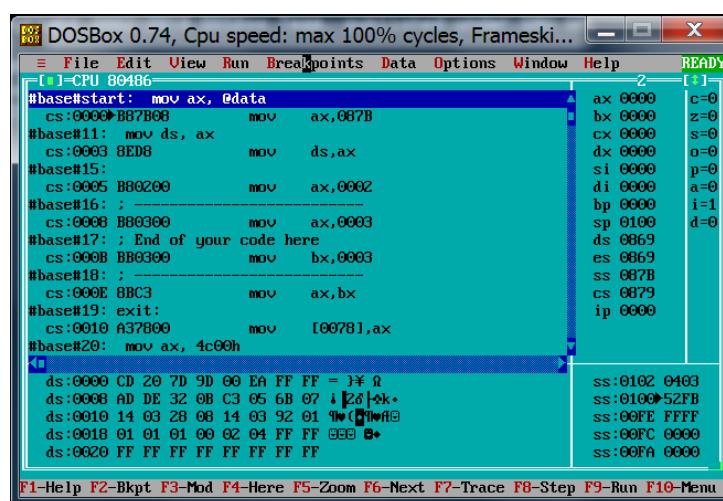
כעת, נרין את הקובץ base.exe באמצעות תוכנת דיבוג בשם Turbo Debugger.

Turbo Debugger – TD

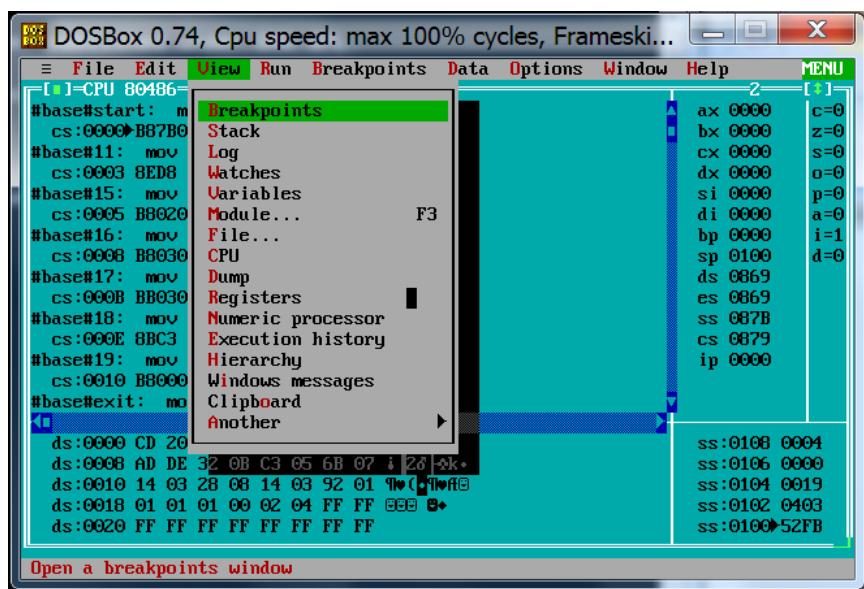
כדי להפעיל את תוכנת הדיבוג שלנו, נכתוב בדוסבוקס:

td base

וופיע המשך הבא:



אם במקומות המ██ זהה מופיעים דברים אחרים, היכנסו לחרטוטים למעלה (לחיצה על מקש F10), בחרו view ובחרו CPU שונפתה, בחרו את האפשרות :



השימוש ב-Turbo Debugger

הסבירים על כל הפקציות של Turbo Debugger, או בקיצור TD, ניתן למצוא בתפריט ה-**Help**. אנחנו נתיחס לפונקציות העיקריות של TD:

הרצת הקוד שורה אחרי שורה – ניתן לעשות באחד משני מצבים. מצב **Step**, על-ידי לחיצה על F8, עבר לשורת הקוד הבאה. אם שורת הקוד הנוכחית היא פרוצדורה, בהרצה של שורה יחידה נ被执行 את כל הפרוצדורה ונמצא ממנה. כדי לדבג פרוצדורה, משתמש במצב **Trace**, על-ידי לחיצה על F7. במצב זה, כשאנו מגעים לתחילה של פרוצדורה, הדיבאגר יתקדם כל פעם שורה יחידה בתוך הפרוצדורה.

נסו זאת – הריצו את base.exe באמצעות TD ובצעו הרצת של הקוד שורה אחרי שורה.

אפשרויות הרצת שימושיות נוספת הן F4, שמשמעותו – **Go to cursor** עד למקום שאנו עומדים עליו. הפקודה F9 מרים את התכנית עד עצירה.

אפשרויות View – באמצעות האפשרויות השונות אפשר לחקר כל אלמנט בתוכנית. הדיאלוג השימושים ביותר הם:

- ה-CPU, שמציג לנו את שורות הקוד ותרגום לשפה מכונה, כמו כן את סגמנט ה-**DATA**:

```
[CPU 80486]
cs:FFF8 0000 add [bx+si],al
cs:FFF9 0000 add [bx+si],al
cs:FFFF 0034 add [si],dh
cs:0001 124523 adc al,[di+23]
#base#start: mov ax, @data
cs:0004 B87A08 mov ax,087A
#base#13: mov ds, ax
cs:0007 8ED8 mov ds,ax
#base#exit: mov ax, 4C00h
cs:0009 B8004C mov ax,4C00
#base#23: int 21h
cs:000C CD21 int 21
cs:000E 0000 add [bx+si],al
cs:0010 0000 add [bx+si],al
cs:0012 0000 add [bx+si],al

ds:0000 CD 20 7D 9D 00 EA FF FF = }¥Ω
ds:0008 AD DE 32 0B C3 05 6B 07 i | 2d | ok.
ds:0010 14 03 28 08 14 03 92 01 9e | 1wff |
ds:0018 01 01 01 00 02 04 FF FF 0000 0+
ds:0020 FF FF FF FF FF FF FF FF FF
```

- הרגיסטרים והדגלים:

J=Regs=1	J=
ax 0000	c=0
bx 0000	z=0
s=0	
dx 0000	o=0
si 0000	p=0
di 0000	a=0
bp 0000	i=1
sp 0100	d=0
ds 0069	
es 0069	
ss 007A	
cs 0079	
ip 0004	

ניח לשונת את ערко של כל רגיסטר על-ידי סימונו והחלצת ערד כלשון

ה-STACK, שמאז לא את סמנט ה-STACK – כרגע אנחנו לא ידעים עליה הרבה. אבל משתמש במחשב:

```
ss:0102 0403  
ss:0100>52FB  
ss:00FE FFFF  
ss:00FC 0000  
ss:00FA 0000
```

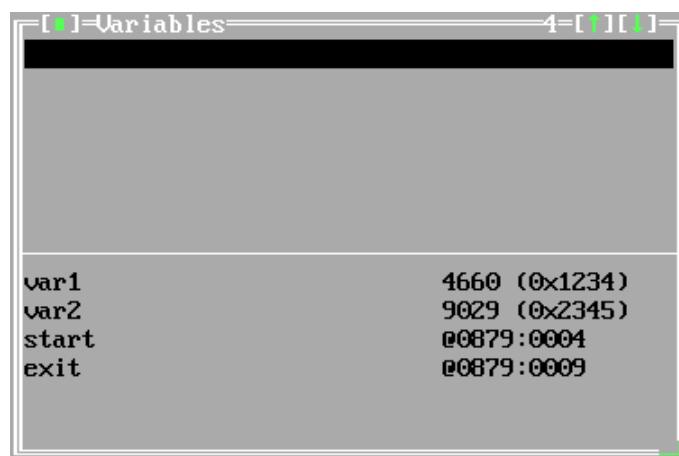
– Watches – אנחנו יכולים לקחת משתנה או ביטוי כלשהו ולהכניס אותו בתור watch. ערך הביטוי יעדכן באופן דינמי. לדוגמה, יצרנו שני משתנים ב-SEG-DATABASE:

Var1	dw	1234h
Var2	dw	2345h

הכנסנו לתוך Watch את הביטוי `var1+var2` וקיבלנו:

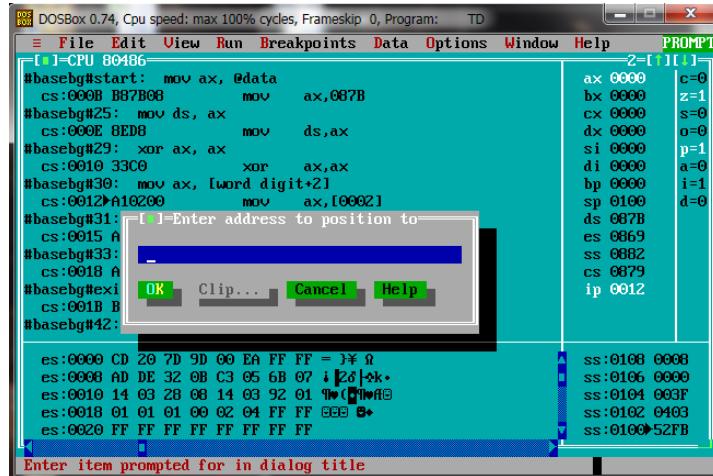


– מציג לנו את רשימת כל המשתנים והתיוויות בתוכנית. לדוגמה:

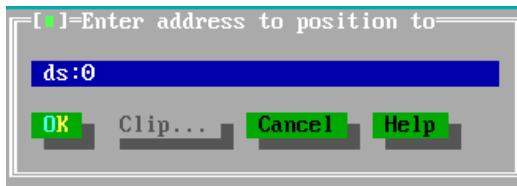


מעבר למקום מבוקש בזיכרון

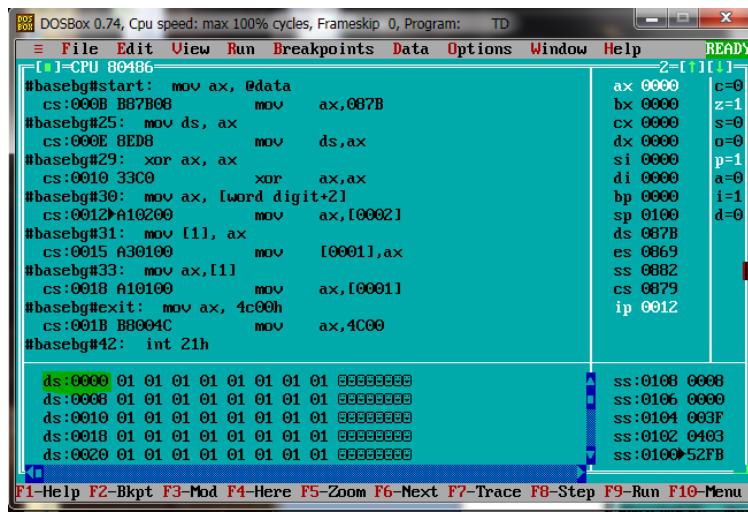
עמדו על החלק התההון של המסר, היכן שכותבים הערכים שנמצאים בזיכרון. לחיצה על CTRL+G תפתח את המסך הבא:



בחלון שנפתח נכניס את הכתובת המבוקשת (נניח, 0:ds), כדי לראות מה יש במקטע הנתונים():



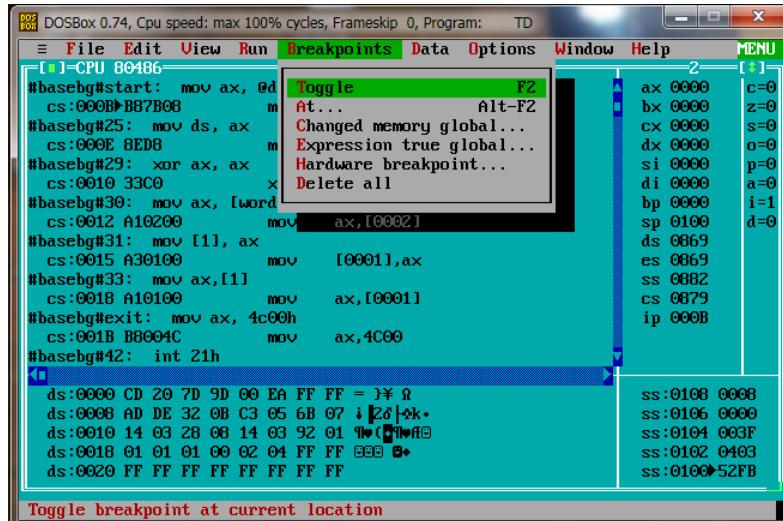
ופעללה זו "תקפי" אותו לאוצר המבוקש:



שימוש לב לכך שבתחלת התוכנית ds עדין אינו מוגדר להצביע על המיקום של DATASEG, רק אחרי שתישורות הקוד הראשונית ds מקבל את הערך הנכון. לכן, עלינו ללחוץ פעמיים על F8 לפני שנבצע את הפעולות  האחרונות.

שימוש ב-breakpoints

אנו יכולים להגדיר נקודת עצירה, או breakpoint בכל שורה שנרצה בקוד. כדי לעשות זאת, علينا לסמן השורה בה אנחנו שהתכוון התוך ובתפריט Breakpoints לבחור ב-Toggle. השורה הופכת לצבועה באדום. חזרה על הפעולה תבטל את נקודת העצירה.



לאחר שקבענו breakpoint, אם נרים את התוכנית בעזרת פקודה Run (F9) היא תעצור בשורת הקוד שיש עליה breakpoint. זה יאפשר לנו לנקודות בקוד שאנו חדשים ושיש בהן באים.

הריצו את base.exe בעזרת הדיבאגר. קיבעו breakpoint ב指令 mov ax, 4C00 בשורה קודה והוא לא שורט הקוד בזמן ריצה תוך שימוש ב-F9. בידקו את הערך של הרגיסטרים השונים והשוו את הערך שלם מה שציפיתם שהם יהיה.



טיפים לעובדה עם TD

- יציאה מהתוכנה אל מסך ה-Dosbox: לחיצה על ALT+X

- הריצה מחדש של התוכנית: CTRL+F2

- דפוד בתרפירים בעזרת המקלדת (למקרה שהעכבר נתקע): לחיצה על F10 ומשימוש במקשי החיצים. לחיצה על Tab למעבר בין החלונות השונים של הדיבאגר.

- מעבר למצב מסך מלא (ויציאה ממנו): לחיצה על ALT+Enter

- יציאה מחלון הדיבאגר לתוכנות אחרות, ללא שימוש בעכבר: מקש "חלונות" (המקש שבין Alt ל-Ctrl)

תרגיל מחקר (הרתהה) – קידוד פקודות אסמלטי ל-**Opcodes**



נחקור כיצד פקודה **mov** מיתרגמת לפקודות מכונה.

השאלה שנחנו מעוניינים להזכיר, אם כך, היא איך פקודה **mov** מיתרגמת לאחדות ואפסים בסגמנט הקוד שבזיכרנו. נחקור את פקודה **mov** כיון שכבר נתקלנו בה בעבר. פקודות אסמלטי אחרות מיתרגמות באופן דומה. עובdot החקיר היא **למצביא את ה-Opcode של הפקודה mov ax, dx בלא רק באמצעות התבוננות בפקודות אחרות.**

הדרך וכליים לביצוע המשימה:

1. הקובץ **base.asm** מכיל שיד של תכנית. הכניסו את הקוד שאותם רוצים להריץ לאחר העראה **.here** שמופיעה לאחר שורת הקוד

```
mov ds, ax
```

2. לאחר ההמרת הקוד מכונה ויצירת קובץ הרצתה, הריצו את תוכנת **TurboDebugger**.
3. התבוננו איך כל פקודה אסמלטי מיתרגמת לקוד מכונה. לדוגמה, הפקודה **2 mov ax, 2** היתרגמה לקוד המכונה **.B80200**.

Register	Value
ax	0000
bx	0000
cx	0000
dx	0000
si	0000
di	0000
bp	0000
sp	0100
ds	0869
es	0869
ss	087B
cs	0879
ip	0000
ds	0000 CD 20 7D 9D 00 EA FF FF = 3FFF
ss	0000 AD DE 32 0B C3 05 6B 07 i [28]ak.
ds	0010 14 03 28 08 14 03 92 01 90 00 00 00
ss	0018 01 01 01 00 02 04 FF FF 000 000
ds	0020 FF
ss	0108 0004
ss	0106 0000
ss	0104 0019
ss	0102 0403
ss	0100 52FB

נשאל את עצמנו כמה שאלות מוחות:

.1. לכמה בתים מתורגמת פקודת `mov`? ננסה גרסאות שונות של פקודת `mov` ונבדוק כמה בתים הן תופסות בזיכרון.

לדוגמה:

`mov ax,5`

`mov ax, bx`

`mov [120], ax`

.2. ננסה לטעון קבועים שונים לרגיסטר ונבדוק איך משתנה ה-`Opcode`, Opcode, לדוגמה:

`mov ax, 5`

`mov ax, 6`

.3. ננסה לטעון קבוע לרגיסטרים שונים ונבדוק את ההשפעה על ה-`Opcode`:

`mov ax, 2`

`mov bx, 2`

`mov cx, 2`

`mov dx, 2`

.4. נבדוק איך ה-`Opcode` משתנה שכטוענים רגיסטר לתוכך רגיסטר אחר:

`mov bx, ax`

`mov ax, cx`

זורא למשימת המחבר: נתונה הפקודה `mov dx, ax`. יש למצוא את התרגום שלו לשפת מכונה. יש למצוא את הפתרון בלי לתרגם את הפקודה ולבדוק איך האסמבולר מתרגם אותה לשפת מכונה, אלא רק עליידי מחבר איך האסמבולר מתרגם לשפת מכונה פקודות דומות.

סיכום

בפרק זה הכרנו את כלי העבודה שלנו ואת סביבת התוכנות באסמלבי, שנעבود אליה מעכשו וайлד. קיימות סביבות עבודה שונות ומגוונות, אך למען האחדות המקדנו בסביבת עבודה אחת. הכלים שקיבלנו בפרק זה הם:

Editor Notepad++ -

הרצה דרך Command line, כולל פקודות DOS -

אמולטור DOSBOX -

אסמלבר Turbo Assembly -

لينקר Tlink -

דייבאגר Turbo Debugger -

שליטה טובה בכלים אלו היא הכרחית להמשך. זה הזמן לוודא שככל הכלים מותקנים אצלכם במחשב ועובדים כמו שצריך. בפרק הבא אנחנו מתחילה לתוכנת.

פרק 5 IP, FLAGS – 5

מבוא

הລקם מפרק זה – רגיסטר IP ודגל האפס – נדרשים עבור ייחודת המעבדה שנלמדת בתכנית גבהים. יתר הנושאים מסומנים

 כנושאי הרחבה (תמצאו לידם את הסימן ), הם אינם נדרשים לייחודת המעבדה של תכנית גבהים אך הם נדרשים על ידי משרד החינוך בבחינת הבגרות במחשבים.

כשערכנו היכרות עם המעבד ומרכיביו, הזכרנו בקצרה שני רגיסטרים ייעודיים, **FLAGS** ו**REGISTER IP**.

IP - Instruction Pointer- מצביע ההוראה

FLAGS – דגליים

הבנה של הריגיסטרים הללו חשובה לטובת נושאים שנעמיק בהם בהמשך: רגיסטר ה-IP ישמש אותנו בין היתר בנושאי פרוצדורות ופסיקות. רגיסטר ה-FLAGS ישמש אותנו בין היתר לכתיבת תנאים לוגיים ולולאות.

בפרק זה נפרט אודות הריגיסטרים הללו, ונשתמש בידע שרכשנו אודות סביבת העבודה על מנת ללמוד את הריגיסטרים הללו בצורה מעשית.

IP – Instruction Pointer

רגיסטר ה-IP מזijk את הכתובת של הפקודה הבאה לביצוע. זהו רגיסטר של 16 בית שŁmuש משמש כמצביע (pointer) להpekודה .

IDEAL

MODEL small

STACK 100h

DATASEG

CODESEG

start:

```
    mov ax, @data
    mov ds, ax
    mov ax, 1234h
    mov bx, 0
    mov bl, 34h
    mov cx, 0
    mov ch, 12h
```

exit:

```

mov  ax, 4c00h
int   21h
END start

```

באו נראה מה קורה ל-IP עם ריצת התוכנית: הוכנית נמצאת כרגע בשורה 0, IP=0005, כלומר IP מצביע על הבית השלישי בסגמנט הקוד.

Register	Value	Description
ax	087B	c=0
bx	0000	z=0
cx	0000	s=0
dx	0000	o=0
si	0000	p=0
di	0000	a=0
bp	0000	i=1
sp	0100	d=0
ds	087B	
es	0869	
ss	087B	
cs	0879	
ip	0005	

כעת נקיש על F8, פעולה שתקדם את הוכנית שלנו בשורת קוד אחת:

Register	Value	Description
ax	1234	c=0
bx	0000	z=0
cx	0000	s=0
dx	0000	o=0
si	0000	p=0
di	0000	a=0
bp	0000	i=1
sp	0100	d=0
ds	087B	
es	0869	
ss	087B	
cs	0879	
ip	0008	

השינויים שהתרחשו הם:

- .1. הפס הכהול בפסק, שמציג את השורה הבאה שהמעבד יריץ, התקדם בשורה אחת (לשורה 0, (mov bx, 0).

.2. הפקודה `mov ax, 1234h`, וערך של `ax` השתנה בהתאם.

.3. רגיסטר `IP` השתנה מ-`0005` ל-`0008`, כלומר עלה בשלושה בתים.

נתעכט רגע על מנת להסביר את השימוש שחל ב-`IP`.

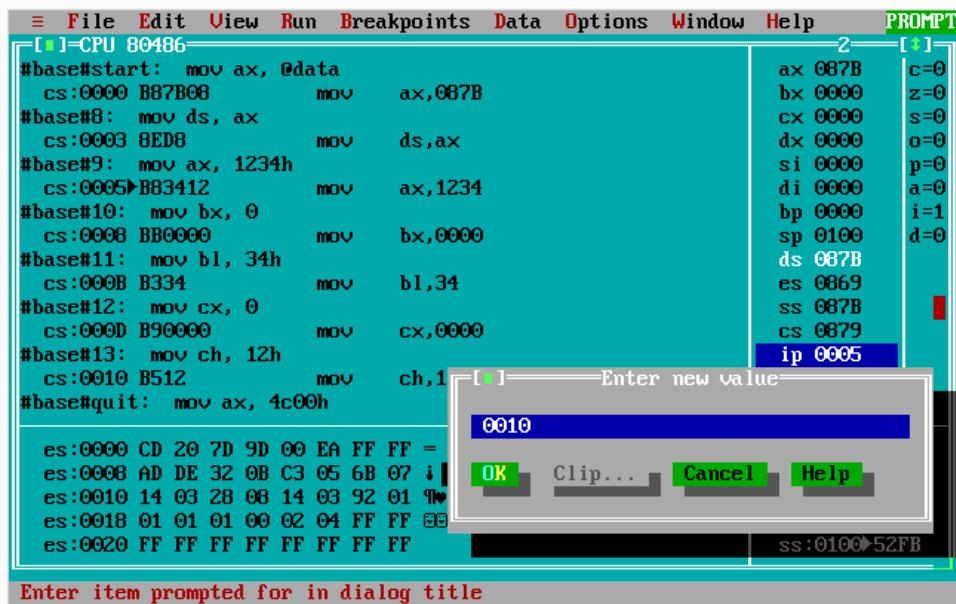
הסיבה שהתקדמות הייתה בשלושה בתים, היא משומם שהפקודה `mov ax, 1234h` תופסת שלושה בתים בזיכרון התוכנית. לא כל הפקודות תופסות שלושה בתים בזיכרון התוכנית – במקרים אלו `IP` יכול להשתנות בבית אחד או בשניים. בנוסף, נראה בהמשך מצבים בהם הערך של `IP` יקפוץ קדימה או אחורה, לפי הצורך.

תרגיל 5.1



A. המשיכו להריץ את שורות הקוד של התוכנית ועייקבו אחרי השימוש של `IP` בין פקודה לפקודה. האם אתם מזהים מצבים שבהם `IP` אינו משתנה בשלושה בתים?

B. הריצו את התוכנית מההתחלת, ויעזרו כאשר `IP=0005h`. כדי לשנות את ערכו של `IP` באופן ידני, הקליקו עליו עם המOUSE (או עימדו עליו ולהצוו על מקש enter) והכנסו ערך כלשהו. ייפתח לכם כדוגמת החלון הבא:

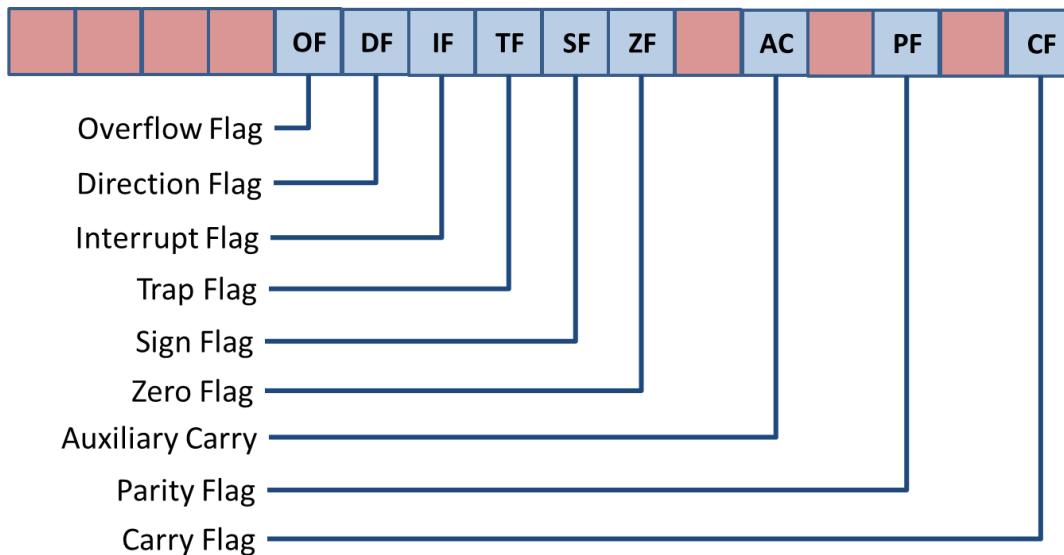


G. שנו את ערכו של `IP` ל-`0010h`, ליהצוו על `enter` וודאו ש-`IP` מקבל את הערך שהזנטם. עיברו לפקודה הבאה על-ידי לחיצה על `F8`. לאיו פקודה הגעתם? האם הפקודות שבין `0005` לבין `0010` במקטע הקוד אכן הטענו (シימו לב לערכים של הרגיסטרים `ax, bx, cx`?)

Processor Status Register – FLAGS

הרגיסטר Processor Status Register, מוכר יותר בכינוי רגיסטר הדגלים או FLAGS. רגיסטר FLAGS שונה מיתר הרגיסטרים. הוא אינו מחזיק ערך באורך 8 או 16 ביטים, אלא הוא אוסף של אותות חממיים בני בית אחד, שעוזרים לקבוע את מצב המעבד. למרות שיש ב-SFLAGS 16 ביטים, רק חלקם יש שימוש.

להלן שמות הדגלים והמקום שלהם בתחום רגיסטר ה-FLAGS:



מ בין הדגלים השונים, קיימים ארבעה דגלי תנאים – Condition Codes :

Zero Flag -

Overflow Flag -

Carry Flag -

Sign Flag -

לדגלי התנאים החשובות מיוחדת, מכיוון שהמעבד משתמש בהם כדי לבצע פקודות שכוללות תנאים לוגיים ופקודות בקרה (פקודות מסווג "אם התנאי הבא מתקיים אז בצע..."). כשןלמד פקודות קפיצה ובקרה נראה איך משתמשים בהם.



דגל האפס – Zero Flag

דגל האפס יהיה 1 אם הרצת הפקודה האחורה גרמה לאיפוס של אופrnd היעד. בכל מקרה אחר, ערכו יהיה 0. (הערה: לכל זה קיימים שני חיריגים: פעולות כפל וחילוק ופקודת העתקה mov. לאחר פעולות כפל וחילוק לא ניתן להסתמך על ערכו של דגל האפס לקבעת התוצאה, ואילו פקודה mov אינה משפיעה על מצב הדגלים).

אופrnd היעד - Destination Operand – המוקם אליו מועתקת התוצאה. מקום זה יכול להיות רגיסטר או כתובת בזיכרון.

דוגמה: נתען את הערך 4Bh גם לתוך al וגם לתוך ah (חישבו: איך אפשר לעשות את זה באמצעות פקודה יחידה?). כעת נבצע פעולה חיסור (subtract) בין הרגיסטרים. שימוש לב לתוצאה של הרצת הפקודות על הדגמים:

```
mov al, 4Bh      ; 75 decimal
mov ah, 4Bh      ; 75 decimal
sub al, ah       ; subtract al minus ah, result is 0
```

[↑	=	c=0
			z=1
			s=0
			o=0
			p=1
			a=0
			i=1
			d=0

דגל האפס מסומן בתוכנה באות 'z'

דוגמה נוספת:

```
mov al, 0FFh      ; 255 decimal
mov ah, 01h      ; 1 decimal
add al, ah       ; add al and ah, result is 256
```

דוגמה זו ראויה לחשומת לב מיוחדת. תוצאה תרגיל זה הייתה 100h, אך כיוון שתוצאה החיבור נשמרת ב-al, רגיסטר שגודלו בית, הספירה השמאלית "גולשת" ולרגיסטר נשמר רק 00h, לכן גם במקרה זה הדגל האפס מקבל ערך 1.

דגל האפס שימושי בעיקר בזמן פועלות השוואה בין זוג ערכים. כל פעולה השוואתית גורמת למעבד להריץ "מאחורי הקלעים" פעולות חישור. אם תוצאה החישור היא אפס, דגל האפס מקבל ערך 1. על-ידי בדיקת דגל האפס ניתן לדעת אם קיים שווון בין שני ערכים, ומכאן החשיבות העיקרית של ה懿כות עימיו.

תרגיל 5.2



כתבו קוד שמדליק את דגל האפס בעזרת פועלות חישור, השתמשו ברגיטרים של 16 ביט. כתבו קוד שונה שמודליק את דגל האפס בעזרת פועלות חיבור, השתמשו ברגיטרים של 16 ביט.



דגל הגלישה – Overflow Flag

פקודות שהתוכאה שלתן "גולשת" מתחום ערכם מוגדר, גורמות לדגל הגלישה לקבל את הערך 1, כלומר "לדлок". תחום הערכים תלוי במספר הביטים המשמשים לייצוג מספרים בפועל, כאשר מתייחסים אל הייצוג של המספרים בתור signed. לדוגמה, אם אנחנו מבצעים פועלות בעזרת משתנים או רגיטרים בגודל של 8 ביטים, שכור יוכלים לשמור מספרים signed בתחום מ -128 עד $+127$, דגל הגלישה יידלק אם התוצאה הנשמרת באופרנד השני חורגת מהתחום $(-128) \text{ עד } +127$. אם תוצאה הפועלה המתמטית אינה יוצרת גליישה, המעבד ינקה את הדגל ויציב בו את הערך 0.

נוקח לדוגמה את al, שגודלו 8 ביט:



```
mov al, 64h      ; 100 decimal
mov ah, 28h      ; 40 decimal
add al, ah       ; result is 140, out of 8 bit signed range
```

	1	0	1	0	0	0	0
c=0							
z=0							
s=1							
o=1							
p=0							
a=0							
i=1							
d=0							

דגל הגלישה מסומן בתוכנה באות '0'

חישבו – עבור משתנה (או רגיטר) בגודל 16 ביטים, מהו תחום הערכים שמעבר לו יידלק דגל הגלישה?



תשובה: הערך signed הנמוך ביותר שנייתן לייצג על ידי 16 ביטים הוא (-2^{15}) , כלומר $-32,768$. הערך הגבוה ביותר הוא $(2^{15}-1)$, כלומר $+32,767$.

5.3 תרגיל

העתיקו את שורות הקוד שבסעיף זה לתוך `base.asm`, הריצו ועיקבו אחרי השינוי בדגל הגלישה.



דגל הנשא – Carry Flag

כמו דגל הגלישה, גם דגל הנשא, Carry Flag, נדלק עקב חריגה מתחום ערכאים מוגדר, אך תחום הערכאים שמודגדר לו הוא שונה. תחום הערכאים תלוי במספר הביטים שימושיים ליצוג מספרים בפועל, כאשר מתייחסים אל הייצוג של המספרים בתור `unsigned`. לדוגמה, אם אנחנו מבצעים פעולות בעזרת משתנים או רגיסטרים בגודל של 8 ביטים, שכוכר יכולים לשמר מספרים `unsigned` בתחום מ-0 עד +255, דגל הנשא יידלק אם התוצאה הנשמרת באופרנד היעד חורגת מתחום 0 עד +255. עבור אופרנד יעד בגודל 16 ביטים התוחם הוא מ-0 ועד +65,535.

לדוגמא:

```
mov al, 0C8h      ; 200 decimal
mov ah, 64h      ; 100 decimal
add al, ah       ; result is 300, out of 8 bit unsigned range
```

[]
c=1		
z=0		
s=0		
o=0		
p=0		
a=0		
i=1		
d=0		

דגל הנשא מסומן בתוכנה באות 'c'.

ביצוע פעולה חיסור בהמחסר (המספר שאותו מהחסירים) גדול מהמחוסר (המספר ממנו מתבצע החיסור) דורש שימוש בנשא שלילי (היזכרו בהסבר על פעולות חיסור בפרק על שיטות ספירה) ולכן ידליק את דגל הנשא.

לדוגמא:

```
mov al, 1h
mov bl, 2h
sub al, bl
```



לאחר ביצוע פעולה החיסור, ערכו של al הוא 0FFh, שבייצוג בתור מספר `unsigned` ערכו שווה 255. התוצאה 255 מתקבלת תוך שימוש בנשא שלילי ("פרטנו" ביט שערכו 256) ולכן נדלק דגל הנשא.

נסתכל על דוגמה אחרת, שאינה מدلיקה את דגל הנשא, ונבון את הסיבה לכך:

```
mov al, -128d
mov ah, 40d
add al, ah      ; result is out of 8 bit unsigned range?
```

לכוארה תוצאה החישוב האחרון צריכה להיות -88, מחוץ לתחום המוגדר בתור `unsigned`, ולהدليل את דגל הנשא. אולם, בפועל זה לא הינו עקביהם - התיחסנו אל `al` בתור מספר `signed` ובדקנו אם התוצאה נכנסת במספר `unsigned`. כדי להיות עקביים, צריך להתייחס אל כל הערכים בתור `unsigned`. נמצא את ערכו של `al`: לאחר שהכננו -128- לתוך `al`, ערכו של `al` הוא `80h`. זה גם הייצוג של +128, כאשר מפרשים את הערך של `al` בתור מספר `unsigned`. כעת אם נחבר +128 ו עוד +40, נקבל +168, מספר שאינו חורג מהתחום המותר.

תרגיל 5.4



העתיקו את שורות הקוד שבסעיף זה לתוך `base.asm`, הריצו ועיקבו אחרי השינוי בדגל הנשא.

דגל הסימן – Sign Flag



ערך של דגל הסימן יהיה 1 אם הביט ה

ללא
 ביתן (השמאלי ביותר) של התוצאה נשמרת באופרנד היעד הוא 1. אחרת ערכו של דגל הסימן יהיה 0. דרכים פשוטות לראות אם מספר הוא שלילי, כאשר מתיחסים אליו כמספר מסומן `Signed`:

- ביצוג בינארי – הביט השמאלי הוא בעל ערך 1.
- ביצוג הקסדצימלי – ה-`nibble` השמאלי נמצא בתחום שבין 8 ל-F. לדוגמה: פועלה שתוצאה שלה היא 0F100h, 0A3h, 088h (משתנים או רגיסטרים בגודל בית), או פועלה שתוצאה שלה היא 0A300h, 08800h (משתנים או רגיסטרים בגודל מיליה).

תרגיל 5.5



שנו את `base.asm` כך שבזמן ההרצה ישנה ערכה של דגל הסימן. הריצו ועיקבו אחרי השינוי.

דגל הכוון – Direction Flag



דגל הכוון משמש בעבודה עם מהרווזות. כשדגל הכוון שווה 0, המעבד עובר על אלמנטים במחזורות מהכתובות הנמוכות אל עבר הכתובות הגבוהות. כשדגל הכוון שווה 1, המעביר הוא מהכתובות הגבוהות אל הנמוכות.



דגל הפסיקות – Interrupt Flag

דגל זה שולט על יכולת המעבד להיענות למאורעות חיצוניים שנקראים "פסיקות" (**Interrupts**). תוכניות מסוימות מכילות קוד שה חייב לרוץ בראץ' ולא הפרעה של מאורעות חיצוניים. לפני הרצת קטעי קוד אלו, מושנים את דגל הפסיקות ל-0, וכך מבטיחים שהקוד ירוץ ללא הפרעה. בסיום ריצת הקוד מאפשרים חזרה את הפסיקות, על ידי קביעת ערך דגל הפסיקות ל-1.



דגל צעד היחיד – Trace Flag

דגל צעד היחיד מכוון את המעבד למצב **Trace**. במצב זה, המעבד מפסיק את פעולה העיבוד לאחר כל שורת קוד ו מעביר את השליטה לתוכנה החיצונית. פעללה זו מאפשרת את פעולהן של תוכנות debugger כגון **turbo debugger** ועוד. אם ערכו של הדגל שווה ל-0, המעבד מרים את התוכנה ללא עצירה.



דגל זוגיות – Parity Flag

זהו אינו דגל שנשאמש בו. ערכו של דגל הזוגיות נקבע לפי כמות האחדות בשמונת הביטים התחתיונים של כל פעולה חישוב. אם בסופה של פעולה חישוב, בשמונת הביטים התחתיונים יש מספר זוגי של '1' (0,2,4,8), דגל הזוגיות מקבל ערך 1. אחרת יתפס.



דגל נשא עוז – Auxiliary Flag

זהו אינו דגל שנשאמש בו. מקבל ערך 1 כאשר יש לווה או שארית מ-4 הסיבות התחתיונות של הרגיסטר AL. בכל מקרה אחר ערכו 0.

תרגיל 5.6



אם קטע הקוד הבא ידליק את דגל הגלישה?

```
mov ax, 0
```

```
mov bx, 8888h
```

```
sub ax, bx
```

תרגיל 5.7

מה תהיה השפעת הפקודות מתרגיל 5.6 על דגל הנשא?

תרגיל 5.8 (אתגר)

תנו דוגמה לקטע קוד שעល ידי פעולה אחת של חיבור או חיסור, מדליק בו זמנית את דגל הגלישה, את דגל הנשא ואת דגל הסימן.

תרגיל 5.9 (אתגר)

תנו דוגמה לקטע קוד שעלי ידי פעולה אחת של חיבור או חיסור, מדליק בו זמנית את דגל הגלישה, את דגל הנשא ואת דגל האפס.

סיכום

בפרק זה למדנו אוזות הרגיסטרים הייעודיים (Special Purpose Registers), רגיסטר ה-IP ורגיסטר הדגלים. ראיינו איך ערכו של IP משתנה בזמן ריצת התוכנית וגילינו מה קורה כמשנים אותו.

חקרנו את רגיסטר הדגלים, והעמקנו במיוחד בדגלים שיישמשו אותן בהמשך:

- דגל האפס
- דגל הנשא
- דגל הגלישה
- דגל הסימן

ראיינו איזה סוג של פעולות גורמות לכך שערךם של הדגלים הללו ישתנה.

בהמשך, נשתמש בשילוב של רגיסטר ה-IP עם רגיסטר הדגלים על מנת לבצע פעולות בדיקה וקפיצה, שהין הבסיס לכתיבה אלגוריתמים בתוכנה.

פרק 6 – הגדרת משתנים ופקודת mov

מבוא

את הפקודות הבסיסיות בשפת אסמלטי אנחנו נלמד בשלושה חלקים:

- בחלק הראשון, בו נעסק בפרק זה, נלמד איך מגדירים משתנים, איך קובעים בהם ערכיהם ואיך משתמשים בפקודת mov כדי להעתיק ערכים אל משתנים בזיכרון או אל רגיסטרים.
 - בחלק השני נלמד פקודות אריתמטיות (פעולות חיבור), פקודות לוגיות ופקודות הזזה.
 - בחלק השלישי נלמד איך יוצרים תוכנית עם תנאים לוגיים ("אם מתקיימים תנאי זה, בצע פעולה זו..."), באמצעות פקודות השוואה, קפיצות ולולאות.
- עם סיום הפרקים הללו נוכל לכתוב תוכניות צנועות. לדוגמה, מציאה של האיבר הגדול ביותר מתוך רשימה איברים, מינון של איברים מהגדול לקטן, ספירת מספר איברים ברשימה וכדומה.
- או קדימה, באו נראה איך מגדירים משתנים באסמלטי.

הגדרת משתנים

כפי שראינו בפרק על ארגון המחשב, ניתן לגשת לכל מקום בזיכרון על-ידי כתובות הזיכרון. הפקודה:

`mov al, [ds: 1h]`

תטען לתוך al את הערך שבsegment ds ובOFFSET 1h (כלומר בית אחד לאחר תחילת הsegment DS). ה索גרים המרובעים או מירים לאסמלט שצורך להתיחס לערך שנמצא בכתובת שבתוך הסוגרים. ככלומר אם בכתובת `ds:1h` נמצא הערך 5, או הערך 5 יועתק לתוך al. הבעייה היא שם התוכנית שלנו מלאה בהצבות זיכרון בשיטה זו, יהיה לנו קשה לdebug אותה. נctrיך לזכור בראש מה אנחנו שומרים במקום `h`. כמו כן, אם נחליט לשנות מעט את הסדר שבו אנחנו שומרים את הערכים, לדוגמה לשים משתנה אחר במקום `h`, או נctrיך לשנות את הקוד של התוכנית.

אחד מתקידיו החשובים של האסמלט הוא לאפשר לתוכנת להשתמש בשמות משמעותיים בשביל מקומות בזיכרון. לדוגמה, אם הכתובת `ds:1h` תיקרא age, אז כל פעם שנפנה לכתובת `age:ds`. כתע נתן יהיה לכתוב את הקוד:

שلونו כך:

`mov al, [age]`

הasmBLR לא רק נותן תווית שם למקומות בזיכרון, הוא גם דואג בשביבלו להקצתה המקום ואפלו – אם נרצה כך – לקביעת ערך התחלתי למקום שהקצנו בזיכרון. התוויות שאנחנו נותנים לכתובות בזיכרון קרוויות **משתנים**



. (**Variables**)



משתנה הוא שם שניתן לאורו מוגדר בזיכרון, שיכל לקבל טווח מוגדר של ערכים.

אפשר לחשב על קובייה בתווך משתנה שמקבל ערך בטווח 1 עד 6,

וקובייה כפולה היא משתנה בגודל שתי קוביות, שמקבל ערך בטווח 2 עד 12.

הגדרת המשתנים הגלובליים נעשית בתוך סגמנט DATA. היוצרים בקובץ base.asm – מיד לאחר התווית "start" יש פעולה של האצת ערך לתוך ds:

```
mov ax, @data
```

```
mov ds, ax
```

פקודות אלו גורמות ל-ds להחזיק את ערך סגמנט DATA. בכל פעם שאנחנו פונים למשתנה, אם לא הגדרנו אחרת, האsmBLR יחפש אותו בסגמנט שכתוותו שמורה ב-ds. כדי למנוע בעיות מיותרות, אנחנו פותחים את התוכנית בהצבה של הערך הנכון בתוך ds. אחרת, פניה למשתנה age תגרום לטעינת ערך עם אופטט נכון, אך סגמנט אקראי כלשהו, ולכן age יועתקו ערכיו "זבל".

הקצתה מקום בזיכרון

הגדרת המשתנים תלואה בגודל המקום בזיכרון שאנחנו מבקשים להקצות למשתנה. נתחיל בהגדרת משתנים פשוטים, שכוללים רק ערך אחד.

כדי להגדיר משתנה בגודל בית אחד בתוך סגמנט DATA, משתמש בהגדירה כזו:

```
ByteVarName db ?
```

במקום ByteVarName יבוא שם המשתנה. ההגדרה db מסמנת לkompileר שהמשתנה תופס מקום בגודל בית אחד (DB – קיצור של Define Byte). סימן השאלה מציין שאנו רק מקצים מקום בזיכרון למשתנה ByteVarName, אך לא קבועים ערך למשתנה. במקרים אחרים איננו משנה את הערך האකראי שישנו בזיכרון. חשוב לזכור שבזיכרון תמיד יש ערך כלשהו, ואם נכתב:

```
mov al, [ByteVarName]
```

זו לא תהיה שגיאה; יעתיק לתוך הרגיסטר al ערך כלשהו, שהיה בזיכרון בזמן הקריאה, אך סביר שהערך זה יהיה "זבל" – אוסף סתמי של ביטים שהיה בזיכרון במקום המוגדר ולא ערך בעל משמעות.

אם אנחנו צריכים להגדיר משתנים נוספים, נוכל להוסיף עוד שורות כראצוננו:

DATASEG

```
ByteVarName db ?
```

```
ByteVar2 db ?
```

```
ByteVar3 db ?
```

ארגון המשתנים בזיכרון יהיה כך:

המקום הראשון (כתובת מספר 0) יוקצה למשתנה הראשון, ByteVarName, שהוא בכתבوبة ds:0. מיד לאחריו יוקצו יתר המשתנים. במקרה זה ByteVarName הוא בגודל בית אחד ולכן המשתנה הבא אחריו, ByteVar2, יהיה בכתבوبة ds:1 וכן הלאה.

אם נרצה לשלב משתנים מגדים שונים, כל מה שנוצרך לעשות הוא להגדיר אותם:

DATASEG

```
ByteVarName db ? ; allocate byte (8 bit) - DB: Define Byte
```

```
WordVarName dw ? ; allocate word (16 bit) - DW: Define Word
```

```
DoubleWordVarName dd ? ; allocate double word (32 bit) - DD:  
; Define Double
```

תרגיל 6.1



- א. הגדרו ב-DATASEG משתנה בשם var (קיצור של variable, משתנה) בגודל byte העתיקו את הקוד הבא לתוכה התוכנית שלכם בסegment CODESEG. הקוד מעתיק את הערך '5' לOMEM.var. כת קמפלו את התוכנית, היכנסו ל-Turbo Debugger (TD) והריצו את התוכנית. בדקו ש-var אכן מקבל את הערך '5'.

CODESEG

start:

```
mov ax, @data
mov ds, ax
mov [var], 5
```

exit:

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

END start

- ב. לפניה נן הסברנו על חשיבות הטעינה של כתובות סמנט נתוניים לתוך DATASEG. הכניסו את השרה mov ax(ds, ax) להערכה בעזרת הוספה נקייה פסיק בתחילת השורה. כת קמפלו שנייה, היכנסו ל-Turbo Debugger (TD) והריצו את התוכנית. האם קיבלתם את הערך שציפיתם לו?

Signed, Unsigned

שימוש לב שהגדירות אלו אינן קבועות אם המשתנה שהגדרנו הוא signed או unsigned – קלומר מה טווה הערכים שניתן להכניס למשתנים. כל מה שהגדירות אלו קבועות, הוא כמה בתים Bytes בזיכרונו יוקצו לטובות המשתנים.



נראה את הטענה האחורונה שלנו בעזרת דוגמה. נגידר משתנים בגודל Byte אחד:



DATASEG

Var1 db ?

Var2 db ?

בתוך CODESEG נניס למשתנים אלו ערכים:

mov [Var1], -120

mov [Var2], 136

נסתכל על הזיכרון ב-DATASEG לאחר הצבota אל. אפשר לראות שני המשתנים המ בעליהם אותו ערך! שני המשתנים מכילים את הערך 8h.

```
[ ]=Dump
ds :0000 88 88 00 00 00 00 00 00
ds :0008 00 00 00 00 00 00 00 00
ds :0010 00 00 00 00 00 00 00 00
ds :0018 00 00 00 00 00 00 00 00
```

איך יכול להיות שminus 120 שווה לפולס 136? יכול להיות שמצוינו באג במעבד?

התשובה היא, שאם נמיר את (120-) ואת 136+ לבינארי, בשיטת המשלים לשתיים, נקבל את אותם הביטים: 10001000. נסו זאת!

האם העובדה של שני מספרים שונים יש את אותו הייצוג אינה גורמת לשגיאות חישוב במעבד? בואו נבדוק את העניין היטיב. אנחנו נראה שאפשר לנקח מספר, +120, לחבר אותו פעמיים אחד לminus 120 ופעם אחרת לפולס 136 ולקבל בשני המקרים את אותה תוצאה:

- בפועלות חיבור של פולס 120 עם minus 120, התוצאה היא כמובן אפס.
- בפועלות חיבור של פולס 120 עם פולס 136, התוצאה היא 256, או 100h. כיוון שמדובר במשתנה בגודל בית, לא ניתן לשמור בתוכו את התוצאה במלואה, ונשמרים רק שמות הביטים הימניים – 00h, ככלומר אפס.
- בעזרה דוגמה זו ראיינו, שתת הערכים ששמורים בזיכרון המחשב אפשר לפרש בתור מספר signed או unsigned – האחריות לפרשנות היא של המשתמש. במעבד אין באג.

קבעת ערכים תחילה למשתנים

לא חייבים להזכיר ל-SEGCODE בשביל לטען ערכים תחילה למשתנים. כבר בזמן ההגדרה, אנחנו יכולים לקבוע למשתנים ערכים תחילה.

שימוש לבן כרך שנותן להגדיר כל ערך שנותן לשומר בדמות הביטים שמוגדרת למשתנה, והאSEMBLER מתייחס לכל המספרים כאילו שהם בסיס עשרוני, אלא אם נכתב לו אחרת.



DATASEG

ByteVarName1	db	200	; store the value 200 (C8h)
ByteVarName2	db	10010011b	; store the bits 10010011 (93h)
ByteVarName3	db	10h	; store the value 16 (10h)
ByteVarName4	db	'B'	; store the ASCII code of the letter B (42h)
ByteVarName5	db	-5	; store the value -5 (0FBh)
WordVarName	dw	1234h	; 34h in low address, 12h in high address
DoubleWordVarName	dd	-5	; store -5 as 32 bit format (0xFFFFFFFFBh)

כך ייראה הזיכרון לאחר פעולת ההקצאה:

```
[I=Dump
ds:0000 C8 93 10 42 FB 34 12 FB
ds:0008 FF FF FF 00 00 00 00 00
ds:0010 00 00 00 00 00 00 00 00
ds:0018 00 00 00 00 00 00 00 00]
```

שימוש לבן ייחודי לכך שהגדרנו שני משתנים שקיבלו את הערך מינוס 5. הראשון בגודל בית, השני בגודל מילה כפולה (ארבעה בתים). למרות שנראה לנו שלשניהם אותו ערך – כל אחד מהם מיוצג בזכרון בצורה אחרת.



אנחנו יכולים גם להגדיר משתנה בגודל בית, אבל לשים בו אוסף של תווים:

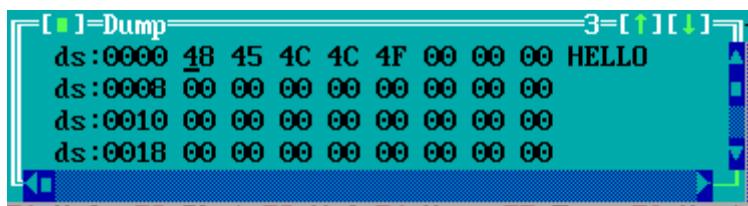
DATASEG

```
ByteVarName db 'HELLO'
```

לכוארה יש כאן בעיה, מכיוון שהגדרנו משתנה בגודל בית, שמכיל חמישה תווים ASCII, כלתו לפני עצמו הוא בגודל בית. למעשה, האSEMBLER יודע להתייחס להגדירה זו כאילו הגדרנו חמישה תווים שונים ושמרנו אותם בזכרון בזיה אחר זה:

DATASEG

ByteVarName1	db	'H'
ByteVarName2	db	'E'
ByteVarName3	db	'L'
ByteVarName4	db	'L'
ByteVarName5	db	'O'



יצרנו כאן אוסף של חמישה איברים מסוג בית. לאוסף של משתנים זהים יש שם מיוחד – **מערך (ARRAY)**. ספציפית, מערך שבכל אחד מהאיברים שלו שומר קוד ASCII נקרא **מחרוזת (STRING)**. כתה אנו בשלים לדיוון מפורט יותר על מערכים.



תרגיל 6.2: הקצת זיכרון



באמצעות התוכנית `base.asm`,agyro-base.asm מעתה ב-DATASEG משתנים בגודלים שונים – משתנה בגודל בית, משתנה בגודל מילה, משתנה בגודל מילה כפולה, וכן לחלקם ערכיהם, הוסיףamo שומר אוסף של תוכי ASCII. בסיום, עיברו על הזיכרון ב-DATASEG ומיצאו כל אחד מהמשתנים שהגדרכם.

הגדרת מערכים

מערכותם הם צורה נפוצה מאוד לשימוש מידע. מה שמייחד מערך מסתם, נניח, שמירה של משתנים שונים בזיכרון, הוא שבמרכז כל האיברים הם בעלי אותו גודל. כל משתנה שהוא חלק ממערך נקרא אלמנט ולכל אלמנט יש אינדקס, שקובע מה המיקום שלו במערך. המערך נשמר בזיכרון המחשב בצורה טורית, כאשר האלמנט הראשון, בעל אינדקס אפס, נמצא בכתובת הנמוכה ביותר ויתר האלמנטים בכתובות עוקבות אחריו.

כתובת הבסיס של המערכת היא הכתובתmana המערך מתחילה, השווה בדיקות לכתובת של האלמנט הראשון במערך. אפשר לדעת מה הכתובת של כל אלמנט במערך, בעזרת כתובות הבסיס של המערך, אינדקס האלמנט וגודלו האלמנט, באמצעות חישוב פשוט:

$$\text{ElementAddress} = \text{ArrayBaseAddress} + \text{Index} * \text{ElementSize}$$

לדוגמא, אם יש לנו מערך של מילים (words), והמערך מתחילה בכתובת 0200h בזיכרון, האלמנט באינדקס 0 במערך יהיה בכתובת 0200h (וימשך כМОון לתוך כתובת 0201h, שכן כל אלמנט הוא בגודל שני בתים), האלמנט בעל אינדקס 1 בכתובת 0202h, האלמנט בעל אינדקס 5 בכתובת Ah 020Ah וכן הלאה.

הגדרת מערך בסגמנט DATA מתבצעת בצורה הבאה:

ArrayName SizeOfElement N dup (?)

הוּא שם המערך, שניתן לקבוע לפי רצוננו.

SizeOfElement קובע מה גודל הזיכרון שמקצה לאלמנט, והוא צריך להיות אחד מסוגי הגדים dd, dw, db – תלוי אם אנחנו רוצים אלמנטים בגודל בית, מילה או מילה כפולה.

N הוא כМОון כמה איברים במערך. N חייב להיות מספר שלם וחובי.

Dup הוא קיצור של **duplicate**, שכפול.

במוקם סימן השאלה אנחנו יכולים לשים כל ערך חיובי ושלם, והוא ישוב נ פעמים. שימוש לב שערכים גדולים במיוחד עלולים לחרוג מהמקום שהוקצה לסגמנט הנתונים, אולי מבחינה מעשית בתוכניות אחרות אנו נתכוна אין זו מגבלה.

לדוגמא:

ArrayOfTenFives db 10 dup (5)

יצור מערך בן עשרה איברים, כל איבר בגודל בית, ערכו של כל איבר הוא 5:

```
[ ]=Dump-----[ ]=3
ds:0000 05 05 05 05 05 05 05 05 05 *****
ds:0008 05 05 00 00 00 00 00 00 00 **
ds:0010 00 00 00 00 00 00 00 00 00 00
ds:0018 00 00 00 00 00 00 00 00 00 00
```

האסמלר ישבול לתוכה את כל מה שכתבנו בסוגרים, גם אם זה יותר מאשר אחד.

אם נגידו מערך כזה:

ArrayOf1234 db 8 dup (1,2,3,4)

התוצאה תהיה:

```
ds:0000 01 02 03 04 01 02 03 04 000000000000
ds:0008 01 02 03 04 01 02 03 04 000000000000
ds:0010 01 02 03 04 01 02 03 04 000000000000
ds:0018 01 02 03 04 01 02 03 04 000000000000
```

כפי שאנו רואים הוגדר בזיכרון מערך בגודל 32 בתים – שמונה פעמים הרץ' 1,2,3,4, בגודל ארבעה בתים.

תרגיל 6.3: הקצת זיכרון לערך



בכל התרגילים הבאים, הריצו את התוכנית שיצרהם ב-**TD**, ומיצאו את המקום בזיכרון בו נשמרים המשתנים שהגדרתם.

- א. הגירו ב-**SEG** מערכים בגודלים שונים: 3, 5 ו-7 בתים (Bytes).
- ב. הגירו מערך של 10 בתים שמאותחלים לערך '5'. הגירו מערך של 10 מילימ' שמאותחלות לערך '5'. השוו את תומנת הזיכרון בשני המקרים!
- ג. הגירו מערך ששומר 20 פעמים את הרץ' 4,5,6, כל משתנה בגודל בית.

פקודת MOV

עד עכשיו סקרנו איך מגדירים משתנים בזיכרון. כעת נראה איך מבצעים העתקת זיכרון.

בפרקם הקודמים הזכרנו בקצתה את פקודת **mov**, קיצור של "move". הסברנו עליה בperfopf, רק כדי לאפשר דיוון במספר נושאים חשובים. כעת הגיע הזמן להסביר מפורט אודות הפקודה.

פקודת **mov** היא פקודה פשוטה למדי ומאוד שימושית. היא תמיד מקבלת את הצורה הבאה:

mov Destination, Source

פקודה זו יוצרת העתק של **Source** ומכניסה אותו לתוך **Destination**. הערך המקורי של **Source** נשאר ללא שינוי. גם ה-**Source** וגם ה-**Destination** נקראים **אופרנדים (Operands)**. במלחינים אחרים, פקודת **mov** מעתקה את המידע שבאופרנד המקורי לתוך אופרנד השני.



לדוגמה, כדי להכניס לרגיסטר **ax** את הערך 22 (דצימלי) נכתב:

mov ax, 22

אנו יכולים להכניס את הערך 22 באמצעות שימוש בהציגה הקסדצימלית או הבינארית שלו:

mov ax, 16h

mov ax, 00010110b

באופן דומה אנחנו יכולים להכניס ערכים ליתר הרגיסטרים:

```
mov bx, 199
```

```
mov cx, 2321
```

```
mov dx, 10
```

אפשר גם להעתיק לרגיסטר את הערך שנמצא ברגיסטר אחר:

```
mov ax, bx
```

פקודה זו תעתיק ל-ax את הערך שנמצא ב-bx. שימו לב לכך שה-bx לא ישנה בעקבות פעולה העתקה. הערך שבתוכו פשוט ישבכפל ל-ax. באופן דומה ניתן גם לכתוב:

```
mov ax, cx
```

```
mov ax, dx
```

```
mov ax, ax
```

הפקודה האחורונה היא חוקית, למרות שלא תנסה כלום. היא פשוט תעתיק את ערכו של ax חוזה לתוך ax.

בתתייחסים הבאים נסקור את הצורות המותרות לשימוש בפקודת mov:

```
mov register, register
```

```
mov register, constant
```

```
mov register, memory
```

```
mov memory, register
```

```
mov memory, constant
```

כפי שאתם רואים, לא ניתן לבצע העתקה memory אל memory. במקרה אחרות, כל העתקה מהזיכרון אל הזיכרון צריכה לעבור דרך רגיסטר. לאחר שקראותם את הפרק על מבנה המחשב והזיכרון אתם ודאי מבינים את ההיגיון בדבר: פסי הבדיקה והנתונים, שנוחזים להעברת מידע, מקשרים רק בין המעבד לזיכרון. כמו כן ה-opcode של פקודת mov לא תומכת בהעתקה מהזיכרון לזיכרון.

העתקה מרגיסטר לרגיסטר

`mov register, register`

הוראת זו מעתקה רגיסטר בן 16 ביט (או יותר, במעבדים מתקדמים יותר) לתוך רגיסטר אחר, שהייבב להיות בגודל זהה. דוגמאות לשימוש:

`mov ax, bx ; 16 bit registers`

`mov cl, dh ; 8 bit registers`

`mov si, bp ; The mov instruction works with ALL general purpose registers`

אם לא נקפיד על רגיסטרים בגודל זהה, לדוגמה:

`mov ax, bl`

נקבל שגיאת קומpileציה.

נציין כי העתקה בין שני רגיסטרים סגמנט איננה חוקית, לדוגמה `ds, cs mov`. יש צורך להעזר ברגיסטר כללי כלשהו כדי לבצע את העתקה. כמו כן הרגיסטר `cs` אינו יכול לשמש כאופרנד ייעד. לדוגמה: `mov cs,ax` שגויה. החלפת `cs` ב `ds` תהפכה לחוקית.

תרגיל 6.4



א. העתיקו את `ax` לתוך `bx`.

ב. העתיקו את `bx` לתוך `ax`.

ג. העתיקו את `ah` לתוך `ch`.

ד. העתיקו את `al` לתוך `dl`.

העתקה של קבוע לרגיסטר

`mov register, constant`

הוראה זו מעתקה קבוע לתוך רגיסטר.

שימוש לבן שגודל הרגיסטר צריך להיות בהתאם לנודל הקבוע – ניסיון להעתיק ערך שנייתן לשומר רק ב-16 ביט (לדוגמה 257) לתוך רגיסטר של 8 ביט, יוביל לשגיאת קומפילצייה.



דוגמאות לשימוש נכון:

`mov cl, 10h`

`mov ah, 10` ; Note the difference from last command! 10 decimal, not 10h (=16)

`mov ax, 555`

תרגיל 6.5



העתיקו לתוך al את הערך 100 (דצימלי), בשלוש דרכים ספירה: בייצוג העשרוני שלו, בייצוג הקסדצימלי וביצוג הבינארי (הערה: ייצוג בינארי מסוימים באות b, לדוגמה b1111b). הריצו את התוכנית ב-DS ובדקו שה-ax מקבל את הערכים הרצויים.

העתקה של רגיסטר אל תא בזכרון

`mov memory, register`

פקודה זו מעתקה את ערכו של הרגיסטר לתוך הכתובת בזכרון שמוחזקת על ידי אופרנד היעד. אופרנד היעד יכול להיות ערך קבוע, שווה לכתובת אליה אנחנו רוצים לפנות (Direct addressing (שיטת (шиיטת (Indexed addressing (Indirect addressing).

`mov [1], ax` ; Direct addressing

`mov [Var], ax` ; Another form of direct addressing, using a variable

`mov [bx], ax` ; Indirect addressing

`mov [bx+1], ax` ; Indexed addressing

בדוגמה הראשונה, ערכו של ax יועתק לתוך הזיכרון לכתובת מספר 1.

בדוגמה השנייה, ערכו של ax יועתק לתוך הזיכרון לכתובת אליה מצביע המשתנה Var.

בדוגמה השלישית, ערכו של ax יועתק לתוך הזיכרון לכתובת השמורה ב-ax. הפקודה זו:

```
mov [1], ax
```

שколоּה לפקודות הבאות:

```
mov bx, 1
```

```
mov [bx], ax
```

בדוגמה הריבועית, ערכו של ax יועתק לתוך הזיכרון לכתובת השמורה ב-ax ועוד 1, כלומר בית אחד אחרி הכתובת השמורה ב-ax.

תרגיל 6.6



- א. הגדרו בתוך DATASEG את המשתנה var בגודל בית, קבעו את ערכו ההתחלתי בתור 0. העתיקו לתוך al את הערך 100 (דצימלי) ולתוכן bx את הערך 2. הוסיפו את שורות הקוד הבאות לתוכנית:

```
mov [Var], al
```

```
mov [1], al
```

```
mov [bx], al
```

```
mov [bx+1], al
```

- ב. הריצו את התוכנית ב-IDT ועייקבו אחרי השינויים ב-DATASEG בזמן ריצת התוכנית. וודאו שככל ארבעת הבטים הראשונים ב-DATASEG מקבלים את הערך 100 (בייצוג הקסדצימלי שלו, כמובן).

העתקה של תא בזיכרון אל רגיסטר

השיטה להעתיק תא בזיכרון לתוך רגיסטר הן לבדוק כמו השיטות להעתיק רגיסטר לתוך תא בזיכרון, בהבדל אחד – האופרנדים הפלכמים:

```
mov register, memory
```

לכן, כל הדוגמאות שננתנו בסעיף הקודם תקפות, רק בהיפוך אופרנדים:

```
mov ax, [1]
```

```
mov ax, [Var]
```

```
mov ax, [bx]
```

```
mov ax, [bx+2]
```

תרגיל 6.7



כמפורט לתרגיל הקודם, הוסיפו לתוכנית את שורות הקוד הבאות (אחרי שורות הקוד של התרגיל הקודם):

```

mov  [Var], al
mov  [1], al
mov  [bx], al
mov  [bx+1], al
; -----1-----
mov  al, 0
mov  al, [var]
; -----2-----
mov  al, 0
mov  al, [1]
; -----3-----
mov  al, 0
mov  al, [bx]
; -----4-----
mov  al, 0
mov  al, [bx+1]

```

הristolו את התוכנית ב-IDT ועיקבו אחרי השינויים ב-al בזמן ריצת התוכנית. וודאו שבכל אחד מארבעת הקטעים al מקבל את הערך 100 (ביצוע הקסדצימלי שלו, כמובן).

העתקה של קבוע לזיכרון

`mov memory, constant`

לדוגמה כדי להכניס את הערך 5 לכתובת השמורה על-ידי `bx`:

`mov [bx], 5`

הערה: צורת הכתיבה המדויקת לפקודה זו היא:

`mov [byte ptr bx], 5`

או:

`mov [word ptr bx], 5`

הסביר על כך בהמשך.

רегистרים חוקיים לגישה לזיכרון

לא כל רגיסטר יכול לשמש לגישה לזיכרון. אי לכך, נשתמש תמיד ב-`bx`, `si` או `di`. שימוש ב-`ax`, `cx` או `dx` יגרום לשגיאת קומפילציה. לדוגמה הפוקודה:

`mov cx, [ax]`

תיזיר שגיאה:

```
Assembling file: base.asm
***Error*** base.asm(11) Illegal indexing mode
```

לכן, תמיד כשנרצה לגשת אל כתובת בזיכרון, נעבד לפי החוקים הבאים:

- נשתמש ב-`ax` כדי לשמר את הכתובת. דוגמאות להעתקות מהזיכרון ואל הזיכרון:

`mov ax, [bx]`

`mov [bx], ax`

- אם נרצה להגיע לכתובת שהיא בהיסט קבוע מהכתובת ששמורה על-ידי `ax`, נוסיף ל-`ax` ערך קבוע. דוגמאות:

`mov ax, [bx+2]`

`mov [bx+2], ax`

- אם נרצה להציג כתובות בהיסט משתנה מהכתובת שמורה עלי-ידי ax, נוכל להוסיף לו את si או את ds, אבל לא את שניהם יחד. לדוגמה:

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

```
mov ax, [bx+di]
```

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

```
mov [bx+di], ax
```

- אם יהיה צורך, נוכל להוסיף לי ax bx יחד את si או ds, וכן להוסיף קבוע. לדוגמה:

```
mov ax, [bx+si+2]
```

```
mov ax, [bx+di+2]
```

```
mov [bx+si+2], ax
```

```
mov [bx+di+2], ax
```

תרגום אופרנד לכתובת בזיכרון

בסעיפים הקודמים רأינו מספר דוגמאות לפקודת mov. בחלקן אופרנד היעד או אופרנד המקור ייצגו כתובות בזיכרון. לדוגמה:

```
mov [1], ax
```

```
mov [Var], ax
```

```
mov [bx], ax
```

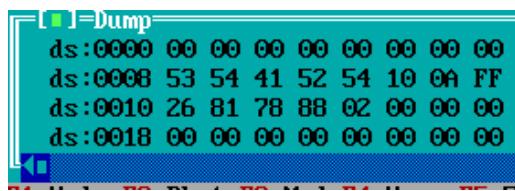
השאלה היא צו: אנו יודעים שככל כתובות בזיכרון מוצגת באמצעות 20 בית. איך בדיק האסמלר ממיר את הקבוע "1", את המשתנה var או את הרגיסטר ax (שכוורת, מכיל 16 ביטים), לכתובת בת 20 בית?

התשובה מתחולקת לשניים. הדבר הראשון שהאסמלר עושה, הוא לתרגם את הביטוי שבסוגרים לכתובת בת 16 בית – זהו האופסט בתחום הסגמנט. כולם הביטוי [1] אומר לקומפיילר שיש כאן אופסט של בית אחד מתחילת הסגמנט. אבל איזה סגמנט? ההנחה השנייה של הקומפיילר, היא – אלא אם כתבנו בפירוש אחרת – שהסגמנט הוא סגמנט הנתונים, DATASEG. לכן המעבד יעתיק את ax לתוך הכתובת שבאופסט 1 בתחום DATASEG.

השאלה הבאה שעליה היא: איך המעבד מכניס את ax, בגודל 16 בית, לתוך כתובות בזיכרון בגודל בית אחד? התשובה היא, שהוא יודע להשתמש גם בבית הבא בזיכרון. נסתכל איך נראה זיכרון המחשב לאחר פקודת mov. רק בשביל להפוך את הקראיה בDATASEG לברורה יותר, נגיד בתחילת DATASEG מערך בן שמונה בתים, שמאותחל לאפסים:

DATASEG:

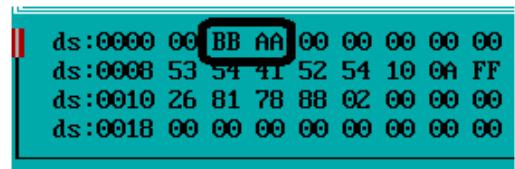
```
ZeroArray db 8 dup (0)
```



עכשו בתוך הקוד נריצ' את השורות הבאות:

```
mov ax, 0AABBh  
mov [1], ax
```

ושינויו את DATASEG:



זוכרים שאמרנו שהאסטמבלר תמיד יתיחס לזיכרון כאילו הוא ב-DATASEG, אלא אם כתבנו אחרת?

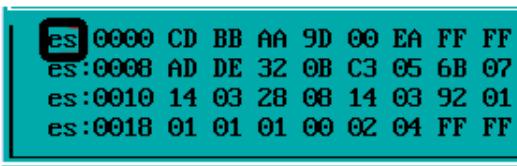
על מנת להתייחס לזכרון שלא כאילו הוא ב-DATASEG, נחליף את הפקודה:

```
mov [1], ax
```

בפקודה:

```
mov [es:1], ax ; as you recall, ES is the pointer to the Extended Segment
```

ולאחר ההרצאה, ניתן לראות את הערך 'AABBh' בכתובת המתאימות, הפעם בתחום הסגמנט Extended Segment אשר גישט ES מצביע עליו:



Little Endian, Big Endian

ראינו שכשאנו מעתיקים מעティקים רגיסטר לזכרון, הרגיستر נראה "הפוך" – לדוגמה כשהעתקנו לזכרון את ah, המעבד שומר את ah ב порядке הכתיבה המקורי ולא בכתיבה הנומוכה? הסיבה היא שכך מוגדר למעבד לבצע. אפשר גם היה להגדיר הפוך ולשמור את ah בכתיבה הנומוכה.

כאשר יש רגיستر או משתנה כלשהו, בגודל של יותר מבית אחד (לדוגמה ax, ax שגודלם שני בתים וכו'), אנחנו מעתיקים אותו לזכרון, העתקה זו יכולה להיות להיות באחת משתי שיטות:

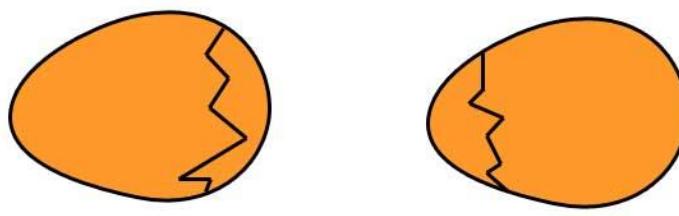
- הבית ה-High Order (כלומר bh, ah וכו') נשמר בכתיבה הנומוכה יותר בזיכרון. שיטה זו נקראת **Big Endian**.

- הבית ה-Low Order (כלומר bl, al וכו') נשמר בכתיבה הנומוכה יותר בזיכרון. שיטה זו נקראת **Little Endian**.

נציין שהשיטות הללו תקפות עבור כל רגייסטרים או משתנה שגודלם יותר מבית אחד.

יש מעבדים שעובדים בשיטה כזו ויש בשיטה אחרת. מהדוגמה האחרונה אנחנו מבינים, שמעבד ה-8086 עובד בשיטת Little Endian. הסבר מפורט ניתן למצוא בוויקיפדיה:

<http://en.wikipedia.org/wiki/Endianness>



BIG ENDIAN - The way
people always broke
their eggs in the
Lilliput land

LITTLE ENDIAN - The
way the king then
ordered the people to
break their eggs

המושגים Big Endian ו-Little Endian לקוחים מהספר "מסעות גוליבר"

מלך ליליפוט חורך חוך שני במלחוקת לגבי האוון בו יש לשבור ביצה

העתקה ממערכיים ואל מערכיים

לעתים אחד האופרנדים בפקודת mov הוא איבר במערך. זהו מקרה פרטי של העתקה מזכרון לרגיستر או של העתקה מרגיستر לזכרון. דוגמה נתון המערך:

DATASEG:

Array db 0AAh, 0BBh, 0CCh, 0DDh, 0EEh, 0FFh

כעת נרצה להעתיק את האיבר באינדקס 2 לתוך ax (שים לב, ax ולא ax – כל איבר במערך הוא בגודל בית). פקודת העתקה תיכתב:

```
mov al, [Array+2]
```

האSEMBLER יתרגם את הפקודה זו לכתובת בזיכרון. אם במקרה Array מוגדר ממש בתחום DATASEG, האינדקס השני של Array נמצא באופסט 0002 מתחילה DATASEG וmoצאת התרגום לשפת מכונה תהיה:

#base#10: mov al, [Array+2]
cs:0005 A00200 mov al, [0002]

התוצאה של פקודה זו היא ש-al מקבל את הערך 0CCh, הערך של האיבר באינדקס 2 במערך. שימו לב שהוא למעשה המיקום השלישי, שכן את המיקום הראשון מסמן אינדקס 0, ואת המיקום השני מסמן אינדקס 1.

אם היינו רוצים לבצע העתקה מרגיסטר למערך, היינו צריכים פשוט לשנות את סדר האופרנדים. לדוגמה:

```
mov [Array+1], al
```

והתוצאה הייתה שערכו של al, שבעקבות הפקודה הקודמת הוא שווה לערך של אינדקס מס' 2 במערך, יועתק לתוך האינדקס הראשון במערך.

פקודת offset

צורה נפוצה לשימוש בטיפול במערכות הוא להכנס לתוך ax את האופסט של תחילת המערך, באמצעות הפקודה offset:

```
mov bx, offset Array
```

ואז פקודת העתקה ממערך לריגיסטר מקבלת את הצורה:

```
mov al, [bx]
```

במקרה זה אופרנד היעד היה al, שגודלו בית יחיד, בעוד Array מוגדר כמערך של בתים. אילו Array היה מוגדר כמערך של מילים words, אופרנד היעד היה ax או כל רגיסטר כללי אחר שגודלו מילה.

נשאל את עצמנו למה טוב השימוש זהה? כתבנו בשתי פקודות מה שלפני כן היה כתוב בפקודה בודדת. התשובה היא, שאם נרצה לבצע פעולה על כל איברי המערך, אז נוכל לעשות זאת ביתר קלות כאשר ax שומר את כתובת התחלה של המערך וכל פעם מקדמים את ax כך שהוא מצביע על האיבר הבא במערך.

פקודת LEA

לעתים במקום הפקודה offset תיתקלו בפקודה lea, קיצור של Load Effective Address. מושית, הפקודות האלו מבצעות את אותה הפעולה. אם ניקח לדוגמה את שתי הפקודות

```
mov bx, offset Array
```

```
lea bx, [Array]
```

כפי שאפשר לראות – שתי הפקודות תורגם לאותו קוד מכונה, BB0000:

```
#base#10: mov bx, offset Array
    cs:0005 BB0000          mov     bx,0000
#base#11: lea bx, [Array]
    cs:0006 BB0000          mov     bx,0000
```

הנחה word ptr / byte ptr

ניקח את אותו מערך של בתים שהגדכנו לפני כן:

```
Array db 0AAh, 0BBh, 0CCh, 0DDh, 0EEh, 0FFh
```

נתבונן בפקודה הבאה:

```
mov ax, [Array+2]
```

הישבו: האם היא תקינה? אם לא – מדוע לא?

תשובה:

הינו מקבלים הודעה שגיאיה של הקומפיאילר.

```
Assembling file: base.asm
***Error*** base.asm(10) Operand types do not match
```

הסיבה היא שאופרנד המקור הוא בגודל בית (כלתו במערך שהגדכנו הינו באורך בית, שכן צינו db) ואילו אופרנד היעד ax הוא בגודל מילה (שכנים הריגיסטר הוא באורך 16 בית).

אי לכך, עלינו לידע את האסמלבר שאנו מנסים לבצע העתקה של שני בתים – ובכך להמנע מהשגיאיה. שימוש לביצירת הכתובת הבאה:

```
mov ax, [word ptr Array+2]
```

מה בעצם עשינו? הודיענו לאסמלבר להתייחס אל המיקום בזיכרון לא בתור byte בלבד, אלא בתור word (שני בתים). חישבו – מה יכול הריגיסטר ax לאחר הרצת שורת הקוד זו?

תשובה: לתוך ax תועתק כמהות זיכרון בגודל word. מיקום תחילת הזיכרון הוא בכתובת Array+2. התוצאה הסופית תהיה:

2
ax DDCC
bx 0000
cx 0000
dx 0000
...

וקיבלנו את מה שרצינו – העתקה של שני בתים מהמערך לתוך הרגיסטר, בפעולה ייחודית.

הערה: במקרים לכתובת word ptr או byte ptr אפשר ל凱ץ ולכתוב רק word או byte והאסמלבר יתרגם את הirection בצורה נכונה. עם זאת, כדי לשמר על泰安ות עם התצוגה ב-DT, נכתב את הנוסח המלא: .byte ptr word ptr או word ptr word ptr.

ازהרת type override

אם ננסה לבצע פקודה `mov` שהיא חוקית אך נתונה ליותר מפרשנות אחת, האסמלבר יחזיר לנו אזהרת `.type override` לדוגמה, ניקח את צורתה העברית החוקית הבאה:

```
mov    memory, constant
```

נניח שהיינו רוצים להכניס את הערך 5 לכתובת השמורה עליידי `ax`, כך:

```
mov    [bx], 5
```

האסמלבר היה מוחזר לנו אזהרה:

```
*Warning* basebg.asm(28) Argument needs type override
```

מה לא בסדר בפקודה זו? נתנו לאסמלבר כתובת להכניס אליה את הערך 5, אבל את הערך 5 אפשר לשומר במשתנה בגודל בית אחד, מילה, מילה כפולה... האסמלבר צריך לדעת כמה בתים להקצות לטובה שמירת הערך 5.

הפתרון הוא פשוט להוראות לאסמלר כמה בתים להקצות. שימוש לב להגדרות הבאות:

```
mov    [byte ptr bx], 5
```

```
mov    [word ptr bx], 5
```

ההגדרה הראשונה אומרת לאסמלר, שנחנו רוצים לגשת לאזור בזכרון שגודלו בית יחיד. הקומפיילר ייצור פקודות מכונה שמשמעותן למבוד היא: גש אל הזיכרון בכתובת `ax`, הכנס לתוכה 00000101 (היצוג הבינארי של 5 בגודל בית אחד).
ההגדרה השנייה אומרת לאסמלר, שנחנו רוצים לגשת לאזור בזכרון שגודלו מילה. האסמלר ייצור פקודות מכונה שמשמעותן למבוד היא: גש אל הזיכרון בכתובת `ax`, הכנס לתוכה 00000101, גש אל הזיכרון בכתובת `bx+1`, הכנס לתוכה 00000000.

נשאף להיות מפורטים על מנת למנוע משגיאות. אי לך, בכל פעם שנחנו מעתיקם משתנה באופן שגודל ההעתקה נתון לפשרות, נכוון את האסמלר באמצעות הסימון `word ptr` `byte ptr` או `word`.

פקודת mov – טעויות של מתחילהים

ישנן מספר טעויות נפוצות של מתחילהים העושים שימוש לא נכון בפקודת mov:

1. גדיי המקור והיעד אינם זהים. לדוגמה:

```
mov al, bx
```

```
mov ax, bl
```

פקודות אלו יגרמו לאסמלר להחזיר שגיאה. שימו לב שבפקודה השנייה גודל המקור יותר קטן מהיעד. לכארה יש ביעד מקום להכניס את תוכן המקור, אך הוראה זו אינה מקובלת על האסמלר.

2. הכנסת קבוע לתוך רגייסטר סגמנט. לדוגמה:

```
mov ds, 1234h
```

גם במקרה זה האסמלר יזכה אותו בשגיאה. הפתרון הוא שימוש בשתי פקודות (היזכרו בתחילת התוכנית):

```
mov ax, 1234h
```

```
mov ds, ax
```

3. העתקה מהזיכרון לשירות אל הזיכרון. לדוגמה:

```
mov [var1], [var2]
```

אי אפשר להעתיק מהזיכרון אל הזיכרון באופן ישיר. לכן, כדי להעתיק את var2(var1 לתוכו), נשתמש ברגייסטר עזר:

```
mov ax, [var2]
```

```
mov [var1], ax
```

4. גישה אל הזיכרון באמצעות רגייסטר לא מתאים. לדוגמה:

```
mov [ax], 5
```

פקודה זו תגרום אף היא להודעת שגיאה של האסמלר. רק bx או register משמשים לגישה אל הזיכרון (למעט גישה לזכרון של המחסנית, עליה נלמד בהמשך).

5. העתקת מידע לגודל לא מוגדר בזיכרון. לדוגמה:

```
mov [bx], 5
```

הКОМПИЛЯР לא יודע אם להעתיק את הערך 5 ל זיכרון בגודל של בית או של מילה (במיעדים מתקדמים יותר גם גודל של 32 או 64 ביט).

6. ניסיון להעתיק מידע לתוך קבוע. לדוגמה:

`mov 5, ax`

שינוי קוד התוכנית בזמן ריצה (הרחבה)

מה דעתכם ליצור תוכנית שהקוד שלה משתנה תוך כדי ריצה? אחד היתרונות המעניינים של אסמבלי על פני שפות עיליות, הוא שיש לנו גישה מלאה לזכרון – כולל זיכרון הקוד – ואנחנו יכולים לעשות בו כרצונו. בואו נראה איך עושים את זה. נתחיל בהסביר תיאורטי קצה:

נניח שאנו פונים למקום בזיכרון [1], לדוגמה על ידי הפקודה

`mov [1], al`

איך האסמבלי יודע לתרגם את [1] לכתובת מלאה בגודל 20 ביטים?

היויצו בהסביר על שיטת הסגמנט והאופסת. הפעולה הראשונה שהאסמבלי עושה היא להניח שאתם מתכוונים לפנות למקומות שנמצא בסגמנט הנתוני. כיוון ש-`ds` מכיל את כתובות סגמנט הנתוני, הפעולה שהאסמבלי יבצע היא לחת את `ds`, להכפיל ב-16 (הזכירו מדוע) ולהוסיף לו 1 (האופסת).

למעשה הפקודה האחרונה זהה לפקודה הבאה:

`mov [ds:1], al`

כל ההבדל הוא שהפעם צינו במפורש מה הסגמנט שאליו אנחנו כותבים, בעודו לאסמבלי להניח באיזה סגמנט מדובר.

כעת ננסה משהו קצת משוגע:

`mov [cs:1], al`

מה עשינו כרגע? הורינו לאסמבלי להעתיק את הערך שיש ב-`al` לתוך הזיכרון אליו מצביע `cs`, כלומר סגמנט הקוד. אפשרות זו פותחת בפנינו חזדיות מעניינות. מה אם נשנה ערכיהם בסגמנט הקוד, כך שהערכים החדשים יהיו זרים לפקודות בשפת מוכנה? ומה אם נשנה פקודות שהמעבד צפי להריץ?

כן, אם נעשה זאת נכון, נצליח לשנות את אופן פעולה התוכנית ולגרום לה לעשות דברים שהוא אינה מתוכננת לבצע.

תרגיל 6.8: תרגיל אטגר- שינוי תוכנית תוך כדי ריצה

לפניכם קטע קוד. העתיקו אותו לתוך התוכנית שלכם בשלוםות:

`xor ax, ax`

`xor bx, bx`

`add ax, 2`

`add ax, 2`

א. כפי שניתן לראות, בסיום קטע הקוד ערכו של `ax` הינו 4. המשימה שלכם היא לגרום לכך שבסיום הריצה של קטע הקוד, ערכו של `ax` יהיה 3. אך ישן מספר מוגבלות:

○ יש להעתיק את קטע הקוד בשלוםותו ובלי להוסיף שורות קוד באמצע (מותר לפני ואחרי)

○ אין להשתמש בפקודות קופיצה או בתוויות

○ יש להשתמש אך ורק בפקודות `mov`

ב. כתע גירמו לכך שבסיום הריצה של קטע הקוד, ערכו של `ax` יהיה 3 וערך של `bx` יהיה גם הוא 3. כל הכלליםensususivi' א' תקפים גם כתע.

סיכום

בפרק זה למדנו:

- להגדיר משתנים מסוגים שונים ולתת להם ערכים התחלתיים
- לעבוד עם מערכים
- לבצע העתקות מידע שונות, כוללן רגיסטרים ומקומות בזכרון, בעזרת שימוש בפקודה `mov`:
 - העתקה מריגיסטר לריגיסטר
 - העתקה של קבוע לריגיסטר
 - העתקה מהזיכרון לריגיסטר
 - העתקה ריגיסטר לזיכרון
 - העתקה של קבוע לזיכרון

בפרק הבא נעסוק בסוגים שונים של פקודות שאפשרו לנו לבצע מגוון חישובים: פקודות אריתמטיות, פקודות לוגיות ופקודות הזזה.

פרק 7 – פקודות אРИתמטיות, לוגיות ופקודות הזזה

מבוא

בפרק הקודם למדנו פקודה יחידה – פקודת `MOV`. בפרק זה נוסיף ליכולת התכנות שלנו באסמבלי מגוון רחב של פקודות, שיאפשרו לנו לבצע חישובים שונים:

- פקודות אРИתמטיות: חיבור, חיסור, כפל וחילוק
- פקודות לוגיות: `and`, `or`, `xor`, `not`
- פקודות הזזה: `shr`, `shl`

הפקודות הללו יהיו הבסיס לכל תוכנית שנרצה לכתוב.

נתחילה מפקודות אРИתמטיות. הפקודה הראשונה שנלמד – חיבור.

פקודות אРИתמטיות

משפחת מעבדי ה-`80x86` כוללת פקודות לביצוע פעולות חשבון שונות: חיבור, חיסור, כפל וחילוק (מנה או שארית). הפקודות הללו הן: `ADD`, `SUB`, `MUL`, `DIV`, `IDIV`, `INC`, `DEC`, `IMUL`, `IMUL`. הצורה הכללית של אחת מפקודות אלו מקבלת היא:

<code>add</code>	<code>dest, src</code>	<code>; dest = dest + src</code>
<code>sub</code>	<code>dest, src</code>	<code>; dest = dest - sub</code>
<code>inc</code>	<code>dest</code>	<code>; dest = dest + 1</code>
<code>dec</code>	<code>dest</code>	<code>; dest = dest - 1</code>
<code>mul</code>	<code>src</code>	<code>; ax = al * src</code>
<code>imul</code>	<code>src</code>	<code>; ax = al * src</code>
<code>div</code>	<code>src</code>	<code>; al = ax / src (ah stores the remainder)</code>
<code>idiv</code>	<code>src</code>	<code>; al = ax / src (ah stores the remainder)</code>
<code>neg</code>	<code>dest</code>	<code>; dest = 0 - dest</code>

בסעיפים הבאים נדון בפירוט בכל פקודה.

פקודה ADD

הפקודה add מחברת את הערך של אופרנד המקור (source) עם ערך אופרנד היעד (destination), ושומרת את התוצאה באופרנד היעד. האופרנדים יכולים להיות מסוג רגיסטר, משתנה או קבוע. מבין כל האפשרויות, הפקודות הבאות הן חוקיות:

תוצאה	דוגמה	הפקודה
$ax = ax + bx$	add ax, bx	add register, register
$ax = ax + var1$	add ax, [var1]	add register, memory
$ax = ax + 2$	add ax, 2	add register, constant
$var1 = var1 + ax$	add [var1], ax	add memory, register
$var1 = var1 + 2$	add [var1], 2	add memory, constant

הערות:

- את כל החישובים מומלץ לבצע בעזרת הרגיסטר ax, המעבד מבצע אותם מהר יותר מאשר באמצעות רגיסטרים אחרים. בידקו בעזרת הדיבאג'ר את הסיבה לכך!
- אפשר להשתמש ברגיסטרים של 8 או 16 ביט.

תרגיל 7.1: פקודת add



- א. צרו מערך בן 6 בתים. הכניסו לשם ערכים כלשהם. הכוינו לתוך al את סכום כל האיברים במערך. הרכזו את התוכנית ב-TD, מיצאו את המערך בזיכרון וצפו בשינוי ב-al.
- ב. בתרגיל הקודם שפתרתם, עלול להיות מצב שבו al אינו מתאים לשימרת התוצאה (מתי לדוגמה?). שנו את התוכנית, כך שסכום האיברים יכנס לתוך ax.
- ג. הגדרו שלושה משתנים:

- 1 var1 בגודל בית

- 2 var2 בגודל בית

- sum

הכניסו לתוך sum את סכום שני המשתנים האחרים (באיזה גודל צריך להיות sum?)

פקודה SUB

הפקודה sub (קיזור של deduction) מחסרת את הערך של אופרנד המקור source מערך אופרנד היעד destination. כפי שאנו מזכירם, צורות הכתיבה זהות לצורות הכתיבה של add. האופרנדים יכולים להיות מסוג register, משתנה או קבוע. מבין כל האפשרויות, הפקודות הבאות הן חוקיות:

תוצאה	דוגמה	הפקודה
$ax = ax - bx$	sub ax, bx	sub register, register
$ax = ax - var1$	sub ax, [var1]	sub register, memory
$ax = ax - 2$	sub ax, 2	sub register, constant
$var1 = var1 - ax$	sub [var1], ax	sub memory, register
$var1 = var1 - 2$	sub [var1], 2	sub memory, constant

תרגיל 7.2: פקודה sub



a. הגדרו שלושה משתנים:

var1 בגודל בית -

var2 בגודל בית -

diff -

הכנסו לתוך diff את הפרש שני המשתנים האחרים (באיזה גודל צריך להיות ?diff)

b. צרו שלושה מערכims בני 4 בתים. אתחלו את שני המערכims הראשונים עם ערכים כלשהם. הכנסו לתוך המערך השלישי את החיסור של שני המערכims הראשונים (לדוגמא המערך הראשון 9,8,7,6 המערך השני 6,7,8,9 חיסור המערכims הוא (3,1,-1,-3)

פקודות INC / DEC

פקודה inc (קייזר של increase) מעלה את ערכו של אופrnd היעד ב-1. פקודה dec (קייזר של decrease) מביאה את הפעולה הפוכה ומורידה ב-1 את ערכו של אופrnd היעד. אפשר, כמובן, לבצע הוראות אלו באמצעות פקודות add או sub, אבל כיוון שהוספה 1 או חיסור 1 הן פעולה נפרזות ביותר, הוקדשו להן פקודות נפרדות. הפקודות הבאות הן חוקיות:

תוצאה	דוגמה	הפקודה
$ax = ax + 1$	inc ax	inc register
$var1 = var1 + 1$	inc [var1]	inc memory
$ax = ax - 1$	dec ax	dec register
$var1 = var1 - 1$	dec [var1]	dec memory

פקודות MUL / IMUL

פקודה mul (קייזר של multiply) מבצעת המכפלה בין שני איברים. שימו לב, שכאשר כופלים שני איברים בני 8 ביט התוצאה יכולה להיות בגודל 16 ביט וכשכופלים שני איברים בני 16 ביט התוצאה יכולה להיות בגודל 32 ביט. במקרה של מכפלה באיבר בגודל 8 ביט, האסמלר יעתיק את התוצאה לתוך ax. במקרה של מכפלה באיבר בגודל 16 ביט, האסמלר יעתיק את 16 הביטים הנמוכים לתוך ax ואת 16 הביטים הגבוהים לתוך dx.

לדוגמה, מכפלה של שני רגיסטרים בני 8 ביטים: al=0ABh, bl=10h. תוצאה המכפל ביניהם היא 0A0B0h. האסמלר דואג שהתוצאה תועתק לתוך ax. ככלומר ah=0Ah, al=0B0h, dx=0ABh.

לדוגמה, מכפלה של שני רגיסטרים בני 16 ביטים: ax=0AB0h, bx=1010h. תוצאה המכפל ביניהם היא h0.ABAB00. האסמלר דואג שהתוצאה תחולק בין ax ו-dx. ax מקבל את שני הביטים הנמוכים ax=0AB0h וαιלו dx מקבל את שני הביטים הגבוהים dx=0ABh.

להלן דוגמאות לפקודות חוקיות:

תוצאה	דוגמה	הפקודה
$ax = al * bl$	mul bl	mul register (8 bit)
$dx:ax = ax * bx$	mul bx	mul register (16 bit)
$ax = al * ByteVar$	mul [ByteVar]	mul memory (8 bit)
$dx:ax = ax * WordVar$	mul [WordVar]	mul memory (16 bit)

בפועל הפעלה עליינו להגיד למעבד אם אנחנו עוסקים במספרים **signed** או **unsigned**, כיוון שהתווצה משנתה. ניקח לדוגמה את המספר מינוס חמיש. מספר זה מיוצג באמצעות הרץ' 11111011 בשיטת המשלים ל-2. זה ייצוג זהה לייצוג של 251. אבל אם נכפול כל אחד מהמספרים בשתיים, לדוגמה, התוצאות הייבות להיות שונות.

הפקודה **mul** מבצעת הפעלה של מספרים **unsigned** (לא מסומנים), ואילו הפקודה **imul** מבצעת הפעלה של מספרים **signed** (מסומנים). נבחן את ההבדל ביניהן.

הקוד הבא טוען לתוך **al** את 11111011 (ניתן לפירוש כ 5- או +251), לתוך **bl** את 00000010. הקוד מבצע בינהם שתי הפעולות. פעם אחת **unsigned mul** ופעם נוספת **signed imul** באמצעות אמצעות 

הפקודה **imul** (כזכור שבין ההפעולות מאפסים את **ax**). שימוש לב לערכו של **ax** לאחר כל אחת מההפעולות:

CODESEG:

```
mov    ax, 0
mov    bl, 00000010b
mov    al, 11111011b
mul    bl
mov    ax, 0
mov    al, 11111011b
imul   bl
```

Z=[↑][↓]	
ax 00FB	c=0
bx 0002	z=1
cx 0000	s=0
dx 0000	o=0
si 0000	p=1
di 0000	a=0
bp 0000	i=1
sp 0100	d=0
ds 087B	
es 0869	
ss 087E	
cs 0879	
ip 000B	

הרегистרים **ax**, **bx**, **al** לפני הפעולות ההפלה

Z=[1][1][1]	
ax 01F6	c=1
bx 0002	z=0
cx 0000	s=0
dx 0000	o=1
si 0000	p=1
di 0000	a=0
bp 0000	i=1
sp 0100	d=0
ds 087B	
es 0869	
ss 087E	
cs 0879	
ip 000D	

בכפלת על-ידי פקודה *imul*, האסמבילר מתייחס לערכו של *ai* בתו 251 ולכן תוצאה הפעולה היא *01F6h*, שווה ל-502.

Z=[1][1][1]	
ax FFF6	c=0
bx 0002	z=1
cx 0000	s=0
dx 0000	o=0
si 0000	p=1
di 0000	a=0
bp 0000	i=1
sp 0100	d=0
ds 087B	
es 0869	
ss 087E	
cs 0879	
ip 0013	

בכפלת על-ידי פקודה *imul*, האסמבילר מתייחס לערכו של *ai* בתו מינוס חמיש, ולכן תוצאה הפעולה היא *0FFF6*, שווה למינוס עשר.

תרגיל 7.3: פקודה *mul*



- הדרירו שני משתנים בגודל byte, הכניסו לתוכם ערכים בתחום 0-255, כיפלו אותם זה בזו והכניסו את התוצאה למשנה שלישי (חישבו באיזה גודל צריך להיות המשתנה השלישי?)
- הדרירו שני משתנים בגודל byte, הכניסו לתוכם ערכים בתחום +127 עד -128. כיפלו אותם זה בזו והכניסו את התוצאה למשנה שלישי (חישבו באיזה גודל צריך להיות המשתנה השלישי?)
- הדרירו שני מערכים, בכל מערך 4 ערכים מסוג בית, signed. שימו בהם ערכים התחלתיים. בצעו כפל של המערכים והכניסו את התוצאה לתוך sum. לדוגמה שהתוצאה נכנסת לתוך word. הדרכה: אם שמות המערכים הם a ו-d, אז

$$\text{sum} = a[0]*b[0]+a[1]*b[1]+\dots$$

פקודות DIV, IDIV

פקודה div (קיצור של divide) מבצעת חילוק בין שני מספרים. אם האופרנד הוא בגודל 8 ביט, div מחלקת את ax באופרנד, שומרת את מנת החלוקה ב-al ואת השארית ב-ah. אם האופרנד הוא בגודל 16 ביט, div מחלקת באופרנד את 32 הביטים המוחזקים על-ידי ax:dx, שומרת את מנת החלוקה ב-ax ואת השארית ב-dx.

דוגמה לחילוק של רגיסטרים בני 8 ביט: al=7h, bl=2h. תוצאה החילוק ביניהם היא 3 עם שארית 1. לאחר ביצוע הפעולה, ah=1, al=3.

דוגמה לחילוק רגיסטרים בני 16 ביט: ax=7h, bx=2h. תוצאה החילוק ביניהם היא 3 עם שארית 1. לאחר ביצוע הפעולה, dx=1, ax=3.

שים לב – בגלל הדרך שבה div מוגדרת, אי אפשר פשוט לחלק ערך של 8 ביט בערך אחר של 8 ביט (או לחלק ערך של 16 ביט בערך אחר של 16 ביט). 

אם המכנה (המחלק) הוא בגודל 8 ביט, המונה (המחלוק) צריך להיות בגודל 16 ביט. לכן כדי לחלק את al, נdag לפני החלוקה ש-ah יהיה שווה אפס: פקודה mov ah, 0 תהיה מספיק טובה. רק צריך לזכור לשים אותה לפני החילוק. באופן דומה, אם המכנה הוא בגודל 16 ביט, המונה צריך להיות בגודל 32 ביט. לכן, כדי לחלק את ax במכנה בגודל 16 ביט, פשוט נdag לפני החלוקת ש-dx יהיה שווה לאפס: 0 mov dx, 0. אם נשכח לעשות את האיפוסים הללו, סביר שנגרום לטעות להחזיר לנו תוצאות לא נכונות!

מתי עוד תתעורר בעיה? באחד מן המקרים הבאים:

חלוקת באפס -

התוצאה אינה כניסה ברגיסטר היעד (האם תוכל לחשב על דוגמה מסוימת כזו?) -

דוגמאות לפקודות חוקיות:

תוצאה	דוגמה	הפקודה
al = ax div bl ah = ax mod bl	div bl	div register (8 bit)
ax = dx:ax div bx dx = dx:ax mod bx	div bx	div register (16 bit)
al = ax div ByteVar ah = ax mod ByteVar	div [ByteVar]	div memory (8 bit)
ax = dx:ax div WordVar dx = dx:ax mod WordVar	div [WordVar]	div memory (16 bit)

הristol את התוכנית הבאה ועקבו אחריו ערכם של הרגיסטרים לפני ואחרי ביצוע פעולה ה-div:

IDEAL

MODEL small

STACK 100h

DATASEG

CODESEG

start:

```
mov ax, @data
```

```
mov ds, ax
```

```
mov al, 7
```

```
mov bl, 2
```

```
mov ah, 0
```

```
div bl
```

```
mov ax, 7
```

```
mov dx, 0
```

```
mov bx, 2
```

```
div bx
```

quit:

```
mov ax, 4c00h
```

```
int 21h
```

END start

פקודת div זהה לפקודת idiv, פרט לכך שהיא מבצעת חילוק של מספרים מסוימים (signed), בניגוד ל-div שמחלקת מספרים לא מסוימים (unsigned).

תרגיל 7.4: פקודה div



- א. הגדרו שני משתנים בגודל byte, התייחסו אליהם כ-`unsigned`, הכניסו את תוצאת החלוקה שלהם למשתנה שלישי ואת השארית למשתנה רביעי.
- ב. הגדרו שני משתנים בגודל byte, התייחסו אליהם כ-`signed`, הכניסו את תוצאת החלוקה שלהם למשתנה שלישי ואת השארית למשתנה רביעי.
- ג. הגדרו שני משתנים בגודל word, התייחסו אליהם כ-`unsigned`, הכניסו את תוצאת החלוקה שלהם למשתנה שלישי ואת השארית למשתנה רביעי.

פקודה NEG

פקודה ה-`neg` (קיצור של negative) מקבלת אופרנד יחיד, בגודל בית או מילה, והופכת את הערך שלו למשלים לשתיים של הערך המקורי. הפקודה:

`neg dest`

תבצע את החישוב הבא:

`dest = 0 - dest`

דוגמאות לשימוש חוקי בפקודה:

תוצאה	דוגמה	הפקודה
<code>al = 0 - al</code>	<code>neg al</code>	<code>neg register (8 bit)</code>
<code>ax = 0 - ax</code>	<code>neg ax</code>	<code>neg register (16 bit)</code>
<code>ByteVar = 0 - ByteVar</code>	<code>neg [ByteVar]</code>	<code>neg memory (8 bit)</code>
<code>WordVar = 0 - WordVar</code>	<code>neg [WordVar]</code>	<code>neg memory (16 bit)</code>

פקודות לוגיות

נלמד עכשוו על ארבע פקודות לא מסובכות אבל עם המון חשיבות. פקודות לוגיות הן שימושיות מאוד כשאנו רוצים לשנות את ערכו של בית, או מספר ביטים, ובאותו זמן להשאיר ערכים אחרים בלי שינוי. פעולה זו נקראת **MASKING**, ונרחיב עליה בהמשך הפרק.

למה בכלל נרצה להתעסק עם ביטים בודדים? בגלל הצפנות, לדוגמה. כמשמעותם מידע, המידע שמיועד להצפנה נשמר בצורה "זחוסה" – **Packed data**. נסתכל לדוגמה על מערך של שמונה בתים, כל אחד מהם שומר ערך שהוא 0 או אחד:

```
00000000 00000001 00000000 00000001 00000000 00000001 00000000 00000000
```

את אותו המידע אפשר לשמר בבית אחד בצורה דחוסה:

```
01110010
```

בצורה דחוסה אנחנו יכולים להאיין את מהירות ההצפנה והפענוח, יחסית למהירות שניתן להשיג כשבודדים בצורה לא דחוסה בה מוקצת בית נפרד לשימירת כל בית.

על הבית הזה מבצעים פעולות לוגיות שונות שהופכות אותו למוצפן. לדוגמה, אנחנו יכולים להפוך את ערכו של כל בית שני, ולקבל:

```
00100101
```

מי שלא יודע מה הפעולה שעשינו, לא יוכל לשחזר את הערכיהם המקוריים ולפענה את המסר המוצפן. בשילוב בכך פעולה כזו, צריך לדעת "לשחק" עם ביטים בודדים. כתע נראה איך עושים את זה.

קיימות ארבע פקודות לוגיות – **not**, **xor**, **or**, **and**. מיד נסביר מה עושה כל פקודה.

צורת הרישום של הפקודות האלו היא:

and dest, src ; dest = dest and src

or dest, src ; dest = dest or src

xor dest, src ; dest = dest xor src

not dest ; dest = not dest

אפשר לכתוב את הפקודות הלוגיות עם רגיסטר, זיכרון או קבוע. הצורות הבאות חוקיות:

and register, register

and memory, register

and register, memory
 and register, constant
 and memory, constant

הפקודות **or** ו**xor** נכתבות בדיק באותה צורה כמו **and**.

הפקודה **not** יכולה להיכתב באחת מבין הצורות:

not register
 not memory

פקודה AND

מבחן לוגית, **and** מקבלת כקלט שני ביטים. אם שניהם שווים 1, התוצאה תהיה 1. אחרת, התוצאה תהיה 0. נעשה היכרות עם מונח **שנקרא** "טבלתאמת". בשורה העליונה הערכים האפשריים לביט הראשון, בטור השמאלי הערכים האפשריים לביט השני. בתוך הטבלה – תוצאה ה-**and** על הביטים האלה.

AND	1	0
1	1	0
0	0	0

טבלתאמת של פעולה **and**

פקודת האסמלבי **and** מקבלת שני אופרנדים, שיכולים להיות להם 8 או 16 ביטים, וביצעת **and** לוגי בין הביטים שלהם – הביט שבמוקום 0 באופרנד הראשון עם הביט שבמוקום 0 באופרנד השני, הביט שבמוקום 1 באופרנד הראשון עם הביט שבמוקום 1 באופרנד השני, הביט שבמוקום 2 באופרנד הראשון עם הביט שבמוקום 2 באופרנד השני וכך הלאה.

לדוגמה:



0000 0111 and

1001 0110

0000 0110

נדגים יישום פשוט של פקודה **and** – בדיקה אם מספר כלשהו הוא זוגי.

כדי לבדוק אם מספר כלשהו הוא זוגי, אנחנו יכולים לבדוק רק את הבית האחרון ביצוג הבינארי שלו. אם בית זה הוא 0 – המספר הוא זוגי. אם הבית הוא 1 – המספר הוא אי-זוגי.

לצורך הבדיקה, נגידר אמצעי חדש שנקרא **מסכה (MASK)**. המסכה היא אוסף של ביטים, שמאפשר לנו לבדוק ולפעול על ביטים מסוימים. מסכה יכולה להיות כל אוסף של ביטים, אבל אנחנו נגידר אוסף מיוחד של ביטים: 100000001.

נעשו **and** בין המספר שאנו רוצhim לבודוק לבין המסכה. כיוון שהגדכנו את שבעת הביטים העליונים כאפס, תוצאה ה-**and** שלהם תהיה 0 בכל מקרה. אי לכך, אנו מתעלמים למעשה מהערך של שבעת הביטים האלה. כתוצאה לכך, תוצאה ה-**and** של הבית השמיני תלויה רק במספר שאות הזוגיות שלו אנחנו בודקים – אם הבית השמיני שלו הוא 1, ה-**and** שלו יחד עם הבית השמיני במסכה, שהוא גם 1, יהיה שווה ל-1. אחרת, התוצאה תהיה 0.

תרגיל 7.5: פקודה **and**



איך אפשר לבדוק בעזרת פקודה **and אם מספר מתחלק ב-4? בידקו זאת על-ידי משתנים שונים.**

פקודת OR

טבלת האמת של פקודת **or** נראה כך:

OR	1	0
1	1	1
0	1	0

כלומר, מספיק שאחד הביטים בקלט שווה ל-1, והפלט הוא 1.

פקודת **or** שימושית כשורצים "להדליך" בית כלשהו בלי לגעת בשאר הביטים. נניח שיש רכיב שמנל תקשורת עם שמונה הtekנים חיצוניים – הוא יכול לשלווח ולקבל מהם מידע, או להפסיק את התקשרות איתם (במשך, כשולמד על פסיקות, ולמד על רכיב זה). מצב התקשרות עם כל רכיב חיצוני מוצג עליידי בית – 1 קובע שיש תקשורת עם הרכיב, 0 קובע שהרכיב מושתק. מצב התקשרות עם כל שמונה הרכיבים נשמר בבית מסוים.

כעת, אנחנו רוצים לאפשר תקשורת עם הtekן מס' 4 (הtekנים ממוספרים מ-0 עד 7). נקרא את זיכרונו הרכיב. נניח שקיבלנו:

1100 0100

בית מס' 4, אשר מייצג את הtekן מס' 4, קבוע ומסומן בצהוב. כתע, ברצוננו להדליך את הבית הזה.

על מנת לעשות זאת, נבצע **or** עם המסה 0000 0001 (הבית במיקום 4 דולק) ונקבל:

1101 0100

כך ווילאנו שבית 4 יהיה דולק, מבלי להיות תלויים בערך המקורי שלו, או בערכם של הביטים האחרים.

את התוצאה נעתיק חוזה לתוכ רכיב התקשרות והתוואה היא שאיפשרנו תקשורת עם הtekן חיצוני מס' 4.

תרגיל 7.6: פקודה **or**



א. בדוגמה שנתנו, באיזו מסכה צריך להשתמש כדי לאפשר תקשורת עם רכיב מס' 2 ? עם רכיבים 2 ו-4 בלבד

(בפקודה **or** אחת?)

ב. איך אפשר להוראות לרכיב להפסיק את התקשרות עם הtekן מס' 4 ?

פקודת XOR

פקודת xor, (קיצור של or exclusive), היא בעלת טבלת האמת הבאה:

XOR	1	0
1	0	1
0	1	0

מבחן מתמטית, xor שקול לפועלות חיבור ומודולו 2 (חלוקת בשתיים, ולקיים השארית בלבד).

פקודת xor היא פקודה שימושית מאוד, לדוגמה כ商量בטים הצפנה. הסיבה היא התכונה הבאה שלה:

התוצאה של xor של רצף ביטים עם רצף ביטים זהה, היא תמיד 0!

הຕכונה המינוחת הזו תשרת אותנו ביצירת הצפנה. נניח שאנו רוצים להצפין את המידע הבא:



1001 0011

נדיר מפתח הצפנה, שידוע רק לנו ולא מי שאמור לקלוט את השדר שלנו. המפתח יהיה (לדוגמה):

0101 0100

נצפין את המידע באמצעות פעולה xor. התוצאה שתתקבל היא:

1001 0011 xor

0101 0100

1100 0111

הצד השני יקבל את המסר הזה, ועל מנת לפענה אותו יבצע פעם נוספת את פעולה xor עם מפתח ההצפנה שידוע לו:

1100 0111 xor

0101 0100

1001 0011

המיוחד בשיטת הצפנה זו הוא, שאיפלו אם מישו מכיר את השיטה שבה השתמשנו בשביל להציג את המידע שלנו, הוא לא יצליח לשבור את הצופן ולפענח את המידע. רק מי שידוע מה היה מפתח ההצפנה יכול לשחזר את המידע.

אגב, ה-*opcode* של xor קצר יותר מאשר *mov*, ולכון מעטה ואילך כשרצחה לאפס רגיסטר נעשה זאת על-ידי xor של ערך הרגיסטר עם עצמו. מכאן שבמוקום לבצע:

```
mov ax, 0
```

נכתב:

```
xor ax, ax
```

שתי הפעולות גורמות לכך שב-*ax* יהיה הערך 0, אך הפעולה השנייה לוקחת פחות מקום מהראשונה, ולכון נועד להשתמש בה.

תרגיל 7.7: פקודה xor



- א. הסבירו מדוע התוצאה של xor של אוסף ביטים עם אוסף ביטים זהה – היא תמיד אפס.
- ב. הגדרו מערך בשם msg שהוא אוסף של תווים ASCII LIKE ASSEMBLY\$'.'!. הגדרו מפתח הצפנה בן 8 ביטים, לבחירתכם. הצפינו את המידע באמצעות מפתח ההצפנה (תו ה-\$ יסייע לכם לדעת שהગעתם לסוף המערך ייש להציגן, וכן להדפסה). לכו ל-DATASEG במקומם שהגדיתם את המערך והסתכלו איך המחשב מתרגם את המערך לתווים ASCII לפניהם ואחריו פועלות ההצפנה שלכם. פענחו את המידע שהצפנתם באמצעות מפתח ההצפנה. חיזרו על הבדיקה ב-DATASEG ועודאו שקיבלתם חוזה את המסר המקורי שלכם.

למעוניינים להדפיס את המסר ואת המסר המוצפן, ניתן להשתמש בקטע הקוד הבא:

print:

```
mov dx, offset msg
mov ah, 9h
int 21h
mov ah, 2 ; new line
mov dl, 10
int 21h
mov dl, 13
int 21h
```

פקודת NOT

פקודת not פשוטה הופכת את כל הביטים באופרנד. טבלת האמת של not:

NOT	
1	0
0	1

את צורות הכתיבה של not סקרונו בתחילת הסעיף. פקודת not מאפשרת גמישות נוספת בעובדה עם ביטים בודדים.

פקודות הזהה

פקודות הזהה מקובלות אופרנד ו"מוזיזות" את הביטים שלו. לדוגמה, הביט במיקום 0 מוזז למיקום 1, הביט במיקום 1 מוזז למיקום 2 וכן הלאה. מיד נפרט איך לבדוק עובדת ההזהה.

יש אוסף של פקודות הזהה, שבערךון די דומות אחת לשניה. אנחנו נתמקד בשתי פקודות הזהה השימושית ביותר: shr (shift left) shl (shift right).

פקודת SHL

פקודת הזהה שמאליה, shl, מקבלת אופרנד יעד וכמותם ביטים להזהה. האופרנד יכול להיות משתנה בזיכרון או רגיסטר, וכמותם הביטים להזהה יכולה להיות מספר קבוע או רגיסטר cl. הוצאות הבאות תקינות:

shl register, const

shl register, cl

shl memory, const

shl memory, cl

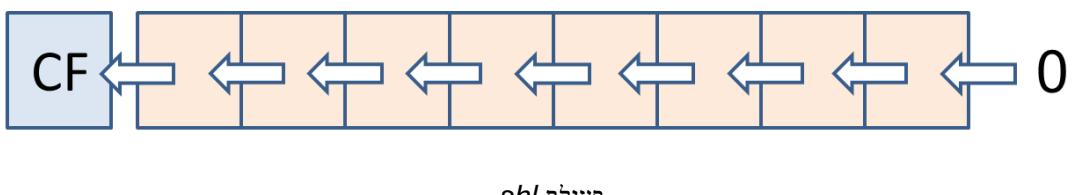
פעולה של shl משפיעה באופן הבא:

- על כל הזהה, הביטים שבאופרנד היעד זרים מקום אחד שמאליה.
- על כל הזהה נכנס 0 אל הביט הימני ביותר (אם ההזהה היא של בית אחד, נכנס 0 אחד. אם ההזהה היא של Ch ביטים, נכנסים Ch אפסים).
- הביט האחרון שיצא מצד שמאל נכנס אל דגל הנשא CF.
- דגל הגלישה OF יקבל ערך 1 במקרה שהבית הcy שמאלי השתנה. כלל זה נכון רק עבור הזהה של בית יחיד.
- דגל האפס ZF יקבל ערך 1 אם לאחר הזהות ערכו של האופרנד הוא אפס.

- דגל הסימן יהיה שווה לערך של הביט הcy שמאלית.

- דגל הזוגיות יקבל ערך 1 אם יש כמות זוגית של אחדות ב-8 הביטים הנמוכים של האופרנד.

כך נראה פעלת `shl` על אופרנד של 8 ביט. אם היינו רוצים לצייר את הפעולה על אופרנד של 16 ביט השינוי היחיד היה 16 ריבועים במקום 8:



פקודת SHR

פקודת ההזהה ימינה, `shr`, כMOVZ דומה בכתיבתה שלה ובפעולתה שלה ל-`shl`. הפקודה מקבלת אופרנד יעד וכמות ביטים להזהה. האופרנד יכול להיות משתנה בזיכרון או רегистר, וכמות הביטים להזהה יכולה להיות מספר קבוע או הרגיסטר `cl`. הזרות הבאות תקינות:

`shr register, const`

`shr register, cl`

`shr memory, const`

`shr memory, cl`

פעולת ה-`shr` מושפעה באופן הבא:

- על כל הזהה, הביטים שבאופרנד היעד זזים מקום אחד ימינה.

- על כל הזהה נכנס 0 אל הביט השמאלי ביותר (אם ההזהה היא של בית אחד, נכנס 0 אחד. אם ההזהה היא של כבitemים, נכנסים ח אפסים).

- הביט האחרון שיוצא מצד ימין נכנס אל דגל הנשא CF.

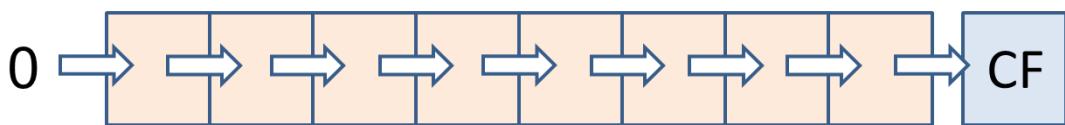
- דגל הגלישה OF יקבל את ערך הביט הcy השמאלי של האופרנד לפני ההזהה. כלל זה נכון רק עבור הזהה של בית יחיד.

- דגל האפס ZF יקבל ערך 1 אם לאחר ההזהות ערכו של האופרנד הוא אפס.

- דגל הסימן יהיה שווה לערך של הביט הcy השמאלי, שהוא תמיד אפס.

- דגל הזוגיות יקבל ערך 1 אם יש כמות זוגית של אחדות ב-8 הביטים הנמוכים של האופרנד.

כך נראה פעלת `shr` על אופרנד של 8 ביט. אם היינו רוצים לצייר את הפעולה על אופרנד של 16 ביט השינוי היחיד היה 16 ריבועים במקום 8:

פעולת *shrlt***שימושים של פקודות הזזה**

פקודות הזזה שיטות מיוחדות למילוי ביצוע פעולות:

- ביצוע כפל וחילוק בקלות בחזקות של שתים
- ביצוע פעולות עזר הקשורות להצפנה
- דחיסה של מידע ופרישה (הפעולה הפוכה לדחיסה)
- גרפיקה

כפל וחילוק: בסיסי עשרוני, כשורצים לכפול בעשר, כל מה שצורך לעשות זה להוסיף אפס בטור הספרה הcy ימנית (כלומר, להזין את הספרות ספרה אחת שמאללה). בסיסיים שתים, אם נוסיף אפס בטור הספרה הcy ימנית, נקבל כפל בשתיים. ככלומר כל הזזה שמאללה בבית אחד כופלת את הערך בשתיים. כיוון שערכאים שמורים בשתיים עם גודל מוגבל, באיזשהו שלב נהרג מגודל של המשטנה ואז התכוונה הוא תאבד, אבל אם נעבוד נכון ונdagג לגדים נכנים של המשטנים, נוכל להשתמש בהזזה כדי לבצע כפל בשתיים, וכן הלאה – כפל בארבע באמצעות הזזה שני ביתים, במשמונה וכו'. כמו שהזזה שמאללה היא כפל בשתיים או בחזקות של שתים, הזזה ימינה היא חילוק בשתיים או בחזקות של שתים. גם כאן צריך לשים לב שאם הבית הcy ימני שלנו הוא 1, ככלומר המספר אי זוגי, אז הזזה ימינה לא בדוק תחלק אותו בשתיים – התוצאה תהיה חלוקה בשתיים, מעוגלת כלפי מטה.

הצפנה: ישנו סוגים רבים של הצפנה. אחד מסוגי ההצפנה הצבאים הנפוצים הוא LFSR. בהצפנה זו ישנו מערך של ביטים שבכל פעם דוחפים בית אחד שלו ומוציאים בית מצידו השני. בין הביטים במערך מבוצעות פעולות XOR שונות, ועם התוצאה עושים XOR לביט רוצים להצפן. אתם מוזמנים לקרוא על LFSR באינטרנט, ויקיפדיה היא מקום טוב להתחילה ממנו:

http://en.wikipedia.org/wiki/Linear_feedback_shift_register

בכל אופן, כדי למש את הפעולה של הזות הביטים בצורה יעילה משתמשים בפקודות הזזה.

דחיסה ופרישה של מידע: הסבר מפורט הינו מחוץ להיקף של ספר זה. קיימים אלגוריתמי דחיסה ופרישה רבים, מומלץ להתחילה מאלגוריתם למפל זוּן:

<http://en.wikipedia.org/wiki/Lempel%20%93Ziv%20%93Welch>

גרפיקה: נדון בנושא זה בהמשך, כחלק מהפרק על כלים לפרויקטי סיום. באמצעות פקודות הזזה ניתן להגיע בקלות לפיקסל נתון על המסך.

תרגיל 7.8: פקודות הזזה



- א. הכניסו ל-`af` את הערך 3. בעזרת פקודות הזזה, כיפלו את `af` ב-4.
- ב. הכניסו ל-`af` את הערך 120 (דצימלי). בעזרת פקודות הזזה, חלקו את `af` ב-8.
- ג. הכניסו ל-`af` את הערך 10 (דצימלי). בעזרת פקודות הזזה והיבור, כיפלו את `af` ב-20. הדרכה: התיחסו ל-20 בתור סכום של 16 ו-4. השתמשו ברגיסטרים נוספים כדי לשמר חישובי ביניים.

סיכום

בפרק זה למדנו:

- לבצע פעולות חשבון, `signed` ו-`unsigned`:

 - חיבור
 - חיסור
 - כפל
 - חילוק

- לבצע פעולות לוגיות:

 - And
 - Or
 - Xor
 - Not

- באמצעות הפעולות הלוגיות למדנו לבצע שינויים ברמת בית בודך, כגון הדלקת וכיבוי בית, הצפנה של מידע באמצעות מפתח.
- לבצע פעולות הזזה, שמשמשות לכפל וחילוק בחזקות של שתיים:

 - Shr
 - Shl

בפרק הבא נלמד איך להוסיף לתוכניות שלנו תנאים לוגיים, שיאפשרו לנו לכתוב אלגוריתמים.

פרק 8 – פקודות בקרה

מבוא

בפרק זה נלמד ליצור תוכנית עם תנאים לוגיים ("אם מתקיים תנאי זה, בצע פעולה זו...").

כדי לכתוב פקודות שיש להן תנאים, אנחנו צריכים ללמידה כמה נושאים:

- איך קופצים למקום כלשהו בתוכנית (פקודת קפיצה jmp על סוגיה השונות).
- איך בודקים אם תנאי כלשהו מתקיים (פקודת השווא – cmp).
- איך לחזור על ביצוע קוד מוגדר כל עוד מתקיים תנאי שהגדנו (פקודת לולאה – loop).

באופן רגיל, התוכנית מתקדמת שורה אחריו שורה. כפי שלמדנו קודם לכן, לאחר כל שורה, מצביע התוכנית, שנמצא ברגיסטר IP, מוקדם אל שורת הפקה בתוך CODESEG. חפקין של הוראות הבקרה הוא לאפשר קפיצה קדימה או אחורה אל שורות קוד בתוכנית. קפיצה קדימה, ככלומר דילוג על שורות בתוכנית, הכרחית כדי לאפשר לנו להריץ קוד כלשהו רק אם מתקיימים תנאים מסוימים – "אם ערכו של המשתנה שווה ל-1, המשך ביצוע התוכנית. אחרת – דלג על המשך שורות קוד והמשך ממשם". קפיצה אחורה הכרחית כדי לאפשר לנו לחזור על הרצאה של קוד מספר פעמים, אם תנאי כלשהו מתקיים – "אם ערכו של המשתנה אינו אפס, קופץ עשר שורות קוד אחורה והמשך ממשם".

הוראות הבקרה מתחולקות לשני סוגים: הוראות בקרה עם תנאי קיום ("אם מתקיים מצב מסוים, אז קופוץ ל...") והוראות בקרה ללא תנאי קיום ("קופוץ ל..."). לכל סוג יש שימוש ייעיל במצבים שונים. למען הפשטות, נתחיל בהוראות בקרה ללא תנאי קיום.

פקודת JMP

פקודת jmp שולחת את המעבד, ללא תנאי, לנקודת הרצאה אחרת בתוכנית. פקודת jmp מקבלת כתובות בזיכרון, בתוך CODESEG. לאחר ביצוע פקודת jmp, תועתק אותה הכתובת המובאת לה בתור אופרנד לתוך רגיסטר IP והרצאה תמשיך מכוחות זו.

לדוגמה:

DATASEG

```
address      dw      000Ah
```

CODESEG

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

```
mov  ax, 1
jmp  [address]
```

כך נראהים הרגיסטרים לפני ביצוע פקודה jmp:

ax	0001
bx	0000
cx	0000
dx	FFCD
si	0000
di	0000
bp	0000
sp	0100
ds	087C
es	0869
ss	087D
cs	0879
ip	0028

כך נראהים הרגיסטרים לאחר ביצוע פקודה jmp, שימושו לב לערכו החדש של ip:

ax	0001
bx	0000
cx	0000
dx	FFCD
si	0000
di	0000
bp	0000
sp	0100
ds	087C
es	0869
ss	087D
cs	0879
ip	000A

וחתנית תמשיך את הריצה מהבית העשוי ב-SEG.CODE.

קפיצות FAR ו-NEAR

בדוגמה שהשתמשנו בה כדי להסביר את פקודה jmp, מסרנו לאסמלר רק את האופטט שהוא צריך ל קופץ אליו – 000Ah – בעזרת שימושו במשתנה address. בזמן שהחכנית הגיעו לשורת ה-jmp שלנו, היא רצתה מתוך סגמנט הקוד ופקודת jmp שלחה אותה לאחר באותו סגמנט. מצב זה, של קפיצה בתוך אותו סגמנט (במקרה זהה סגמנט הקוד אל חלק אחר בסגמנט הקוד), נקרא קפיצה near. כדי לבצע קפיצות מסווג near, מספיק שניתן לפקודת jmp את האופטט – הסגמנט הרוי לא משתנה.

קפיצות מסווג far הן במקרים שחלק מהקוד שלנו נמצא בסגמנט אחר. עקרונית, האסמלר מאפשר לנו לחלק את הקוד שלנו לסגמנטים שונים, אם הוא גדול מדי להיכנס בסגמנט אחד. במקרה זה, הכרחי להודיע לאסמלר לאיוזה סגמנט קוד אנחנו רוצים לkopioz. צורת הכתיבה היא:

```
jmp  cs:offset      ; for example cs:000A
```

כאשר לפני הקפיצה יש לוודא ש-CS מצביע על כתובות הסגמנט.

אין סיבה להיות מוטרדים מנוסה זה – לא סביר שנכתב קוד שסגןנט קוד אחד לא יספק עבورو. מצד שני, הבנת התיאוריה של נושא קפיצות `near` וה-`far` השובה להמשך.

תרגיל 8.1: פקודה jmp



נתונה הוכנית הבא (העתקו אותה לתוך ה-`CODESEG` שבתוכנית `base.asm`):

```
xor    ax, ax
add    ax, 5
add    ax, 4
```

בעזרת הוספה פקודה `jmp` לתוכנית, גירמו לכך שבסוף ריצת הוכנית `ax=4`.

תוויות LABELS

ברוב הפעמים לא נבעיר ל-`jmp` ממש כתובות בזיכרון. הסיבה היא, שכל שינוי בתוכנית שלנו ישנה את מקום הפקודות ב-`CODESEG` ואז המעבד יקפוּן מקום לא נכון בתוכנית. לא נרצה לעדכן את הכתובות בכל פעם שנבצע שינוי בתוכנית שלנו. על מנת להקל علينا, שפת אסמבלי מאפשרת לנו לתת **תוויות (label)** לשורה בקוד, ובמקום לחת פקודת `jmp` את כתובות השורה, אפשר לחת לה את ה-`label` של השורה. רצוי מאוד ש-`label` של שורה יהיה בעל שם בעל משמעות, שיאפשר לנו ולמי שקורא את הקוד אחרינו להבין מה עושה קטע הקוד אחרי ה-`label`. לדוגמה:

:LoopIncAx

```
inc    ax
jmp    LoopIncAx
```

קטע קוד זה יירוץ אינסופי פעמים, ובכל פעם יקדם את הערך הנמצא ברגיסטר `ax` באחד.

שימוש לב לאינדנטציה – הרוחים בתחילת כל שורה. כל ה-`label`ים תמיד נמצאים בעמודה השמאלית ביותר של הקוד, ופקודות נמצאות טاب אחד ימינה.



אנחנו יכולים לקבע כל רצף של תוים בתור `label`, כל עוד הוא מתייחס בספרה ואינו כולל רווח. כמו כן האסמבילר אינו מפheid בין אותיות גדולות לקטנות – מבחןנו `StartLoop` זהה ל-`startloop`. ועדיין, האפשרות הראשונה (תחילת כל מילה באות גדולה וכתיבת שאר המילה באות קטנה) יותר קריאה ועל-כן מומלצת.

דוגמאות לשמות "מוצלחים" של `labels` יכולים להיות: `Check`, `Back`, `PrintResult`, `Wait4Key`, `Next_Level`, `Not_Positive`. בעזרת שמות אלו, גם בלי לדעת מה כתוב בהמשך אפשר לקבל מושג מה עשויה הקוד.

דוגמאות לשמות לא מוצלחים הם: Label1, MyLabel, Shooki וכו'. אלו שמות כלליים וחסרי משמעות. האסמבלי כנובן לא יעיר עליהם, היות שהם תקניים מבחינה טכנית, אבל הבעיה תהיה שלנו (ושל מי שינסה להבין את הקוד שכתבנו). אבל יש סוג נוסף של שמות ל-label, שהאסמבלי יתריע עליהם כי ישנה שגיאה. אסור לקרוא ל-label בשם של רегистר (למשל: ax) או בשם של פקודה אסמבלי (למשל: MOV), ואין להשתמש בשמות מהשומות המופיעים להלן – אלו שמות שמורים.

נסים בטעות נפוצה של מתחילה: אין להשתה שורות קוד שונות את אותו שם label. האסמבלי פשוט לא

יבין لأنם רוצחים לкопאן.



The list of words in this table are words that have special meaning to the assembler. You cannot use them for any purpose other than that defined by the assembler.

.186	.286	.286C	.286P	.287	.386
.386C	.386P	.387	.8086	.8087	ALIGN
.ALPHA	AND	ASSUME	AT	BYTE	CATSTR
.CODE	COMMENT	COMMON	.CONST	.CREF	.DATA
.DATA?	DB	DD	DOSSEG	DQ	DT
DUP	DW	DWORD	ELSE	ELSEIF	ELSEIF1
ELSEIF2	ELSEIFB	ELSEIFDEF	ELSEIFDIF	ELSEIFDIFI	ELSEIFE
ELSEIFIDN	ELSEIFIDNI	ELSEIFNB	ELSEIFNDEF	END	ENDIF
ENDM	ENDP	ENDS	EQ	EQU	.ERR
.ERR1	.ERR2	.ERRB	.ERRDEF	.ERRDIF	.ERRDIFI
.ERRE	.ERRIDN	.ERRIDNI	.ERRNB	.ERRNDEF	.ERRNZ
EVEN	EXITM	EXTRN	FAR	.FARDATA	.FARDATA?
FWORD	GE	GROUP	GT	HIGH	IF
IF1	IF2	IFB	IFDEF	IFDIF	IFDIFI
IFE	IFIDN	IFIDNI	IFNB	IFNDEF	INCLUDE
INCLUDELIB	INSTR	IRP	IRPC	LABEL	.LALL
LE	LENGTH	.LFCOND	.LIST	LOCAL	LOW
LT	MACRO	MASK	MEMORY	MOD	.MODEL
NAME	NE	NEAR	NOT	OFFSET	OR
ORG	%OUT	PAGE	PAGE	PARA	PROC
PTR	PUBLIC	PUBLIC	PURGE	QWORD	.RADIX
RECORD	REPT	.SALL	SEG	SEGMENT	.SEQ
.SFCOND	SHL	SHORT	SHR	SIZE	SIZESTR
.STACK	STACK	STRUC	SUBSTR	SUBTTL	TBYTE
.TFCOND	THIS	TITLE	.TYPE	WIDTH	WORD
.XALL	.XREF	.XLIST	XOR		

In addition to the above words, TASM also reserves these:

ARG	%BIN	CODESEG	%COND	CONST	%CREF
%CREFALL	%CREFREF	%CREFUREF	%CTL	DATAPTR	DATASEG
%DEPTH	DISPLAY	DP	EMUL	ERRIF	ERRIF1
ERRIF2	ERRIFB	ERRIFDEF	ERRIFDIF	ERRIFDIFI	ERRIFE
ERRIFIDN	ERRIFIDNI	ERRIFNB	ERRIFNDEF	EVENDATA	FARDATA
FARDATA?	GLOBAL	IDEAL	%INCL	JUMPS	LARGE
%LINUM	%LIST	LOCALS	%MACS	MASM	MASM51
MODEL	MULTERRS	%NEWPAGE	%NOCONDS	%NOCREF	%NOCTL
NOEMUL	%NOINCL	NOJUMPS	%NOLIST	NOLOCALS	%NOMACS
NOMASMS51	NOMULTERRS	%NOSYMS	%NOTRUNC	NOWARN	P16
P286	P286N	P286P	P287	P386	P386N
P386P	P387	P8086	P8087	%PAGESIZE	%PCNT
PNO87	%POPLCTL	%PUSHLCTL	PWORD	QUIRKS	RADIX
SMALL	%SUBTTL	%SYMS	SYMTYPE	%TABSIZE	%TEXT
%TITLE	%TRUNC	UDATASEG	UFARDATA	UNION	UNKNOWN
WARN					

רשימת מילים שמורות

תרגיל 8.2: label



שנו את התוכנית שכתבתם בתרגיל ה-jmp, כך שפקודת jmp תהייה אל label. תנו ל-label שם משמעותי.

CMP פקודת

פקודת ההשוואה cmp (קיצור של המילה compare) משמשת להשוואה בין שני אופרנדים. המעבד לא "יודע" אם האופרנדים האלו שוים, או שהאחד גדול מהשני. הדרך בה המעבד מגלה את היחס בין האופרנדים, היא היסור האופרנדים זה מזה. אם נדלק דגל האפס – האופרנדים שוים. הדלקה של דגלים אחרים (ג'ישה, נשא, סימן) מאפשרת למעבד לבדוק איזה אופרנד גדול יותר. בכך דומה פקודת cmp לפקודה sub, בהבדל אחד – היא לא משנה את ערכם של האופרנדים.

צורות כתיבה חוקיות של פקודת cmp:

תוצאה	דוגמה	הפקודה
שינוי מצב הדגלים בהתאם לייחס בין האופרנדים	cmp al, bl	cmp register, register
	cmp ax, [WordVar]	cmp register, memory
	cmp [WordVar], cx	cmp memory, register
	cmp ax, 5	cmp register, constant
	cmp [ByteVar], 5	cmp memory, constant

שינוי מצב הדגלים באמצעות הוראת cmp הוא נושא חשוב – בעוד כמה שורות נראה מה השימוש שלו.

בטבלה הבאה יש דוגמה למספר פקודות, שמורצות אחת אחרי השניה, והשפעתן על מצב הדגלים:



Code	CF	ZF	SF
mov al, 3h	?	?	?
cmp al, 3h	0	1	0
cmp al, 2h	0	0	0
cmp al, 5h	1	0	1

- בשורה הראשונה אנו פוקדים לטעון לתוך al את הערך 3. כיוון שפקודת mov אינה משנה את המצב הדגלים, בשלב זה איננו יודעים מה מצב הדגלים.

- בשורה השניה, אנו משווים את al ל-3. כפי שראינו, cmp מדמה היסור של שני האופרנדים. תוצאה היסור היא אפס (היות שהערך של al היה 3) ולכן נדלק דגל האפס. דגלי הנשא והסימן בהכרח יהיו 0 לאחר פקודה זו.

- בשורה השלישיית אנו מושווים את `a` ל-`2`. התוצאה של `a` פחות `2` היא חיובית ולכון דגל האפס כביה.
- בשורה הרביעית אנו מושווים את `a` ל-`5`. תוצאה פעלת החישור `a` פחות `5` היא בעלת בית עליון שערכו '1' לנילך דגל הסימן, כמו כן כיון שהמחסר גדול מהמחסר יש צורך בנשא שלילי לחישוב התוצאה ולכון נילך דגל הנשא.

קפיצות מותניות

פקודות של קפיצות מותניות הן כל' עבודה בסיסי ביצירת לולאות (קטע קוד שחזור על עצמו מספר פעמים) והוראות מותניות ("אם מתקיים... אז בצע..."). באופן רחב יותר אפשר להגיד שקפיצות מותניות מאפשרות לנו לבנות תוכנית מעניינת – תוכניות שיש בהן קבלת החלטות, לוגיקה וטיפול במצבים שונים.

קפיצות מותניות עובדות בדרך הבאה:

- לפני פקודת הקפיצה, מבצעים בדיקה השוואתית של שני אופרנדים באמצעות פקודה `cmp`. תוצאה פעלת `cmp` תהיה קביעת הדגלים לפי היחס בין האופרנדים, כפי שראינו קודם.
- פקודת הקפיצה בודקת אם דגל כלשהו, או כמה דגלים, מקיימים תנאי מוגדר. לדוגמה, האם דגל האפס שווה ל-`1`.
- אם התנאי מתקיים, הקוד קופץ לכתובת שהוגדרה עליידי המשמש. בדרך כלל כתובות זו תצוין באמצעות `label`.
- אם התנאי לא מתקיים, המעבד ממשיך את ביצוע הפקודות לפי הסדר (כלומר, עובר לפקודה הבאה שלאחר פקודת הקפיצה).

הערה: בשלב הראשון, ביצוע פקודת `cmp`, איןו תמיד הכרחי מבחינה תיאורטית. יש קפיצות מותניות שבודקות ישירות מצב של דגל זהה או אחר. עם זאת, שילוב פקודת `cmp` מקל על התוכנות ועל הקריאה של הקוד.

יש מגוון לא קטן של תנאים לקפיצה, כאשר כל תנאי בודק יחס אחר בין האופרנדים שהתייחסנו אליהם בפקודת `cmp`. הדבר הראשון שצריך לשיםالي לב, הוא שאין דין השווה של אופרנדים `signed` כדי השווה של אופרנדים `unsigned`.

להלן שאלה למחשבה:

איזה מספר יותר גדול – `1b` או `b10000001`?



התשובה היא – כמו שבטח ניחשתם בשלב זה – שתלייך בודקים. אם מתיחסים אל שני המספרים בתור `unsigned`, הרי `10000001b` (שווה `129` בבסיס עשרוני) גדול מ-`1b`. לעומת זאת, בהשואת מספרים `signed`, מיתרגם למינוס `127`, ולכון קטן מ-`1b`.

ברור, אם כך, ש כדי לבצע קפיצות מותנות בתוצאות של cmp, אנחנו חייבים להודיעו לאסמבולר איזה סוג השווה ביצענו, או במקרים אחרים – אילו דגלים לבדוק. לכן התפתחו שני סיטים מקבילים של פקודות קפיצה, שבנויות כך:

כל פעולות הקפיצה מתחילה באות L, קיצור של jump. פקודה קפיצה שבאות אחרי cmp של מספרים unsigned יכולת את האותיות B או A – קיצורים של Below ושל Above. לעומת זאת, פקודות קפיצה שבאות אחרי cmp של מספרים signed יכולות יכילה את האותיות L או G – קיצורים של Less ושל Greater. בנוסף לאות L ולאות G, יכולות הזכרנו, יכולות להתווסף האותיות N בשביל Not ו-E בשביל Equal.

רכיב האפשרויות נמצא בטבלה הבאה. שימו לב להגדלת סדר האופrndים:

cmp Operand1, Operand2

מספרים Unsigned	מספרים Signed	משמעות הפקודה
JA - Jump if Above	JG - Jump if Greater	קפוץ אם האופrnd הראשון גדול מהשני
JB - Jump Below	JL - Jump if Less	קפוץ אם האופrnd הראשון קטן מהשני
JE - Jump Equal		קפוץ אם האופrnd הראשון והשני שווים
JNE - Jump Not Equal		קפוץ אם האופrnd הראשון והשני שונים
JAE - Jump if Above or Equal	JGE - Jump if Greater or Equal	גדול או שווה לאופrnd השני
JBE - Jump if Below or Equal	JLE - Jump if Less or Equal	קטן או שווה לאופrnd השני

טיור בדיקת הדגלים (הרחבה)



כל פקודת קפיצה מבוססת על בדיקה ותנאים לוגיים של דגל אחד או יותר. רק לטובת הבנה כללית, נסתכל לדוגמה על פקודת `cmp` בין שני המספרים שהזכרנו בתחילת הסעיף:

```
mov al, 10000001b
mov bl, 1b
cmp al, bl
```

פעולה `cmp` בין מספרים אלו תעביר את הדגלים במצב הבא:

J	=1
c=0	
z=0	
s=1	
o=0	
p=0	
a=0	
i=1	
d=0	

פעולה JA, שמיועדת למספרים `unsigned`, צריכה להחזיר תשובה חיובית (129 גול מ-1). התנאים שיבדקו הם אם CF שווה לאפס ואם ZF שווה לאפס. תוצאה הבדיקה תהיה חיובית, ולכן תבוצע קפיצה. פעולה JG, שמיועדת למספרים `signed`, צריכה להחזיר תשובה שלילית (מינוס 127 אינו גול מ-1). התנאים שיבדקו הם אם SF=OF או ש-ZF שווה לאפס. התוצאה תהיה שלילית ולכן לא תבוצע קפיצה.

תרגיל 8.3: הוראות בקרה (השוואה וקפיצה)



הערה: אלא אם נכתב אחרת, הכוונה למשתנים בגודל בית.

- כתבו תוכנית שבודקת אם `ax` גדול מאפס (יש להתייחס לנרך של `ax` ביצוג שלו כ-`signed` כמוון), ואם כן מוריידה את ערכו באחד.
- כתבו תוכנית שבודקת אם `ax=bx`, ואם לא – מעתקה את `ax` לתוך `ax`.
- כתבו תוכנית שבודקת אם `Var1` גדול מ-`Var2` (שימו לב – יש להתייחס אל הערכים במשתנים כ-`signed`). אם כן, `ax=0`, אחרת `ax=1`.
- כתבו תוכנית שמבצעת את הפעולה הבאה: מוגדרים שני משתנים בגודל בית, `var1` ו-`var2`. אם שני המשתנים שוים – התוכנית תחשב לתוך `ax` את `var1+var2`. אם הם שונים – התוכנית תחשב לתוך `ax` את `var1-var2`.
- קטע הקוד הבא מדפס למסך את התו 'x':

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

כיתבו תכנית, שמודגר בה משתנה בגודל בית בשם TimesToPrintX. תנו לו ערך התחלתי כלשהו (חיובי). הדפיסו למסך כמהות א'ים כערכו של TimesToPrintX. הדרכה: החזיקו ברגיסטר כלשהו את כמהות הפעמים שי-'א' כבר הודפס למסך. צרו label שמדפיס x למסך ומקדם את הרגיסטר ב-1. לאחר מכן בצעו השוואה בין הרגיסטר ל-TimesToPrintX, ואם לא מתקיים שוויון – קופזו ל-label.

מה יקרה אם TimesToPrintX יאותחל להיות מספר שלילי? או אפס? טפלו במקרים אלו.

פקודת LOOP

לעתים קרובות אנחנו צריכים לבצע פעולה כלשהי מספר מוגדר של פעמים. לדוגמה, בתרגיל האחרון היינו צריכים להדפיס למסך א' כמהות של פעמים שנמצאת במשתנה TimesToPrintX. כדי לבצע את הפעולה זו, נדרש להחזיק ברגיסטר כלשהו את כמהות הפעמים שי-'א' כבר הודפס למסך, להשוו את כמהות הפעמים זה TimesToPrintX, ואם לא מתקיים שוויון – לkopioן אחריה למקום בו מודפס עוד א' למסך. פעולה זו – של ביצוע פעולה מוגדרת מספר פעמים – נקראת **לולאה (Loop)**. כיון שפעולה זו נפוצה למדי, ישנה פקודה מיוחדת שעשוה בשילוב חלק מהעבודה.

פקודת loop מבצעת את הפעולות הבאות:

- מפחיתה 1 מערכו של cx.
- משווה את cx לאפס.

אם אין שוויון (כלומר, ערכו של cx אינו אפס) – מבצעת jmp ל-label שהגדכנו.

הפקודה הזו:

```
loop SomeLabel
```

זהה הקוד הבא:

```
dec cx
cmp cx, 0
jne SomeLabel
```

דוגמה לשימוש בפקודת loop לביצוע התכנית שמדפיס א' למסך:

```
xor cx, cx ; cx=0
mov cl, TimesToPrintX ; we use cl, not cx, since TimesToPrintX is byte long
```

```
    mov  dl, 'x'
```

PrintX:

```
    mov  ah, 2h
    int  21h
loop PrintX
```

זיהוי מקרי קצר

נקודה למחשבה: העתיקו את הקוד של מעלה והריצו אותו, עבר ערך חובי כלשהו של TimesToPrintX. האם



אתם מזהים מקרה כלשהו, שבו התכנית לא תבצע בדיקת מה שרצינו?

פתרון:

אתחלו את TimesToPrintX לאפס.

כמו שאמרנו, הפקודה loop קודם כל מפתיחה את ערכו של CX וזו משווה אותו לאפס. לכן, אם לפני פקודת הד-loop הערך של CX היה אפס, לפני ההשוואה עם אפס בפעם הראשונה, ערכו של CX הוא כבר 65,535 (היצוג של מינוס 1 כ-unsigned). בפעם הבאה CX כבר יהיה שווה 65,534 וכך הלאה – לאחר 65,536 פעמים ערכו של CX יתאפס ורק אז תנאי העצירה יתקיים.

כדי שהתוכנית תעבור בכל מקרה, צריך להוסיף לה בדיקה לפני הכניסה לולאה (מודגש בצהוב):

```
xor  cx, cx
mov  cl, TimesToPrintX      ; we use cl, not cx, since TimesToPrintX is byte long
cmp  cx, 0
je   ExitLoop
```

PrintX:

```
...
; Some code for printing 'x'
Loop PrintX
```

ExitLoop:

...

לולאה בתוך לולאה Nested Loops (הרחבה)



לעתים נרצה לבצע לולאה בתוך לולאה – קוד שרצץ מספר מוגדר של פעמים, ובתוכו רין קוד אחר מספר מוגדר של פעמים.

התבוננו בקוד הלא-תקין הבא:

```
mov cx, 10
```

LoopA:

```
mov cx, 5
```

LoopB:

```
... ; Some code for LoopB
loop LoopB
... ; Some code for LoopA
loop LoopA
```

הכוונה במקרה זה היא די ברורה – התכנית צריכה לרוץ 10 פעמים על LoopA, ובכל פעם שהוא בתוך LoopA לרוץ 5 פעמים על LoopB. בטע הכל, LoopB צריך להיות מבוצע 50 פעמים. מה לא תקין בקוד הזה?

הבעיה היא ששתי הוראות ה-loop משנות את ערכו של cx. בסיום ריצת LoopB ערכו של cx הוא אפס. הקוד מגיע לסוף LoopA ומוריד 1 מערכו של cx, מה שגורם לו לרוץ 65,536 פעמים. אך בכל ריצה כזו cx מתאפס... אי לך התכוון לעולם לא תיגמר.

כדי לפתור את הבעיה, אין ברירה אלא להשתמש ברגיסטר אחר לטובת שמירת כמות הפעמים שאחת הלולאות רצתה (או לשומר את cx במחסנית- נושא שנלמד בהמשך). דוגמה לאותו הקוד, רק תקין:

```
mov bx, 10
```

LoopA:

```
mov cx, 5
```

LoopB:

```
...      ; Some code for LoopB

loop    LoopB

...      ; Some code for LoopA

dec     bx

cmp     bx, 0

jne     LoopA
```

תרגיל 8.4: לולאות



- א. סידרת פיבונצ'י: סדרת פיבונצ'י מוגדרת באופן הבא – האיבר הראשון הוא 0, האיבר השני הוא 1, כל איבר הוא סכום שני האיברים שקדמו לו (לכן האיבר השלישי הוא $1+0=1$, האיבר הרביעי הוא $1+1=2$ וכו'). צרו תוכנית שמחשבת את עשרת המספרים הראשונים בסדרת פיבונצ'י ושומרת אותם בזיכרון בעל 10 בתים. בסיום התוכנית המערך צריך להכיל את הערכים הבאים:

0,1,1,2,3,5,8,13,21,34

- ב. שרוט הקוד הבאות קולטותתו מהמשתמש:

```
mov    ah, 1h
int    21h          ; al stores now the ASCII code of the digit
```

- צרו תוכנית שקולטת 5 תווים מהמשתמש ושומרת אותם לוחץ מערך. בידקו את התוכנית על-ידי הכנסת התווים HELLO ובדיקה שהמערך מכיל תווים אלו.

- ג. צרו תוכנית שמחשבת את המכפלה $Var1 * Var2$, משתנים בגודל בית שיש להתייחס אליהם כ-`unsigned`. אך מבצעת זאת על-ידי פועלות חיבור. הדרכה: יש לבצע פועלות חיבור של `sum=Var1+sum` ולהזoor עליה על-ידי `Var2 loop` פעמיים.

- ד. שאלת אתגר: צרו תוכנית שמקבלת שני מספרים מהמשתמש ומדפסה למסך מטריצה של 'X'ים בגודל המספרים שנקלטו – המספר הראשון הוא מספר השורות והשני הוא מספר העמודות במטריצה. לדוגמה עבור קלט 5 ואחר כך 4, יודפס למסך:

xxxx
xxxx
xxxx
xxxx
xxxx

הדרך:

קליטת ספרה – שורות הקוד הבאות קולטות ספרה מהמשתמש, מmirות אותה למספר בין 0 ל-9 ומעתיקות את התוצאה ל-**al**:

```
mov ah, 1h
int 21h          ; al stores now the ASCII code of the digit
sub al, '0'       ; now al stores the digit itself
```

הדפסה למסך – קטע הקוד הבא מדפיס למסך את המספר 'x':

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

מעבר שורה – הפקודות הבאות גורמות להדפסה של מעבר שורה:

```
mov dl, 0ah
mov ah, 2h
int 21h
```

קפיצה מוחוץ לתחום

סקרנו מספר פקודות שמקפיצות אותנו למקום אחר בקוד:

- פקודה jmp

- פקודות קפיצה מותנית כגון jb, jg, ja וכו'

- פקודה loop

במקרים מסוימים, שימוש בפקודת קפיצה מותנית או בפקודת loop עלול לגרום לשגיאה הבאה בזמן ביצוע האסמלבלר: Error- Relative jump out of range.

האסמלבלר מתרגם כל פקודה ל-opcode. בעוד שלפקודת jmp האסמלבלר מקצת 16 ביט לייצוג טווח הקפיצה (כלומר טווח של מינוס 32768 עד פלוס 32767 בתים מהמקום הנוכחי), האסמלבלר מקצת רק 8 ביט לייצוג טווח הקפיצה של פקודות קפיצה מותנית ושל loop (כלומר טווח של 127 בתים קדימה עד 128 בתים אחוריה בזיכרון). לכן, אם ננסה לכתוב פקודה relative jump out of range לתוכית שנמצאת בmphak גדול מהטוויה שnitן נכתב ב-8 ביט, נקבל שגיאת .range

ישנו מספר דרכי להימנע עז בעיה זו- כולם סוגים של מעקפים:

1. ההתחמודדות הכי פשוטה היא לא לייצר קפיצות מותניות שהורגות מהטוויה וכך להימנע מהבעיה.

2. שיטה פשוטה אך לא אלגנטית- לקפוץ למקום קרוב וממנו מיד לקפוץ למקום הבא, עד שmagיעים למקום הנכון בקוד.

3. להמיר את פקודות הקפיצה המותנית שגורמות לשגיאה, בפקודת קפיצה בלתי מותנית.

נדגים את השימוש בהמרת קפיצה מותנית בקפיצה בלתי מותנית. כתבו את הקוד הבא, נניח שפקודת הקפיצה ja גרמה לשגיאה:

```
cmp ax, bx
```

```
ja my_label
```

```
.... ; more than 127 bytes in code memory
```

```
my_label:
```

```
.... ; some code here
```

cut נבצע המرة מ ja ל jmp:

```
cmp ax, bx
```

```
jbe help_label
```

```
jmp my_label
```

`help_label:`

```
.... ; more than 127 bytes in code memory
```

`my_label:`

```
.... ; some code here
```

יצרנו מגננו שמחליף את הקפיצה המותנית בקפיצה בלתי מותנית. בואו נראה כיצד מתנהג הקוד החדש שכתבנו. בקוד המקורי התרחשה קפיצה ל-`-ax>jb` אם התקיים התנאי `my_label` (הפקודה `ja` משמעה "קפוץ אם גדול ממש", והיא הפוקודה ההפוכה ל-`je`, שמשמעותה "קפוץ אם קטן או שווה"). בקוד החדש, כאשר `ax>jb` התנאי `je` אינו מתקיים, כותצאה מכך התוכנית מגיעה לשורת הקוד `jmp my_label` והקפיצה מתקיימת.

סיכום

בפרק זה:

- למדנו על פקודת `jmp`, המשמשת לביצוע קופיזות. למדנו להיעזר ב-`labels` כדי לסמן מקומות בקוד שנרצה קופוץ אליהם.
- למדנו על פקודת `cmp`, שמשמשת להשוואה בין שני ערכים ומשנה את מצב הדגלים בהתאם.
- למדנו סוגי שונים של קופיזות מותניות ("קפוץ אם ערך גדול מ...", "קפוץ אם ערך קטן או שווה...", "קפוץ אם ערך שווה אפס...").
- עמדנו על הבדלים בין השוואת מספרים `signed` לבין השוואת מספרים `unsigned` וראינו שישנן פקודות שונות לאללה ולאלה.
- ראיינו איך מגדירים לולאות `loop` שימוש בפקודת `loop` וברגיסטר `cx` (או `cl`).

בסוף הפרק הגענו לרמה שמאפשרת לנו לכתנת תוכניות שכוללות אלגוריתמים, תנאים לוגיים וחזרות על קטעי קוד. בשלב זה הקוד שאנו יוצרים מתאים לתוכניות קצרות, ועדיין לא לביצוע תוכניות ארוכות ומורכבות. חסירה לנו עדין היכולת לכתוב קטעי קוד שמקבלים פרמטרים ומבצעים פעולות שונות – פרוצדורות. על כך נלמד בפרק הבא.

פרק 9 – מחסנית ופראוצדורה

מבוא

המטרה של הפרק זהה היא ללמד איך לכתוב תוכניות מודולריות. ראשית, כדאי שنبין – מה זאת אומרת לכתוב תוכנית מודולרית? ומדוע זה חשוב?

תוכנית מודולרית היא תוכנית שבנויות ממודולים – חלקים של קוד, שיש להם נקודת כניסה אחת ונקודת יציאה אחת, והם מבצעים פעולה מוגדרת. מודול זה נקרא **פראוצדורה (Procedure)** (בשם העברי "תת תוכנית") או **פונקציה**



בשפה אSEMBLAI אין הבדל מעשי בין פראוצדורה לפונקציה ולכל נתיחה יכול בטור פראוצדורות. כאשרנו כותבים קוד מודולרי – קוד שבוני מפראוצדורות עם קוד שמקשר ביניהם – אנחנו מתכוונים מראש איך לחלק את מה שהתוכנית שלנו צריכה לעשות לכמה פראוצדורות, כאשר לכל פראוצדורה יש תפקיד מוגדר. לעובדה בשיטה זו יש כמה יתרונות:

יתרון ראשון – הקוד שלנו קצר יותר. נניח שחלק מהתוכנית שלנו דורש שנקודות סיסמה מהמשתמש, ואנחנו צריכים לעשות את הפעולה הזו לעיתים קרובות. זו לא פעולה מסובכת, אבל עדין – צריך להגיד לו לאו שרצה כמה פעמים, להפעיל בכל פעם פקודה של קליטתתו ולדאוג לשמור את התווים שנקלטו. אם נגיד פראוצדורה ש יודעת לקלוט תווים מהמשתמש, במקום לכתוב את כל הקוד של קליטת התווים, נוכל פשוט לקרוא לפראוצדורה שלנו. בסוף הפראוצדורה התוכנית תקופץ חוזרת למיקום האחרון הייתה בו לפני הקראיה. כך חסכנו את כתיבת חזרות של קוד קראית התווים.

רגע, למה שלא פשוט נגיד `label`, נקרא לו, לדוגמה – `ReadPassword`, ובכל פעם שנצטרכן לקלוט סיסמה נעשה אליו `jmp`? אנחנו בהחלט יכולים לעשות את זה, וזה גם יעבוד, אבל لأن **נמשיך** אחורי קראית הסיסמה? אם לא ניתן לתוכנית הוראה אחרת, היא פשוט תמשיך אל שורת הקוד הבאה. אולי נפתר או זה באמצעות פקודת `jmp` בסוף קראית הסיסמה? זה יעבד. אבל רגע – מה אם יש יותר מקום אחד בקוד שדורש להקליד סיסמה? דבר זה בעייתי, שכן בכל פעם נרצה לקרוא לקוד שיקרא את הסיסמה ולהמשיך במקום אחר בתוכנית. נגיד שאנחנו צריכים לכתוב קוד שקולט סיסמה לגישה למחשב, ולאחר כך סיסמה לחישבון המיל. אחורי קראית הסיסמה, איך נדע לנו לקפוץ?

המחשת הבעיה של שימוש ב-`label` שאפשר לקרוא לו מיותר מקום אחד:

OpenComputer:

```
jmp     ReadPassword
```

```
...           ; Code for signing into computer
```

OpenEmail:

```
jmp     ReadPassword
...
; Code for signing into email
```

ReadPassword:

```
...
; Code for reading password from user
jmp     ???      ; Where should we jump back to???
```

גם לבעה זו יש פתרון – אנחנו יכולים להגדיר משתנה שקובע לאן צריך לקפוץ ולבדק אותו עליידי פקודת cmp – אבל כל הבדיקות האלה מתחילה להיות יותר ארכוכות מאשר לכתוב את הקוד של קליטת הסיסמה בכל המיקומות שאנו צריכים אותו... פְּרָוֶצְדוֹרוֹת מהוות פתרון אלגנטי ויעיל לסוגיה זו.

יתרון שני – אנחנו יכולים לבדוק חלקיים מהקוד שלנו ולדעת שם עובדים בצורה טובה. נגיד שהגדכנו פְּרָוֶצְדוֹרָה ReadPassword. כתבנו אותה, בדקנו שהיא עובדת כמו שאנו רוצים. זהו. לא צריך לגעת בה יותר. אם יש לנו באג בתוכנית, אז נחפש אותו במקומות אחרים. אם במקרה מצאנו באג בפְּרָוֶצְדוֹרָה ReadPassword, מספיק שנתקן אותו פעמי – והוא יתוקן עבור כל הקריאות בתוכנית. העבודה שיש לנו נוספת של קטעי קוד בדוקים מקטינה משמעותית את המאמץ שלנו לגלות באגים.

יתרון שלישי – קל יותר לקרוא את הקוד שלנו. במקום לכתוב הרבה קוד, אנחנו פשוט כותבים פקודה שנקראת call ואת שם הפְּרָוֶצְדוֹרָה. כך, לדוגמה:

call ReadPassword

יתרון רביעי – יכולת לשף קוד גם בין תוכניות וגם בין אנשים. נניח שכתבנו קטע קוד שמבצע משהו שימושי ונפוץ, קליטת סיסמה המשמש. אם נשים את הקוד בתוך פְּרָוֶצְדוֹרָה ואת הפְּרָוֶצְדוֹרָה נשמר בקובץ נפרד, יוכל להשתמש בה בכל תוכנית שנכתוב וגם לשף בה אחרים. העיקרון שלא צריך "להמציא את הגלגל מחדש" בכל פעם שאנו רוצים לתכנת משהו, עומד בסיס התוכנות מרובות הקוד שקיימות היום.

מושלם, לא? ובכן, עציו נראה קצת חסרונות של פְּרָוֶצְדוֹרוֹת.

חיסרון ראשון – היתרון של שיתוף קוד הוא פתח לביאות. אם יש באג בקוד שמשהו כתוב והקוד שותף עם הרבה אנשים, שהשתמשו בקוד בתוכנות שלהם, הבאג נפוץ בכל התוכנות הללו וכוחזאה מכך קשה לתקן אותו. הקושי בתיקון הבאג הוא הריף במינוח אם הוא נמצא בתוכנה שמוקטנת אצל המון גודלה של משתמשים, שנדרשים לעדכן את התוכנה שברשותם.

חיסרון שני – כתיבת פְּרָוֶצְדוֹרָה דורשת השקעה מסוימת מצד המתכנת. יותר קל פשוט לכתוב בתוכנית הראשית את הקוד מאשר להגדיר אותו בתחום פְּרָוֶצְדוֹרָה, במיוחד בתוך מתכנתים מתחילה.

חיסון שלישי – הקריאה לפrozצדרות והזרה מהן דורשת **משאבים** של המחשב. לדוגמה, ביצוע פעולות קפיצה והזרה. לדוגמה, שימוש בזיכרון. מכאן שקריאה לקוד בתוך פרוזצדרה היא "יראה" יותר עבר המחשב מאשר הריצה של הקוד שלא מתוך פרוזצדרה.

חרף החסרונות שסקרנו, התוצאה שכטיבת קוד מודולרי, שבוסס על פרוזצדרות, עולה על החסרונות שלו. לכן נקבע לכתוב את הקוד שלנו בצורה זו.

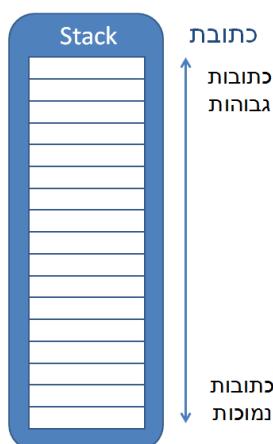
המחסנית STACK

המחסנית (**STACK**), היא סגמנט בזיכרון. המחסנית משמשת לאחסון לזמן קצר של משתנים, קבועים וכותבות

ולכן היא כלי חשוב בעבודה עם פרוזצדרות ויצירת תוכניות מודולריות.



כדי שנוכל להבין באופן מלא איך לכתוב פרוזצדרות, נצרך קודם כל להכיר היטב את המחסנית. לכן, נפתח בהסבר על המחסנית, הגדרה של מהסנית ופקודות הקשורות לשימוש במחסנית.



תיאור סכימי של *Stack Segment*

הגדרת מהסנית

כמו כל סגמנט בזיכרון, המחסנית היא אזור בזיכרון שמתחליל בכתובת כלשהו ותופס גודל מוגדר של זיכרון. גודל האזור בזיכרון שמצוצה למחסנית נקבע עלי ידי המתכונת בתחלת התוכנית. הקצתה המקום נעשית בדרך הבאה:

STACK number of bytes

לדוגמה, כדי להקצות מהסנית בגודל 256 בתים, נגיד (כפי שמצוגד גם בקובץ *asm*):

STACK 100h

למחסנית יש שני רגיסטרים שקשורים אליה. עשינו אותם הিירות קטרה בעבר, כשסקרנו את כל הרגיסטרים שיש למעבד:

- **stack segment – ss**

- מצביע stack pointer – **sp**

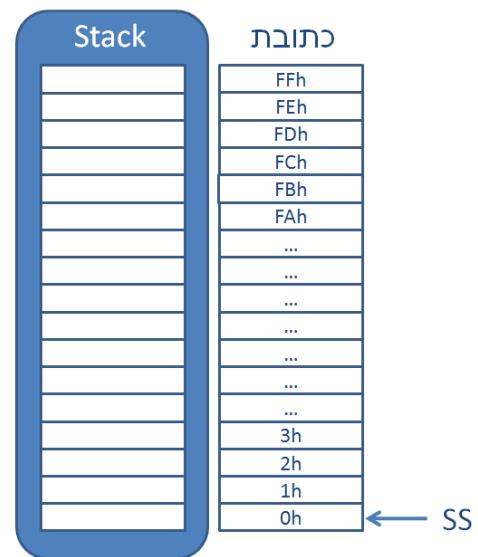
מצביע המהסנית, **sp**, הוא רגיסטר שمحזיק אופסט בזיכרון. בכך אין הוא שונה מהרגיסטר **ax**. כמו שאפשר להגיע לכל כתובות ב-**DATASEG** בעזרת ה指挥 **ax:ds**, כך אפשר להגיע לכל כתובות ב-**STACK** בעזרת הוספה האופסט הנתנו ב-**ds** לכתובת הסגמנט הנתונה ב-**ss**.

בתחילת התוכנית, הרגיסטר **sp** שווה לגודל המהסנית. בדוגמה שנשתמש בה, מהסנית בגודל **100h**, לפני שהכנסנו ערך כלשהו לmahסנית **sp** מאותחל להיות שווה **100h**.

שימוש לב שבמהסנית בגודל **100h**, לעומת 256 בתים, מרחב הכתובות אינו מגיע עד **100h** אלא הוא בין 0 ל-**OFFh**. לעומת, הערך ההתחלתי של **sp**, **100h**, מצביע על כתובות שהיא בדיקת בית אחד מעלה קצה המהסנית.



mahסנית בגודל **100h**. הרגיסטר **ss** מצביע על תחילת סגמנט mahסנית. בתחילת התוכנית הרגיסטר **sp** מאותחל לערך **100h**, לעומת לערך שנמצא מחוץ לmahסנית.



העובדת ש-**sp** מאותחל למצביע לא על תחילת mahסנית אלא על הקצה שלה נראה כרגע קצר מזמן, אבל היא קשורה בדרך שבה mahסנית מנוהלת. mahסנית מנוהלת בשיטת **LIFO** – **Last In First Out**, לעומת הערך שנכנס אחרון הוא הראשון לצאת מהmahסנית. לפני הכנסה של נתונים לmahסנית, ערכו של **sp** יורד ולאחר הוצאה של נתונים מהmahסנית ערכו של **sp** עולה. נבין זאת כאשר נלמד על הפקודות המשמשות אותנו בעת הכנסה והויצאה של נתונים מהmahסנית, ונראה דוגמאות.

אילו פקודות מאפשרות הכנסה והויצאה של נתונים מהmahסנית?

PUSH פקודת

פקודת `push` גורמת להכנסת של ערך למחסנית. הפקודה נכתבת כך:

`push operand`

מה יבוצע?

- ערכו של `sp` יורד בשתיים: $sp=sp-2$.

- ערכו של האופרנד יועתק למחסנית, לכטובה `ss`.

שים לב: ערכו של `sp` תמיד יורד בשתיים עם פקודת `push`, כלומר הוא מצביע על כתובות שרחוקה שני בתים מהכתובת الأخيرة עלייה הצבוי. המשמעות היא שאפשר לדוחף למחסנית רק משתנים רק מודול של שני בתים – `word`. כל ניסיון לבצע `push` לכמות אחרת של בתים – יוביל לשגיאה.

דוגמאות לשימוש בפקודת `push`:

`push ax`

`push 10`

`push var`

הפקודה הראשונה תדוחף למחסנית את `ax`.

הפקודה השנייה תדוחף למחסנית את הערך 10 (בצורתו כ-`word`, לא כ-`byte`).

הפקודה השלישית תדוחף למחסנית את תוכן המשתנה `var` – בתנאי שהוא מוגדל `word`.

האם הפקודה הבאה היא חוקית?



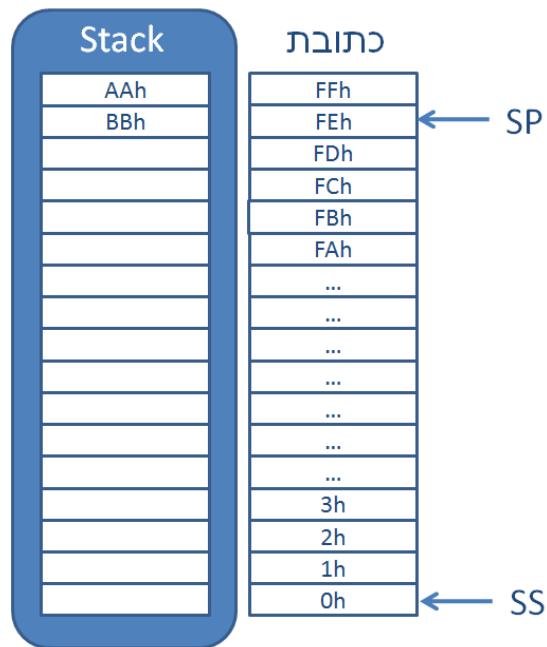
`push al`

תשובה: פקודה זו היא שגויה. `al` הוא לא בגודל מילה ופקודת `push` מקבלת רק רегистרים בגודל מילה.

נחזיר למחסנית שהגדכנו, בגודל `100h`. מה יהיה מצב המחסנית לאחר ביצוע הפקודות הבאות?

`mov ax, 0AABBh`

`push ax`



פקודת `push` גורמת לירידת ערכו של `sp` ב-2

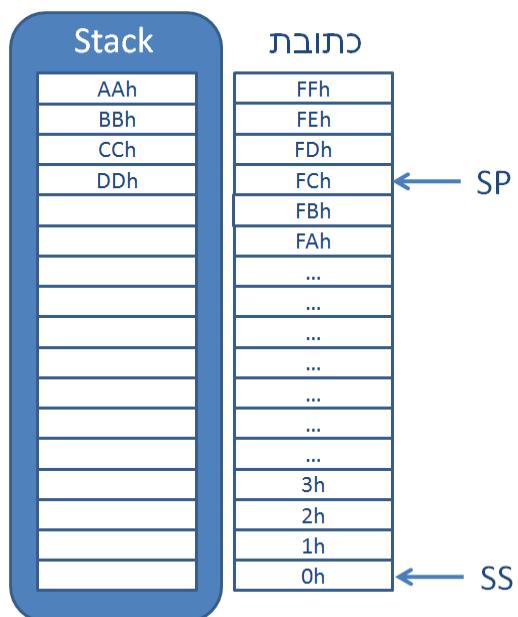
והוא מצביע על כתובת נמוכה יותר במחסנית

שימו לב לכך שהביטים הגבוהים של `ax`, אלו שהמוראים ב-`hh`, נדחפו למחסנית ראשונים בכתובת גבוהה יותר.

אנו יכולים להמשיך לדחוף ערכים למחסנית לפי הצורך. כל דחיפה כזו תוריד עוד 2 מערך `sp`. לדוגמה:



`push 0CCDDh`



פקודת **POP**

פקודת **pop** היא הפקודה הפוכה ל-**push**. פקודה זו גורמת להוצאה של מילה (שני בתים) מהמחסנית והעתקה שלה לאופרנד:

pop operand

מה יבוצע?

- מילה ראש המחסנית תועתק לאופרנד.

- ערכו של **sp** יעליה ב-2.

דוגמאות לשימוש בפקודת **pop**:

pop ax

pop [var]

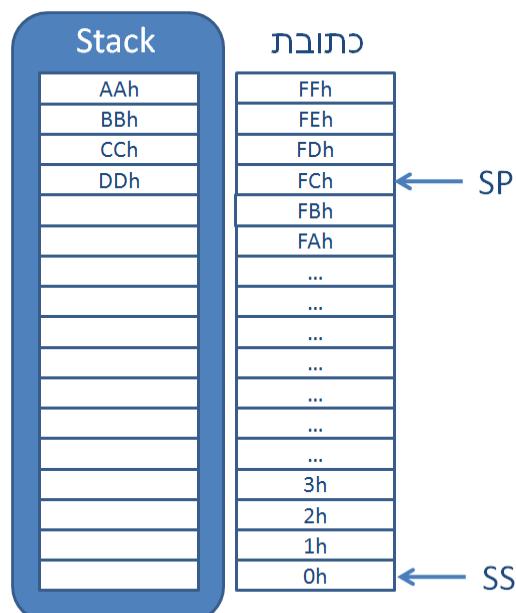
הפקודה הראשונה תעתק לתוכן **ax** את המילה שבראש המחסנית.

הפקודה השנייה תעתק לתוכן המשתנה **var** את המילה שבראש המחסנית (כਮובן שכדי לא לקבל שגיאת קומפילציה, **var** צריך להיות מטיפוס מילה).

pop al

זהו כצפי פקודה לא חוקית – פקודת **pop** לא יודעת לקבל כאופרנד רגיסטר או זיכרון בגודל בית.

המחסנית שהשתמשנו בה בדוגמה נמצאת כרגע במצב זה:



pop bx

שאלה: מה יהיה ערכו של sp לאחר ביצוע הפקודה הבאה? ומה יהיה ערכו של ax?



תשובה: ערכו של ax יהיה bx, ערכו של sp יעלה בשתיים ויהיה שווה 0FEh.

pop var

שאלה: לאחר ביצוע הפקודה הקודמת, בוצעה פקודת הדוקן הבאה:



מה יהיו הערכים של sp ושל var לאחר ביצוע הפקודה הבאה?

תשובה: ערכו של var יהיה 0AABBh, ערכו של sp יעלה בשתיים ויהיה שווה 100h.

מה קרה לערכים שבתווך המחסנית? האם הם נמחקו?



לא! הערכים שהכנסנו למחסנית עדין קיימים בתוכה, אך כעת אין למעבד דרך לגשת אליהם כיוון ש-ds אינו מצביע עליהם יותר. טיורטית אנחנו עדין יכולים לגשת אליהם אם היינו מבצעים את הפקודה הבאה:

sub sp, 4

אך מעשית לא מקובל "לשחק" עם ערכו של sp, מסיבות שנעמדו עליו בהמשך.

תרגיל 9.1: מהסנית, push ו-dsop



- א. הגדרו מהסנית בגודלים שונים. 10h, 20h, 1234h. ראו כיצד משתנה ערכו של sp עם תחילת ההרצה.
- ב. הכניסו את הערך 1234h לתוך ax. בוצעו push של ax. איך השתנה sp? הסתכלו על הזיכרון שבמחסנית ומיצאו את הערך שדוחף там.
- ג. בוצעו pop למחסנית לתוך ax. איך השתנה sp? הסתכלו על הזיכרון שבמחסנית – האם הערך 1234h נמחק?
- ד. בוצעו push לערך 5678h. האם עכשו נמחק הערך ?1234h?
- ה. העתיקו את ax לתוך bx בעזרת המחסנית ללא שימוש בפקודת mov.

פראוצדורה

הגדרה של פראוצדורה

פראוצדורה היא קטע קוד שיש לו כניסה אחת, יציאה אחת (רצוי), והוא מבצע פעולה מסוימת. יש כמה רכיבים שהופכים "סתם" קטע קוד לפראוצדורה:

- קוראים לפראוצדורה באמצעות הפקודה `call .`
- אפשר להעביר פרמטרים לפראוצדורה. לדוגמה, פראוצדורה שמחברת שני מספרים ומחזירה את סכומם – אפשר להעביר לה כפרמטרים שני משתנים – `num1, num2`.
- לפראוצדורה יש מנגנוןים להחזיר תוצאות העבודה. למשל, `num1+num2` לא רק ייחסב, אלא גם יועבר חוזה לתוכנית שזימנה את הפראוצדורה.
- פראוצדורה יכולה ליצור משתנים מקומיים (שימושים את הפראוצדורה בלבד) ולהיפטר מהם לפני החזרה לתוכנית הראשית.

פראוצדורה מגדירים או מיד בתחילת CODESEG, או בסופה של CODESEG. בפראוצדורה יהיה קטע קוד:

```

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

```

דוגמה לפראוצדורה – להלן תוכנית שכוללת פראוצדורה בשם ZeroMemory, פראוצדורה שמפעסת 10 בתים מתחילה DATASEG (כלומר, הופכת את ערכם ל-0). שימו לב למיקום הפראוצדורה בתוך CODESEG, להגדרת הפראוצדורה ולקריאה לפראוצדורה:

IDEAL

MODEL small

Stack 100h

DATASEG

```
digit    db 10 dup (1)      ; if we do not allocate some memory we may run over
                                ; important memory locations
```

CODESEG

```
proc ZeroMemory          ; Copy value 0 to 10 bytes in memory, starting at location bx
    xor    al, al
    mov    cx, 10
```

ZeroLoop:

```
    mov    [bx], al
    inc    bx
    loop   ZeroLoop
    ret
endp ZeroMemory
```

start:

```
    mov    ax, @data
    mov    ds, ax
    mov    bx, offset digit
    call   ZeroMemory
```

exit:

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

END start

The screenshot shows the DOSBox interface with assembly code and registers.

Registers:

ax	0800	c=0
bx	0008	z=0
005D	s=0	
dx	0000	o=0
si	0000	p=0
di	0000	a=0
bp	0000	i=1
sp	00FE	d=0
ds	087B	
es	0869	
ss	0882	
cs	0879	
ip	0008	

Stack Dump:

ds:0000	00 00 00 00 00 00 00 00	ss:0106 0000
ds:0008	01 01 01 01 01 01 01 01	ss:0104 0089
ds:0010	01 01 01 01 01 01 01 01	ss:0102 0403
ds:0018	01 01 01 01 01 01 01 01	ss:0100 52FB
ds:0020	01 01 01 01 01 01 01 01	ss:00FE 0016

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

לאחר שהפְּרוֹצְׂדָּוֶרֶת הספיקה לאפס 8 בתים ראשוניים (מודגש בצהוב)

תרגיל 9.2



העתיקו את קוד הדוגמה של `ZeroMemory`. הריצו אותו ב-DOSBox. תראו שורה אחרת שורה וצפו בשינויים שתרחשים ב-`ds` וב-`ds` בשלבי התוכנית השונים

פקודות CALL, RET

בדוגמה ראיינו שפקודה `call` משמשת לזמן הפְּרוֹצְׂדָּוֶרֶת. נראה לבדוק מה מבצע הפקודה:

`call ZeroMemory`

לפני פקודה `call` לפְּרוֹצְׂדָּוֶרֶת, הרגייסטר `ip=12h`, `CODESEG`, כלומר מצביע על שורה 12h, שהיא השורה שאחרי פקודה `call` (בדוגמה שלנו: (...).

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TD
File Edit View Run Breakpoints Data Options Window Help READY
[CPU 80486]
#baseproc#start: mov ax, @data
    cs:000B B87B08      mov     ax,087B
#baseproc#22:   mov ds, ax
    cs:000E 8ED8        mov     ds,ax
#baseproc#24:   mov bx, ds
    cs:0010 8CDB        mov     bx,ds
#baseproc#25:   call ZeroMemory
    cs:0012 E8EBFF      call    #baseproc#zeromemory
#baseproc#exit:  mov ax, 4c00h
    cs:0015 B8004C      mov     ax,4C00
#baseproc#28:   int 21h
    cs:0018 CD21        int    21
    cs:001A 0000        add    [bx+si],al
    cs:001C 0000        add    [bx+si],al
    cs:001E 0000        add    [bx+si],al
es:0000 CD 20 ?D 9D 00 EA FF FF = 3￥ ?
es:0008 AD DE 32 0B C3 05 6B 07 i [2d] ok.
es:0010 14 03 28 08 14 03 92 01 !ok( 01!ok?
es:0018 01 01 01 00 02 04 FF FF 0000 ok
es:0020 FF FF FF FF FF FF FF FF FF FF

```

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

שימוש לב לשינוי בערכו של ip מיד לאחר ביצוע הוראות ה-call:

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TD
File Edit View Run Breakpoints Data Options Window Help READY
[CPU 80486]
#baseproc#zeromemory: xor al, al
    cs:0000 32C0      xor    al,al
#baseproc#12:   mov cx, 100
    cs:0002 B96400    mov    cx,0064
#baseproc#zeroloop: mov [bx], al
    cs:0005 8807      mov    [bx],al
#baseproc#15:   inc bx
    cs:0007 43        inc    bx
#baseproc#16:   loop ZeroLoop
    cs:0008 E2FB      loop   #baseproc#zeroloop (0005)
#baseproc#17:   ret
    cs:000A C3        ret
#baseproc#start: mov ax, @data
    cs:000B B87B08      mov     ax,087B
#baseproc#22:   mov ds, ax
    cs:0010 8ED8        mov     ds,ax
es:0000 CD 20 ?D 9D 00 EA FF FF = 3￥ ?
es:0008 AD DE 32 0B C3 05 6B 07 i [2d] ok.
es:0010 14 03 28 08 14 03 92 01 !ok( 01!ok?
es:0018 01 01 01 00 02 04 FF FF 0000 ok
es:0020 FF FF FF FF FF FF FF FF FF FF

```

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

ערך של ip השתנה ל-0000. זהו המיקום של הפְּרָוֶצְדוֹרָה ZeroMemory בזיכרו (זכרו, שמדובר ב-*offset* בתוך CODESEG, ככלומר הפְּרָוֶצְדוֹרָה נמצאת ממש בתחוםו של ה-G-SEG). כמובן, פקודת call משנה את ערכו של ip כך שתוכנית קופצת אל תחילת הפְּרָוֶצְדוֹרָה. אבל את הפעולה זו אפשר היה להשיג גם באמצעות פקודת jmp. פקודת jmp עשויה משווה נוספת. כדי שאולי שמתם לbp, רגיסטר נוספת השתנה – הרגיסטר sp. פקודת ה-call הקטינה את ערכו בשתיים. לפני פקודת ה-call ערכו היה 100h וכעת ערכו הוא 0FEh. שימוש לב לעוד נתון שהשתנה. בחלק הימני התהווון

של המסק, יש ערכאים שונים בסגמנט מהשנית ss:00FEh (מודגש בצהוב) הוא 0015h. לפני פקודה ה-call ערכו היה 0000h. מה משמעותו של ערך זה? מיד נראה.

הפְּרוֹצְׂדָּוֶרֶת כמעט סיימה את ריצתה – היא איפסה עשרה בתים בזיכרון והגיעה לפקודה ret. בנקודה זו הרегистר ip=0Ah.

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TD
File Edit View Run Breakpoints Data Options Window Help READY
[CPU] CPU 80486
cs:FFFF 0032 add [bp+si],dh
cs:0001 C0B9640088 sar byte ptr [bx+di+0064],88
cs:0006 07 pop es
#baseproc#15: inc bx
cs:0007 43 inc bx
#baseproc#16: loop ZeroLoop
cs:0008 E2FB loop #baseproc#zeroloop (0005)
#baseproc#17: ret
cs:000A C3 ret
#baseproc#start: mov ax, @data
cs:000B B87B08 mov ax,087B
#baseproc#22: mov ds, ax
es:0000 CD 20 7D 9D 00 EA FF FF = 0000:CD207D9D00EAFFFF
es:0008 AD DE 32 0B C3 05 6B 07 i 2dok.
es:0010 14 03 28 08 14 03 92 01 1403280814039201
ss:0100 52FB
ss:00FE 0015

```

הגיע הזמן להזoor לתוכנית הראשית – לאחר ביצוע פקודת ret, הרегистר ip מציביע על שורת הקוד הבאה מיד אחרי שורת הקוד שקרה לפְּרוֹצְׂדָּוֶרֶת והתכנית ממשיכה ממנה באופן טורני:

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TD
File Edit View Run Breakpoints Data Options Window Help READY
[CPU] CPU 80486
#baseproc#22: mov ds, ax
cs:000E 8ED8 mov ds,ax
#baseproc#24: mov bx, ds
cs:0010 8CDB mov bx,ds
#baseproc#25: call ZeroMemory
cs:0012 E8EBFF call #baseproc#zeromemory
#baseproc#exit: mov ax, 4c00h
cs:0015 B8004C mov ax,4C00
#baseproc#28: int 21h
cs:0018 CD21 int 21
cs:001A 0000 add [bx+si],al
cs:001C 0000 add [bx+si],al
es:0000 CD 20 7D 9D 00 EA FF FF = 0000:CD207D9D00EAFFFF
es:0008 AD DE 32 0B C3 05 6B 07 i 2dok.
es:0010 14 03 28 08 14 03 92 01 1403280814039201
ss:0102 0403
ss:0100 52FB

```

התוכנית חוזרת מהפְּרוֹצְׂדָּוֶרֶת, ip קופץ מ-0Ah ל-15h.

... 15h ? נכון, זה בדיקת הערך שנכנס למחסנית עם ביצוע פקודת ה-call .call.

נסכם מה ראיינו שפקודות ה-call וה-ret מבצעות:

פקודת call

1. מוריידה את ערכו של sp בשתיים (יש מקרה שבו היא מוריידה את ערכו של sp באربע – כאשר מדובר בפראוצדורה FAR – נראה מקרה זה בהמשך).
2. מעיטהה לתוך הכתובת ds:ss, את הכתובת בזיכרון אליה יש לחזור בסיום ריצת הפראוצדורה.
3. מעיטהה לתוך ק'ו את הכתובת של הפקודה הראשונה של הפראוצדורה (פעולה זו שולחה לביצוע jump אל תחילת הפראוצדורה)

פקודת ret

1. קוראת מהכתובת ds:ss, את הכתובת בזיכרון אליה יש לחזור.
2. מעלה את ערכו של sp בשתיים (כך במקרה זה; נראה מקרים אחרים בהמשך).
3. משנה את הדק'ו אל הכתובת שנקרה מ-sp:ss, ובכך מזיזה את התוכנית לשורת הקוד שבה מיד אחרי הקראיה לפראוצדורה.

כמו שראינו, זו לא מקרים שבסיום ריצת הפראוצדורה, הערך של ק'ו חוזר אל השורה הבאה בתוכנית. פקודות ה-call וה-ret דאגו לשמר את ערכו של sp במחסנית ולשזר אותו בסיום ריצת הפראוצדורה!

פראוצדורות NEAR, FAR

התוכנית שלנו ריצה מתוך ה-**CODESEG**. בשלב מסוים היא מגיעה לפקודת **call** אל פראוצדורה שהגדנו. הקוד עצמו של הפראוצדורה יכול להימצא באחד משני מקומות:

1. בתחום ה-**CODESEG**, יחד עם שאר הקוד של התוכנית הראשית.
2. מחוץ ל-**CODESEG** (בsegment אחר כלשהו שהגדנו).

באופן מעשי, כיוון שאנחנו מגדירים מודל זיכרון **small model**, בעל סגמנט קוד יחיד, כל הפראוצדורות שנגידיר יהיו בתחום ה-**CODESEG**, אבל יש היגיון להבין גם מה קורה במקרה השני – אחרי הכל, תוכניות יכולות לקבל גם ספריות של פראוצדורות מקומפלות, שאין נמצאות בתחום ה-**CODESEG**.

אם הפראוצדורה נמצאת בתחום ה-**CODESEG**, היא נקראת פראוצדורה **near**. הפראוצדורה **ZeroMemory** היא דוגמה לפראוצדורה מסוג **near**. במקרה כזה, פקודת **call** מכינסה לתוך המחסנית רק את האופסט של כתובות החזרה – אחרי הכל, הסגמנט כבר ידוע – **CODESEG**. כיוון שהוא א鬰ן שני בתים, הריגיסטר sp משתנה ב-2.

לעומת זאת, אם הפראוצדורה נמצאת מחוץ ל-**CODESEG**, היא נקראת פראוצדורה **far**. במקרה כזה, פקודת **call** מכינסה למחסנית גם את האופסט וגם את הסגמנט של כתובות החזרה, כדי שהמעבד יוכל לחשב בדיקות לאן לחזור. האופסט והסגמנט תופסים ביחד ארבעה בתים, ולכן במקרה זה ערכו של הריגיסטר sp משתנה ב-4.

איך אנחנו יכולים לשולוט בבחירה פקודה `call` לפראוצדורה `near` או `?far`?

אנו מודיעים לאסמבילר מהו סוג הפראוצדורה בזמן הגדרת הפראוצדורה, עלייה הוספה המילים `near` או `.far`. לדוגמה:

```
proc ProcedureName      near
proc ProcedureName      far
```

אם אנו לא כתבים לא `near` ולא `far`, ברירת המחדל היא `near`.

נכתוב גירסה חדשה של תוכנית הדוגמה שלנו. כל השינוי הוא הוספה אחת לפראוצדורה `:far` – `ZeroMemory`

```
proc ZeroMemory      far
```

חוץ מזה, הגירסה החדשה זהה לגרסה הקודמת. להלן צילום מסך של הכניסה לפראוצדורה. שימוש לב' לדברים הבאים:

- במקום `cs:0012` בתוכנית, האסמבילר הוסיף שורת קוד: `"push cs"`. שורת קוד זו גורמת להעתקה של רגיסטר `cs` לתוך המחסנית. כתוצאה לכך נדחף הערך `0879h`, שהוא ערכו של רגיסטר `cs` (מודגש בכחול).

- לאחר מכן נדחף למחסנית הרגיסטר `sp`, ערכו `15h` (ולא `16h`) כמו בגרסה הקודמת, משום שננוספה שורת הקוד `("push cs")`.

- עם הכניסה לפראוצדורה, ערכו של `sp` הוא `00FCh` (ולא `00FEh`, כמו בגרסה הקודמת).

Register	Value
ax	087B
bx	087B
cx	0000
dx	0000
si	0000
di	0000
bp	0000
sp	00FC
ds	087B
es	0869
ss	0882
cs	0879
ip	0000

בכל הדוגמאות בהמשך הפרק, נניח שהפראוצדורה היא מסוג `near`. ככלומר אנחנו משתמשים את הכניסה של `cs` למחסנית. למעט הבדל זה, אין הבדל בין השימוש בפראוצדרות `near` ו-`.far`.

שימוש במחסנית לשימירת מצב התוכנית

נתקבון בקוד הבא, שאמור להדפיס שלוש שורות, בכל שורה ארבעה תווי 'X'. בשלב זה נתעלם מהשורות הצבועניות – אלו פשוט קטעי קוד שעוסקים בהדפסה למסך, השורות הירוקות מדפיסות למסך X ומעבר שורה. את כל יתר הפקודות אנחנו כבר מכירים:

CODESEG

```
proc Print10X
```

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

PrintXLoop:

```
    mov dl, 'X'
```

```
    mov ah, 2h
```

```
    int 21h ; Print the value stored in dl ('X')
```

```
    loop PrintXLoop
```

```
    ret
```

```
endp Print10X
```

start:

```
    mov ax, @data
```

```
    mov ds, ax
```

```
    mov cx, 3 ; 3 lines of 'X'
```

Row:

```
    call Print10X
```

```
    mov dl, 0ah
```

```
    mov ah, 2h
```

```
    int 21h ; New line
```

```
    loop Row
```

```
exit: mov ax, 4c00h
```

```
    int 21h
```

```
END start
```

התוכנית הזו, לצערנו, לא מבצעת את מה שאנחנו רוצים. במקומות זאת, היא לעולם לא תפסיק לרווץ. מה הגורם לבעה? העתיקו את התוכנית, קמפלו אותה והריצו ב-**TD**. עיקבו אחרי הערך של הרגיסטר **CX**.



הסביר: בתחילת התוכנית **CX** מאותחל ל-3. בתוך הפְּרוֹצְדוּרָה ערכו משתנה ל-4. ביציאה מהפְּרוֹצְדוּרָה ערכו הוא 0, ואז הפקודה **loop Row** מפחיתה את ערכו באחד והופכת את ערכו ל-55 (זכור, זהו הייצוג ה-**signed** של מינוס אחד). כיוון שתנאי העצירה של **loop Row** לא מתקיים (**CX** אינו שווה לאפס), היא ממשיכה לרווץ ולקרוא שוב לפְּרוֹצְדוּרָה, ששוב מוחזרה את **CX** עם ערך 0 וכן הלאה...

לכן, אנו זקוקים למנגנון שיאפשר לנו לשמור את מצב הרגיסטרים בתוכנית לפני הכניסה לפְּרוֹצְדוּרָה, ולשזרור את הרכים של הרגיסטרים (אם נרצה בכך) לפני החזרה לתוכנית.

נראה איך בעזרת פקודות **push** ו-**pop** אפשר לשנות את הפְּרוֹצְדוּרָה **Print10X** כך שיתבצע בדיקת מה שאנחנו רוצים. התכניתה היא פשוטה: בכניסה לפְּרוֹצְדוּרָה צריך לשמור את הרגיסטרים במאסנית, ביציאה מהפְּרוֹצְדוּרָה לשלוף את הרכים מהמאסנית ולהחזיר את הרגיסטרים במצבם טרם הפְּרוֹצְדוּרָה.

CODESEG

```
proc Print10X
```

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

PrintXLoop:

```
    mov dl, 'X'
    mov ah, 2h
    int 21h        ; Print the value stored in dl ('X')
    loop PrintXLoop
```

```
    pop cx
```

```
    ret
```

```
endp Print10X
```

start:

```

mov  ax, @data
mov  ds, ax
mov  cx, 3           ; 3 lines of 'X'

```

Row:

```

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

```

הוספנו בסך הכל פקודות `push` ופקודת `pop` אחת (מודגשות בצהוב). כעת מה שיקרה, הוא שבתחילת הפראוצדורה יועתק הערך של `CX` למחסנית. רק אז ישונה ערכו של `CX` ל-4. לאחר שהלולאה `PrintXLoop` תסתיים, ערכו של `CX` יהיה שווה לאפס. פקודת `h-pop` תעתק לתוכה `CX` את הערך שנשמר במכינה לפראוצדורה, ואז התוכנית תחזור לתוכנית הראשית.

העתיקו את הקוד, הריצו אותו ב-`TD` וצפו בערכו של `CX` בשלבי הריצה השונים של התוכנית!

תרגיל 9.3: שימוש במחסנית לשמרות מצב התוכנית

נתונה התוכנית הבאה:

CODESEG

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

start:

```
    mov ax, @data
    mov ds, ax
    xor ax, ax
    xor bx, bx
    xor cx, cx
    xor dx, dx
    call ChangeRegistersValues
exit: mov ax, 4c00h
      int 21h
END start
```

הפרוצדורה `ChangeRegistersValues`, משנה את ערכי הרגיסטרים. הוסיפו שורות קוד לפראצדרה (במקומות בהם יש ???) כך שבסיום ריצת הפרוצדורה ערכי הרגיסטרים יישארו כפי שהיו טרם הקראיה לפרוצדורה.

העברת פרמטרים לפראצדרה

לא תמיד נרצה שפראצדרה תבצע את אותה הפעולה בכל פעם שאנחנו קוראים לה. ראיינו דוגמה לפראצדרה שמאפסת עשרה בתים בזיכרון. נניח שעכשווי אנחנו רוצים לאפס תשעה בתים בזיכרון – האם אנחנו צריכים עכשווי כתוב פרוצדורה חדשה? ואם אחר כך נרצה לאפס שמונה בתים בזיכרון, שוב נדרש כתוב פרוצדורה חדשה? לא הגיוני לעבוד בצדזה זו. למה שלא נבנה פרוצדורה ש יודעת לקבל את מספר הבטים שעיליה לאפס בתים נתון, וכדי לאפס מספר שונה של בתים פשוט נציג

להודיע ל프וצדורה כמה בתים עליה לאפס בזיכרונו? לנตอน זהה, שהפרוצדורה מקבלת מוקד שקורא לה (למשל התוכנית הראשית), קוראים פרמטר.

פרמטר יכול להיות כל דבר שאחנו רוצים להעביר לפרוצדורה. לדוגמה, שני מספרים שפרוצדורה צריכה לחבר ביניהם, יכולים לעבור כפרמטרים. מה לגבי תוצאה החיבור? איך מחזירים אותה לתוכנית הראשית? גם את תוצאה החיבור, ובאופן כללי כל תוצאה שהפרוצדורה צריכה להחזיר, אנחנו יכולים להעביר בעזרתו פרמטר.

יש יותר מטיטה אחת להעביר פרמטרים לפרוצדורה, השיטות הן:

- שימוש ברגיסטרים כלליים
- שימוש במשתנים שמוגדרים ב-SEG
- העברת הפרמטרים על גבי המהשנית

הטיטה הראשונה, שימוש ברגיסטרים כלליים, היא פשוטה ביותר. הפרוצדורה ZeroMemory, לדוגמה, מקבלת בתוך bx כתובת בזיכרון שמנתה עליה להתחיל את איפוס הזיכרון. באותה שיטה היינו יכולים להגיד גם את כמהת הבטים שעלייה לאפס בתור פרמטר ולהעביר אותו ברגיסטר ax.

לדוגמה:

```
proc ZeroMemory
    mov cx, ax      ; ax holds the number of bytes that should become zero
    xor al, al

```

ZeroLoop:

```
    mov [bx], al
    inc bx
    loop ZeroLoop
    ret

```

endp ZeroMemory

כעת כל מה שאחנו צריכים לעשות בתוכנית הראשית הוא לזרג ש-ax יחויק את כמהת הבטים שאחנו רוצים לאפס.

הערה: אפשר כמובן לבצע את אותה פעולה על-ידי ה הכנסת הפרמטר ישירות לתוך cx, אך כדי להקל על ההסביר נעשה בדוגמה זו שימוש ב-ax.

למרות פשוטות השימוש ברגיסטרים כלליים להעברת פרמטרים לפרוצדורה, לשיטה זו יש חסרונות. ראשית כל, מספר הרגיסטרים מוגבל. מתייחסו ax, bx, cx, dx – ואו מה? שנית, מי שכותב את התוכנית הראשית צריך להכיר את הוראות הפרוצדורה ולדעת באילו רגיסטרים היא משתמשת בתור פרמטרים.

השיטה השנייה, שימוש במשתנים שמווגדרים ב-SEG, פותרת את בעיית המספר המוגבל של רגיסטרים כליליים. מספר המשתנים שאנו יכולים להגדיר בזיכרון הוא כמעט בלתי מוגבל (באופן ייחסי לצרכים המועטים שלנו בשלב זה).
מדובר שלא נubbyר פרמטרים לפְּרוֹצְדּוֹרָה בשיטה זו?

לדוגמה:

```

proc ZeroMemory

    mov cx, [NumOfZeroBytes]      ; NumOfZeroBytes is defined in DATASEG
    xor al, al

ZeroLoop:
    mov bx, [MemoryStart]        ; MemoryStart is defined in DATASEG
    mov [bx], al
    inc [MemoryStart]
    loop ZeroLoop
    ret

endp ZeroMemory

```

בתוכנית זו החלפנו במשתנים את השימוש בשני רגיסטרים. המשתנה NumOfZeroBytes מחליף את הרגיסטר ax,bx. לפנינו כן היה פרמטר שקבע את כמות הבתים שעל התוכנית לאפס. המשתנה MemoryStart מחליף את הרגיסטר bx,ax. שלפנוי כן היה פרמטר שקבע את המיקום בזיכרון שאנו רוצים לאפס. הצלחנו לפנות שני רגיסטרים לטובת שימושות אחרות (נכון, אנחנו עדיין משתמשים ב-al וב-ab בפְּרוֹצְדּוֹרָה, אבל כרגיסטרים כליליים לחישובי עזר ולא כפרמטרים, וזה אומר שהתוכנית הראשית כבר לא צריכה להיות עסוקה בקביעת הערכיהם הנכונים לרגיסטרים לפני הקראאה לפְּרוֹצְדּוֹרָה).

העברה פרמטרים על מהשנית

גם לשיטה הקודמת ישנו חסרונו – כותב התוכנית הראשית צריך להזכיר הוראות הפרוצדורה, לדעת שצורך להגדיר ב-DATASEG שני משתנים, ולא סתם שני משתנים אלא עם שמות קבועים שאי אפשר לשנות בלי לשנות את הפרוצדורה. התוכנית הופכת לקשה לשיתוף, יותר מכך – מה אם נרצה להשתמש בשתי פרוצדורות שלקחנו מקורות שונים, ושתיهنן עושות שימוש במקרים שונים ומהם? ומה אם נרצה לכתוב פרוצדורה שקוראת עצמה (רקורסיה)?

נלמד קצת איך להעביר פרמטרים לפרוצדורה על ידי מהשנית. זהה השיטה המקובלת להעברת פרמטרים לפרוצדורות, משום שהוא פותרת את הבעיה שהצגנו בשיטות הקודמות: אין מגבלה ממשית של מקום ב מהשנית (אפשר להגדיר מהשנית יותר גודלה, ולא סביר שנציג רק יותר מ-64K, מגבלת גודל סגמנט), אין הגבלה על מספר הפרמטרים שאחננו שלוhimם לפרוצדורה, אנחנו לא מוגבלים על-ידי כמות הרегистרים, ואחרון חביב – מי שקורא לפרוצדורה שלנו לא צריך לדעת איך אנחנו קוראים למשתנים בתחום הפרוצדורה.

ישנן שתי שיטות להעברת פרמטרים אל פרוצדורה:

1. העברה לפי ערך – Pass by Value

2. העברה לפי ייחוס – Pass by Reference

Pass by Value

בשיטת זו מועבר לפרוצדורה ערך הפרמטר. כמובן, על מהשנית נוצר העתק של הפרמטר. לפרוצדורה אין גישה לכתובת בזיכרון שמכיל את המשתנה המקורי, ולכן אין היא יכולה לשנות את ערכו – רק את ערכו של העתק. ביציאה מהפרוצדורה, ערכו של הפרמטר שנשלח לפרוצדורה ישאר ללא שינוי.

נמחיש זאת באמצעות דוגמה. נדמיין פרוצדורה בשם SimpleAdd, שמקבלת פרמטר אחד ומוסיפה לערכו 2. נניח שהפרוצדורה הזו היא חלק מספירה של פרוצדורות שימושו כתוב, קימפל ומסר לנו. הנתון זהה חשוב, כדי להציג של אסטブル שקיימפל את SimpleAdd לא יהיה מושג מה הכתובות של המשתנים שאחננו מגדרים ב-DATASEG של התוכנית שלנו. יכול להיות שהפרוצדורה SimpleAdd נכתבה בכלל לפני שאחננו כתבנו את התוכנית שלנו.

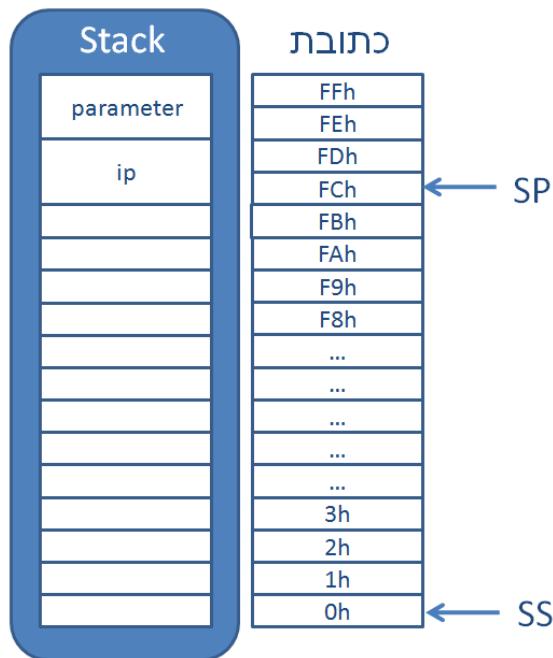
בתוכנית שלנו מוגדר משתנה, נקרא לו parameter. כתוב אנחנו רוצים SimpleAdd-Sh- parameter.parameter. כתוב בתוכנו את הערך שנמצא בתחום parameter.

כדי להעביר את ערכו של הפקט לפְּרוֹצְדוּרָה, התוכנית הראשית צריכה לדוחף אותו למחשנית ואז לקרוא לפְּרוֹצְדוּרָה:

```
push [parameter]
```

```
call SimpleAdd
```

מצב המחשנית עם הכניסה לפְּרוֹצְדוּרָה:



בתוך הפְּרוֹצְדוּרָה המשתנה `parameter` אינו מוכר. כל מה שהפְּרוֹצְדוּרָה מכירה זה את הערך שלו, שנמצא על המחשנית. הפְּרוֹצְדוּרָה לא יודעת ש-`parameter` נמצא בכתובת כלשהו-ב-DATASEG. מבחינה זו, היא מכירה רק את ההעתק שלו שנמצא בכתובת כלשהו על המחשנית. לאחר שהפְּרוֹצְדוּרָה תוסיף 2 לערכו של ההעתק, ערכו של `parameter` המקורי לא ישנה כלל.

למרות שפְּרוֹצְדוּרָה שמקבלת ערכים בשיטת Pass by Value לא יכולה לשנות אותם, לעיתים שיטה זו מתאימה עבורינו. בדוגמה הבאה נראה איך פְּרוֹצְדוּרָה משתמש בפקטים שהועברו אליה בשיטת Pass by Value. נניח שיש פְּרוֹצְדוּרָה בשם `SimpleProc`, שמקבלת שלושה פְּרַמְטִירִים: `i`, `j` ו-`k` ומהשבת בתוך `ax` את $k - j + i$. קוד אסmbלי מתאים לקרוא לפְּרוֹצְדוּרָה (בשיטת Pass by Value) יכול להיות:

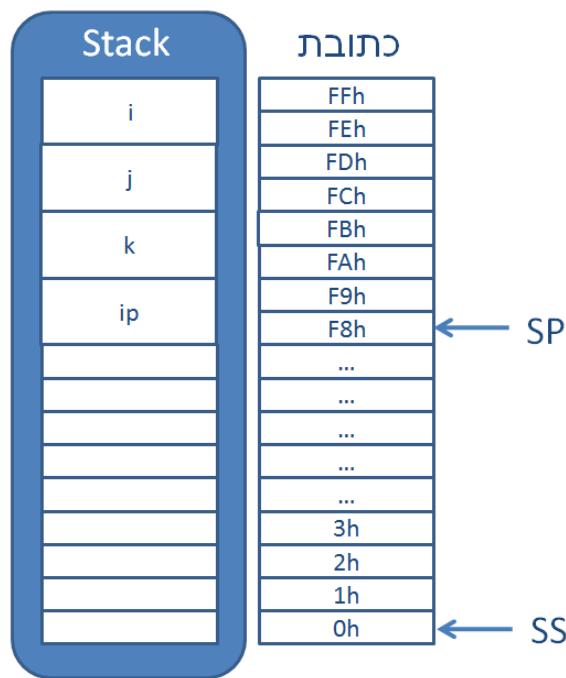
```
push [i]
```

```
push [j]
```

```
push [k]
```

```
call SimpleProc
```

עם הכניסה לפְּרוֹצְדוּרָה, המהסנית תראה כך (בහנחה שלא הכנסנו למחסנית מידע נוסף):



מצב המהסנית לאחר דחיפה *i, j, k*

בתוך הפְּרוֹצְדוּרָה SimpleProc אנחנו יכולים להוציא את הפרמטרים שהועברו אליה באמצעות פקודה pop. יש רק עניין אחד – ראש המהסנית מצביע על כתובת החזרה מהפְּרוֹצְדוּרָה, קו', שנדרף למחסנית עם פקודה ה-call. ככלمر הדוק הראשוון שנעשה יוציא את כתובת החזרה מהמחסנית.

אנחנו נראה שיטה פשוטה לטיפול בכתובות החזרה. זו **איינית שיטה שימושים בה בפועל** ואנחנו מדגימים אותה בקצרה, רק כדי שנוכל להבין איך המהסנית עובדת. בהמשך נלמד שיטה אחרת, שהיא השיטה המקובלת לפניה **לפרמטרים שהועברו על המהסנית**.



נשמר את הכתובות בצד ונדרוף אותן למחסנית לאחר ביצוע הדסקן לכל הפרמטרים שהעברנו לפְּרוֹצְדוּרָה.

```

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

```

ret

endp SimpleProc

```

הו משנתנה כלשהו שהגדנו ב-DATASEG. אם נרצה, אפשר להחליף אותו ברегистר. בשיטה זו הצלחנו לקרוא את הפרמטרים שהועברו במחסנית אל הפראוצדורה ועדיין לחזור אל המיקום הנוכחי בתוכנית הראשית. כאמור – בקרוב נחליף את השיטה זו בשיטה נוספת יותר.

תרגיל 9.4 Pass by Value



א. צרו פראוצדורה שמקבלת פרמטר אחד בשיטת pass by value ומדפיסה למסך 'X' אם לפि הנתון בפרמטר. דגשים: יש לבדוק קודם הערך הפרמטר היובי! בסיום הריצה יש לשחזר את ערכי הרגיסטרים שנעשה בהם שימוש. עזרה – הדפסת תו למסך:

```

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

```

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

ג. צרו תוכנית שמוגדרים בה ארבעה מספרים קבועים בתחילת התוכנית, ומשתנים בשם max ו-chin. צרו פראוצדורה שמקבלת כפרמטרים pass by value את ארבעת המספרים הקבועים ומכניסה למשנה max את הערך המקסימלי מביניהם ולמשנה chin את הערך המינימלי מביניהם.

Pass by Reference

בשיטה זו מועברת לפראוצדורה הכתובת של הערך בזיכרון. כלומר, על המחסנית לא נוצר העתק של הערך אלא רק הכתובת שלו בזיכרון מועתקת למחסנית. הפראוצדורה לא יודעת מה ערך של הערך אליה אבל יש לה גישה לכתובת בזיכרון שמכילה את המשתנה, ולכן היא יכולה לשנות את ערכו – היא פשוט צריכה לגשת למיקום בזיכרון. במקרה מהפראוצדורה, ערכו של הערך שנשלח לפראוצדורה עשוי להשתנות.

נמחיש בעזרת דוגמה: שימו לב לפוקודה חדשה שמו פיעעה בה. אנחנו הולכים לקרוא לפראוצדורה SimpleAdd, אבל הפעם בשיטת Pass by Reference:

```

push offset parameter ; Copy the OFFSET of "parameter" into the stack
call SimpleAdd

```

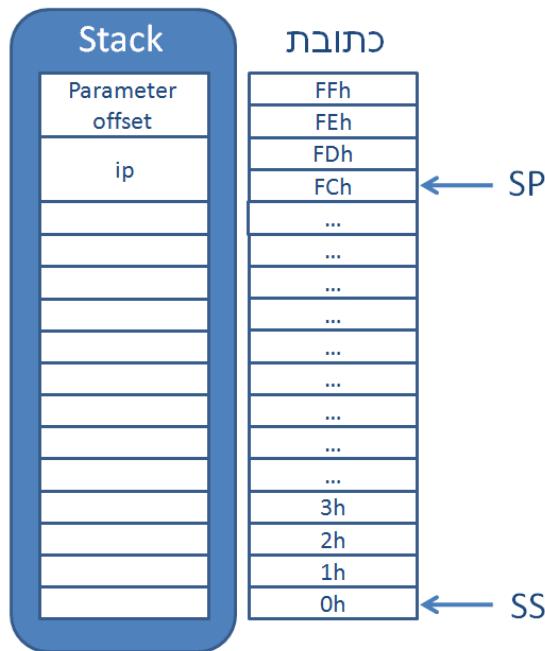
אם היה לנו יותר מסegment נתונים אחד – מה שסבירן איןנו המצב – היינו צריכים להכנס למחסנית גם את כתובתו של סegment הנתונים ש-parameter מוגדר בתוכו. כיון שיש לנו רק סegment נתונים אחד, אין צורך בפקודה הבאה והיא

לידע כללי בלבד:



```
push seg parameter ; Copy the SEGMENT of "parameter" into the stack
```

בחזורה למחסנית שלנו. מצב המחסנית עם הכניסה לפראוצדורה:



כעת הפראוצדורה יכולה לשנות את הערך של `:parameter`

```
proc SimpleAdd
```

`;Takes as input the address of a parameter, adds 2 to the parameter`

```

pop ReturnAddress ; Save the return address
pop bx           ; bx holds the offset of "parameter"
pop es           ; es holds the segment of "parameter"
add [byte ptr es:bx], 2 ; This actually changes the value of "parameter"
push ReturnAddress
ret

```

```
endp SimpleAdd
```

תרגיל 9.5 Pass by Reference



א. צרו פראוצדורה שמקבלת פרמטר בשיטת `pass by reference` ומעליה את ערכו ב-1.

ב. צרו פראוצדורה שמקבלת ארבעה פרמטרים בשיטת `pass by reference` ומニアפת אותם.

ג. צרו פְּרוֹצְׂדּוֹרָה שמקבלת שני פרמטרים בשיטת pass by reference, ומהליפה ביניהם (לדוגמא – לפני .var1=5, var2=4. אחרי הפְּרוֹצְׂדּוֹרָה var1=4, var2=5)

שימוש ברגיסטר BP

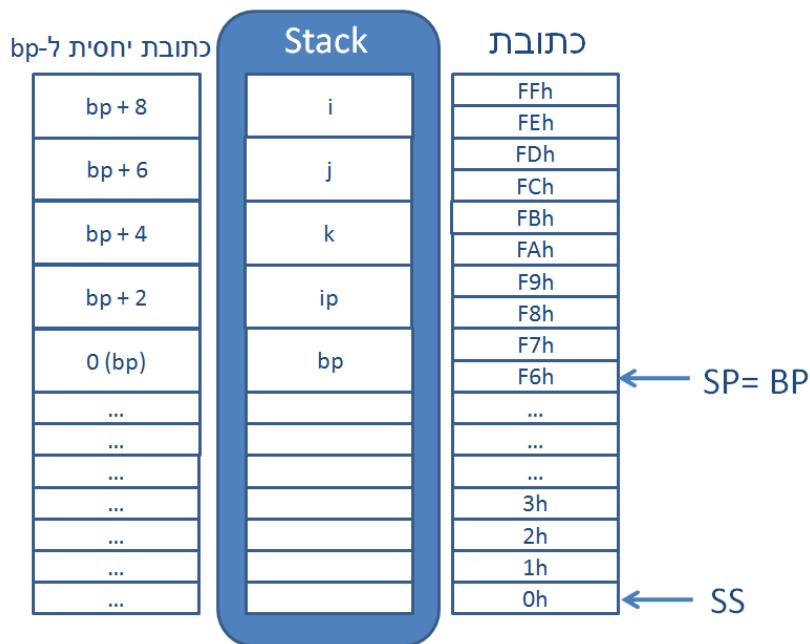
כשהצנו את השימוש ב-[ReturnAddress] pop צינו שזו שיטה שאינה מקובלת ולא נעשה בה שימוש בפועל. הסיבה היא, שככל פעם צריך לדאוג להוציא, לשמר ולהחזיר למחסנית את הרגיסטר ip.

הרגיסטר bp, קיצור של Base Pointer, מסייע לנו לגשת לפרמטרים שהתוכנית הראשית הכניסה למחסנית מבלי להתעסק עם הרגיסטר ip. שמו לב לשורות הקוד שאנחנו מוסיפים לפְּרוֹצְׂדּוֹרָה (מודgeshot):

```
proc SimpleProc
    push bp
    mov bp, sp
    ...
    ;Code of the stuff the procedure does
    pop bp
    ret 6
endp SimpleProc
```

נסביר את הפקודות חדשות בזו אחר זו.

שתי הפקודות הראשונות מבצעות שמירה של bp למחסנית והעתיקת sp לתוך bp. בשביל מה זה טוב?
יצרנופה בעצם מגנון, ששמור את ערכו ההתחלתי של sp. מעכשיו, גם אם sp ישנה כתוצאה מדחיפה או הוצאה של ערכיהם מהמחסנית, bp נשאר קבוע ותמיד מצביע לאותו מקום. מגנון זה פותח לנו אפשרות לקרוא לכל ערך במחסנית לפי הכתובת היחסית שלו לbp.



כעת המשתנה *i*, שדחפנו אותו ראשון למחסנית, נמצא בכתובת שהיא 8 בתים מעל bp. המשתנים *j* ו-*k* מצויים בכתובות שהן 6 ו-4 בתים, בהתאם, מעל bp. מה שחשוב הוא שהמרחקים מ-bp נשארים קבועים למשך כל חי הפראצדרה.

נראה איך משתמשים במסקנה האחרונה. נדגים זאת שוב על הפראצדרה SimpleProc, שבוצעת את הפעלה *k=j+i=ax*. הקוד הבא מבצע זאת:

```

proc SimpleProc
    push bp
    mov bp, sp
    ; Compute I+J-K
    xor ax, ax
    add ax, [bp+8]    ; [bp+8] = I
    add ax, [bp+6]    ; [bp+6] = J
    sub ax, [bp+4]    ; [bp+4] = K
    pop bp
    ret 6
endp SimpleProc

```

נשתמש עכשו בפקודה של אסמבלי, שתשורג את הקוד שלנו. הוראת *equ* אומרת לקומpileר שבלל פעם שהוא נתקל בצירוף התווים שהוגדר, עליו להחליף אותו בצירוף תווים אחר שהוגדר. לדוגמה:

```

iParm equ [bp+8]
jParm equ [bp+6]
kParm equ [bp+4]

```

עכשו הפעולה שלנו פשוטה לא רק פשוטה, אלא גם פשוטה לקריאה:

```
proc SimpleProc
    push bp
    mov bp, sp
    ; Compute I+J-K
    xor ax, ax
    add ax, iParm
    add ax, jParm
    sub ax, kParm
    pop bp
    ret 6
endp SimpleProc
```

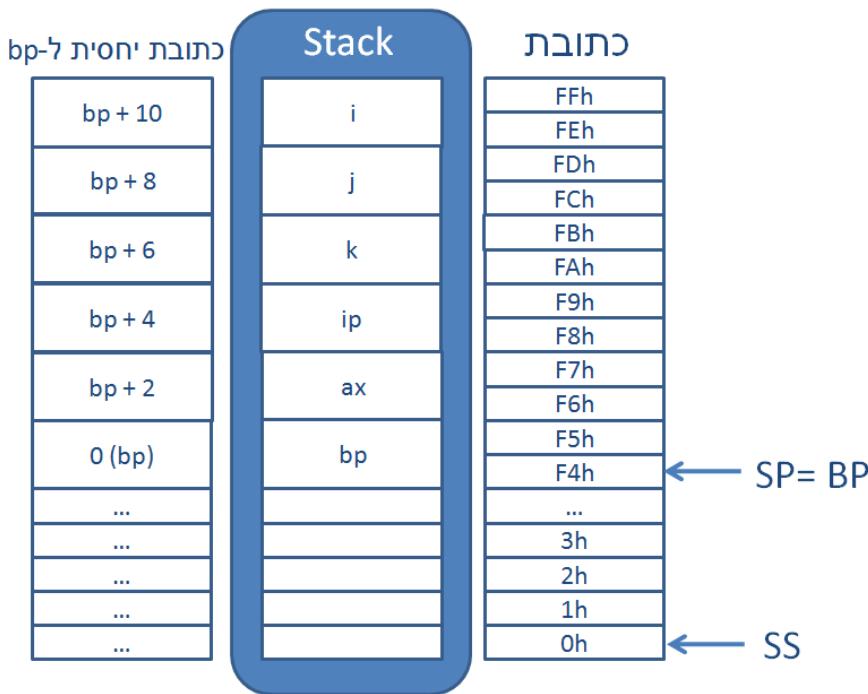
טיפ: השימוש ב-`equ` נוח במיוחד כאשר עובדים בשיטת `pass by value`. כך, לכל משתנה שדחפנו למחסנית יש שם מוגדר בפְּרוֹצְׂדּוֹרָה. לו היינו עובדים בשיטת `pass by reference`, היינו מרויחים פחות שימוש ב-`equ`, משום שבשיטת `pass by reference` לא פונם יישירות אל הערך שנשמר במחסנית, אלא מעתיקים אותו לתוך רגיסטר שמשמש לפניה לזכרון (לדוגמא – מעבירים מערך ומעתיקים את כתובות תחילת המערך ל-`ax`. במקרה זה אין תועלת לחת שם מיוחד לתחילת המערך משומש שאנחנו תמיד ניגשים אל המערך דרך `ax`). 

שימוש לב: כדי שהמבנה שייצרנו יယוב, העתקה של `bp` למחסנית ופקודת `mov bp, sp` חייבת להיות הפקודות הראשונות בפְּרוֹצְׂדּוֹרָה. 

הסיבה היא, שאם בטעות נבצע פקודת `push` או `pop` לא מאוזנת (דחיפה והוצאה של כמות לא שווה של בתים) לפני פקודות אלו, כל המבנה שייצרנו, של יצרת משתנים במרקדים מ-`bp`, יהיה משתמש. הנה דוגמה לשימוש לא נכון: `bp`:

```
proc WrongBP
    push ax
    push bp
    mov bp, sp
    ...

```

*המהשנית של ret*

בעקבות הקריאה של ax לפני bp, מבנה המרחקים מ-bp שיצרנו אינו נכון. עכשו bp+8 מצביע על j במקום על ax וכולא.

נסים בהסביר על פקודת ret-6 שבסוף הפראוצדרה.

ראינו שפקודת ret מבצעת את התהליך הפוך מאשר פקודת call: פקודת ret מוציאה את כתובות החזרה מהמהשנית, מעלה את ערכו של sp ב-2, ו קופצת אל כתובות החזרה. הפקודות הבאות שקולות לפקודת ret (למעט העובדה ש-ret אינה משנה את ax):

pop bx ; pop increments sp by 2

jmp bx

כשאנו מוסיפים מספר ליד פקודת ה-ret, לאחר ביצוע ה-pop, נוסף ל-sp הערך שרשמנו ליד ה-ret. הפקודות השקולות ל-ret-6 הן:

pop bx ; pop increments sp by 2

add sp, 6 ; sp is incremented by a total of 8

jmp bx

השימוש בספרה ליד פקודת ה-ret נועד לשחרר מקום במחשנית שתאפשר פקודות push לפני הכניסה לפראוצדרה. בדוגמה שלנו עשינו push לשולש המשתנים בגודל 2 בתים כל אחד, כלומר הכנסנו בסך הכל 6 בתים למחשנית. הפקודה ret-6 מזירה את sp למספר המקורי טרם הכניסה לפראוצדרה ובכך "משחררת" את הזיכרון שבחשנית. באותה מידה היינו

יכולים לבצע `ret` וגיל בלי ספרה לידי, ולאחר כך שלוש פעמים `pop`, אך השימוש ב-`6 ret` יותר אלגנטי, כיון שהוא לוקח פחות מקום ומבצע את הפעולה המבוקשת באמצעות פקודה אחת בלבד.

סיכום ביניים של יתרונות השימוש ב-`bp`:

1. בתחילת הפרוצדורה לא צריך לעשות `pop` לכנתובת החזרה ולשמור אותה.
2. לא צריך לעשות `pop` לכל הערכים שדוחפנו מהשנית. פשוט ניגשים ישר אל הכתובות שלהם מהשנית בעזרת `bp`.
3. אפשר ליצא כל תא בזיכרון המהשנית עלי ידי שם קרייא ובעל משמעות.

יתרונו נוסף של `bp`, שנראה אותו מיד, הוא ש-`bp` עוזר לנו לגשת למשתנים מקומיים.

תרגיל 9.6: פרוצדורה ושימוש ב-`bp`



- A. כתבו פרוצדורה שמקבלת שני משתנים, `pass by reference`, בשיטת `var1=5, var2=4`. מחליפה ביניהם (לדוגמה – לפני הפרוצדורה `var1=5, var2=4` לאחר הפרוצדורה `var1=4, var2=5`). בתוך הפרוצדורה יש להתייחס למשתנים רק בעזרת `bp`.
- B. כתבו פרוצדורה שמקבלת שלושה משתנים: `max, var1, var2`. המשתנה `max` נשלח בשיטת `reference` ויתר המשתנים בשיטת `value`. ביציאה מהפרוצדורה, `max` יוכל את הערך הגבוה `var1, var2` מבין.

שימוש במחסנית להגדרת משתנים מקומיים בפראוצדורה (הרחבה)



משתנה מקומי הוא משתנה שהפראוצדורה מדירה, והוא אינו מוכר מחוץ לפראוצדורה. משתנים אלו הם בשימוש של הפראוצדורה בלבד, ואין להם שימוש מחוץ לפראוצדורה. במקרה של משתנה מקומי, הפראוצדורה מקצת עboro המשנה זיכרון על המחסנית וגם דואגת לשחרר את הזיכרון לפני החזרה לתוכנית שקרה לה.

איך זה מתבצע בפועל? כמו שראינו, כל הקצת זיכרון על המחסנית מוריידה את ערכו של sp. גם כאן, כדי להקצתו משתנים מקומיים הפראוצדורה פשוט מוריידה את ערכו של sp. אם, לדוגמה, הפראוצדורה רוצה להקצתו למשתנים מקומיים 6 בתים, הקצתה תבוצע באמצעות הפקודה:

```
sub    sp, 6
```

כמובן שלפנוי הייצה מהפראוצדורה צריכה להתבצע הפעולה ההפוכה, כדי לשחרר את הזיכרון (וכדי להביא את sp לערך הנכון – שיצביה על המקום במחסנית בו שמור לו):

```
add    sp, 6
```

נראה דוגמה לפראוצדורה שעושה שימוש במשתנים מקומיים.

נניח פראוצדורה שמקבלת שני פרמטרים – y,x. הפראוצדורה מדירה שני משתנים מקומיים AddXY ו-SubXY ומכניסה לתוכם את הסכום וההפרש של x ו-y, בהתאם.

```

varX    equ    [bp+6]
varY    equ    [bp+4]
AddXY   equ    [bp-2]
SubXY   equ    [bp-4]

proc    XY
        push   bp
        mov    bp, sp
        sub    sp, 4      ; Allocate 4 bytes for local variables
        push   ax          ; Save ax value before we change it
        mov    ax, varX
        add    ax, varY

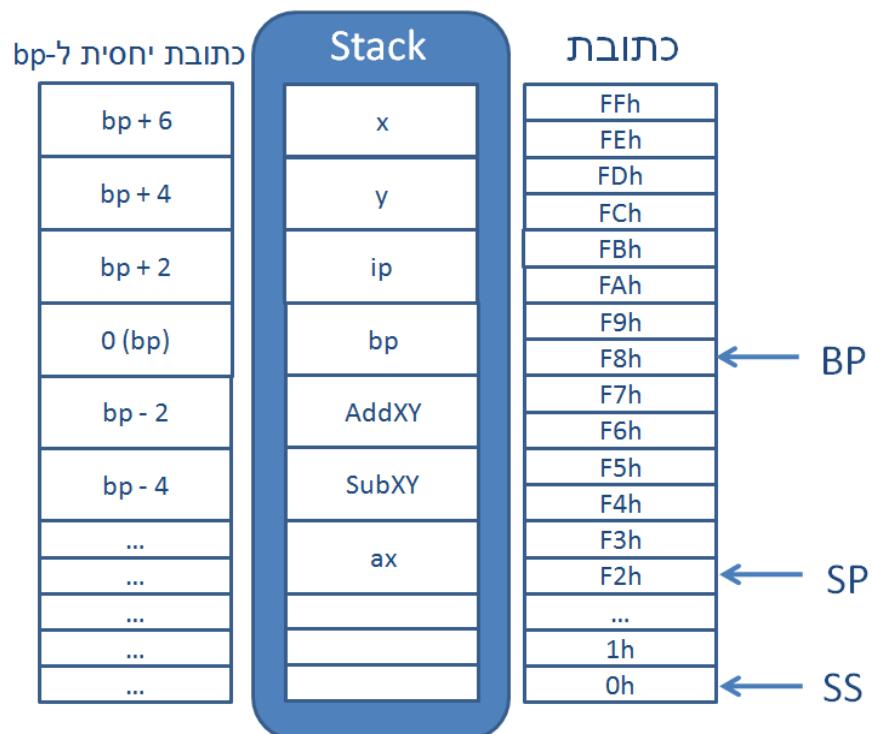
```

```

mov    AddXY, ax
mov    ax, varX
sub    ax, vary
mov    SubXY, ax
pop    ax          ; Restore ax original value
add    sp, 4       ; De-allocate local variables
pop    bp
ret    4
endp  XY

```

מצב המחשנית לאחר ביצוע הפקודה `push ax` :



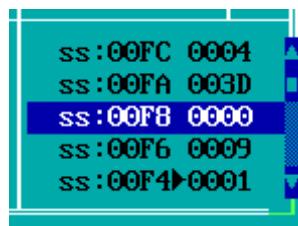
אם לדוגמה אנחנו קוראים ל- ZX בתוכנית הראשית באמצעות שורות הקוד הבאות:

```
push 5
```

```
push 4
```

```
call ZX
```

כך נראה המהסנית, לאחר ביצוע החישובים ושמירת הערכים בתוך ZX AddXY ו- ZX Sub. השורה המודגשת בכהל היא השורה עליה מצביע bp:



דחיפנו למחסנית את הערך 5 ואת הערך 4. עשינו הקזאה של מקום לשמר את הסכום ואת הפרש של שני הערכים. ואננו כפי שאפשר לראות, המהסנית מכילה הנה את הסכום (0009) והנה את הפרש (0001), במקומות זיכרונו בו הגדרנו את המשתנים המקומיים ZX AddXY ו- ZX Sub, בהתאם.

שיםו לב לדמיון שבין צילום הזיכרון במחשב לבין האירור שמתאר את מצב המהסנית: במקום ss:00F6, bp, שהוא [bp-2], נמצא הסכום של שני הפרמטרים. במקום ss:00F4, bp, שהוא [bp-4] נמצא החיסור של שני הפרמטרים.

תרגיל 9.7: ייצרת משתנים מקומיים במחסנית



כתבו תוכנית ראשית, שדוחפת למחסנית שלושה ערכים כלשהם וקוראת לפראצדרה בשם $ZXYZ$. הפראצדרה כוללת שלושה משתנים מקומיים – LocalX, LocalY, LocalZ. הפראצדרה תעתק כל ערך לתוך משתנה מקומי אחר. הריצו את התוכנית ובדקו במחסנית שההעתקה בוצעה כנדרש.

שימוש במחסנית להעברת מערך לפראוצדורה

לעתים נרצה שפראוצדורה תקבל אוסף של איברים, הדוגמה הקלאסית היא מערך. השיטה הבסיסית ביותר להעביר מערך לפראוצדורה היא פשוט לדוחף למחסנית את האיברים של המערך אחד אחד, בשיטת `pass by value`. הסרונות השיטה זו:

- מיגעת, אם המערך ארוך.
 - תופסת הרבה מקום במחסנית.
 - הפראוצדורה לא יכולה לשנות את הערכים של המערך המקורי.
- בשיטת `pass by reference`, לעומת זאת, מעבירים לפראוצדורה רק את:
- כתובת של האיבר הראשון במערך
 - מספר האיברים במערך

כעת נראה דוגמה. נגידו מערך בתוך DATASEG:

DATASEG

```
num_elements      equ   15
Array             db    num_elements      dup (?)
```

בתוך CODESEG, לפני הקראיה לפראוצדורה, נדחוף למחסנית את כתובת האיבר הראשון במערך ומספר האיברים במערך:

```
push  num_elements
push  offset Array
call  SomeProcedure
```

בתוך הפראוצדורה אפשר להתייחס לכל אלמנט במערך על-ידי כתובת הבסיס שלו, שהוועתקה אל המחסנית, בתוספת היחס של האלמנט מתחילה המערך.

תרגיל 9.8: העברת מערכם לפראוצדורות



שימו לב - בתרגילים הבאים, אין להשתמש בתוך הפראוצדורות במשתנים שהוגדרו בסגמנט הנתונים. את כל המידע הדרוש לפראוצדורה יש להעביר על גבי המחסנית.

א. צרו פראוצדורה שמקבלת מערך ואת המשתנה `sum` ומכניסה לתוך `sum` את סכום האיברים במערך. לדוגמה, עבור המערך 2,2,3,4,5 התוצאה תהיה `sum=16`.

ב. כתבו פרכזדרה SortArray שמקבלת מצביע לערך ומספר איברים בערך, וממיינת את המערך מהאיבר הקטן לגדול. לדוגמה עבור המערך 3,6,5,2,1 הfraczdraha תגרום למערך להכיל את הערכים: 1,2,3,5,6. הדרכה לכתיבת הfraczdraha:

- כתבו פרכזדרה עוז FindMin שמקבלת מצביע לערך, מוצאת את האיבר הקטן ביותר ומהזירה את האינדקס שלו.

- כתבו פרכזדרה עוז Swap שמקבלת שני פרמטרים באמצעות שיטת pass by reference ומחליפה את הערכים שלהם.

- הfraczdraha תרצה בולולה על המערך ותקרה ל-`FindMin`. לאחר מכן תקרה ל-`Swap` עם שני פרמטרים: האינדקס שהזירה `FindMin` והאינדקס הראשון בערך.

- לאחר שהfraczdraha העבירה את האיבר הקטן ביותר לאינדקס הראשון, היא תקרה ל-`FindMin` כאשר המצביע לערך הוא על האינדקס השני בערך. לאחר מכן, SortArray ל-`Swap` עם שני פרמטרים: האינדקס שהזירה `FindMin` והאינדקס השני בערך.

- הfraczdraha תמשיך בשיטה זו עד לסיום מיוון המערך.

ג. כתבו פרכזדרה שנקראה Sort2Arrays, שמקבלת מצביעים לשני מערכים ומספר איברים בכל מערך, ומצביע נוסף לערך יעד sorted אליו יש להכניס את תוכנת המיוון של שני המערכים, כאשר ערכים כפולים מסוננים החוצה. לדוגמה:

`Array1 = 4,9,5,3,2`

`Array2 = 3,6,4,1`

לאחר הרצת הfraczdraha:

`Sorted = 1,2,3,4,5,6,9`

הדרכה לכתיבת הfraczdraha:

- כתבו פרכזדרה בשם Merge שמקבלת מצביעים לשני מערכים ומעתיקה אותם לערך אחד, ללא מיוון או סינון.

- קיראו ל-`Merge` עם המערך שיצרה SortArray.

- כתבו פרכזדרה בשם Filter שעוברת על המערך הממוין ומאפסת את כל הערכים שモפייעים יותר מפעם אחת.

גִּלְשַׁת מַהֲסִנִּית - Stack Overflow (הרחבה)



הינו מונה חשוב מתחום אבטחת המידע. באמצעות ביצוע פעולות שונות ניתן לשנות את אופן פעולה של תוכנית ולגרום לה להריץ קוד שונה مما שתוכנן (ולכן בהיבט של אבטחת מידע מדובר בעיה קשה). איןנו מעודדים כל שינוי קוד מקור של תוכנות והדבר אף אינו חוקי כמובן. מאידך מובה לפניכם הרקע התיאורטי לנושא, מהסיבות הבאות:

1. מודעות לביעות אבטחה עשויה תעוזד אתכם לתוכנת קוד ברמה גבוהה יותר, שאינו חשוף לביעות אבטחה קלאסיות.
2. הבנה של הנושא תדרושים מכל חורה והתעמקות בחומר הלימוד בפרק זה, שהינו אחד הפרקים החשובים ביותר להבנת אופן הפעולה של תוכנות מחשב.

ראשית נגידיר את המונח **Buffer Overflow**. מונה זה מתיחס למצב בו לתוכן מערך בגודל מסוים, מה שנקרא "באפר", מנסים לרשום יותר מידע מאשר המערך יכול להכיל. דמיינו קרטון ביצים, שמיועד להכיל 12 ביצים. קרטון הביצים הוא הבאפר שלנו, ו-12 הוא כמות המקומות הקיימים באפר. מה יקרה אם ננסה להכניס לkarton 13 ביצים? נקבל **Buffer Overflow**. באותו אופן, אילו נקצתה מערך בגודל – נניה – של 100h, וננסה להעתיק לתוכנו 257 בתים, הבית ה-257 יירוג מיקום הזיכרון ונקבל **Buffer Overflow**.



Buffer Overflow הוא השם הכללי ביחס לכל גלישת זיכרון מחוץ לבאפר. אם הבאפר שלוש הוגדר על המהסנית, הגלישה נקראת **Stack Overflow**.



הבה נראה איך עשויה להתרחש **Stack Overflow**.



התבוננו בתוכנית הבאה. עיברו על שורות הקוד ובררו לעצמכם, מה התוכנית עושה?

```
; -----
; Program StackOF – demonstration of stack overflow
; Author: Barak Gonen 2015
; -----
IDEAL
MODEL small
STACK 100h
DATASEG
msg1 db 'Please enter your name, press enter to finish',13,10,'$'
```

```
msg2 db 13,10,'Program finished$'
msg3 db 13,10,'Here be dragons$'
```

CODESEG

proc GetName

```
; Get user input and store it on the stack
push bp
mov bp, sp
sub sp, 10 ; Allocate a buffer of 10 bytes on the stack
mov di, sp
mov ah, 1
xor bx, bx
```

get_char:

```
int 21h
cmp al, 13 ; Is it the 'enter' key?
je quit_proc
mov [ss:di+bx], al ; Copy user input to the buffer on the stack
inc bx
jmp get_char
```

quit_proc:

```
add sp, 10 ; De-allocate buffer
pop bp
ret
```

endp GetName

start:

```
mov ax, @data
mov ds, ax
mov ah, 9
mov dx, offset msg1
int 21h
```

```

call    GetName
mov     ah, 9
mov     dx, offset msg2
int     21h

exit:
mov     ax, 4c00h
int     21h

; This code should not be reached at all, as the program should have
; already exited

nops   db 20E8h dup (90h) ; Fill a part of the memory with NOP (90h)-
; NOP - a command which does nothing (No Operation)

mov     ah, 9
mov     dx, offset msg3
int     21h
jmp     exit

END start

```

הסבר אודות התוכנית: התוכנית מדפסה למסך בקשה לקלוט את שם המשתמש. לאחר שהמשתמש סיים עלייו להקשין `.enter` או התוכנית מדפסה למסך הודעה `Program finished`. כך:

```

Please enter your name, press enter to finish
Jon Snow

Program finished

```

את קליטת שם המשתמש מבצעת פְּרָוֶצְׁדָּרָה, שומרת את הקלט על המחסנית, בابر בגודל 10 בתים.

שימוש לב שלמרות שלמהשנית אפשר לעשות `push` רק בנסיבות של שני בתים, בגישה ישירה לזכורן אפשר להעתיק אל המחסנית גם ערכים בגודל בית יחיד.



באופן לא שגרתי, בסיום קטע הקוד שמסים את ריצת התוכנית, יש קטע קוד נוספת, שמדפסה הודעה שונה למסך. זה קטע קוד מוזר, מכיוון שהוא לא אמור להיות מוזר. אין בתוכנית שום פעולה שמקפיצה את זו כך שייגיע אל קטע הקוד הזה.

אם כך, איך אפשר להסביר את ההרצתה הבאה?



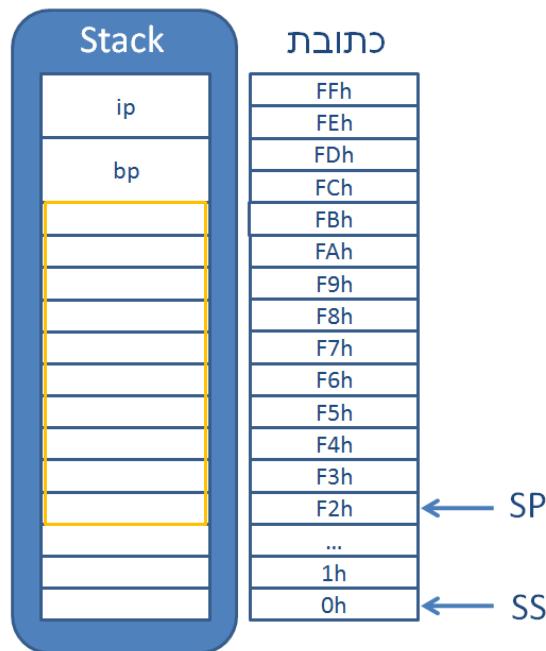
```

Please enter your name, press enter to finish
Jon Snow !  

Here be dragons

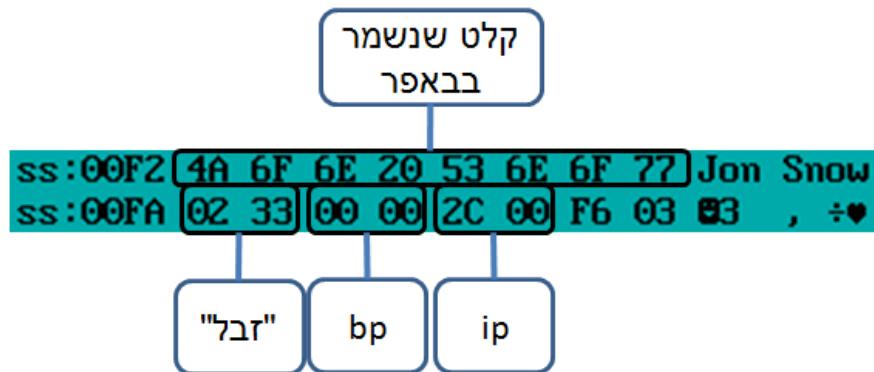
```

לאחר שנוכחנו שיש כאן התחנות מוזרה, נסביר מה התרחש כאן שלב אחריו שלב. הפְּרֹזְצָדָרָה `GetName` באפר בגודל 10 בתים על המהסנית. בתוך הפְּרֹזְצָדָרָה, המהסנית נראה כך:



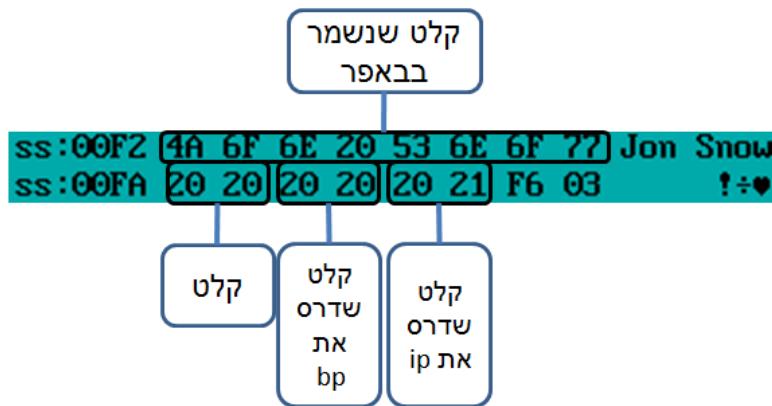
המהסנית בתחום `GetName`. באפר בגודל 10 בתים.

בהרצתה הראשונה משתמש הzin את שמו, `Snow`, סך הכל 8 תווים (כולל הרווח בין המילים). ערכי ה-ASCII של התווים נכנסו לmahsinit בזו אחר זו והמהסנית נראה如此:



נבחן את מצב המהסנית. הקלט הראשון – האות J (4Ah: ASCII 4) – נכנסה לתחילה באפר במקום ss:00F2. יתר האותיות שנקלטו הועתקו לזכור עד מקום ss:00F9, כולל. כיוון שהבאפר הוגדר בגודל 10 בתים אף הוועתקו אליו רק 8 בתים, נשארו בו שני בתים של "זבל", ערכיהם לא מאותחלים, שהיו בזיכרון טרם ריצת התוכנית. מעל ה"זבל" נמצא bp ומעליו ip, כתובות החזרה.

כעת נניח שהמשתמש שלנו ממשיך להזין תווים. לאחר שהוא כתב את המחרוזת 'Jon Snow', המשתמש מזין 5 תווים נוספים ואז סימן קראיה. כתובות מהעתקה של 14 בתים לתוך באפר שגודלו 10 בתים, נוצר מצב של Stack Overflow. בשלב זה המהסנית נראה如此:



ארבעת הבתים האחרונים שניכנו למחסנית, דרכו את `kp` ואת כתובת החורה, הערכיהם המקוריים הוחלפו בערכי ה-ASCII של התווים שהוקלו: ערך ה-ASCII של تو הרווח הוא `20h` ואילו של סימן קריאה-`21h`. פגולה זו לא גרמה למעבד לשגיאה, כיוון שלא התבכשה פעולה לא חוקית, ההתקאה לזכרו בוצעה באופן תקין מבחינה שפת אSEMBL. עם זאת, בפעולה זו שוכתבה כתובת החורה של הפרוזדורה. חישבו - מה יהיו ההשלכות של מצב זה?

התו הבא שהמשתמש מזין הוא `enter` ובשלב זה הפרוזדורה מסתiyaת ומגיעה לקוד היציאה. בשלב הראשון נמחק הבאפר שהוקצתה בזיכרון. בשלב השני מבוצע `pop kp`. כמובן, שהערך שנכנס לתוכו `kp` אינו הערך המקורי שנשמר במחסנית אלא הערך החדש-`2020h`. לבסוף מבוצעת פקודת `ret`. הערך `2120h` מועתק לתוכו `ki` ובמוצעת קפיצה לכתובת `2120h` (מדוע לא `?2021h`? הזכור – הזכור פועל בשיטת `little endian`).

הקפיצה לכתובת `2120h` מביאה את `ki` היישר אל קטע הקוד שمدפיס את הודעת הסיום 'Here be dragons'. המשפטbekoh מופיעות עתיקות והמשמעות הרווחת שלו היא "אזר לא מוכך, שאין לדעת מה נמצא בו. צפו לסקנות...".



מפה מ-1265. בחתימת המפה,
מחוון לטרייטוריה המוכרת לאדם, דרקונים.

לסייעם בערך זה, ראיינו שבוזרת שימוש ב-Stack Overflow ניתן להקפיין את התוכנית להוראות שהמתקנה לא תכнן שיתבצעו. אולם במקרה הזה הפקודות שאלייהן קופצים נמצאות בתוכנית, אולם באמצעות טכניקות שונות (שאין בהיקף של ספר זה) ניתן לשלוט פקודות חדשות בתוך הקוד.

תרגיל 9.8: מניעת אפשרות של Stack Overflow



שפרו את התוכנית, כך שלא ניתן יהיה לבצע בה Stack Overflow. הדרכה: העתיקו את קוד התוכנית והוסיפו שורות קוד מתאימות. עלייכם למנוע לא רק את האפשרות שתודפס המחרוזת 'Here be dragons' אלא גם כל אפשרות ל-Overflow אחר, לדוגמה כזו שעלול לגרום לקריסת התוכנית.

Calling Conventions (הרחבה)



רוב הקוד בעולם נכתב בשפות עיליות, לא בשפת אסמבלי. לעיתים חלק מהקוד נכתב בשפה עילית וחלקו נכתב באסמבלי. הקישור בין הקוד שבספה עילית לבין הקוד המקורי מתבצע על ידי הLINKER – אותו הכרנו בפרק אודות סביבת העבודה – אשר מקשר בין מספר קבצים ליצירת קובץ הרצה היחיד בשפת מכונה. כתוצאה לכך, יש צורך בתאימות בין הקוד בשפה העילית לקוד בשפת האסמבלי. מהי תאימות של קוד? נתקלנו כבר ב מקרה שבו נדרשת תאימות. היזכרו ב-**endians** – בושא הגדרת משתנים ופקודת **mov**). המעבד והתוכנה יכולים לעבוד או בשיטת **little endian** או בשיטת **big endian**. אין הדבר משנה כלל, כל עוד גם המעבד וגם התוכנה מבצעים את כל הפעולות בשיטה אחת. עירוב של שיטות יגרום לכך שהתוכנה לא תתפרק.

קונבנצייה – Convention – מוסכמה. התנוגות רוחות.

כפי שלמדנו על קונבנציות הקשורות לזכרון, ישנן גם קונבנציות של הפעלת פרוצדורות, ואלו נקראות **Calling Conventions**. תחילת נמחיש מדויק ייש בכלל צורך ב-**Calling Conventions**:

נניח האדרה של פונקציה בשפת C:

```
int MyProc (int a, int b);
```

מה אנחנו יכולים לדעת על הפרוצדורה **MyProc**? אפשר לקבוע שהיא מקבלת שני פרמטרים מטיפוס **integer** ומהזירה ערך מטיפוס **integer**. מה לבדוק עשה הפרוצדורה איננו משנה כרגע.

קריאה לפרוצדורה יכולה להיות דוגמה:

```
int c = MyProc(1,2);
```

קטע הקוד שקורא לפרוצדורה (ה"אבא") נקרא **Caller** ואילו קטע הקוד שנקרא, ככלומר הפרוצדורה עצמה (ה"בן"), נקרא **Callee**.

כעת הבה נדמיין שהפרוצדורה **MyProc** נכתבה בשפת אסמבלי ואילו הקריאה לפרוצדורה נכתבה כחלק מקוד בשפת C אשר קומpileר ממיר לשפת אסמבלי. חישבו – מה יכול להשתבש בין ה-**Caller** ל-**Callee**?

1. העברת הפרמטרים על גבי המחשנית:

הקומpileר יכול להמיר את הקריאה לפרוצדורה ביותר מדרך אחת. אפשרות א':

```
push 1
```

```
push 2
```

```
call MyProc
```

אפשרות ב':

```
push 2
```

push 1

call MyProc

אפשרות א' נקראת העברת ממשאל לימין Pass Left to Right ואילו אפשרות ב' נקראת העברת מימין לשמאלי Right to Left. בהמשך נראה דוגמאות של קונבנציות נפוצות.

אילו ה-caller יעביר את הפרמטרים למחסנית בשיטת העברת שונה מאשר השיטה שבה הcallee קורא את הפרמטרים מהמחסנית, ערכיהם של הפרמטרים יהלפו. המסקנה היא, ש כדי למנוע תקלות, ה-caller והcallee חייבים לעבור לפי אותה הקונבנצייה.

2. החזרת ערך מהפרוצדורה:

הדרך הנוחה ביותר להחזיר את הערך שהיחסבה MyProc אל הקוד שקרה לה, היא להשתמש ברגיסטר. כלומר, הפרוצדורה MyProc תעתק את תוכנת הפעולה שלה אל רגיסטר כללי כלשהו, והערך יועתק על ידי הcallee מהרגיסטר אל המשתנה שומר את ערך החזרה, במקרה זה – המשתנה C.

כמובן שה-callee וה-caller חייבים להיות מתואימים לגבי הרגיסטר שמשמש להחזרת הערך. אין זה משנה באיזה רגיסטר כללי יוחזר הערך, רק שה-callee וה-caller יתיחסו אותו הרגיסטר.

3. ניקוי המחסנית:

כזכור, כאשר מעבירים פרמטרים לפרוצדורה על גבי המחסנית, יש צורך לנוקוט את המחסנית בסיום ריצת הפרוצדורה. פעולה זו מתבצעת על ידי העלאת ערכו של sp בהתאם למספר הבתים שהועתקו למחסנית. למדנו שישנן שתי דרכים לעשות זאת. האחת, היא להוסיף לפקודה ret קבוע. לדוגמה, ניקוי 4 בתים מהמחסנית:

ret 4

הזריך השנייה היא לקדם את ערכו sp:

add sp, 4

ה-callee וה-caller צריכים להיות מתואימים לגבי מי מנקה את המחסנית. אם ה-callee מבצע את הניקוי, בתוך הפרוצדורה תהיה פעולה ret עם קבוע. אם ה-callee מבצע את הניקוי, לאחר החזרה מהפרוצדורה יבוצע הניקוי. לדוגמה כך:

call MyProc

add sp, 4

גם במקרה זה אפשר לעבוד בכל שיטה, העיקרי שה-caller וה-callee יעבדו לפי אותה קונבנצייה.

חישבו: מה היה קורה לו ה-caller וה-callee לא היו מתואמים בנושא ניקוי המחסנית? אילו בעיות היו עלולות להגרם?

קונבנציות נפוצות

לאחר שראינו שיש צורך בקונבנציות, נסקור שתים נוספות. ישנן קונבנציות נוספות, שניתן לקרוא עליה בוויקיפדיה (https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions) ובמגון אתרי אינטרנט, אך לצורך המחתה הנושא נסתפק בפירוט של STDCALL ו-CDECL.

לפי קונבנציית CDECL:

- Register הזרה הוא ax (או גיסטר ax מורחב, אם מדובר על מעבד של למעלה מ-16 ביט).

- ה-caller הוא שאחראי על ניקוי המחסנית.

לפי קונבנציית STDCALL:

- Register הזרה הוא ax (או גיסטר ax מורחב, אם מדובר על מעבד של למעלה מ-16 ביט).

- ה-callee הוא שאחראי על ניקוי המחסנית.

היתרון של STDCALL על פני CDECL, הוא שאפשר לשלווה לפרוצדורה כמהות לא קבועה של פרמטרים. מדוע? משום שה-caller אחראי על ניקוי המחסנית, כלומר מי שביצע את פעולה עדכון sp הוא ה-caller. כיוון שהוא יודע כמה פרמטרים הוא עבריר לפרוצדורה והוא גם יכול לבצע את הניקוי. ה-callee, לעומת זאת, אינו יכול לבצע את הניקוי כיוון שמספר הפרמטרים אינו קבוע מראש.

מתי נרצה להביר לפרוצדורה כמהות לא קבועה של פרמטרים? לדוגמה, כאשר אנחנו קוראים לפרוצדורת print בשפה עילית, אנחנו רוצים גמישות לקרוא לה עם כמה פרמטרים שאנחנו צריכים להדפיס. כל פרמטר יועבר לפרוצדורה.print ולכן לא ניתן לכתוב את print כך שהיא תנקה את המחסנית בסוף הריצתה.

מהו היתרון של STDCALL על פני CDECL? כאשר כמהות הפרמטרים היא קבועה וידועה מראש, הפרוצדורה יכולה לנוקוט את המחסנית על ידי פקודת ret עם קבוע. זאת לעומת המצב שבו הפרוצדורה מבצעת ret ואז ה-caller מבצע פקודה שמעלה את sp. כמובן חסכנו פקודה. אמנם חסכו של פקודה אחת נראה כמו משהו צנוע, אך כשמכפילים את החיסכון בכמות הקריאה לפרוצדורות עשויה להתקבל חיסכון משמעותי.

לקריאה נוספת: <http://www.codeproject.com/Articles/1388/Calling-Conventions-Demystified>

סיכום

פרק זה הוקדש ללימוד של כמה נושאים מתקדמים, בהם הכרחיים כדי להבין איך בניות תוכניות מודולריות. כשתעבדו עתיד עם שפות תוכנה עיליות, חלק מהמנגנוןים שלמדנו בפרק זה יהיו נסתורים מעיניכם. לדוגמה, כדי להגיד משתנים מקומיים לא צריכים להזכיר מקום על גבי המחסנית – אולם הרקע שנותן בפרק זה מאפשר לכם להבין לעומק את פעולות המעבד, וההבנה העומקה היא שתיתן לכם את הכלים הנדרשים לעבודה בעולם הסיבר.

פתחנו את הפרק עם סקירה של המחסנית, אופן הפעולה שלה, הרגיסטרים שקשורים אליה ופקודות `pop` ו-`push`.

המשךו בהסבר על פראוצדורות – איך מגדרים פראוצדורה, מה המעבד מבצע עם הקריאה לפראוצדורה, ההשפעה של פראוצדורה על מצב המחסנית. למדנו על פקודות `call` ו-`ret`. לאחר מכן ניתן למדן איך אפשר לשולח פרמטרים לפראוצדורה. סקרנו שיטות שונות:

- העברה לפי ערך – **Pass by value**

- העברה לפי ייחוס – **Pass by reference**

- שימוש במחסנית

לאחר מכן למדנו להגיד משתנים מקומיים בעזרת שמירת מקום במחסנית, איך אפשר להקל על העבודה בעזרת הרגיסטר `bp` ושימוש בפקודת הגדרת הקבועים `equ`.

לבסוף עסקנו בשני נושאים בעלי חשיבות בעולם התוכנה. על מנת להבין את הנושא הראשוני, **Stack Overflow**, השתמשנו באבני בניין אותן למדנו בתחילת הפרק וראינו כיצד ניתן לנצל את הידע החדש שלנו על מנת להבין בעיות אבטחה. כתע נוכל להשתמש בידע זה על מנת לכתוב קוד מאובטח יותר. הנושא השני, **Calling Conventions**, מאפשר לנו להבין את המשקדים שבין קוד בשפה עילית לבין קוד בשפת אסמביל.

בפרק הבא נלמד על פסיקות. בפרק הלימוד רأינו כמה דוגמאות קוד שלא הבנו – למשל, השימוש בקוד של הדפסתתו למסך או קריאתתו מהמשתמש, אך לא הבנו איך הקוד פועל. לאחר שנלמד פסיקות נבין את קטיעי הקוד הללו ונדע לכתוב בעצמנו קטיעי קוד דומים.

פרק 10 CodeGuru Extreme – 10 (הרחבה)



מבוא

הינה תחרות ארצית יוקרתית, בה קבוצות מתחמות זו בזו על כתיבת תוכנה שתשלוט בזירה וירטואלית. התוכנות נכתבות בשפת אסםבי ונקראות "שורדים". התחרות מתקיימת אחת לשנה והשתתפות בה מתחזעת בקבוצות של 2-5 תלמידים.



לינו תחרותExtreme

אתר התחרות: <http://www.codeguru.co.il/xtreme>

השתתפות בתחרות היא בעלת יתרונות רבים. ראשית, זהה הזדמנות מעוללה לחדר את כישורי התכנות שלכם בשפת אסםבי. שנייה, זהה הזדמנות להתנסות במחקר תוכנה על ידי Reverse Engineering. שלישיית, השתתפות בתחרות היוקרתית היא פרט שנייה להוסיפה נוספת לקורות החים ולצין אותו בראינוע העבודה ובמיונים לתפקידים שונים. אחרון- התחרות היא חוות כיף ומנה, גם מי שלא זוכים בה זוכים את החוויה.

הסביר קצר על חוקי התחרות:

כל קבוצה מגישה לתחרות קוד אסםבי, שנקרא שורד. השורדים שהגיבו כל הקבוצות נתענים אל זירה וירטואלית, קטע זיכרון שגודלו 64 קילו בתים בסך הכל. כל שורד מוגדר למיקום כלשהו בזירה. בכל חור, כל שורד מקבל אפשרות להריז פקודת אסםבי אחת. באמצעות פקודת האסםבי השורד יכול לבצע דברים שונים, לדוגמה- לנסות לפגוע בקטע קוד של שורד יריב. אם שורד מנסה להריז פקודת לא חוקית, הוא נפסל ומנוע המשחק מסלך אותו מהזירה. השורד שנשאר אחרון בזירה מוכרז כמנצח.

נוסף על השורדים, צוות התחרות מכניס לזרה "זומבים". אלו הן תוכנות זדוניות, שתופסות חלק מזירת המשחק. הן כוללות הידה תכנית או מתימית, שפתרון שלה מאפשר השתלטות על הזומבי ושימוש בו נגד שורדים מתחרים.

נסתפק בהסביר קצר זה, כיוןSCP הפתרים והדרכה על התחרות מוצאים באתר קודגورو אקסטרים ובמיוחד בחוברת ההדרכה ובמצגת, שבקישוריהם הבאים:

[חוברת הדרכה](http://www.cyber.org.il/assembly/codeguru-guide.pdf)

[מצגת](http://www.cyber.org.il/assembly/codeguru-slides.pdf)

כמו כן מומלץ להעזר בפורום שבאתר קודגورو אקסטרים, שם גם מפורסמים עדכוניים שותפים לגבי התחרות:

[פורום](http://www.codeguru.co.il/wp/?forum=%D7%90%D7%A7%D7%A1%D7%98%D7%A8%D7%99%D7%9D)

בפרק זה תמצאו נושאים שאינם מפורטים בחוברת ההדרכה ובמצגת, אך הם עשויים לסייע לכם רבות בהכנה לתחרות. הנושא הראשון הוא מספר פקודות שימושיות באסמלி. הנושא השני הוא ביצוע Reverse Engineering לזרמים על מנת להשתלט עליהם ולשורדים מתחרים על מנת לאתר נקודות חולישה שלהם.

פקודות אסמללי שימושיות

פקודות אלו אינן הכרחיות לטובת ייחודה המעבדה באסמללי, כיון שניתן לכתב תוכנית אסמללי עובדת היטב גם ללא שימוש בהן. עם זאת הן יכולות לבצע פעולה שבשבילו לבצע אותן בדרך אחרת יידרשו מספר רב של שורות קוד ולפניהם רצוי להזכיר אותן כדי לכתוב קוד יעיל לשורה.

1. XCHG: פקודה XCHG מקבלת שני רגיסטרים, או רגיסטר וטא בזיכרון, ומחליפה בין הערכיהם שלהם

לדוגמה:

xchg ax, bx

מחליפה בין הערכיהם של ax ו-bx. פעולה זו חוסכת מספר פעולות mov.

2. XLAT: על מנת להבין את פקודה XLAT נבון קודם מהו Look Up Table, או בקיצור LUT. LUT היא טבלה שמחזיקה אוסף של ערכים, כאשר יש קשר בין הערך לאינדקס שלו. לדוגמה- אפשר להציג ב-LUT את ערכי סדרת פיבונצ'י. כיון שסדרת פיבונצ'י היא

0, 1, 1, 2, 3, 5, 8, 13 ...

או לדוגמה הערך באינדקס 6 יהיה 8, הערך באינדקס 7 יהיה 13 וכך הלאה (שים לב- האינדקסים מתחילה מאינדקס אפס).

בשביל מה זה טוב? נניח שיש לנו תוכנית שכוללת המרת מסט אחד של ערכאים לסט אחר של ערכאים. לדוגמה- אנחנו רוצים להציג טקסט על ידי צופן החלפה (צופן בו כל תו מוחלף על ידיתו אחר. לדוגמה a מוחלף ב-m, b מוחלף ב-f וכו'). דרך אחת היא ליצר פרוטזורה שיש בה הרבה מאד תנאים (אם האות היא a החרור m, אם האות b החרור f וכן הלאה...). דרך נוספת הרבה יותר היא לשמר ב-LUT את צופן החלפה כך שכל אינדקס ב-LUT מצביע על הערך אותו יש להחרור. לדוגמה, קוד ה-ASCII של a הוא 97 ואנחנו רוצים להחליף אותה עם האות m, שקוד ה-ASCII שלה הוא 109. את האות b, שקוד ה-ASCII שלה הוא 98, אנחנו רוצים להחליף באות f, שהקוד שלה הוא 102. נdag שבאינדקס 97 יישמר הערך 109 ובאינדקס 98 יישמר 102. הנה כך:

Cipher db 97 dup (0), 'mf'

כמובן שאנו יכולים להוסיף אחרי mf גם את הצופן של יתר האותיות. כתוב בתוך סגמנט הקוד נכתב:

mov bx, offset Cipher

mov al, 'a'

xlat

ואם יכיל את ערך ה-ASCII של m, כפי שרצינו.

.3. **NOP**: זה קיצור של **No Operation**, כלומר פקודה שאומרת למעבד "אל תבצע כלום". פקודה זו יכולה להיות שימושית כאשר צריך למלא מקום זיכרון בפקודה חוקית, אך בלי שהיא תשפיע על מצב התוכנית.

.4. **STD / CLD**: הפקודה STD מדיליקה את דגל הכוון ואילו CLD מכבה אותו. לשם מה זה נחוץ? בשבייל פקודה **.MOVSW**.

.5. **MOVSW**: פקודה זו מסיימת לנו להעתיק בצורה יעהה מידע מאזור זיכרון אחד לאחר. לדוגמה, יש לנו מהרוות של 200 איברים, ואני מעוניינים להעתיק אותה למקום אחר בזכרון. דרך אפשרית היא ליצור לו לאה שתறוץ על המהרוות הראשונה תוך כדי שהיא מקדמת בכל פעם אינדקס שימוש שמשמש לקריאה מהזיכרון, ובעזרת פקודה mov תעתק את המהרוות למקומות החדש, תוך קידום אינדקס שימוש שמשמש לכתיבה לזכרון. זה יעבד, אולם פקודה MOVSW עוזרת לפשט את התהליך.

פקודה זו מעתקה מילה מכתובת **ds:di** אל הכתובת **es:si**, תוך עדכון ערכי הרגיסטרים **si**, **di**. כאמור, נחסכה מאייתנו המתכנתים פעולה עדכון רегистר המקור ורגיסטר היעד. אך כדי לעדכן את ערכי הרגיסטרים **si**, **di**, נדרש

לדעת האם להעלות אותם או להוריד אותם. לשם כך קבענו את ערכו של דגל הכוון בפקודת CLD או STD טרם ההעתקה.

6. REP: כדי להעתיק מספר תאים בזיכרון ניתן להוסיף לפני הוראת MOVSW את הקידומת REP, קיצור של Repeat. ראשית מתחילה את ערכו של CX וכך שיכיל מספר הפעמים שהוראת העתקה צריכה להתבצע ואו כותבים

rep movsw

קוד זה שקול לפועלות הבאות:

my_label:

```
movsw
dec    cx
jnz    my_label
```

פקודת אסמלבי נוספת ניתן למצואBNNSPFH A'.

Reverse Engineering

נושא ה-RE, Reverse Engineering, או בקיצור RE, הוא עמוק ורחב מכדי לסתם אותו בפרק אחד. לכן לצערנו אין ביכולתו ללמד לביצוע RE במסגרת לימודי האסמלבי. מאידך, כן נלמד איך מבצעים RE לשורדים ולזומבים של קודגورو אקסטרים. זהו משימה בהיקף מצומצם, שמאז אחד ניתן לכנות בפרק ייחד ומצד שני מספקת טיעמה מילולית מעניינת זו.

הזומנים שבפרק זה נמצאים בקישור:

www.cyber.org.il/assembly/zombies.zip

שימו לב- בסעיפים הבאים נלמד כיצד לחקור קוד של תוכנה. מחקר של תוכנה הוא חוקי כל עוד הוא נעשה על תוכנה שנמסרה לנו בכוונה שנחקרו אותה. מחקר של תוכנה מסחרית, בכוונה לפרוץ סיסמאות או לשנות דברים בתוכנה, הוא אסור לפי החוק. היזהרו והישארו מצד הטוב של החוק.



duck.com

נתחיל בניתו זומבי פשוט. הורידו את הזומבי duck.com. כתע ענו על השאלה - מה מבצע הזומבי? נסו לגלות.



אפשרות אחת היא להפעיל את קובץ הרצה. מתוך חלון ה-cmd הקישו `duck` ואו `enter`. התוכנית רצה אך אינה מוגיבה... הבהיה היא שאין לנו את קוד המקור של התוכנית אלא רק את הקובץ הבינארי שלה, את שפת המכונה. מה אפשר לעשות עם הקובץ הבינארי? ובכן, אפשר להריץ אותו בדיבאגר שלנו. הדיבאגר יודע לתרגם את שפת המכונה לשפת אסמבלי.

הנה כך:

```
[ ]=CPU 80486=
cs:0100>EBFE      jmp    0100 ↓
cs:0102 16          push   ss
cs:0103 A6          cmpsb 
cs:0104 08830672    or     [bp+di+7206],al
cs:0108 090A          or     [bp+sil],cx
cs:010A C74602FB09    mov    word ptr [bp+02],09FB
cs:010F 2E8E1E0000    mov    ds,cs:[0000]
cs:0114 A1F028    mov    ax,[28F0]
cs:0117 894604    mov    [bp+04],ax
cs:011A 5E          pop    si
cs:011B 5F          pop    di
cs:011C 5A          pop    dx
cs:011D 59          pop    cx
cs:011E 5B          pop    bx
cs:011F 58          pop    ax
```

אפשר לראות את תרגום שפת המכונה לשפת אסמבלי. התוכנית מתחילה בכתובת `cs:100h`. הפקודה הראשונה היא `jmp` `100h`. לאחר מכן יש פקודות אחרות שלא ברור מה הקשר ביניהן. מיד נבין.

נתחיל בהרצה התוכנה באמצעות `f7`. כאשר נריץ את התוכנה נגלה שהיא נשארת באותה שורת קוד. הסיבה לכך היא שפקודה `jmp 100h` נמצאת בכתובת `100h`. למעשה, אפשר להבין שהקוד שייצר את התוכנית היה:

`start:`

`jmp start`

`end start`

כעת, מהן אותן שורות קוד שמופיעות החל מכתובת `102h`? זה התרגום לשפת מכונה של התאים הללו מאותחלים שישם בזיכרון המחשב, למשל זיכרון "זבל". זה, סימנו את ניתוח הזומבי הראשון שלנו.

coffee.com

לאחר שהבנו את העקרון הכללי של RE, נעבור לזומבי הבא, שמו coffee.com. הפעם גם נבצע RE כדי להבין מה הזומבי מבצע וגם נלמד איך אפשר לגרום לו להריזן קוד שאנחנו כתבו.



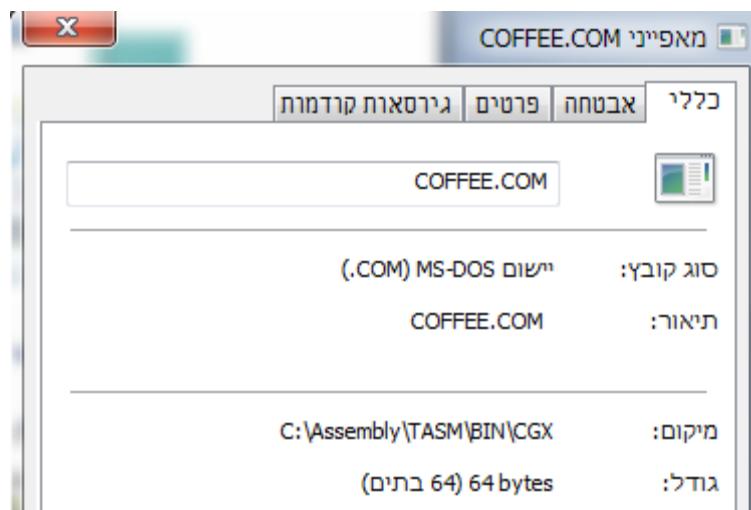
```

CPU:80486
cs:0100→0E      push   cs
cs:0101 07      pop    es
cs:0102 CD87      int    87
cs:0104 8A160000    mov    dl,[0000]
cs:0108 80FA43      cmp    dl,43
cs:010B 75F7      jne    0104
cs:010D 8A160100    mov    dl,[0001]
cs:0111 80FA30      cmp    dl,30
cs:0114 75F7      jne    010D
cs:0116 8A160200    mov    dl,[0002]
cs:011A 80FA46      cmp    dl,46
cs:011D 75F7      jne    0116
cs:011F 8A160300    mov    dl,[0003]
cs:0123 80FA46      cmp    dl,46
cs:0126 75F7      jne    011F
cs:0128 8A160400    mov    dl,[0004]
cs:012C 80FA45      cmp    dl,45
cs:012F 75F7      jne    0128
cs:0131 8A160500    mov    dl,[0005]
cs:0135 80FA45      cmp    dl,45
cs:0138 75F7      jne    0131
cs:013A 8B1E0600    mov    bx,[0006]
cs:013E 53          push   bx
cs:013F C3          ret
cs:0140 BD00BB    mov    bp,BB00
cs:0143 B82606    mov    ax,0626
cs:0146 6A00          push   0000
cs:0148 50          push   ax
cs:0149 6A04          push   0004
cs:014B 6A00          push   0000

```

ברור שהקוד של זומבי זה הינו בעל יותר משורה אחת. המשימה הראשונה שלנו היא לזהות את נקודת סיום הקוד. אפשר לראות שיש בקוד רצף חזר של פקודות mov-cmp-jne ו- push ו- ret. פקודה ה-ret היא מועמדת לא רעה לצין את סיום הקוד של הזומבי. גם אילו הינו רואים את פקודות היציאה מ-dos (int 21h, ax=4C00h) הינו יכולים לדעת שמדובר בסיום הקוד. לאחר פקודה ה-ret נפסיק רצף הפקודות ההיגייניות. לדוגמה, הפקודה push 0000 מופיעה פעמיים, סביר להניח שהוא אין קו שמיישחו כתוב.

כדי למצוא בודאות את מקום סיום הקוד, ניגש אל המאפיינים (properties) של הקובץ:



כפי שראים, גודלו של הקובץ הוא 64 בתים, או $40h$. כיוון שהקובץ מתחילה בכתובת $cs:0100h$, הרי שהפקודה המתחילה ב- $h0:0140$ כבר אינה חלק מהקובץ. כך וידאונו באופן סופי את ההשערה שלנו.

כעת – מה עושה הזרמי `?coffee`

שלוש הפקודות הראשונות צריכות להיות מוכרכות לכל מי שקורא את המדריך – פקודת `int 87` היא הפצת חכמה. הזרמי נפטר מהפצת חכמה שלו.

ב- $h0:0104$ `cs` מועתק בית מקום $0000h$ בזיכרון לתוכו `dl`. בפקודה הבאה הערך של `dl` מושווה ל- $43h$ ואם אין שוויון, מתבצעת זורה על העתקה מהזיכרון ופעולה ההשואה. כמובן, כל עוד מקום $0000h$ אינו מכיל את הערך $43h$ (קוד ה- $30h$ ASCII של האות C). אם השווין מתקיים, התוכנית ממשיכה לבדוקה הבאה – האם הבית הבא בזיכרון מכיל את הערך $30h$ (קוד ה- $ASCII$ של הסירה 0). המשיכו מכאן – אילו תנאים צריכים להתקיים כדי שהתוכנית הגיע אל פקודת `h0:013Fh ret`?

נניח שאנו רוצים להשתלט על זומיי זה, כלומר לגרום לו להריץ את הקוד שלנו. איך אפשר לבצע זאת? נctrack לשילוח אותו ל-`h0:013Fh` בו נמצא הקוד שלנו. פקודת `h0:013Fh ret` שבכתובת `cs:0104` יכולה לסייע לנו ביצוע התהליך. אנחנו צריכים לדאוג לשני תנאים. הראשון, שהזרמי יגיע להריץ את שורת `h0:013Fh ret`. ראיינו שעיל ידי הזנת ערכים מתאימים למיקומים מוגדרים בזיכרון, אנחנו יכולים "לשחרר" אותו ולגרום לו להגיע לפקודת `h0:013Fh ret`. התנאי השני, הוא שעם ההגעה לפקודת `h0:013Fh ret` בראש המחסנית יהיה ה-`bx` של הקוד שלנו. לעוזרתו נמצאות הפקודות שמעתיקות את מקום $0006h$ אל `ax` ואז דוחפות את `ax` בראש המחסנית. אם נdag לכך שזמן שהזרמי מריץ את שורות אלה בכתובת $0006h$ נמצאת כתובות של פקודה בקוד שלנו, הזרמי יסיים את הריצה ויקפוץ אליה. שם הוא ימשיך ויריץ את הקוד שלנו.

סיימנו את הטיפול ב-`coffee.com`. זה זומיי חביב ולא מורכב שכל מטרתו למד את הרעיון הכללי של RE לתוכנית קצרה ושליחת המעבד אל קוד במיקום אחר.

כעת נטפל בזומבי אמיתי מתחנות 2015. כנהוג בתחרות, הזומבי מכיל בתוכו חידה מתמטית שיש צורך לפתור לפני שנוכל להשתלט עליו.

```
[CPU 80486]
cs:0100 0E      push   cs
cs:0101 07      pop    es
cs:0102 CD87      int    87
cs:0104 BB1D00      mov    bx,001D
cs:0107 01C3      add    bx,ax
cs:0109 50      push   ax
cs:010A 91      xchg   cx,ax
cs:010B 5A      pop    dx
cs:010C A11520      mov    ax,[2015]
cs:010F 50      push   ax
cs:0110 31C8      xor    ax,cx
cs:0112 D7      xlat
cs:0113 86C4      xchg   ah,al
cs:0115 D7      xlat
cs:0116 00E0      add    al,ah
cs:0118 3C06      cmp    al,06
cs:011A 7FEF      jg    010B
cs:011C C3      ret
cs:011D 0001      add    [bx+di],al
cs:011F 0102      add    [bp+si],ax
cs:0121 0102      add    [bp+si],ax
cs:0123 0203      add    al,[bp+di]
cs:0125 0102      add    [bp+si],ax
cs:0127 0203      add    al,[bp+di]
cs:0129 0203      add    al,[bp+di]
cs:012B 0304      add    ax,[si]
cs:012D 0102      add    [bp+si],ax
cs:012F 0203      add    al,[bp+di]
cs:0131 0203      add    al,[bp+di]
cs:0133 0304      add    ax,[si]
```

- א. כפי שעשינו לפניהם, ניתן לזהות את סיום קוד הזומבי בפקודת `ret` שבמיקום `cs:011Ch`.
- ב. שלושת הפקודות הראשונות נפטרות מהפצת החכמה.
- ג. ארבעת הפקודות הבאות (`cs:0104h` עד `cs:010Ah`) מעתיקות לתוך `cx` את `ax`, כאשר `ax` שווה בקודגورو אקסטרים למיקום ההתחלתי של הזומבי ואילו `bx` מאותחל לקובע `1Dh`.
- ד. לאחר מכן נטען לתוך `ax` המילה שמתחליה במיקום `h 2015` בזיכרון.
- ה. לאחר ביצוע `xor` בין `ax` ו-`cx` ממבצע פעלות `xlat`. נזכיר במה מבצעת `xlat`: היא מעתקה לתוך `ax` את הערך שנמצא במיקום `ax+bx+ds`. נציין כי בנויגוד לתוכניות שאיתן עבדנו בפרקיהם הקודמים, בקודגورو אקסטרים מודול הזיכרון מוגדר כך `sh-ds` ו-`cs-ds` מצביעים על אותו מקום בזיכרון. אם כך, על איזה זיכרון מצביע `bx:ds?` נבדוק

מהו ערכו של ax ערך זה הוגדר להיות מיקום תחילת התוכנית ועוד הקבוע $1Dh$. כיוון שהתוכנית מתחילה במיקום $100h$, ערכו של ax יהיה $11Dh$. ככלומר $ax:ds$ יהיה שווה ל- $11Dh:cs:011Dh$.

נחקור את המיקום בזיכרון $011Dh:cs$. זהו המיקום הראשון שנמצא אחרי פקודת ret . נמצאת שם פקודת add .

משונה. גם יתר הפקודות הן צורות שונות של add . מדובר מטבח $xlat$ למיקומים אלו בזיכרון? נניח $al=al$.

הפקודה $xlat$ תחזיר את הערך שבמיקום $011Dh:cs$. ערך זה הוא 0 . נניח $al=1$, הפקודה $xlat$ תחזיר את הערך

שבמיקום $1Eh:cs$ ערך זה הוא 1 . כך ניתן לראות שה"פקודות" בחלק זה של הזיכרון אינן אלא סדרה של

מספרים.

ז. סדרת המספרים היא $0,1,1,2,1,2,2,3$ וכן הלאה. מה פירוש סדרת מספרים זו? נסו לגלוות את חוקיות הסדרה.

אם איןכם מצלחים, לא נורא- הריצו חיפוש בגוגל על סדרת המספרים. אם הצלחתם למצוא את החוקיות בלי

עוזרה- הרשמו בהקדם לאולימפיאדה למתמטיקה לנוער.

ח. המשקנה עד כאן היא שהזומבי שלנו לוקח את המיקום ההתחלתי שלו בזיכרון ואת הערך שנמצא בתא 2015 בזיכרון, עושה להם ox וממשתמש בתוצאה על מנת לgesht ל- LUT , שהיא סדרה מתמטית בעלת חוקיות מעניינת.

ט. פעולה $xlat$ חוזרת על עצמה פעמיים, פעם עבר al ופעם עבר ah .

י. לאחר מכן מחושב הסכום שלהם- אם הוא איינו עולה על 6 , מטבח ret ייצאה על ידי ret . איזה ערך יוכל?

יא. מהו הערך שיש בראש המחסנית בשלב זה? הערך האחרון שנדחף למחסנית הוא ax , מיד לאחר שהוועתק אליו ה-

$word$ שמתחליל במיקום $2015h$.

לסיכום, אנחנו מבינים שעיל מנת להוציא את הזומבי מהלולאה בה הוא נמצא ולהגיע לפקודת ret , יש צורך להכניס ערך מסוים ל- $word$ במיקום $2015h$. אותו הערך הוא גם ה- cpo שהזומבי יקפוץ אליו בסומו. הבנת הערך ה"נכון" תלולה בהבנת חוקיות הסדרה המתמטית שאיתרנו.

תרגיל: Make it – Break it – Fix it

בתרגיל זה נכתב תוכנית שמקשת סיסמה משתמש ובודקת אם הסיסמה "נכונה". אם כן, תודפס למסך הודעה מתאימה: "Access granted". בשלב הראשון, נסה לכתב תוכנה מסוובכת ככל הניתן לפענו. בשלב השני נמסור את התוכנה לחברינו לכיתה, שנgeo לגרום להדפסת ההודעה "Access granted" על ידי פענו הסיסמה. בשלב השלישי נפיק לקחים מההחברינו עשו ונשפר את התוכנה כך שמציאת הסיסמה תהיה קשה יותר.

שלב א' - Make it

בשלב זה אנחנו מעוניינים לכתב תוכנית שקולטת סיסמה משתמש, אם הסיסמה נכונה היא מדפיסה את הודעה הצלחה למסך. כתיבה של תוכנית כזו דורשת שני דברים טרם למדנו: הראשון, קליטה שלתו מהמשתמש. השני, הדפסה של מחרוזות למסך.

קליטה שלתו מהמשתמש: מכנים לתוכה ah את הערך 1, כותבים את הפקודה int 21h לתוך ah, התו שהמשתמש מקליד נכנס לתוך ah. כך:

```
mov ah, 1
int 21h
```

הדפסה של מחרוזות למסך:

כדי להדפיס מחרוזת למסך אנחנו צריכים קודם כל ליצור מחרוזת, שמתויימת בתו '\$' (זה הסימן ל-ISR להפסיק את ההדפסה למסך – לאחרת יודפס כל הזיכרונו...). התווים 13,10 מורים לרדת שורה בסוף ההדפסה. לדוגמה:

```
message db 'Hello World',13,10,'$'
```

לאחר מכן טוענים לתוך dx את האופסט של המחרוזת:

```
mov dx, offset message
```

נותר לנו רק לקבוע את ah=9h ולהפעיל את הפקודה int 21h את ah=9h

```
mov ah, 9h
int 21h
```

הסברים מפורטים על פקודות אלו נמצאים בפרק אוזות פסיקות, תחת הסעיף פסיקות DOS.

להלן תוכנית פשוטה, שמקשת סיסמה מהמשתמש ועל סמך בדיקת הסיסמה מדפיסה הודעה הצלחה או כישלון. שקולטת תווים, משווה אותם לסיסמה ואם יש שוויון מדפיסה הודעה מתאימה. כפי שאפשר לראות קל למצוא סיסמה מתאימה. אתם יכולים להתבסס על התוכנית זו ולשפר אותה כך שמציאת הסיסמה תהיה ממשימה מורכבת.

```

; -----
; Simple get password program- a very basic code just to help you start
; Author: Barak Gonen 2015
; -----



IDEAL

MODEL small

STACK 100h

DATASEG

Save db (?)  

Welcome db 'Please enter password, press enter to finish',13,10,'$'  

Access db 13, 10, 'Access granted$'  

Wrong db 13, 10, 'Login failed$'

CODESEG

start:  

    mov ax, @data  

    mov ds, ax  

    mov ah, 9  

    mov dx, offset Welcome  

    int 21h  

    xor cx, cx

getChar:  

    mov ah, 1  

    int 21h  

    cmp al, 13  

    je check  

    mov [Save], al

```

```
inc    cx
jmp    getChar
```

check:

```
cmp    [Save], 'X'
jne    fail
cmp    cx, 3
jne    fail
```

success:

```
mov    ah, 9
mov    dx, offset Access
int    21h
jmp    exit
```

fail:

```
mov    ah, 9
mov    dx, offset Wrong
int    21h
```

exit:

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

END start

איך תוכלו לגרום לתוכנית זו להיות יותר חסינה לנטיונות מציאת הסיסמה?

הדרך:

- .1. בחרנו את השורות בהן מתבצעת הבדיקה של הסיסמה. האם תוכלו לפתח מנגנון יותר מורכב, שככלל בדיקות נוספת?
- .2. ביצוע RE לקוד שיש בו השוואה קבוע, לדוגמה 'X', מסגיר מיד מהו הקבוע המבוקש. נסו לבצע את ההשוואה על ידי רегистרים.

.3. השתמשו בפעולות על ביטים כדי להקשות עוד יותר את מציאת הסיסמה הנכונה מתוך הקוד בשפת מכונת.

.4. מכאן המשיכו להלאה בכוחות עצמכם. גלו יצירתיות!

שלב ב' - Break it

קחו את התוכנית שכ כתבו חבריכם (כמובן, רק את קובץ הרצאה - לא את שורות הקוד באסמלוי). השתמשו במידע שצברתם כדי לפצח את סיסמת הכניסה אל התוכנית של חבריכם. הסבירו לחבריכם איך מצאתם את הסיסמה לתוכנית שלהם.

שלב ג' - Fix it

לאחר ששמעתם כיצד חבריכם פיצזו את סיסמת הכניסה לתוכנית שלכם, מיצאו פיתרון שיקשה על חבריכם למצוא את הסיסמה באמצעות方法ה השיטה.

סיכום

בפרק זה סקרנו בקצרה את תחרות קודגورو אקסטרים וקיבלנו מספר כלים חשובים להצלחה בתחרות. הכלי הראשון הינו ידיעת פקודות אסמלוי מיוחדות. הכלי השני הוא יכולת מחקר של קוד שנכתב על ידי מי שהו אחר באסמלוי, הכלי השלישי הוא הבנת טכניקת ההשתלטות על זומבים. יתר המידע הנדרש מרוכז בחומר הדריכה - המשיכו מכאן בכוחות עצמכם.

איהולי הצלחה בתחרות והנאה מההשתתפות ומהתרגול לקרה התחרות!

פרק 11 – פסיקות

מבוא

פסיקה (Interrupt) היא אוטומת המתקבל במעבד ומאפשר לשנות את סדר ביצוע הפקודות בתוכנית שלא על-ידי פקודות בקרה מותנית (פעולות השוואה וקפיצה – כגון cmp ו-jmp).



נסביר למה הכוונה. משתמשים בפסיקה כשותם לשנות את סדר ביצוע הפקודות באופן בלתי צפוי מראש (אחרת היינו משתמשים בפקודות בקרה וקפיצה). למה בכלל רוצים לשנות את סדר ביצוע הפקודות?

לעתים נרצה שהתוכנה שלנו לא תעבד בדיקת אותו אופן כל הזמן – לדוגמה, תמצא את האיבר האגול ביותר במערך – אלא תגיב לאיורוים שונים. דמיינו משחק מחשב, שבו השחקן משתמש במקלדת כדי להפעיל דמות שזזה על המסך. התוכנה צריכה להגיב לפקדות השחקן – כל היזח על המקלדת צריכה לגרום לתוכנה להרים קוד אחר. גם התזמון של היזוח המקלדת לא ידוע מראש – התוכנה לא יכולה להניח שברגע מסוים השחקן יקיים על המקלדת. הצורך ביכולת לשנות את אופן ריצת התוכנה, גורם לכך שנוצר מנגנון שיודיע לשנות את סדר ביצוע הפקודות במעבד באופן דינامي.

במשפחת ה- $80x86$ יש שלושה סוגים של איורוים שנכנים תחת המונח "פסיקה":

- **פסיקות תוכנה, שנקראות Traps.** פסיקות אלו הן חלק מקוד התוכנית, כלומר הן יזומות על-ידי המתכנת.
- **פסיקות חריגה, שנקראות Exceptions.** פסיקות אלו הן כמו פסיקות תוכנה, אבל מתרחשות באופן אוטומטי כתגובה לאיורע חריג. לדוגמה, חילוק באפס יפעיל פסיקת חריגה.
- **פסיקות חומרה, שנקראות Interrupts.** פסיקות אלו הן תוצאה של רכיבי חומרה חיצוניים למעבד (לדוגמה מקלדת או עכבר). פסיקות אלו מודיעות למעבד שיש איורע חיצוני שדורש טיפול. המעבד עוצר את ביצוע הקוד שרצן, מסרת את רכיב החומרה וחזור לתוכנית למקום שעצר בה.

במשך נדון בהרחבה בכל אחת מסוגי הפסיקות.

קריאה לפסקה מתבצעת באמצעות הפקודה int. לכל פסקה יש מספר, שמייחד אותה מיתר הפסיקות. לאחר ה-int יבוא אופרנד, שהוא מספר הפסיקה.

int operand

לדוגמה, הפעלת פסקה מס' 1:

int 1h

בפרקים הקודמים נתנו בפקודה זו. כאשר רצינו לבצע פעולות של קריאתתו מהמקלדת או הדפסתו למסך, למשל, המשמשו בפקודה:

int 21h

במשך השנים חברות שונות פיתחו קוד שכלל בנוסף פסיקות תוכנה וחומרה שימושיות. לדוגמה, חברת מיקרוסופט, פיתחה מערכת הפעלה בשם DOS – קיזור של Disk Operating Systems. מערכת הפעלה DOS מספקת

למתקנים פסיקות שימושיות, שנתקלנו בהן בפרקים קודמים, כגון קריאתתו מהמקלדת והדפסתו למסך. דוגמה נוספת היא חברת אינטל, יצרנית מעבדי ה- $80x86$, שפיתחה קוד שנקרא BIOS – קיצור של Basic Input Output System. BIOS הוא קוד שנמצא בזיכרון מיוחד, ונתען למעבד מיד עם הפעלתו. תפקידו של קוד BIOS הוא לבדוק את תקינות החומרה ולטען את קוד מערכת הפעלה. כמו כן כולל BIOS מספר פסיקות ש解脱ות את השימוש בתפקידים חומרים, לדוגמה קליטת תווים מהמקלדת, נושא שנגע אליו בהמשך.

כחותה לכך שיש לנו פסיקות זמניות לשימוש מקורות שונים, ישנן לעיתים דרכים שונות לבצע את אותה פעולה. לדוגמה, תקשורת עם המקלדת:

- עם המקלדת אפשר לתקשר באמצעות פסיקה מס' 9h.

- אפשר לתקשר עם המקלדת גם דרך פסיקה מס' 16h, שהיא פסיקה BIOS. פסיקה זו למעשה "עוטפת" את פסיקה 9h עם קוד נוסף.

- אפשר לתקשר עם המקלדת גם דרך פסיקה מס' 21h, של DOS. קוד DOS "עוטף" את הקוד של BIOS.

דוגמה נוספת, תקשורת עם שעון המערכת (טיימר):

- פסיקה מס' 8h עובדת מול הטימר.

- פסיקה מס' 1Ch היא פסיקה BIOS, שגם עובדת מול הטימר. פסיקה זו למעשה "עוטפת" את פסיקה 8h עם קוד נוסף.

- פסיקה מס' 21h של DOS, עובדת גם היא מול הטימר. קוד DOS "עוטף" את הקוד של BIOS.

יתרונות וחסרונות של שימוש בפסיקות מסוגים שונים: באופן כללי, כל קוד שעוטף את הפסיקה המקורית עושה את השימוש בה יותר פשוט, אבל אנחנו מפסידים גמישות ולעיתים גם ביצועים.

במסגרת לימוד הפסיקות רק יהיה מודעים לכך שיש דרכים שונות לבצע את אותה פעולה, אבל לא נלמד את כל הדרכים. בדרך כלל נבצע את הפעולות באמצעות פסיקות DOS, וכך נקייש לפסיקות אלו הסבר מפורט.

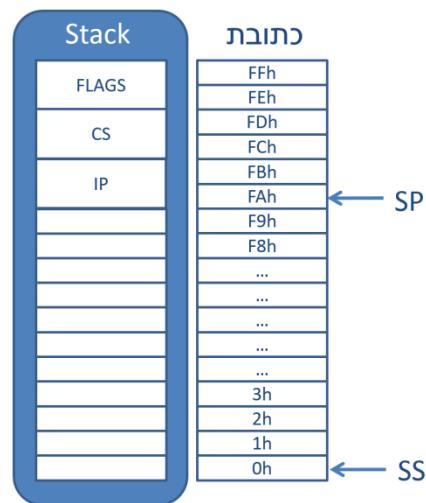
כל פסיקה, יהיה אשר יהיה המקור שלה, מפעילה קוד מיוחד לטיפול בפסיקה. הקוד הוא כמובן שונה בין פסיקה לפסיקה. קוד זה נקרא בשם כללי Interrupt Service Routine או בקיצור ISR. 

כשמעבד מבצע פסיקה, הוא מפסיק את ביצוע התוכנית, פונה אל ISR שנמצא בזיכרון המחשב ולאחר מכן חזר להמשך התוכנית שהופסקה. נסקור כיצד המנגנון שמאפשר למעבד לבצע את הפעולה זו.

שלבי ביצוע פסיקה

תהליך ביצוע פסיקה מורכב מphasבים הבאים:

1. המעבד מסיים את הוראה הנוכחית. ככלומר – אם לדוגמה היהנו באמצעות הוראה `mov ax,5` בזמן שההשתמש לחץ על מקש במקלדת, המעבד יעתיק לתוך `ax` את ערך 5 ואז יתפנה לשרת את הפסיקה מהמקלדת.
2. המעבד שומר במחסנית את תוכן רגיסטר הדגלים ואת כתובות ההוראה הבאה לביצוע בתחום ביצוע חוכנית הפסיקה (ה-ISR). כזכור, כתובת ההוראה הבאה היא צירוף של הרגיסטרים `cs` ו-`ip`.
3. לפני הכניסה ל-ISR המעבד "מנקה" (מאפס) את דגל הפסיקות (Interrupt Flag) ואת דגל המלכודות (Trap Flag). הסיבה לכך – בהמשך.
4. המעבד מחשב את כתובת ISR ומעתיק אותו לרגיסטרים `cs` ו-`ip`. תהליך חישוב כתובת ISR ראוי להסביר בפנוי עצמו ונתיחס אליו בהמשך.
5. המעבד מבצע את ISR.
6. בתום ביצוע ISR, המעבד מוציא מהמחסנית את הערכים שנדחפו אליה (ראו סעיף ב') ומשוחזר את הערכים המקוריים של רגיסטר הדגלים, `cs` ו-`ip`.
7. המעבד ממשיך בביצוע התוכנית ממוקם `ip:cs`.



מצב המחסנית עם הכניסה לפסיקה

(האייר מניח שהמחסנית בגודל `100h` וכמו כן שהמחסנית הייתה ריקה לפני הפסיקה)

נתיחה למשמעות איפוס הדגלים Trap Flag ו-Interrupt Flag:

איפוס דגל המלכודת גורם לכך שהפקודות יבוצעו ללא הפסקה גם אם אנחנו בתחום הדיבאגר. נדמיין מה היה קורה אם דגל זה לא היה מאופס אוטומטית. אנחנו נמצאים בתחום הדיבאגר, מתקדמים בתוכנית שורה אחרי שורה. כל 55 מילישניות, כפי שנלמד בהמשך, מגיעה פסיקה לעדכון השעה. התוכנית שלנו קופצת אל ה-ISR שאחראי לעדכון השעה, אבל מיד עוצרת – אנחנו צריכים ללחוץ על מקש ה-F7 כדי להריץ את התוכנית לשורה הבאה. פעולה זו חוזרת עצמה כל 55 מילישניות... עצם הלחיצה על מקש ה-F7 מפעיל ISR שאחראי לטיפול במקלדת, וכך שלמעשה הפעולה אף פעם לא מסתיימת...

איפוס דגל הפסיקות מונע מפסיקות נוספות להגיע תוך כדי ביצוע הפסיקת הנוכחית. המונח המכווני של פעולה זו הוא disable interrupts. פעולה זו חשובה כדי שלא תהיה "תחרות" בין פסיקות, שבה פסיקה חדשה מגיעה וגורמת למעבד לטפל בה לפני שישים לטפל בפסיקת הקודמת.

לאחר היציאה מה-ISR, המעבד משוחרר מהמחסנית את רегистר הדגלים, ובין היתר את ערכו של הדגל if, וכך שפסיקות הומרה שוב מאפשרות. המונח המכווני של פעולה זו הוא enable interrupts.



המבנה הכללי של ISR הוא כזה:

```
proc ISRname far
```

```
...
```

```
iret
```

```
endp ISRname
```

פקודת iret היא כמו פקודת ret – היא דואגת לשוחזר כתובת החזורה אל המקום שבו הייתה התוכנית לפני הקראיה ל-ISR. ההבדל בין iret ל-ret הוא שהוא pop נוסף, כדי לשוחזר גם את רегистר הדגלים. כפי שאפשר לראות, יש דמיון רב בין הגדרה של ISR להגדרה של פרוצדורה, ואנמנ אופן הפעולה שלהם דומה.

השאלה שאנו רוצים לענות עליה היא: המעבד קיבל פסיקה. עכשיו הוא צריך להפסיק את ריצת התוכנית ולקפוץ למקום כלשהו בזיכרון, שם אמינו שנמצא ה-ISR. איך הוא יודע לאיזו כתובת בזיכרון לקפוץ?

נתחיל בכך שלכל פסיקה יש מספר, וזהו מספר שמייחד אותה מיתר הפסיקות. מספר זה יכול להיות בין 0 לדיימלי, אך נהוג לקרוא לפסיקה לפי הערך הhexagonal שלו. לדוגמה, כשכתבנו את הפקודה:

```
int 21h
```

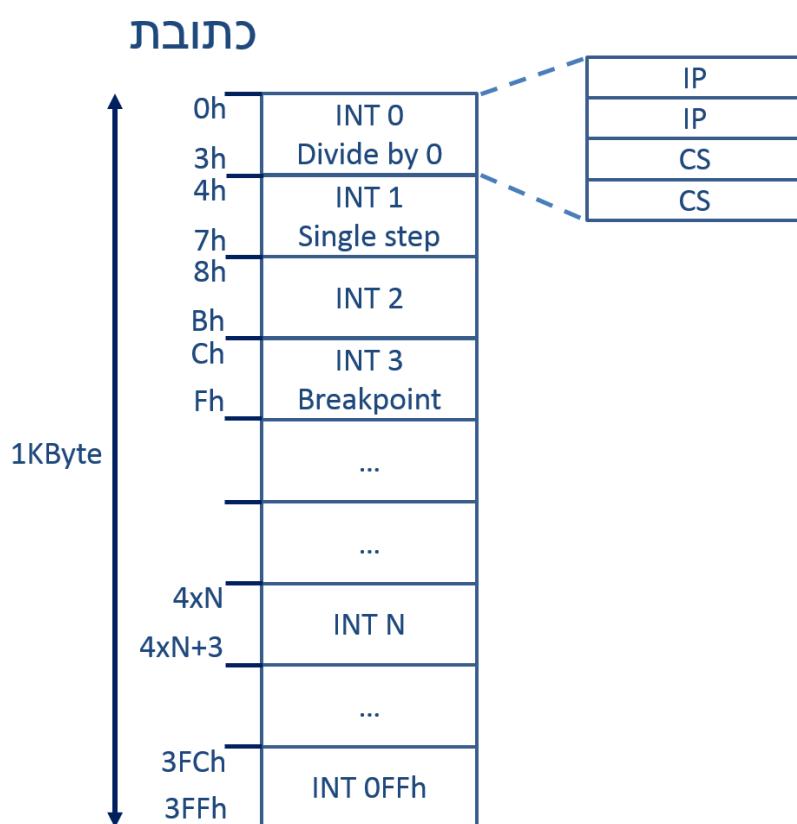
הודיענו למעבד שהוא צריך לשרת את פסיקה מספר 21h.

כעת המעבד ניגש למלאכת תרגום מספר הפסיקה לכטובה בזיכרון. לטובת פעולה התרגם, המעבד פונה לטבלת תרגום, ששומרה אצלו מראש בזיכרון. טבלה זו נקראת **Interrupt Vector Table**, או **IVT**.

ה-IVT הינה טבלה בגודל של 256 Double Words, כלומר 1024 בתים. תחילת ה-IVT היא בסגמנט 0 ובאופסט 0 בזיכרון, או במילים אחרות – ה-IVT נמצא ממש בתחום הזיכרון ותופס את הכתובות שבין 0 ל- $3FFh$ בתים בסך הכל).

איך ה-IVT עוזר למעבד לתרגם את מספר הפסיקה לכטובה בזיכרון? ב-IVT נמצאות הכתובות של 256 פרוצדורות שמצוינות בפסיקות, ISR'ים. כל כתובה מורכבת משתי מילימ. המילה הראשונה היא האופסט של ה-ISR והמילה השנייה היא הסגמנט של ה-ISR. כך לדוגמה, ארבעת הבטים הראשונים ב-IVT הם הכתובת של ISR של פסיקה מס' 0. ארבעת הכתובות הבאים אחריו הם הכתובת של ISR מס' 1 וכן הלאה.

כלומר, כדי לדעת מה הכתובת של ISR כלשהו בטבלה, לוקחים את מספר הפסיקה, כופלים ב-4 וההתוצאה היא הכתובת הפיזית של ה-ISR שהמעבד צריך לקרוא לפניו אליה. דוגמה: פסיקה $21h$ מיתרגרמת למקום $84h$ בזיכרון. המעבד ניגש למקום $84h$, שנמצא כموבן ב-IVT, וקורא ארבעה בתים, שמוררים לו לאיזו כתובה לקרוא לפניו לביצוע ה-ISR של פסיקה $21h$.



מבנה ה-IVT בזיכרון *Interrupt Vector Table*

פסיקות DOS

מערכת הפעלה DOS, קיצור של Disk Operating System, נכתבה על ידי חברת מיקרוסופט. מערכת הפעלה זו שלטה בעולם מערכות הפעלה עד אמצע שנות התשעים, או הוחלפה על ידי מערכת הפעלה Windows.

מערכת הפעלה מספקת לנו שירותים שונים. בין היתר היא מקשרת בין רכיבי החומרה במחשב לבין אפליקציות – תוכנות ומשחקים שריצים על המחשב שלנו. כך היא חוסכת עבודה למי שמתכונת אפליקציות, שבמוקם לגשת יישורות לרכיבי החומרה, מקבל שירותים מערכות הפעלה. יכולת זו גם מונעת התנגשות בין אפליקציות שצרכות לשימוש באותו הזמן רכיבים באותו זמן. לכן, אחד מהדברים שמערכות הפעלה מודרנית כוללת הוא אוסף של IRQ'ים, שיודעים לעבוד מול רכיבי חומרה שונים.

ל-IRQ'ים של DOS הוגדרו מספר מקומות קבועים ב-IVT – טווח הפסקות שבין 20h ל-2Fh.

Interrupt vector	Description
20h	Terminate program
21h	Main DOS API
22h	Program terminate address
23h	Control-C handler address
24h	Critical error handler address
25h	Absolute disk read
26h	Absolute disk write
27h	Terminate and stay resident
28h	Idle callout
29h	Fast console output
2Ah	Networking and critical section
2Bh	Unused
2Ch	Unused
2Dh	Unused
2Eh	Reload transient
2Fh	Multiplex

רשימת הפסקות שמורות ל-DOS בתחום ה-IVT

אחד מהפסיקות היא 21h, שמוגדרת כפסקת שירות של מערכת הפעלה. כתע, כל פעם שנרצה שירות של מערכת הפעלה נוכל לבצע זאת על ידי השורה `int 21h`. אין אפשרות בעורת פסקת תוכנה אחת לבצע את כל השירותים שמערכות הפעלה נותנת לנו? התשובה היא שלפני שאחנו קוראים ל-`int 21h` אנחנו שמים ברגיסטרים פרמטרים שקובעים מה מערכת הפעלה תבצע בשביבינו.

ספקית, הרегистר ah מחזק את סוג השירות שהוא אנחנו רוצים.

באטר <http://spike.scu.edu.au/~barry/interrupts.html> תוכלו למצוא רשימה מסודרת של כל השירותים שאפשר לקבל באמצעות הפעלה של int 21h ואת קוד השירות – ah – המתאים לכל אחד מהשירותים. אנחנו נסקור במסגרת ספר זה רק את השירותים השימושיים ביותר.

AH=1h – קליטתתו מהמקלדת

כדי לקבל בעזרה int 21h שירות של קריאתתו מהמקלדת, נשם בתוך ah את הקוד "1". כך:

```
mov ah, 1
int 21h
```

התו שיופיע ייטען בתוך ah.

יש לציין ש-ah יכול את ערך ה-ASCII של התו שהוקלד. לדוגמה, אם המשתמש הקליד "2", ah לא יכול, אלא 32h (כפי שהוא מופיע בטבלה למטה), שהוא ערך ה-ASCII של התו "2".

		Regular ASCII Chart	character	codes	0 – 127			
000	<nul>	016 ► <dle>	032 sp	048 0	064 ☒	080 P	096 `	112 p
001 ☐	<soh>	017 ▲ <dc1>	033 !	049 1	065 ☑	081 Q	097 a	113 q
002 ☑	<stx>	018 ♫ <dc2>	034 "	050 2	066 B	082 R	098 b	114 r
003 ♥	<etx>	019 !! <dc3>	035 #	051 3	067 C	083 S	099 c	115 s
004 ♦	<eot>	020 ¶ <dc4>	036 \$	052 4	068 D	084 T	100 d	116 t
005 ☈	<eng>	021 § <nak>	037 %	053 5	069 E	085 U	101 e	117 u
006 ☉	<ack>	022 - <syn>	038 &	054 6	070 F	086 V	102 f	118 v
007 ☊	<bel>	023 ¶ <eth>	039 ,	055 ?	071 G	087 W	103 g	119 w
008 ☋	<bs>	024 ↑ <can>	040 <	056 8	072 H	088 X	104 h	120 x
009 ☌	<tab>	025 ↓ 	041 >	057 9	073 I	089 Y	105 i	121 y
010 ☍	<lf>	026 <eof>	042 *	058 :	074 J	090 Z	106 j	122 z
011 ☎	<vt>	027 ← <esc>	043 +	059 ;	075 K	091 [107 k	123 {
012 ☏	<np>	028 ← <fs>	044 ,	060 <	076 L	092 \	108 l	124 :
013 ☐	<cr>	029 ↔ <gs>	045 -	061 =	077 M	093]	109 m	125 >
014 ☒	<so>	030 ▲ <rs>	046 .	062 >	078 N	094 ^	110 n	126 ~
015 ☓	<si>	031 ▼ <us>	047 /	063 ?	079 O	095 _	111 o	127 △

כדי ש-ah יוכל ממש 2, או כל ספרה אחרת שהמשתמש הקליד, הטכניקה המקובלת היא להחסיר ממנו את ערך ה-ASCII של התו "0" (ערך זה הוא 30h).

```
sub al, 30h
```

דוגמה לתוכנית שקולטת تو מהמשתמש ושומרת את ערך ה-ASCII שלו בתוך al:

IDEAL

MODEL small

STACK 100h

DATASEG

CODESEG

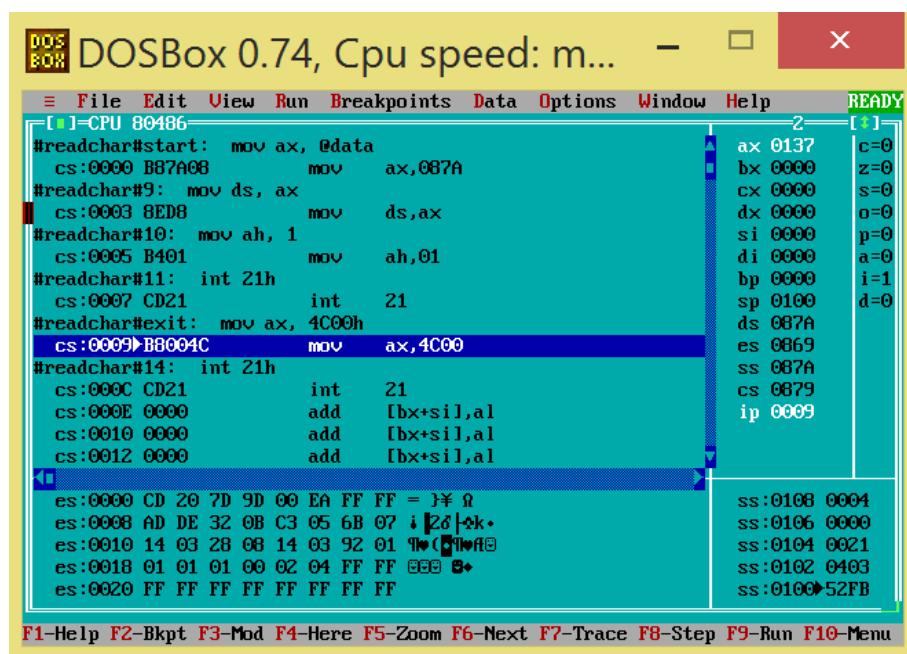
start:

```
    mov ax, @data
    mov ds, ax
    mov ah, 1
    int 21h
```

exit:

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

END start



המשתמש הזין את התוו 7 (קוד al)

הקוד הועתק לתוך al

תרגיל 11.1: קליטת تو מהמקלדת



- א.** כתבו תוכנית שקולטת שני תווים ובודקת לאיזה تو יש ערך ASCII גדול יותר.
- ב.** כתבו תוכנית שקולטת تو מהמקלדת ובודקת אם מדובר בספרה (כלומר ערך ה-ASCII נמצא בטוחה עברית ה-ASCII שבין 0 ל-9).
- ג.** כתבו תוכנית שקולטת עשרה תווים מהמקלדת, ואם כולם ספרות התוכנית מחשבת את סכום הספרות לתוכן הרגיסטר DL (טייפ: השתמשו בולולאה).
- ד.** אתגר: כתבו תוכנית שקולטת מספר בן 4 ספרות. התוכנית תקלוט אותו ספרה אחריו ספרה ותשמר ברגיסטר או משתנה כלשהו את ערך המספר שהתקבל. כדי לפשט את התוכנית, הניחו שמספרים בני פחות מ-4 ספרות ייקלטו עם אפסים בהתחלה (לדוגמה המספר 250 ייקלט כ-0250). נסו לכתוב את התוכנית באמצעות פהות מ-30 שורות קוד.

AH=2h – הדפסת تو למסך

כדי לקבל בעזרה int 21h שירות של הדפסת تو למסך, נשים בתוך ah את הקוד "2" ובתוך dl את התו אותו אנחנו רוצים להדפיס, או את קוד ה-ASCII שלו. לדוגמה כדי להדפיס למסך את התו 'X', שקוד ה-ASCII שלו הוא 58h:

```
mov    dl, 'X'           ; same as:    mov    dl, 58h
mov    ah, 2
int    21h
```

הפסיקה תנסה את ערכו של ah לעורך והוא האחרון שנשלח להדפסה.

קיימים שני קודי ASCII שימושיים, שאינם מדפסים תווים למסך אלא נתונים הוראות בקרה:

1. קוד 10, או Line Feed – 0Ah – מורה להתחליל שורה חדשה, הסמן נמצא באותה עמודה בה הייתה בשורה היישנה.

2. קוד 13, או Carriage Return – 0Dh – מורה לחזור לתחילת השורה.

הצירוף של שני הקודים האלה מאפשר לנו לרדת שורה ולהמשיך את ההדפסה מתחילת השורה.

דוגמה לתוכנית שמדפיסה 'X', יורדת שורה ומדפיסה 'Y':

IDEAL

MODEL small

STACK 100h

DATASEG

CODESEG

start:

 mov ax, @data

 mov ds, ax

 ;print x

 mov dl, 'X'

 mov ah, 2

 int 21h

 ;new line

 mov dl, 10

 mov ah, 2

 int 21h

 ;carriage return

 mov dl, 13

 mov ah, 2

 int 21h

 ;print y

 mov dl, 'Y'

 mov ah, 2

 int 21h

exit:

 mov ax, 4C00h

 int 21h

END start

DOSBox 0.74, Cpu speed: ...

```
C:\TASM\BIN>tasm /zi putchar.asm
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file: putchar.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 466k

C:\TASM\BIN>tlink /v putchar.obj
Turbo Link Version 7.1.30.1. Copyright (c) 1987, 1996 Borland International

C:\TASM\BIN>putchar
X
Y
C:\TASM\BIN>_
```

תרגיל 11.2: הדפסת تو למסך



- א. הדפיסו למסך את התו 'A'.
- ב. הדפיסו למסך את התו 'a'.
- ג. הדפיסו למסך את המילה 'HELLO', تو אחרתו.
- ד. הדפיסו למסך את המילה 'HELLO', تو אחרתו, בשורה מתחתייה.
- ה. כתבו תוכנית שקולטת שתי ספרות ומדפיסה למסך את הספרה הגדולה מביניהן.
- ו. כתבו תוכנית שקולטת שתי ספרות ומדפיסה למסך את החישור של השניה מהראשונה. אם התוצאה שלילית – התוכנית תדפיס סימן מינוס לפני התוצאה. עזרה: אם התוצאה שמתקבלת היא שלילית, הדפיסו סימן מינוס ואז היפכו את התוצאה והדפיסו את הספרה שמתתקבלת. לדוגמה – הספרה הראשונה היא 5 והשנייה 7. התוכנית תמצא שהמספר הראשון יותר קטן מהשני, תדפיס מינוס ולאחר כך את ההפרש בין המספר השני לראשון.

AH=9h – הדפסת מחרוזת למסך

כדי להדפיס מחרוזת למסך אנחנו צריכים קודם כל ליצור מחרוזת, שמשמעותה בתו '\$' (זה הסימן ל-ISR) להפסיק את הדפסה למסך – אחרת יודפס כל הזיכרונות...). לדוגמה:

```
message      db      'Hello World$'
```

לאחר מכן טוענים לתוך ax את האופסט של המחרוזת:

```
mov      dx, offset message
```

נותר לנו רק לחת את הקוד הנוכחי ולהפעיל את הפסיקה:

```
mov      ah, 9h
```

```
int      21h
```

ירידת שורה בסוף המחרוזת:

שיטה נוחה לכתוב מחרוזת שבסוף הדפסה שלא מתחילה שורה חדשה, היא להגדיר את תווי הבקרה בתוך המחרוזת:

```
message      db      'Hello World', 10, 13,'$'
```

תוכנית דוגמה:

IDEAL

MODEL small

STACK 100h

DATASEG

```
message      db      'Hello World',10,13,'$'
```

CODESEG

start:

```
    mov      ax, @data
    mov      ds, ax
    push   seg message
    pop    ds
    mov      dx, offset message
```

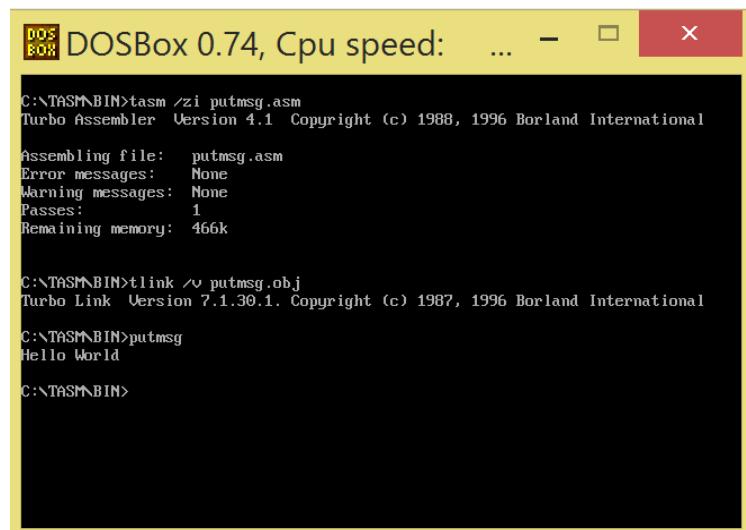
```

mov ah, 9h
int 21h

exit:
    mov ax, 4C00h
    int 21h

END start

```



תרגיל 11.3: הדפסת מהרוזת למסך



- א. כתבו תוכנית שמדפיסה למסך את ההודעה: 'Enter a digit' ואז קולטת ספרה מהמשתמש.
- ב. הרחיבו את התוכנית שכתבתם בסעיף הקודם, כך שאם המשתמש היזן تو שאינו ספרה בין 0 ל-9, יודפס למסך: "Wrong input".
- ג. כתבו תוכנית שמדפיסה למסך הודעה כלשהי, ובשוררה הבאה כותבת את שמהם. לדוגמה:

I like to write assembly code

Barak

קליטת מהרוזת תווים – AH=0Ah –

כדי לקלוט מהרוזת תווים מהמקלדת, צריך קודם כל להזכיר בזיכרון מקום לקליטת התווים - מקום זה נקרא "באפר" (Buffer). בתוך הבית הראשון של הבאפר צריך להכניס את מספר התווים המxisמי שאנו מאפשרים לקלוט. לאחר מכן מכנים לתוכך ah את האופט של הבאפר (יחסית ל-ds) ואז מפעילים את Int 21h כאשר ah שווה לקוד 0Ah. כשההמשתמש סיים

להקליד תווים ולחץ על מקש ה-Enter, הבית השני בבאפר יכול את מספר התווים שהוכנסו, ואילו התווים עצם יהיו מהמקום השלישי והלאה.



דוגמה לתוכנית שקולעת עד 20 תווים מהמשתמש:

(שים לב - אנחנו מזמנים 23 בתים, כיון שני בתיים בתחילת הבאפר חפוסים על-ידי מספר התווים המקורי ומספר התווים בפועל, והבית האחרון תפוס על-ידי קוד ה-ASCII של Enter :(

IDEAL

MODEL small

STACK 100h

DATASEG

message db 23 dup (?)

CODESEG

start:

```

    mov ax, @data
    mov ds, ax
    mov dx, offset message
    mov bx, dx
    mov [byte ptr bx], 21      ;21 not 20, the last input is ENTER
    mov ah, 0Ah
    int 21h

```

exit:

```

    mov ax, 4C00h
    int 21h

```

END start

להלן צילום מסך של הטעסט שהוכנס על ידי המשתמש:

```

DOSBox 0.74, Cpu speed: m...
C:\TASM\BIN>tasm /zi read20.asm
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International
Assembling file: read20.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 466k

C:\TASM\BIN>tlink /v read20.obj
Turbo Link Version 7.1.30.1. Copyright (c) 1987, 1996 Borland International

C:\TASM\BIN>td read20
Turbo Debugger Version 5.0 Copyright (c) 1988,96 Borland International
Hello world! -Barak

```

הטקסט נשמר במשתנה **message**, כפי שהוא ניתן לראות בתחום **ds**. הבית הראשון, **14h**, הוא גודל ההודעה המקסימלית שהתוכנית שלנו מאשרת. הבית השני, **13h**, הוא כמה התווים שהוכנסו בפועל, כולל ה-**enter**. הבית האחרון בהודעה שנראה כמו סימן של צליל, ערכו **0Dh** והוא קוד ה-ASCII של **enter**.

Register	Value	Register	Value
ax	0A7B	c	=0
bx	0000	z	=0
cx	0000	s	=0
dx	0000	o	=0
si	0000	p	=0
di	0000	a	=0
bp	0000	i	=1
sp	0100	d	=0
ds	087B		
es	0869		
ss	087D		
cs	0879		
ip	0011		

Registers (READY):

```

#read20:exit: mov ax, 4000h      ax 0A7B
          mov    cs:0011[B8004C]    bx 0000
#read20:17: int 21h             cx 0000
          add    [bx+sil],al     dx 0000
          add    [bx+sil],al     si 0000
          add    [bx+sil],al     di 0000
          add    [bx+sil],al     bp 0000
          add    [bx+sil],al     sp 0100
          add    [bx+sil],al     ds 087B
          adc    al,13           es 0869
          dec    ax               ss 087D
          insb   gs:              cs 0879
          insb   gs:              ip 0011
          outsw  6F               ss:0108 0005
          and    [bx+6F],dh       ss:0106 0000
          ds:0000 14 13 48 65 6C 6C 6F 20 ??Hello
          ds:0008 77 6F 72 6C 64 21 20 2D world! -
          ds:0010 42 61 72 61 6B 0D 00 00 BarakJ
          ds:0018 00 00 00 00 00 00 00 00
          ds:0020 00 00 00 00 00 00 00 00

```

תרגיל 11.4: קליטת מחזורת תווים



כתבו תוכנית שתקבלת עד 10 תווים מהמשתמש, באותיות קטנות (abc) ומדפיסה למסך את התווים באותיות גדולות (ABC).

תרגילים מסכימים- קלט פלט (קרדייט: פימה ניקוליבסקי)



11.5: כתבו תכנית המציגת על המסר את שם המשפחה ואת שמה הפרטני.

11.6: כתבו תכנית שקולטתתו בודד מהמקלדת ומציגת אותו 10 פעמים בשורה חדשה.

11.7: כתבו תכנית המציגת על המסר את הוצאות הבאות (כל צורה – תכנית חדשה!)

א.	*****	*****	*****	*****	*****
	****	****	*	****	****
	***	***	*	***	***
	**	**	*	***	***
	*	*	*****	***	***

11.8. כתבו תכנית המציגת על המסר את רצף של תווים הבאים:

ABCDEFGE....Z

11.9: כתבו תכנית ובה פעולות:

א. קליטת מספר שלם מן המקלדת. סוף הקלט לחיצה על מקש אנטר.

ב. להציג מספר שלם על המסר.

ג. קליטת מספר שלם שמתאר את מספר הנתונים שיש לקלוט למערך וקליטת מספרים למערך זהה.

ד. להציגו של ערכי מערך על המסר.

11.10 : כתבו תוכנית הקולטתתו אחריותו עד הלחיצה על המקש אנטר מהמקלדת ועובד כלתו מציגו הודעה מתאימה:

Small letter, Capital letter, Number, Other

11.11 : כתבו תוכנית הקולטת רצף של 5 תווים מהמקלדת "קוד סודי", במקומות כלתו שנקלט מוצגתו "*". על התוכנית לבדוק האם הקוד הוא: "12345" ואם כן - להציג הודעה מתאימה. אחרת, להודיע על הטעות ולאפשר עוד 2 ניסיונות.

AH=4Ch – יציאה מהתוכנית

קריאה ל-`int 21h` עם קוד `4Ch` גורמת לשחרור הזיכרון שתפקידו על-ידי התוכנית. כמו כן הפסיקה מעבירה למערכת הפעלה את מה ששמור בתוך `ah` בתור קוד זהה. נהוג לקבוע את ערכו של `ah` כאשר במכשיר והתקנית הסתימה בהצלחה, ולכן יציאה סטנדרטית מתוכנית נראית כמו שורות הקוד המוכנות لكم מ-`base.asm`:

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

קריאה השעה / שינוי השעה – AH=2Ch ,AH=2Dh (הרחבת)

לצד המעבד קיימ רכיב חומרה שתפקידו לתחזקון למעבד – הטימר. הטימר שולח למעבד אותן חמומי כל 55 מילישניות (0.055 שניה), או בערך 18.2 פעמיים בשניה. لكن הוא מכונה גם "שעון 1/18 שניה".

כדי לקרוא את השעון אפשר להשתמש בשירות של DOS, חלק מ-`int 21h`, ולשלוח לו את הקוד `:2Ch`

```
mov ah, 2Ch
int 21h
```



תוצאת הקריאה תהיה:

- `ch` השעות יועתקו לתוך `ah`.
- `dl` דקות השניה יועתקו לתוך `al`.

שימוש לב לך שערך דקות השניה מתעדכן רק כל 55 מילישניות. ככלمر אם קראנו את השעון פעמיים, ובין הקריאה האלו עברו פחות מ-55 מילישניות, יכול להיות שנקלע את אותו הערך.



שירות נוסף של DOS מאפשר לנו לקבוע את ערכו של השעון, באמצעות קריאה ל-`int 21h` עם ערך `ah=2Dh`. פסיקת התוכנה תיקבע את ערכו של הטימר לפי הפרמטרים ברегистרים השונים:

- `ah` יועתק לתוך הדקות.
- `dl` יועתק לתוך דקות השניה.
- `ch` יועתק לתוך השעות.
- `dh` יועתק לתוך שעות.

תוכנית לדוגמה – קריאת הטימר והדפסת השעה למסך:

התוכנית הבאה משתמשת ב-int 21h כדי לקרוא את ערכו של הטימר. לאחר מכן נעשה שימוש בפראצ'דורה שמיירה את הערךם שברגיסטרים לקוד ASCII של ספרות ומדפסה אותם למסך.

```

; -----
; Print time to screen
; Author: Barak Gonen 2014
; Credit: www.stackoverflow.com (printing-an-int, by Brendan)
; -----



IDEAL

MODEL      small
STACK      100h

DATASEG

    hourtxt    db     'Hour:  ','$'
    mintxt     db     13,10,'Mins:  ','$'
    sectxt     db     13,10,'Sec:  ','$'
    mstxt      db     13,10,'1/100sec: ','$'
    savetime   dw     ?
    divisorTable db     10,1,0

CODESEG

proc printNumber
    push ax
    push bx
    push dx
    mov bx,offset divisorTable

nextDigit:
    xor ah,ah
    div [byte ptr bx]           ;al = quotient, ah = remainder

```

```

add    al,'0'

call   printCharacter      ;Display the quotient

mov    al,ah                ;ah = remainder

add    bx,1                 ;bx = address of next divisor

cmp    [byte ptr bx],0       ;Have all divisors been done?

jne    nextDigit

pop    dx

pop    bx

pop    ax

ret

endp  printNumber

```

```

proc  printCharacter

push  ax

push  dx

mov   ah,2

mov   dl, al

int   21h

pop   dx

pop   ax

ret

endp  printCharacter

```

start:

```
mov   ax, @data
```

```

mov ds, ax
mov ah, 2ch
int 21h ;ch- hour, cl- minutes, dh- seconds, dl- hundredths secs
mov [savetime], dx
; print hours
mov dx, offset hourtxt
mov ah, 9
int 21h
xor ax, ax
mov al, ch
call printNumber;
; print minutes
mov dx, offset mintxt
mov ah, 9
int 21h
xor ax, ax
mov al, cl
call printNumber
;print seconds
mov dx, offset sectxt
mov ah, 9
int 21h
xor ax, ax
mov dx, [savetime]
mov al, dh
call printNumber

```

```

;print 1/100 seconds

mov dx, offset mstxt

mov ah, 9

int 21h

xor ax, ax

mov dx, [savetime]

mov al, dl

call printNumber

quit:

mov ax, 4c00h

int 21h

END start

```

תרגיל 11.12: טיימר



- א.** כתבו תוכנית שמדפסה למסך את הספרה 0, ולאחר מכן מדפסה למסך את הספרה 1. שימו לב לכך שיתכן ששעון השניות ישנה למרות שעדיין לא עברה שנייה מהפעם האחרון שקרהתם אותו: לדוגמה, בזמן שהדפסתם את הספרה 0 כמות המילישניות הייתה 960, לאחר עדכון של השעון אחרי 55 מילישניות בלבד ערך השניות התעדכן.
- ב.** כתבו תוכנית שמדפסה למסך את השעה, הדקה והשניות. המספר יעודכן בכל מעבר של שנייה. המוכנית תסתדרים בלבד לאחר פרק זמן מוגדר כרצונכם.

פסיקות חריגה – Exceptions

פסיקות החריגה נמצאות בתחום ה-IVT. פסיקה מסוג exception מתרחשת אוטומטית כתוצאה מאירוע תוכני שהתנה לא ים אותו – בדרך כלל אירוע חריג, ולכן נקראות פסיקות חריגה. נסקור כמה דוגמאות לפסיקות חריגה. אם נבצע **חילוק באפס**, תופעל אוטומטית פסיקת חריגה.

נסו להריץ את קטע הקוד הבא:

```
mov cl, 0
div cl      ; ax=al/cl □ ah= al / cl    al= al % cl
```

האSEMBLER יכול לmpl את התוכנית בלי בעיה, כיוון שבchinתו כל אחת מהפקודות שרשמתם היא חוקית. אך בשלב הריצה התוכנית תיעצר ותופעל פסיקת חריגה של חילוק באפס.

פסיקה זו, של חילוק באפס, נקראת פסיקה מס' אפס או **.int**.

דוגמה שנייה לפסיקת חריגה היא כאשר אנחנו ב**debugger** וMRIIZIM את הקוד שלנו **שורה אחרי שורה**. צריך להיות מנגנון כלשהו שעוזר את ריצת התוכנית אחרי כל פקודה. מנגנון זה הוא פסיקת חריגה, שמשנה את ערכו של אחד הדגלים ברגייסטר הדגלים. המעבד יודע האם דגל זה מופעל, עליו לעזור לריצה עד לקבלת פקודה "המשך".

פסיקה זו, של הריצת צעד בודד, היא **1h.int**.

דוגמה נוספת היא השימוש ב-**breakpoints** בתוכן ה-**debugger**. כל breakpoint גורם למעבד להריץ את הפקודות בלי עצירה עד להגעה לשורת קוד שיש בה breakpoint ואז מתבצע עצירה עד לקבלת פקודה "המשך".

פסיקה זו, של ביצוע breakpoint, היא **3h.int**.

פסיקות תוכנה – Traps

בניגוד ל-**exceptions**, שמרתחשות אוטומטית, מי שיוזם פסיקות תוכנה הוא מי שכותב את התוכנית. הדרך שבה מופעלת

פסיקת תוכנה היא פשוטה ביותר, ועקרונית לא צריך לדעת הרבה בשביב לעבד עם פסיקות תוכנה:

כותבים פקודה **int** ואחריה אופרנד, שהוא מספר הפסיקה:

```
int operand
```

לדוגמה:

```
int 80h
```

מכאן והלאה התהילה של ביצוע פסיקת התוכנה הוא זהה לתהילה שתואר עד כה – שמירת המיקום והדגלים במחסנית, חישוב כתובה ה-**ISR** בعزרת ה-IVT וקפיצה למיקום ה-**ISR**.

בשביל מה בכלל צריך פסיקות תוכנה? הרי אם התוכנת רוצה שבנקודה ידועה בתוכנית המעבד יקפוץ למקום אחר, יבצע את הקוד שכתוב שם ויזוזר – אפשר פשוט להשתמש בפרוצדורה.

התשובה היא, שיש מצבים שבהם לא עשוי להשתמש בפ逻צדורה ובמצבים אלו פסיקת תוכנה היא שימושית. פסיקות תוכנה הון דרך נוחה לתוכנה שלנו להריץ קוד של תוכנות שרצות במקביל לתוכנית שלנו – הדוגמה הבולטת ביותר היא מערכת הפעלה. כשהתוכנה שלנו רוצה להשתמש בשירותים של מערכת הפעלה, היא לא יודעת את הכתובות של הפ逻צדרות של מערכת הפעלה. תיאורטיות היינו יכולים לפתור את הבעיה זו אם היינו מengl'ים את התוכנה שלנו יחד עם הקוד של מערכת הפעלה, אבל אופציית זו לא מעשית. לכן בפועל מערכת הפעלה מדירה את הפ逻צדרות שלה כפסיקות ומנכינה ל-IVT את המיקום בויכרנו אליו צריך לקפוץ. כתע כל מה שנשאר לתוכנה שלנו לעשות הוא לקרוא לפסיקת תוכנה וה-IVT כבר יdag "להקפי" את הקוד שלנו למיקום הנכון.

בסעיף הבא נלמד לכתוב Trap.

כתיבת ISR (הרחבת)



כדי להציג את שלבי כתיבת ה-ISR נכתב ISR פשוט מסוג Trap, שככל מה שהוא עושה זה להדפיס למסך 'Hello World'.

שלב א' – כתיבת ה-ISR.

נתחיל מ-ISR ריק:

```
proc SimpleISR far
    ...
    iret
endp SimpleISR
```

מוסיף ל-ISR שלנו את ההודעה אותה אנחנו רוצים להדפיס.

שים לב, שההודעה צריכה לבוא אחרי פקודת ה-`iret`, לאחרת המעבד עלול להגיע למצב שהוא מנסה לתרגם את ההודעה ל-`opcodes` ולהריץ אותם, מה שעלול להיגמר בקריסה.



```
proc SimpleISR far
    ...
    iret
    message db 'Hello World$'
endp SimpleISR
```

מוסיף קריאה ל-`int 21h`. הקוד שמדפיס מהירותו הוא `ah=9h`, ולפני הקריאה צריך לדאוג לכך שב-`ds` יהיה הסגמנט של המחרוזת וב-`dx` יהיה האופט של המחרוזת.

```
proc SimpleISR far
    mov dx, offset message
    push seg message
```

```

pop    ds
mov    ah, 9h
int    21h
iret
message db  'Hello World$'

endp SimpleISR

```

נוסיף ל-ISR פקודות שיגרמו לכך שבסוף הריצזה מצב הרגיסטרים יהיה כמו לפני הקריאה. גם בלי פקודות אלו ה-ISR יעבדו, אבל התוכנית שקוראת לו עלולתה לא לתפקיד כמו שצורך אחרי שהשליטה תוחזר אליו.

```

proc SimpleISR far
    push   dx
    push   ds
    mov    dx, offset message
    push   seg message
    pop    ds
    mov    ah, 9h
    int    21h
    pop    ds
    pop    dx
    iret
message db  'Hello World$'

endp SimpleISR

```

זהו, סיימנו את כתיבת ה-ISR.

שלב ב' - שתילת כתובת ה-ISR ב-IVT

כדי שה-ISR יעבדו, הוא צריך להיות מוכר על-ידי ה-IVT. כמובן, אנחנו צריכים להוסיף את כתובת ה-ISR לתוך ה-IVT. נבחר לשתול את כתובת ה-ISR במקום האחרון ב-IVT, מקום (255, OFFh). הסיבה שבחורנו דזוקא במקום זה, היא שאנחנו רוצים לשתול את ה-ISR שלנו במקום שאינו תפוס על-ידי כתובת של ISR בשימושו, ובסיום ה-IVT יש מקומות פנויים (חישבו - מה היה קורה אילו היינו שותלים את ה-ISR שלנו במקום (21h)?).

כדי לשלול ב-IVT את הכתובת של ה-ISR החדש שלנו, נשתמש בשירותת נוספת `int 21h`, שירות שזהה הזמן המתאים לסקור אותו. קוד `AH=25h` מכניס כתובת לתוך ה-IVT. מספר הפסיקה צריך להיות בתוך `al`, וככתובת ה-ISR בתוך `dx:ds`. שורות הקוד הבאות מבצעות את הפעולות הדרישות:

```
mov al, 0FFh           ; The ISR will be placed as number 255 in the IVT
mov ah, 25h            ; Code for int 21h
mov dx, offset SimpleISR ; dx should hold the offset of the ISR
push seg SimpleISR
pop ds                 ; ds should hold the segment of the ISR
int 21h
```

זהו. נשאר לנו רק לקרוא ל-`int 0FFh` מתוך התוכנית. להלן התוכנית המלאה:

IDEAL

```
MODEL small
STACK 100h
```

DATASEG

CODESEG

```
proc SimpleISR far
    push dx
    push ds
    mov dx, offset message
    push seg message
    pop ds
    mov ah, 9h
    int 21h
    pop ds
    pop dx
```

```

    iret

    message      db      'Hello World$'

endp  SimpleISR

```

start:

```

    mov  ax, @data

    mov  ds, ax

; Plant SimpleISR into IVT, int 0FFh

    mov  al, 0FFh

    mov  ah, 25h

    mov  dx, offset SimpleISR

    push seg SimpleISR

    pop  ds

    int  21h

; Call SimpleISR

    int  0FFh

exit:  mov  ax, 4c00h

    int  21h

END start

```

תרגיל 11.13: כתיבת ISR



א. כתבו ISR שמקבל בתוקן al ערך ASCII של תו, מעלה אותו באחד ומדפיס למסך את התו החדש. גירמו לכך שהוא יופעל על-ידי הפקודה `.int 0FEh`.

ב. כתבו ISR שמקבל שני רגיסטרים bx, ax ומדפיס למסך:

`-ax' אם ax גדול מ-bx'`

`-bx' אם bx גדול מ-ax'`

- SAME' אם הם שווים.

גirmo לך שהוא יופעל על ידי הפקודה 0F0h.int.

תרגיל אתגר- פיצוח צופן הזזה

נבער תרגיל שהוא חזרה על החומר העיקרי שלמדנו עד כה (פרוצדורות ושימוש בפסיקות DOS לקלט ופלט) ותוך כדי נלמד מעט על צפנים ופיצוח צפינים.



נתחיל בהסביר קצר על צופן הזזה. צופן הזזה הוא צופן עתיק בו כל אות מוחלפת באות אחרת, שנמצאת במרקם קבוע מהאות המקורי (מרקם הזזה). לדוגמה צופן הזזה 1- האות b מוחלפת באות c, האות d מוחלפת באות e וכו' עד שmagivim לאות z, אשר מוחלפת באות a. לדוגמה, בצופן הזזה במרקם 3, המילה cat מוחלפת במילה wfd.

1. כתבו פרוצדורה שמקבלת מהירות ומרקם הזזה, ומיפוי את המחרוזת לפי מרקם הזזה. תוכי רוחה יש להשאיר כמו שם.

2. לפיכם העמוד הראשון מתוך הספר הנפלא Anna Karenina מאת טולסטי. כדי להקל על תהליך ההצפנה הכתוב מובא רק עם אותיות קטנות וללא סימני פיסוק. הצפינו אותו בעזרת צופן הזזה. שימו לב לכך שבסוף העמוד ישנו סימן '\$'. סימן זה נועד גם להקל עליכם למצוא את סוף המחרוזת וגם להקל עליכם בהדפסה של התוצאה. הדפיסו למסך את הטקסט המוצפן.

all happy families resemble one another every unhappy family is unhappy in its own way
 all was confusion in the house of the oblonskys
 the wife had discovered that her husband was having an intrigue with a french governess who had been in their employ and she declared that she could not live in the same house with him
 this condition of things had lasted now three days and was causing deep discomfort not only to the husband and wife but also to all the members of the family and the domestics
 all the members of the family and the domestics felt that there was no sense in their living together and that in any hotel people meeting casually had more mutual interests than they the members of the family and the domestics of the house of oblonsky
 the wife did not come out of her own rooms
 the husband had not been at home for two days
 the children were running over the whole house as if they were crazy
 the english maid was angry with the housekeeper and wrote to a friend begging her to find her a new place
 the head cook had departed the evening before just at dinner time
 the kitchen maid and the coachman demanded their wages\$

3. כתבו פרוצדורה שמקבלתתו ומחזירה כמה פעמים מופיעתו בתוך המחרוזת.

4. באמצעות הפרוצדורה שכתבتم, סיפו כמה פעמים מופיע כל אות באلف' בית הלועזי בטקסט המוצפן. פעולה זו נקראת ניתוח תדיירויות. הדפיסו למסך את התוצאה עבור כל אות.

5. צרו פרוצדורה שמקבלת את ניתוח התדיירויות שעשיתם, ובאמצעות טבלה של ניתוח תדיירויות שבשפה האנגלית מחליטה מה הייתה המקורית שהוצפנה. מצורפת טבלה של ניתוח תדיירויות אחרות בשפה האנגלית (לדוגמה, האות E היא 12.02% מהאותיות בשפה האנגלית. האות T היא 9.1% וכו'). שימו לב שככל שטקסט שבחרתם ארוך יותר כך האחוזים קרובים יותר לאותיהם שבטבלה:

Letter	Frequency (%)
E	12.02
T	9.10
A	8.12
O	7.68
I	7.31
N	6.95
S	6.28
R	6.02
H	5.92
D	4.32
L	3.98
U	2.88
C	2.71
M	2.61
F	2.30
Y	2.11
W	2.09
G	2.03
P	1.82
B	1.49
V	1.11
K	0.69
X	0.17
Q	0.11
J	0.10
Z	0.07

סיכום

התחלנו מהסביר על הצורך בפסיקות, ועל כך שלא תמיד אפשר לצפות מראש מתי התוכנה תצורך לבצע פעולה מסוימת. לאחר מכן למדנו על ה-ISR, הפרוצדורה לטיפול בפסיקות, ועל ה-IVT, המערך בזיכרון שמכיל את המידע לגבי מקום כל ה-ISRs'ם.

למדנו על פסיקה 21h של מערכת הפעלה DOS. סקרנו את הקודים השימושיים לביצוע פעולות שונות כגון קליטה של تو ומחרוות, הדפסה של تو ומחרוות וקריאה השעה מהשעון הפנימי של המחשב. ראיינו איך כותבים ISR ואיך שותלים אותו ב-IVT.

עסכנו בצורות שונות של פסיקות:

- פסיקות תוכנה (Traps)

- פסיקות חריגה (Exceptions)

על הסוג האחרון של הפסיקות, פסיקות חומרה (Interrupts), נפרט בפרק הבא.

פרק 12 – פסיקות חומרה (הרחבה)



מבוא

כשענסקנו בנושא הפסיקות ראיינו שקיימות פסיקות DOS SMB ציעות פעולות שונות מול רכיבי חומרה, לדוגמה קריית תווים מהמקלדת. כמו כן למדנו להשתמש בכמה פסיקות DOS שימושיות. עם זאת, לא ענסקנו בשאלת איך המידע מהתקני החומרה מגיעה אל המעבד? איך, לדוגמה, הקשה על המקלדת גורמת להופעת תווים בזיכרון המחשב? בפרק זה נבין יותר לעומק את הדרך שבה המעבד מתקשר עם התקני החומרה.

כדי להבין זאת אנו נלמד:

- תיאוריה של פסיקות חומרה – מדוע צריך פסיקות חומרה ובאיו דרכן מגיעות פסיקות החומרה אל המעבד.
- פורטים I/O – תאים מיוחדים בזיכרון ששמשים לעובודה מול התקני קלט / פלט.

לאחר מכן, נעבר לדוגמה מעשית על עובודה מול התקני חומרה ונפרט אודות פעולה המקלדת, תוך סקירת הדרכים השונות לעובודה מולה:

- קליטת המידע דרך הפורטים של המקלדת
- שימוש בפסיקות BIOS
- שימוש בפסיקות DOS

פסיקות חומרה – Interrupts

המעבד שלנו מקשור להתקני חומרה שונים כגון מקלדת, עכבר, שעון המערכת (טיימר) ועוד. התקנים הייחודיים אלו מייצרים אירועי חומרה. אירוע חומרה יכול להיות לחיצה על המקלדת, הזזה העכבר או עדכון של הטימר. אירועים החומרה האלו דורשים שירות מצד המעבד – לדוגמה, לנקחת את התו שהוקלד במקלדת ולהדפיס אותו למסך, להזיז את הסמן של העכבר על המסך או לעדכן את השעה שמוצגת על המסך. מבחינת המעבד, כל האירועים הללו בלתי צפויים מראש – המעבד לא יכול לתכנן מראש متى תהיה לחיצה על המקלדת ובאיזה קצב המשמש יקליד תווים. גם מי שכותב את התוכנה לא יכול לדעת מראש באיזה תזמון יקרה אירוע חומרה ולהתכוון אליו. אם כך, איך המידע שmagיע מהתקנים הייחודיים מועבר לתוכנה בזמן ריצה?

קייםות שתי גישות לפתורן הבעיה.

הגישה הראשונה נקראת **משיכה** – **Polling**. בשיטה זו, אחת לזמן מוגדר מראש, התוכנה שרצה על המעבד שואלת כל התקן חומרה אם יש לו מידע חדש. כך המעבד עובר התקן אחריו התקן, בצורה מעגלית, ובודק אם יש



התיקן חומרה שצורך שירות. המקלט צריכה שירות? אם כן – המעבד מטפל בה, אם לא – ממשיך לבדוק אם העכבר צריך שירות. כך המעבד עובר הלאה על כל התקנים עד שהוחזר אל התיקן הראשוני.

היתרון של Polling הוא שזו גישה פשוטה יחסית למימוש, ולא צריך רכיבי חומרה נוספים שיעזרו לנו בתהליך (להבדיל מפסיקות חומרה, שנגיעה אליהן מיד). החיסרון של Polling הוא שהתיקן החומרה צריך להמתין שיפנו אליו. כמובן יש עיכוב בין הזמן בו התקן החומרה צריך טיפול לבין הזמן שהמעבד פונה אל התקן החומרה ובודק אם יש אירוע שדורש טיפול. יש מקרים שבהם העיכוב הזה עלול להיות עז, מקרים שבהם הפעולה התקינה של תוכנה שלנו תלויה בזמן שלוקח לה לשרת אירועים מהתיקן חומרה כלשהו. לדוגמה, במערכות SMB3.0 מסחר אלקטронני במניות, יכולים להיות הבדלים של פרקי זמן קצרים ביותר בין הוראות קניה ומכירה והתוכנה חייבת לדעת מה הסדר הנכון של ההוראות. לכן, עליה לשמור על קצב גבוה של טיפול בפנויות מהטיימר. דוגמה אחרת – נניח שיש מערכת SMB3.0 ברגע שימושו לווחץ על כפתור במקלדת, במקרה זה מובן שלמהירות הטיפול באירוע הלחיצה על המקלט יש חשיבות רבה. פתרון פשוט הוא להעלות את קצב התשאול של התקני החומרה. נניח שבמוקם לשאול את המקלט 10 פעמים בשניה אם יש מקש לחוץ, התוכנה תישאל את המקלטים 1000 פעמים בשניה אם יש מקש לחוץ. שינוי זה יוריד את הזמן שהמקלט ממתינה לשירות מעשייתנית שנייה לפחות שנייה. הבעיה היא, שככל שהתוכנה שלנו מתחאלת את רכיבי החומרה לעיתים קרובות יותר, כך היא מבזבזת זמן רב יותר בבדיקה של התקנים, שלאروب לא יהיה להם צורך בשירות. ככל התוכנה שלנו תגרום לכך שהמעבד יקצת חלק משמעותית מהזמן לביקורת התקני חומרה ולא לפועלות אחרות שאנחנו רוצים שיבצעו.



גישהPolling – אחת לזמן מוגדר, חומרה שואלת כל תלמיד אם יש לו שאלת. כל עוד חומרה לא פונה אליהם, התלמידים מהיכים.

הגישה השנייה היא פסיקת חומרה, שנקראת **אינטרupt** (Interrupt). מעכשו כשנכתווב "אינטרupt" נתכוון לפסיקת חומרה. כדי לחשוף להתקן החומרה צורך בשירות של המעבד, התקן החומרה שולח אותה חשמלי שmagיע אל המעבד וגורם לו לעצור לאחר סיום הפעולה הנוכחית שלו ולבצע קוד, ISR, שנכתב מראש לטיפול באירוע החומרה. בסיום ISR התכנית תמשך בריצתה. היתרון של Interrupt הוא שהוא שומר על גודל שווה נמוך (בקשות של התקני חומרה מטופלות עם שייחי נמוך יחסית) וגם חסכווי במשאבי מעבד – כל עוד לא מגיע אינטרupt, המעבד פשוט מריזץ את הקוד שנקבע לו להריצ' בלי לדאוג לגבי התקני החומרה.

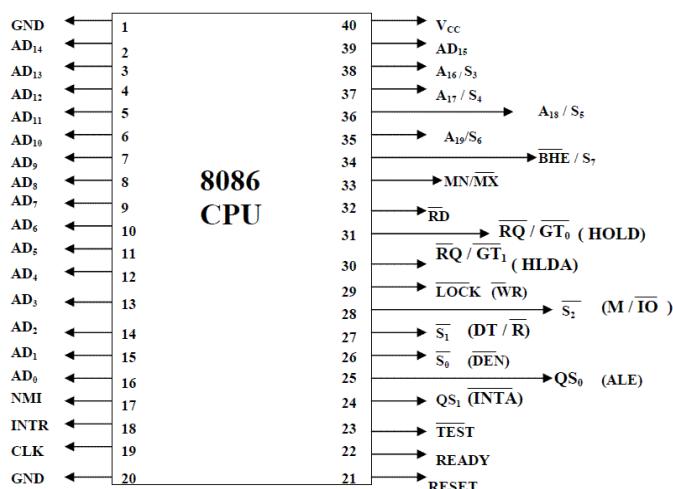




גישת תלמיד שיש לו שאלת מרים יד. המורה עוצרת את השיעור וממשיכה את השיעור מהמקום שהפסיקה אותו.

כמו שתלמיד שיש לו שאלת מרים יד, רכיב החומרה שצריך טיפול שלו הוא חשמלי. לمعالג יש "רגליים", או "פינים" – חוטי מתכת דקים שמחברים אותו אל העולם החיצוני. שנייה המתה החשמלי על אחת הרגליים גורם לمعالג להניע תהליך של אינטראפט.

Pin Diagram of 8086



דיagramת פינים של מעבד ה-8086

핀 מס' 18, שמוסמן "INTR", מקבל אינטראפטים מרלבבי חומרה חיצוניים

בקר האינטראפטים – PIC

כפי שראינו באирו של מעבד ה-8086, יש למעבד רgel אחת שמיועדת לאינטראפטים. בעזרתו הרgel הzo המעבד מקבל אינטראפטים מכל התקנים שקשורים אליו. איך בעזרתו רgel אחת המעבד יכול להיות קשור למספר רב של התקני חומרה?

Programmable Interrupt הוא **בקר האינטראפטים**, **PIC Controller**, או בקיצור **PIC**.

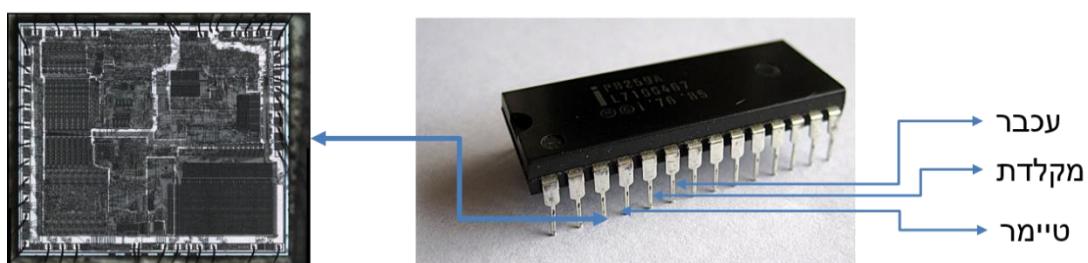


תמונה של PIC מדגם 8259A משנת 1976, מהסוג ששימש ליבורה עם מעבד ה-8086.

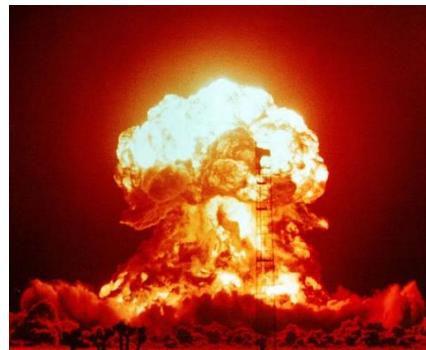
כפי שאפשר לראות בתמונה, ל-PIC יש אוסף של רגליים. הרגליים שמסומנות IR0 עד IR7, בסך הכל 8 רגליים, כאשר כל אחת מהן יכולה להיות מקושרת לרכיב חומרה בודד. כלומר PIC מדגם 8259A מסוגל לעבוד עם עד 8 רכיבי חומרה במקביל. לדוגמה – רgel IR0 מחוברת לטימר, רgel IR1 מחוברת למקלדת, רgel IR6 מחוברת לכונן דיסקטים, ורgel IR2 מחוברת לעכבר (לא באופן ישיר, אבל לצורך הפשטות נכוון להגיד שאינטראפטים מהעכבר מגיעים ל-IR2).

ל-PIC יש גם רgel שנקראת INT. רgel זו מחוברת חשמלית אל הרgel של המעבד שנקראת INTR. כעת אפשר להתבונן בمسلسل שעושה אינטראפט. ניקח בתור דוגמה אינטראפט של המקלדת. האות החשמלי עובר מהמקלדת אל רgel IR1 של PIC. כתגובה, ה-PIC מוציא את החסמי מרgel INT, שמגיעה לרgel INTR של מעבד ה-8086.

מעבד



עד עכשו תיארנו את ה-PIC כגורם הקשור בין מספר רכיבי חומרה למעבד. אולם ה-PIC לא רק הקשור, אלא גם קובע עדיפות לטיפול בתключи החומרה. דמיינו מערכת שמקושרת לשני התקני חומרה: מקלדת, וכור גרעיני. המשתמש הקיש על המקלדת והכור הגרעיני מדוחה על בעיה... במאי המעבד צריך לטפל קודם?



الطיפול לא מתחבץ בשיטת כל הקודם זוכה. ה-PIC מחזק תור של אינטראפטים שמחייבים לטיפול. אם הגיע אינטראפט חדש, הוא לא יזחף לסוף התור אלא ייכנס לתור בהתאם לעדיפות שלו. ככל שהאינטראפט הגיע מרגל בעלת מספר נמוך יותר, כך העדיפות שלו גדולה.

עם קבלת אינטראפט ה-PIC מבצע את הפעולות הבאות:

- שולח למעבד אות חממי שמסמן שיש אינטראפט.
- שולח מידע שקורא לאינטראפט לאזור מיוחד בזיכרון המעבד, אזור שנקרא Port O/I או בקיצור – פורט.
- מפסיק לשולוח אינטראפטים למעבד.
- ממחה לאות חממי end of interrupt מהמעבד, ובינתיים שומר בתור אינטראפטים חדשים שמנגנים אליו מרכיבי חומרה.
- חוזר לשולוח אינטראפטים למעבד.

אובדן אינטראפטים

כזכור, כשהמעבד מטפל בפסיקות הוא מבצע פעולה של disable interrupts על-ידי איפוס דגל הפסיקות. פעולה זו היא בעלת משמעות מיוחדת עבור התקני חומרה. ברגע לפסיקות תוכנה, שתתוכניתה מפעילה באופן צפוי מראש (כיוון שהתוכנתה קבועה באופן קבוע), פסיקות חומרה מגיעה בזמן לא צפויים מראש. מה קורה אם בזמן שהמעבד מטפל באינטראפט מגיע אינטראפט נוסף? לדוגמה – לחצנו על המקלדת והעכבר בהפרש זמן קצר ויצרנו אינטראפט אחד מהמקלדת ואינטראפט אחד מהעכבר. האינטראפט מהעכבר מתרחש בזמן שבו המעבד כבר מטפל באינטראפט של המקלדת וחוסם אינטראפטים נוספים. האם האינטראפט של העכבר "ילך לאיבוד"?

ובכן, כפי שציינו קודם, ה-PIC מרכז את האינטראפטים של כל רכיבי החומרה ושומר אותם אצלו בתור. בדוגמה שלנו, עם סיום האינטראפט של המקלדת, ה-PIC יעביר למעבד את האינטראפט של העכבר. התור של ה-PIC עוזר בכך שבו מספר

איןטרפטים הגיעו בפרש זמינים קצר. לעומת זאת, אם המעבד לא מצליח לעמוד בקצב הפסיקות, התור של ה-PIC יילך ויתארך עד לנקודת מסויימת בה לא יהיה ל-PIC מקום לשמר אינטרפטים חדשים והם ילכו לאיבוד.

דוגמה תיאורטית – שעון המערכת, הטיימר, שולח עדכון שעה כל 55 מילישניות. נדמיין מעבד איטי, שלוקח לו יותר מ-55 מילישניות לבצע את אינטרפט הטיימר. לאחר זמן מה אינטרפטים מהטיימר ילכו לאיבוד ופעולתו התקינה של המעבד תשתחש.

I/O Ports – קלט / פלט זיכרון

בפרק אודות מבנה המחשב סקרונו את הפסים (buses) השונים שיש למעבד, ומשמשים אותו לתקשורת עם הזיכרון ועם רכיבי הקלט / פלט. בין היתר, סקרונו את פס המענים – address bus – שמשמעותו כל בית בזכרון לכתובת.

למעשה, למעבד יש שני פסי מענים. נוסף על פס המענים המשמש לגישה לזכרון, משפחת ה- 86×80 כוללת פס מענים מיוחד ונפרד בגודל 16 ביט, המשמש לתקשורת עם רכיבי חומרה. את הזיכרון המשמש לתקשורת עם רכיבי חומרה מכנים גם זיכרון קלט / פלט, או O/I (קיזור של Input / Output).

התקשורת עם זיכרון O/I עובדת בצורה דומה מאד לתקשורת עם הזיכרון הרגיל, למעט מספר הבדלים:

- כתובות של זיכרון O/I נקראת **פורט (Port)**.
- כתובות של זיכרון O/I מיצג עליידי 16 ביט בלבד (כלומר יש בסך הכל 64K פורטים).
- במקומות פקודות mov, משתמשים בפקודות in ו-out.
- פס הבקרה דואג שפקודות in ו-out יעבירו כתובות על פס המענים של ה-O/I (לעומת פקודות mov, שמתיחסות לפס המענים של הזיכרון).

פקודות out, in: פקודה in משמשת לקריאה מפורט, ופקודה out משמשת כתיבת הפורט. פקודה out מעתיקה זיכרון בגודל מילה או בית מפורט מסוים אל ax או al. פקודה out מעתיקה זיכרון בגודל מילה או בית מ-ax או al אל פורט.

- יש שתי שיטות כתיבה של מספר הפורט בפקודות out, in:
 - ישירות: רושמים את מספר הפורט, בתנאי שהוא בין 0 ו-255 בלבד.
 - בעקיפין: אם אנחנו רוצחים לתקשר עם פורט בתחום שמעל 255, במקום פורט משתמשים ברגיסטר ax.

צורות הכתיבה הבאות חוקיות:

in ax/al, port

in ax/al, dx

out port, ax/al

out dx, ax/al

דוגמה לשימוש בפקודות **out** / **in** לעבודה מול פורט שמספרו קטן מ-255:

```
in    al, 61h          ; read the status of the port
or    al, 0000001b      ; change some bits
out   61h, al          ; copy the value in al back to the port
```

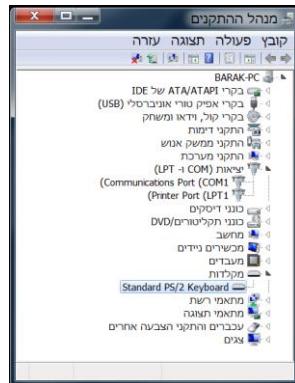
משמעות הביטים ששינו איננה חשובה כרגע, (נגע אליה בהמשך, כشنעוק בפרויקט סיום). החשיבות היא בתהליך – אנחנו קוראים באמצעות פקודה **in** את הסטוס של פורט כלשהו, משנים בו כמה ביטים וכותבים חזרה את הסטוס אל הפורט על-ידי פקודה **out**.

דוגמה לעבודה מול פורט שמספרו גדול מ-255:

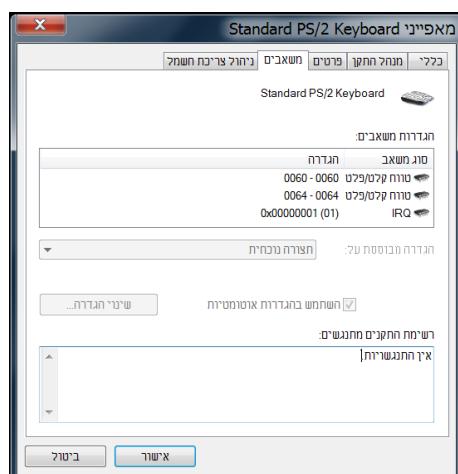
```
mov  dx, 300h
in   al, dx
```

כיוון שמספר הפורט גדול מ-255, לא ניתן לפנות אליו ישירות אלא נדרש שימוש ב-**dx** לשימרת מספר הפורט.

דוגמה – מציאת פורט התקשרות עם המקלדת: ניכנס למנהל התקנים (hiposh tcnit-<Device manager



בתוך מanager התקנים, נסמן את המקלדת ונ קישר על עכבר ימני – נבחר "מאפיינים" (Properties). בתוך מאפיינים, נבחר "משאבים". ניתן לראות שהמעבד מתקשר עם המקלדת דרך פורט 60 ופורט 64.



המקלדת

הקדמה

בחלק זה נפרט את שרשרת הפעולות שמתבצעת מרגע הקשה על מקש במקלדת ועד קבלת התו ומיון לשימוש בזיכרון המעבד. לפני שנציגו אל שלבים השונים, כדי לעשות סדר בדברים, התהליך הוא כזה:

1. המקלדת יוצרת מידע – הקשות על מקשים וחרור של מקשים. מידע זה נקרא **Scan Codes**.
2. המקלדת מקושרת ל זיכרון ה-O/I של המעבד והמידע שמגיע ממנו מועתק אל פорт קבוע בזכרון, פорт **60h**.
3. פסיקת חומרה, אינטראפט, אוספת את ה-**scan codes** מפורט **60h** ומעתיקה אותם אל באפר מיוחד שמנוגדר **Type Ahead** לשימירת המידע מהמקלדת. מספרו של האינטראפט הוא **9h** והשם של הבאפר של המקלדת הוא **Buffer**.



כמו שאמרנו בمبוא לנווט הפסיקות, למי שמתכוна באסmbly יש יותר דרך אחת לעבוד מול חומרה. ספציפית לגבי המקלדת יש שלוש דרכים לעבוד מולה:

1. לעבוד ישירות מול הפורט של המקלדת בשיביל לקבל את המידע על המקשים שהוקשו.
2. להשתמש בפסיקה של BIOS (חומרה – BIOS) היא חבילת תוכנה של אינטל, שנתקלה בקצתה בראשית הפרק הקודם. אחד מתפקידיה הוא קישור אל רכבי חומרה). ל-BIOS יש פסיקה מס' **h16** שנותנת לנו לבצע פעולות שונות מול המקלדת – כמו קליטתתו – בפשטות יחסית.
3. להשתמש בפסיקה של DOS, פסיקה **h21**, שכבר הכרנו חלק מהשירותים שהוא נותן בנושא העבודה מול המקלדת.

מקוצר הירעה, ניתן הסבר תמציתי בלבד לנווט המקלדת. מומלץ להעמיק את ההבנה בספר *Art of Assembly*, פרק 20 (המקלדת).

יצירת Scan Codes ושליחתם למעבד

בתוך המקלדת יש רכיב, ששולח ל-PIC קוד עם כל לחיצה או שחרור של מקש. המידע על שחרור המקש במקלדת חשוב מאוד לטובת פעולה תקינה – לדוגמה, כשהאנחנו מבצעים צירוף של מקשים יחד. יש הבדל בין לחיצה בנפרד על המקשים, alt, control, delete לבין לחיצה על שלושתם יחד. לכן המקלדת צריכה לסמן באופן נפרד פעולות של לחיצה ופעולות של שחרור של כל מקש.

הסימן הזה מבוצע על-ידי טבלת קודים, שנראית如下. בטבלה יש לכל מקש במקלדת שני קודים – קוד לחיצה down וקוד לשחרור up. כמו שאפשר לראות בטבלה הבאה, ההבדל בין scan code של לחיצה לשחרור הוא 80h, כלומר בית מס' 7 של הדיבט scan code מקבל 0 בלחיצה ו-1 בשחרור, יתר הביטים נשארים קבועים בין לחיצה ושחרור של אותותו (אך משתנים כਮובן בין תווים שונים).

Key	Down	Up	Key	Down	Up	Key	Down	Up	Key	Down	Up
ESC	1	81	[{	1A	9A	, <	33	B3	center	4C	CC
1 !	2	82] }	1B	9B	, >	34	B4	right	4D	CD
2 @	3	83	Enter	1C	9C	/ ?	35	B5	+	4E	CE
3 #	4	84	Ctrl	1D	9D	R shift	36	B6	end	4F	CF
4 \$	5	85	A	1E	9E	PrtSc	37	B7	down	50	D0
5 %	6	86	S	1F	9F	alt	38	B8	pgdn	51	D1
6 ^	7	87	D	20	A0	space	39	B9	ins	52	D2
7 &	8	88	F	21	A1	CAPS	3A	BA	del	53	D3
8 *	9	89	G	22	A2	F1	3B	BB	/	E0 35	B5
9 (0A	8A	H	23	A3	F2	3C	BC	enter	E0 1C	9C
0)	0B	8B	J	24	A4	F3	3D	BD	F11	57	D7
- _	0C	8C	K	25	A5	F4	3E	BE	F12	58	D8
= +	0D	8D	L	26	A6	F5	3F	BF	ins	E0 52	D2
Bksp	0E	8E	;	27	A7	F6	40	C0	del	E0 53	D3
Tab	0F	8F	"	28	A8	F7	41	C1	home	E0 47	C7
Q	10	90	` ~	29	A9	F8	42	C2	end	E0 4F	CF
W	11	91	L shift	2A	AA	F9	43	C3	pgup	E0 49	C9
E	12	92	\	2B	AB	F10	44	C4	pgdn	E0 51	D1
R	13	93	Z	2C	AC	Num	45	C5	left	E0 4B	CB
T	14	94	X	2D	AD	SCRL	46	C6	right	E0 4D	CD
Y	15	95	C	2E	AE	home	47	C7	up	E0 48	C8
U	16	96	V	2F	AF	up	48	C8	down	E0 50	D0
I	17	97	B	30	BO	pgup	49	C9	R alt	E0 38	B8
O	18	98	N	31	B1	-	4A	CA	R ctrl	E0 1D	9D
P	19	99	M	21	B2	left	4B	CB	pause	E1 1D	-

טבלת תווים ו-Scan codes

לדוגמה, לחיצה על המKeySpec ESC תגרום לשילוחת קוד 1h, שחרור של מקש הדיבט ESC יגרום לשילוחת קוד 81h. באופן כללי, כל פעולה של לחיצה או שחרור של מקש גורמת לשילוחת scan code יחד עם אינטראפט. להלן התהליך שמתறחש עם הלחיצה על המקלדת:

1. רכיב חומרה שנמצא במקלדת מעביר לפורט 60h את הדיבט scan code של הלחיצה / שחרור של המKeySpec.
2. הדיבט PIC מקבל דרך IR1 אינטראפט מהמקלדת.
3. הדיבט PIC שולח למעבד אינטראפט, שאומר למעבד שיש מידע בפורטים של המקלדת. האינטראפט שמוופעל הוא .9h.
4. כתגובה לאינטראפט, המעבד מרים ISR שמתפלב באינטראפט 9.

5. ה-ISR מטפל בהעתקה ה-*scan code* אל מקום מוגדר בזיכרון (ה-*Type Ahead Buffer* שהזכרנו בפתח).

6. בסיום ריצת ה-ISR, המעבד שולח ל-*PIC* סימן *end of interrupt* והתהליך מגיע לסיום.

בابر המקלדת Type Ahead Buffer

ה-ISR שמופעל על-ידי האינטרפט *9h* לטיפול במקלדת לוקח את ה-*scan code*, *scan code*, שהוא שלו הוא בית אחד, ומתרגם אותו לקוד ASCII. התרגום ל-ASCII תלוי באילו עוד מקשים היו לחוצים באותו זמן. נניח שהמשתמש לחץ על המקש 'a' יחד עם המקש *shift*, התרגום צריך להיות 'A', שיש לו קוד ASCII שונה מאשר 'a'. לאחר פעולת התרגום, קיבלנו את קוד ASCII גדול יותר. עכשו יש לנו גם *scan code* וגם ASCII *code*, שתופסים בסך הכל שני בתים. ה-ISR לוקח את שני הבטים הללו ומעתיק אותם אל אזור קבוע בזיכרון בשם *babefor* המקלדת, *Type Ahead Buffer*. הבאפר מוגדר בצורה הבאה:

- מיקום A 0040:001A – מצביע על ראש הבאפר

- מיקום C 0040:001C – מצביע על זנב הבאפר

- מיקום E 0040:001E – 16 מילים (words)

כמו שאפשר לראות מגודל הבאפר, הוא יכול להחזיק עד 16 הקלדות של המקלדת (משום שככל הקלדה מתורגם לשני בתים – *scan code*, ASCII *code*). מה קורה אחרי 16 הקלדות? הבאפר הוא באפר מעגלי – יש מצביע לראש ומצביע לזנב שלו. בכל פעם שרץ אינטרפט מקלדת, הוא מגדיל את הערך של זנב הבאפר ב-2. אם הערך יוצא מחתחום של הבאפר, הוא מוחזר לתחילת הבאפר. ככה עובד באפר מעגלי.

כמו שיש מגנון שדווגג להכנס נתונים לבאפר, יש מגנון שדווגג להוציא נתונים מהבאפר. זה פסיקת BIOS, שמעתקה את המידע מהבאפר במיקום שראש הבאפר מצביע עליו, ומעלה את ערכו של מצביע ראש הבאפר ב-2.

כך הראש של הבאפר "רודף" אחרי הזנב של הבאפר במסלול מעגלי שגודלו 16 מילים. יש פסיקת מקלדת שדווגגת לכך את הערך של הזנב ויש פסיקת BIOS שדווגגת לכך את הערך של הראש. מה קורה אם הראש "משיג" את הזנב? המשמעות של המקרה זהה היא שהבאפר מלא, כלומר אם נעתיק לתוכו ערך חדש אנחנו נדרוס הקלדה על המקלדת שעדיין לא טופלה. שמתם פעם לב לכך שאם התוכנית שלכם תקועה, ואתם מנסים ללחוץ על המקלדת הרבה פעמים, מתישחו המחשב ישמיע לכם צליל מעצבן עם כל הקלדה של המקלדת? זה בגלל שימושכם את באפר המקלדת, הקשתם 15 פעמים בדיק על המקלדת (בקלדה הבאה הראש יעקו את הזנב) ובגלל שהמחשב תקוע, הפסיקה של ה- *BIOS* לא מրוקנת את הבאפר.

הסיבה שנותנו את כל הסקירה זו, היא כדי להגיע למסקנה שניקיי הבאפר הוא פעולה הכרחית וצריך לדאוג לכך שהבאפר לא יהיה מלא – לאחרת אנחנו מאבדים את יכולת קלוט תווים חדשים מהמקלדת. זה מה שצורך לזכור, ומיד נשתמש במידע זה.

עכשו, אחרי שסקרנו את התיאוריה של שרשרת הפעולות שגורמות להקלחת להפוך לקוד ASCII זמין לשימוש, נראה איך מבצעים שתי פעולות עיקריות:

- קריאה של TWO מהמקלדת
- ניקוי הבאפר של המקלדת

נראה את הפעולות הללו בשלוש שיטות שונות – יישירות דרך הפורטים של המקלדת, בעזרת פסיקת BIOS ובעזרת פסיקת DOS.

שימוש בפורטים של המקלדת

בקצהה, המעבד עובד מול שני רכיבי חומרה, microcontrollers. אחד נמצא בתחום המקלדת, שני נמצא על לוח האם של המעבד. ישנו שלושה פורטים שמעורבים בעבודה עם המקלדת:

1. פорт **60h** משמש לתקשורת בין המעבד לבין ה-microcontroller של המקלדת. עוברים ביניהם נתונים מסוימים שונים, הנתון שחשוב לנו הוא ה-**scan codes**.
2. פорт **64h** משמש לתקשורת בין המעבד לבין ה-microcontroller של לוח האם. הם מעבירים ביניהם הודעות בקרה, שדרכו אפשר לדעת אם יש **scan code** בפורט **60h**.
3. פорт **61h** הוא פорт בקרה כמו פорт **64h**, אבל ישן יותר. מקלדות מודרניות לא עושים שימוש בפורט זה, אבל ישנו רכיבי חומרה אחרים שעדיין עושים בו שימוש ולכך נגיעה אליו בהמשך (כשנרצה לעבוד מול כרטיס הקול).

קוד שבודק בפורט **64h** אם הגיעתו חדשה מהמקלדת (הבית השני מחזק את הסטטוס):

```
in    al, 64h      ; Read keyboard status port
cmp   al, 10b     ; Data in buffer ?
```

את הקוד הזה ניתן להכניס ללולאה, שרצאה עד שתנאי העצירה – יש מידע חדש מהמקלדת – מתקיים:

WaitForData:

```
in    al, 64h
cmp   al, 10b
je    WaitForData
```

לאחר שבדקנו שיש מידע חדש מהמקלחת, נבצע קריאה מהמיוקם שבו המידע נמצא – פורט 60h:

in al, 60h

לABIי השלב האחרון, ריקון הבAPER, ניתן לבצע אותו עליידי קידום מצבייע ראש המקלחת ב-2.

תוכנית דוגמה – קליטת הקלדות ובדיקה האם מקש ה-ESC נלחץ. שימושו ליב שבדוגמה זו אנחנו לא מנקים את באפר המקלחת אחרי כל הקלדה. אחרי 15 תווים הבAPER כבר מלא – בתו הבא הראש כבר ישיג את הזנב. כתוצאה לכך בסוף התוכנית יודפסו למסך התווים שהקלדנו לפני ה-ESC, כל עוד מדובר בפחות מ-15 תווים. התווים ה-16 והלאה כבר "הילכו לאיבוד" ואינם קיימים יותר בזיכרון. גם אם הבAPER מלא, ברגע שאנחנו עובדים ישירות מול פורט 60h אנחנו עדיין מסוגלים לקרוא תווים חדשים, אבל הם כבר לא נגישים לבAPER.

```

; -----
; Use keyboard ports to read data, until ESC pressed
; Author: Barak Gonen 2014
; -----

IDEAL

MODEL small

STACK      100h

DATASEG

message    db 'ESC key pressed',13,10,'$'

CODESEG

start:

        mov     ax, @data
        mov     ds, ax

WaitForData :

        in      al, 64h          ; Read keyboard status port
        cmp     al, 10b          ; Data in buffer ?
        je      WaitForData     ; Wait until data available
        in      al, 60h          ; Get keyboard data
        cmp     al, 1h           ; Is it the ESC key ?
        jne    WaitForData

ESCPressed:

        mov     dx, offset message
        mov     ah, 9
        int     21h

exit:

        mov     ax, 4C00h
        int     21h

END    start

```

```
C:\TASM\BIN>keyboard
ESC key pressed

C:\TASM\BIN>012345678901234_
```

למרות שהקלטן יותר מ-15 תווים, רק 15 תווים נשמרו ב ArrayBuffer המקלדת

והודפסו למסך עם הייציאה מהתוכנית.

לעתים אנו צריכים לזהות לא רק איזה מקש הופעל במקלדת, אלא גם אם הפעולה היא לחיצה על המKeySpec או שחרור. לדוגמה, כאשר אנחנו רוצים לנגןתו בפסנתר- הנגינה מתחילה עם הלחיצה על המKeySpec ומסתיימת עם שחרור המKeySpec.

על מנת לעשות זאת נתבבש על התוכנית האחרונה אך בשינוי קטן- נוסיף בדיקה האם האמ-ה-code scan code הוא מעל 80h (כלומר שחרור המKeySpec). אם הערך הוא מעל 80h, או ביצוע הפקודה

and al, 80h

יוביל לתוצאה שאינה אפס. אחריה התוצאה תהיה אפס.

להלן קוד דוגמה:

```
; -----
; Identify key press and key release
; Print "Start" when a key is pressed
; Print "Stop" when the key is released
; Exit program if ESC is pressed
; Barak Gonen 2015
; -----
IDEAL
MODEL small
STACK 100h
DATASEG
msg1      db 'Start$'
msg2      db 'Stop$'
saveKey    db 0
CODESEG
start:
        mov ax, @data
```

```
    mov ds, ax
```

WaitForKey:

```
;check if there is a new key in buffer
in al, 64h
cmp al, 10b
je WaitForKey
in al, 60h

;check if ESC key
cmp al, 1
je exit

;check if the key is same as already pressed
cmp al, [saveKey]
je WaitForKey

;new key- store it
mov [saveKey], al

;check if the key was pressed or released
and al, 80h
jnz KeyReleased
```

KeyPressed:

```
;print "Start"
mov dx, offset msg1
jmp print
```

KeyReleased:

```
;print "Stop"
mov dx, offset msg2
```

print:

```
    mov ah, 9h
    int 21h
```

```
jmp WaitForKey
```

exit:

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

END start

שימוש בפסיקת BIOS

פסיקת BIOS מס' 16h נותנת לנו כלים נוחים לבדיקת מצב המקלדת, קריאת התו שהוקלד (אם הוקלד) ו"נקיי" באפר המקלדת (כלומר שינוי מצביו הראש ומצביע הזנב של באפר המקלדת כך ש מצב הבאפר יהיה ריק, אין נתונים).

כדי לקרוא את התו הבא מתיק באפר המקלדת, מפעילים את ah=0h עם קוד 16h. הפסיקה מחזירה בתוך ah את קוד ASCII של התו שנמצא בראש הבאפר ובתוך ah את הדסטוט ah scan code שלו. בנוסף, הפסיקה "מנקה" את התו מהבאפר עליידי קידום ערכו של ראש הבאפר ב-2.

הבעיה היחידה עם הפסיקה זו, היא שאם איןתו שמתין באפר – הפסיקה תחכה לו. כתוצאה מכך, אם אנחנו רוצים לתוכנה משחק שלא עוצר בהמתנה לפועלה של השחקן, הפסיקה זו לבדה לא מתאימה.

הפתרון הוא לשלב את הפסיקה 16h עם קוד ah=1. במקרה זה, הפסיקה מחזירה את הסטוטוס של המקלדת – 0 אם ישתו מוקן לקריאה, 1 אם איןתו מוקן. אם ישתו מוקן, ah ו-ah יקבלו את ערכי ASCII וה-dsটוט ah scan code של התו.

שילוב הפסיקות מאפשר לנו:

- לדעת מתי יש מידע מהמקלדת (בלי לעזר את הריצזה ולהחות למשתמש).
- לקЛОות את המידע.
- לנ��ות את באפר המקלדת.

דוגמאות:

WaitForData:

```
mov ah, 1
int 16h
jz WaitForData
mov ah, 0 ; there is a key in the buffer, read it and clear the buffer
int 16h
```

התוכנית הבאה מבצעת גם היא קליטת הקלדות ויוציא אם הוקלד ESC - בתוספת נקיי באפר המקלדת:

```
; -----
; Use BIOS int 16h ports to read keyboard data, until ESC pressed
; Author: Barak Gonen 2014
```

```

; -----
IDEAL
MODEL small
STACK      100h
DATASEG
message     db 'ESC key pressed',13,10,'$'
CODESEG
start:
        mov    ax, @data
        mov    ds, ax
WaitForData :
        mov    ah, 1
        int    16h
        jz     WaitForData
        mov    ah, 0
        int    16h
        cmp    ah, 1h
        jne    WaitForData
ESCPressed:
        mov    dx, offset message
        mov    ah, 9
        int    21h
exit:
        mov    ax, 4C00h
        int    21h
END   start

```

שימוש בפסיקה DOS

פסיקה ah=0Ch עם קוד ah=0Ch מנקה את הבאפר של המקלדת, ואז מבצעת טרייק נחמד – היא לוקחת הערך ששענו בלא כפרמטר, וMRIIZAH ah=7h עם הקוד הזה. אנחנו נראה דוגמה בה ah=7h, al=7h – לאחר ניקוי באפר המקלדת, הפסיקה תעבור לקוד 7h, שהוא קוד של קליטתתו מהמקלדת ללא הדפסת התו על המסך. באופן זה אנחנו מבצעים שתי פעולה:

- ניקוי באפר המקלדת

- קליטתתו חדש מהמשתמש

בסיום הריצה או יכול את קוד ה-ASCII של התו שהוקלד.

הקוד הבא מבצע את הפעולות הב"ל:

```
; Clear keyboard buffer and read key without echo
mov ah,0Ch
mov al,07h
int 21h
```

לעובודה בשיטה זו יש יתרון בROUTOR מבחן פשוטות תכונות. עם זאת ישנו שני חסרונות:

החיסרון הראשון, יותר ברור מאשר מאליו, הוא שהתוכנה עוצרת בזמן שהוא מנסה לקלט מקלט מהמשתמש. זה יכול להיות בעיתי אם אנחנו מרכיבים משחק וכו'.

החיסרון השני, הוא שישנו מקשים שונים שקובד ה-ASCII שלהם הוא לא קוד נורמלי בגודל בית אחד, אלא קוד ASCII מרווח בגודל שני בתים. מקשים אלו דוחק אמוך שימושיים למשחקים, כמו לדוגמה מקשי החיצים. במקרה זה, אם נרצה לעשות תנאי השוואה ובדיקה על קוד ה-ASCII – יהיה בעיה.

תרגיל 12.1: מקלדת



א. ה-ISR של המקלדת מעתיק את הקוד scan code אל ה-**Type Ahead Buffer** שנמצא במיקום 0040:001Eh בזיכרון המעבד.

כתבו תוכנית שקוראתתו מהמקלדת (השתמשו ב-int 21h עם הקוד המתאים), הריצו את התוכנית ב-**TD** במצב **step by step**, וצפו בשינוי זיכרונו במיקום של ה-**type ahead buffer**. בטור קלט, הכניסו את התו 'a' ומצאו ב-**scan codes** את ה-**type ahead buffer** שלו.

ב. במשחקי מחשב שונים, המקשים wasd משמשים לתזוזות השחקן:

W = up -

A= left -

S = down -

D= right -

כיתבו תוכנית שמאזינה למקלדת. אם הוקש אחד ממקשי wasd, יודפס למסך "Move up", "Move down" וכו'. אם הוקש מקש ה-**Esc**, התוכנית תצא. כל מקש אחר – התוכנה לא תעשה דבר. כדי לדמות משחק מחשב, השתמשו בפסיקה שאינה עוזרת את ריצת התוכנית בהמתנה לקלט.

ג. הקוד הבא גורם לכרטיס הקול להשמיע צליל:

```
in    al, 61h
or    al, 00000011b
out   61h, al
mov   al, 0b6h
out   43h, al
mov   ax, 2394h
out   42h, al
mov   al, ah
out   42h, al
```

הקוד הבא גורם לכרטיס הקול להפסיק את השמעת הצליל:

```
in    al, 61h
and   al, 11111100b
out   61h, al
```

כיתבו תוכנית שברגע שנלחץ מקש כלשהו מוציאה צליל, ועם שחרור המקש מפסיק את השמעת הצליל.
הדרכה: התוכנית תשתמש בפסיקה **16h** כדי לבדוק אם יש מידע חדש מהמקלדת. אם יש מידע חדש, התוכנית תבודק בעזרת פорт **60h** אם ה-**code scan** מתאים להחיצה או לשחרור ובהתאם יופעל קטע הקוד
שמושמע צליל או קטע הקוד שפסיק את השמעת הצליל.

סיכום

התחלנו את הפרק בלימוד התיאוריה של אינטראפטים: למה בכלל יש צורך בפסיקות לטיפול בהתקני חומרה翕ך ה-**PIC**, בקר האינטראפטים, מבצע את עבודת הקישור בין התקני החומרה לבין המעבד.

למדנו על פורטימ, אותן מקומות בזיכרון שדרכם המעבד מקבל ושולח מידע אל רכיבי החומרה.

לבסוף התמקדנו ברכיב חומרה מרכזי – המקלדת. למדנו שכל לחיצה על המקלדת יוצרת **scan code** וסקרנו את התהיליך שגורם לכך שבסוףו של דבר התו שנלחץ מופיע בזיכרון המחשב, ב-**Type Ahead Buffer**. ראיינו דוגמאות לעובודה מול המקלדת במגוון שיטות:

- עבודה ישירה מול הפורטימ של המקלדת, **h60h** ו-**h64h**
- שימוש בפסיקות **int 16h**, BIOS
- שימוש בפסיקות DOS, **h21h**, לנקיי באפר המקלדת

בפרק הבא נעסק בנושאים שימושיים לכנתיבת פרוייקטי הסיום.

פרק 13 – כלים לפרויקטים

מבוא לפרויקטי סיום

זהו, כיסינו בפרקם הקודמים את כל החומר התיאורטי שנכלל בתוכנית הלימודים. כמובן שיש עוד חומר תיאורטי רב שקשרו לשפת אסבלי, אבל מדובר בעיקר על הרחבת נושאים שדנו בהם בקורסה. עכשו אנחנו מתקדמים אל העבודה המעשית, אל הפרויקטים. פרויקט הסיום הוא ההזמנות שלכם לצותוב תוכנה "אמיתית" שמשלבת בין הדברים שלמדתם לדברים שימושיים אתם, להציג שאתם מבינים את החומר היטב ולהוכיח את זה. הפרק הזה מועד לкриאה וללימוד עצמי. הוא אינו מתיימר לכוסות באופן מלא את הנושאים אלא רק לתת הסבר ראשוני ומקורות ללימוד עצמו. קיימות לכך שתי סיבות עיקריות. הראשונה, סיבה מעשית: במסגרת פרק אחד אין אפשרות לכוסות לעומק נושאים, שלאורב מוקדש להם פרק משל עצמו או אפילו ספר. הסיבה השנייה היא הרצון שלכם ליצור העבודה בה תנתנו בעתיד – המידע שאתם צריכים קיים אי שם, עליו לחשוף אותו, להגיע אליו ולהבין אותו בכוחות עצמכם.

בחירת פרויקט סיום

האם כבר בחרתם פרויקט סיום? מומלץ שתבחרו פרויקט לפני קריאת פרק זה, כך שתוכלו להתמקד בדיקת דבריהם שאתם צריכים כדי לעבוד על הפרויקט ותשקעו פחות זמן בעיסוק בנושאים אחרים. כמובן, שם יש לכם עניין למוד את כל הנושאים – דבר זה מומלץ וצפואה לכם הנהה לאורך הדרך.

קיימים מגוון לא קטן של פרויקטי סיום לבחירתם. רוב הפרויקטים הם בסדר גודל של 1000 עד 2000 שורות קוד. כאשר מדובר בפרויקט שכולל מתחת ל-1000 שורות קוד, ייתכן שאתם לא ממצים את מלאו היכולות שלכם. אם הפרויקט הוא הרבה מעל ל-2000 שורות קוד, ייתכן שאתם לוקחים על עצמכם משימה שתיקח זמן רב מדי. אפשר לסתוג את הפרויקטים לשולש קטגוריות עיקריות:

- **משחקים – סנייק,** פונג או כל דבר שעולה על דעתכם. היתרונו בבחירה פרויקט שהוא משחק, הוא שהעבודה צריכה לעשות בפרויקט די מוגדרת ובדרך כלל הפעלת התוכנה די אינטואיטיבית, דבר שמקל על חכונם הפרויקט.

- **אפליקציה קטנה –** לדוגמה כלי נגינה (הקשה על המקלדת או העכבר גורמים להפעלת הכללי, המשמע צלילים ושינויי גרפיקה), מכונית מזויקה (בחירה שירים שהוקלטו מראש), צייר (הוזת העכבר תוך כדי להיזה גורמת לציר על המסך) או כל דבר שעולה על דעתכם. היתרונו בפיתוח אפליקציה הוא שיש מקום רב לייצירויות ואפשר לעשות מה שימושיים אתם.

- **בעיות אלגוריתמיות –** לדוגמה שימוש של קודים לתקן שגיאות, פתרון אוטומטי של סודוקו או פתרון של בעיות מתמטיות שונות. בפרויקטים מסווג זה ישנו סיכון – אם אתם מתכוונים לפתור בעיה כזו, תניחו מראש שאות כל המידע התיאורטי תצרכו ללמידה בכוחות עצמכם. אי לך, כדאי שתיכנסו לנושאים כאלה רק אם אתם נלהבים ללמידה את

כל הידע בעצמכם מהאינטרנט. היתרונו בפרויקט כזה הוא שמדובר בנושאים מאוד מעניינים והידע האלגוריתמי שתרכשו עשוי להיות רלוונטי לדברים שתעשו בעתיד.

הכלים שנלמד בפרק זה הם:

- קבצים:

- עבודה עם קבצים, קריאה מקובץ ושמירה לקובץ

- גרפייה:

- ייצירת גרפייה בעזרת תוכנת ASCII
- ייצירת גרפייה בעזרת הדפסת פיקסלים למסך
- יבוא של תמונות בפורמט BMP

- צילומים:

- המשמעת צילומים בתדרים שונים

- שעון:

- שימוש בשעון למדידת זמן (השניה)
- שימוש בשעון לייצור מספרים אקראיים

- ממתק משתמש:

- טיפים לעובדה עם המקלדת
- קליטת פקודות מהעכבר

- שיטות דיבוג

עבודה עם קבצים

פתחת קובץ

הדבר הראשון שאנו צריכים לעשות כדי לקרוא מקובץ, הוא לפתוח אותו לקריאה (כמו ספר – אי אפשר לקרוא אותו כשהוא סגור, קודם צריך לפתוח את הרכינה...). הדרך פשוטה לבצע פתחת קובץ היא באמצעות פסיקת DOS, עם קוד ah=3Dh:

AL – מטרת הפתיחה

- 0: קריאה בלבד
- 1: כתיבה בלבד
- 2: קריאה וכתיבה

DS:dx – מצביע על שם הקובץ.

שימוש לב שהמחוזות של שם הקובץ צריכה להסתיים ב-0. לדוגמה:



Filename db 'file.txt',0

בסיום הריצה, ax יוכל את ה-dlhandle שהוקצת לו על ידי מערכת הפעלה DOS. אם היה שגיאה, יודלק דגל CF ו-ax יוכל את אחד הערכים הבאים:

- 2: הקובץ לא נמצא.
- 5: יותר מדי קבצים פתוחים.
- 12: אין הרשאה לפתיחת הקובץ.

מומלץ לאחר הפתיחה לבדוק אם הפתיחה הייתה תקינה, ואם לא – להדפיס הודעה שגיאיה. אם ננסה להמשיך לעבוד עם קובץ שהפתיחה שלו נכשלה, סביר שהתוכנה תקרוב.

```
proc OpenFile
; Open file
    mov ah, 3Dh
    xor al, al
    lea dx, [filename]
    int 21h
    jc openerror
    mov [filehandle], ax
    ret
openerror:
```

```

    mov  dx, offset ErrorMsg
    mov  ah, 9h
    int   21h
    ret
endp OpenFile

```

קריאה מקובץ

קריאה מקובץ מתבצעת על ידי קוד ah=3Fh. הפקטרים שצורך לשלווח לפסקה הם:
 bx – שקיבלנו מ-DOS בשלב הפתיחה.
 cx – כמות הבתים שאנחנו מבקשים לקרוא.
 dx – מצביע על באפר (מערך) שאליו יווטק המידע מהקובץ.

שימוש לב: הגודל של dx חייב להיות גדול או שווה לכמות הבתים שאנחנו רוצים לקרוא, לאחרת יידرس הזיכרון

אחרי הבאפר.



```
proc ReadFile
```

```

; Read file
    mov  ah,3Fh
    mov  bx, [filehandle]
    mov  cx,NumOfBytes
    mov  dx,offset Buffer
    int   21h
    ret

```

```
endp ReadHeader
```

ביציאה, cx יחויק את כמות הבתים שנקראו מהקובץ, או קוד שגיאת אם הייתה בעיה.
 שימוש לב Ci יש לקרוא מהקובץ רק לאחר שפתחנו אותו בהצלחה.

כתיבה לקובץ

פעולה הכתיבה לקובץ נראית בדיק כמו פעולה הקריאה – רק להיפך. הקוד לכתיבה לקובץ הוא ah=40h. פקטרים:
 bx – שקיבלנו מ-DOS בשלב הפתיחה.
 cx – כמות הבתים אותם אנחנו מבקשים לכתוב. הערה: אם cx=0 או כל המידע שבקובץ אחרי filehandle, יימחק.

ax – מצביע על באפר (מערך) שמננו ועתק המידע אל הקובץ.

ביציאה, **ax** יחויק את כמות הבתים שנכתבו לקובץ, או קוד שגיאת אם הייתה בעיה.

שימוש לב: אם לא פתחם את הקובץ במצב SMAFSLWR כתיבה (cx=1, cx=2) או יוחזר קוד שגיאת 5
שם שמעותו  **access denied**.

```
proc WriteToFile
```

```
    mov ah,40h
    mov bx, [filehandle]
    mov cx,12
    mov dx,offset Message
    int 21h
    ret
```

```
endp WriteToFile
```

שימוש לב כי יש לכתוב לקובץ רק לאחר שפתחנו אותו בהצלחה.

סגירת קובץ

בזמן היציאה מהתוכנית (פסקה 21h עם קוד ah=4Ch), משוחרר כל הזיכרון שהתוכנית תפסה ונוגדים כל הקבצים שהתוכניתفتحה. אם כך, מודיע בכל לסקור קבצים שאנחנו משתמשים בהם? אחת הסיבות היא שלא תמיד נוכל לסמוק על היציאה מהתוכנית, שתסגור את הקבצים שלנו. לדוגמה, אם התוכנית קרסה בזמן ריצה, היא לא תסגור את הקבצים שפתחנו. זה גם הריגל טוב לעתיד: לסקור משאבי חיצוניים שאנחנו משתמשים בהם (קבצים, זיכרון, קישורים למחשבים אחרים, תוכנות עוזר ועוד). בדומה זו, תוכנות אחרות יכולות לנצל את המשאבי שפינינו והקוד שלנו הופך ליעיל יותר.

סגירת קובץ מתבצעת על ידי קוד ah=3Eh. הפקטור היחיד שצורך לחת לפסקה הוא:

שקיבלנו מ-DOS בשלב הפתיחה – bx

```
proc CloseFile
```

```
    mov ah,3Eh
    mov bx, [filehandle]
    int 21h
    ret
```

```
endp CloseFile
```

פקודות נוספות של קבצים

באמצעות קודים שונים ניתן לבצע פעולות נוספות, בנוסף ללימוד עצמי. עקב ריבוי המקורות והעבודה שמקורות שונים מתמקדים בהסבירים ובדוגמאות על פעולות שונות, עדיף שתתנסו בלבד בחיפוש אחריו המקור שנדרש לכם, בין זהה עמוד הסבר, דוגמאות או פורום לעזרה ושאלות. תוכלו למצוא הדרכה על כל פקודה על-ידי חיפוש בגוגל 'assembly int 21h ah=...' אשר שלושת הנקודות מוחלפות בקוד הרלכני:

AH=3Ch – ייצרת קובץ

AH=41h – מחיקת קובץ

AH=42h – הוזת המצביע בתוך הקובץ

תכנית לדוגמה – filewrt.txt

התוכנית הבאה פותחת קובץ ריק בשם testfile.txt, מעתקה לתוכו את המחרוזת 'Hello world!' וסגרת את הקובץ.

```
; -----
; Write to file
; Author: Barak Gonen, 2014
;

IDEAL

MODEL      small
STACK      100h

DATASEG

filename    db 'testfile.txt',0
filehandle   dw ?
Message     db 'Hello world!'
ErrorMsg    db 'Error', 10, 13,'$'

CODESEG

proc OpenFile
; Open file for reading and writing
```

```

mov ah, 3Dh
mov al, 2
mov dx, offset filename
int 21h
jc openererror
mov [filehandle], ax
ret

```

openererror:

```

mov dx, offset ErrorMsg
mov ah, 9h
int 21h
ret

```

endp OpenFile

proc WriteToFile

; Write message to file

```

mov ah,40h
mov bx, [filehandle]
mov cx,12
mov dx,offset Message
int 21h
ret

```

endp WriteToFile

proc CloseFile

; Close file

```

mov ah,3Eh

```

```
    mov  bx, [filehandle]  
    int   21h  
    ret  
endp CloseFile
```

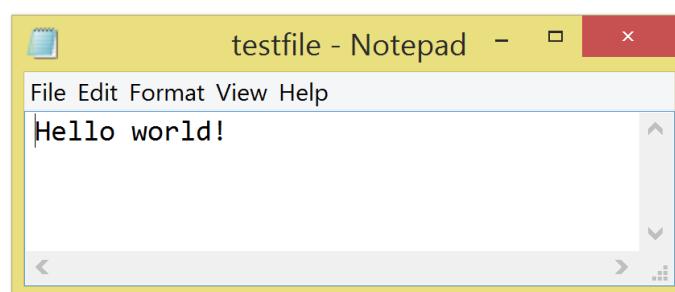
start:

```
    mov  ax, @data  
    mov  ds, ax  
    ; Process file  
    call  OpenFile  
    call  WriteToFile  
    call  CloseFile
```

quit:

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

END start



גרפיקה

יש מספר דרכים להכנס לתוכנית שלכם אלמנטים גרפיים. לפני הכל, ניתן קצת רקע תיאורטי.

הרענון הבסיסי בעובדה עם גרפיקה הוא שישנו אוצר בזיכרון ה-O/I שומר את כל המידע שמוצג על המסך. האוצר הזה נקרא **video memory** והוא הקשור לכל כרטיס המספר. הדינמי **video memory** חoops את המיקום בזיכרון שבין 0000:0000 ל-FFFF:0000. בין 0000:FFFF.

כרטיס המספר יכול לעבוד בשני מצבים, שנקראים **modes**:

- במצב טקסטואלי, או **text mode**, אנחנו קובעים לCards המספר שעליו לקרוא את המידע מה-video memory במקום שמתחליל ב-0000:0000 וגודלו 4K (4,096 בתים). במצב זה, Cards המספר מדפיס 25 שורות כפול 80 עמודות של תוכן ASCII.

- במצב גרפי, או **graphic mode**, אנחנו קובעים לCards המספר שעליו לקרוא את המידע מה-video memory במקום שמתחליל ב-0000:0000, וגודלו 64K (65,536 בתים). במצב זה, Cards המספר מדפיס 200 שורות כפול 320 עמודות של פיקסלים.

אפשר לפרק את תהליך ההדפסה למספר שלושה שלבים:

שלב ראשון – קביעת מצב העבודה של Cards המספר (כ**শক্তিশালী ক্ষমতা** DOSBOX), מצב ברירת המחדל של הגרפיקה הוא מצב טקסטואלי).

שלב שני – תרגום הגרפיקה שלנו לביטים שCards המספר יכול לענה (מה שנקרא פורטט של התמונה).

שלב שלישי – העתקת הביטים שמייצגים את הגרפיקה אל המקום הנכון ב-video memory. כמו עוד דברים באסמבלי, פעולה זו יכולה להתבצע ביותר מדרך אחת:

- אפשר להעתיק את המידע ישירות ל-video memory. זו דרך מהירה מאוד יחסית לשאר השיטות, החיסרון הוא שצורך לחשב בעצמו את הכתובת שצורך לפונט אליה.

- אפשר לקרוא לפסיקת BIOS. פסיקות BIOS הוזכרו בקצרה בפרק שדן בפסיקות. להזכירם, BIOS היא ספריית תוכנה של חברת אינטל. השימוש בפסיקות BIOS נותן לנו ממשק פשוט יחסית להתקני חומרה.

- אפשר להשתמש בפסיקות DOS, שכבר יצא לנו להכיר.

נסביר עכשו את שלושת השלבים, פעם ל-**text mode** ופעם ל-**graphic mode**.

גרפיקה ב-**Text Mode**

כמו שאפשר להבין, מצב טקסטואלי מיועד להציג טקסט העיקרי ולכך הוא מוגבל לדברים שאפשר לייצג באמצעות תווי ASCII. אכן, עם קצת יצרתיות ומשחק עם תווי ASCII אפשר להציג לדברים נחמדים מאוד.

שלב ראשון – קביעת מצב עובדה **text mode**. לשמהתו, זה מצב העבודה ברירת המחדל של DOS. במצב זה המסך מחולק ל-25 שורות ו-80 עמודות של תוויים.

0,0	0, 79
...				...
...				...
...				...
24, 0	24, 79

אם אנחנו לא נמצאים ב-**text mode**, נוכל לעבור אליו באמצעות פסיקת BIOS, int 10h. קיראו לפסקה באופן הבא:

```
mov ah, 0
```

```
mov al, 2
```

```
int 10h
```

לביצוע השלב השני והשלישי יש כמה שיטות. נראה עכשו שתי דרכים – יש יותר מכך, ואפשר גם לבצע שילובים.

שימוש במחוזות ASCII

שיטה פשוטה אבל בעלת תוצאות מקסימות היא להגדיר ב-**DATASEG** אוסף של תוויים שיוצרים ציור או כתובת. לחילוף, במקום לבצע את ההגדלה ישירות ב-**DATASEG**, אנחנו יכולים להגדיר קובץ, וואז לעשות לו include לתוכו **.DATASEG**.

לדוגמה הקובץ **:monalisa.asm**

monalisa db "~-` .-/v\` .-/v@@@A)
db "~-` .-/v\` ,ivJ@@@!
db "~-` .-/v\ i\G@@@Z-
db "~-` .-/v\ , -|B@@@!\
db "~-` .-/v\ tt@@@@A,
db "~-` .-/v\)8@@@@@\
db "~-` .-/v\]Z@@@@d|-
db "~-` .-/v\ KN8@@@@ (.!vGG_
db "~-` .-/v\)8K@@@@K@b@p@~~T4(
db "~-` .-/v\|48@K@@@@d@*@@bVi
db "~-` .-/v\,\Kb@@@d@.~t` !*~!`.
db "~-` .-/v\, 8M@@@@ @` ,gvz`
db "~-` .-/vi\@8@K@@@D
db "~-` .-/ve8d5@00000
db "~-` .-/v8d28@M@@@-
db "~-` .-/v@b@AK@@@b@[
db "~-` .-/v@@@@M@@@@P-
db "~-` .-/v@@@@MA@@@@`- ._)g2i
db "~-` .-/v@@@@K@b@K@)i 'c, Kb@bK
db "~-` .-/v@Kb@@@@A@A@/i- ~M@@@@Mc
db "~-` .-/v@@@@Mb@@@@((c`\ PPK((,i)v|-\v)8XNaD MK@|||||||b@EMK@|||||||[" ,13,10
db "~-` .-@8@8@@MK@d@A@L!--c)s_ , (ZsLB@`-\ ..N]/KM@|||||||d@||@|||||||d@[" ,13,10
db "~-` .08@deKb@@@@Kb@00/- !`~^Vff@N5f`- ,`))KK@|||||||MK@ed@||@|||||||M8d@b@b@([" ,13,10
db "~-` .08@d@b@00@KAK@00002- ,_,JJ@)/|-/(v)NK@8d@|||||||8@|||||||8@|||||||MK@[" ,13,10
db "~-` @8dv@8@K@b@00@000d!, 'VVV)\`)\`7(-)4Jb@8@A@K@d@|||||||8@|||||||8@|||||||[\$"

הקבוץ הזה הוא בסך הכל מחרוזת ארוכה, שמורפרדת לשורות כדי שיהיה יותר נוח לעבוד איתה. שמו לב לסיים ה-\$ בסוף המחרוזת – אתם כבר בטח מבינים מדוע הוא שם.

.ah=9h את הדפסה למסך ובצע בשיטה המוכרת, באמצעות הפקודה של הדפסת מהירות למסך int 21h עם קוד ah=9h

דוגמא לתוכנית קצרה שמדפסה למסך את הקובץ `monalisa.asm` (שימו לב לכך שיש להוריד את הוראה `IDEAL` בראש התוכנית, על מנת שה-`include` יעבד):

```
;-----;  
; Display ASCII graphics  
;  
; Author: Barak Gonen 2014  
;  
-----;  
  
model small  
  
stack 100h  
  
DATASEG  
  
include monalisa.asm  
  
CODESEG  
  
start:
```

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

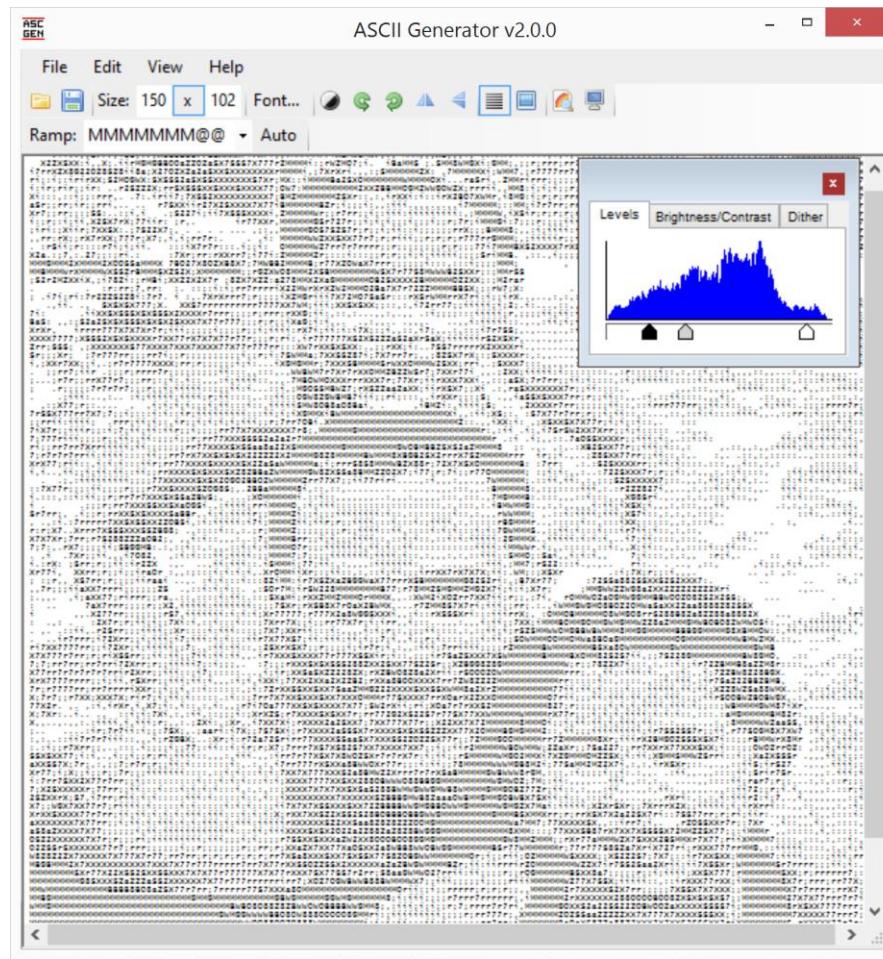
; Print string
mov    ah, 9h
mov    dx,offset monalisa
int    21h

; Wait for key press
mov    ah, 0h
int    16h

exit:
mov    ax, 4C00h
int    21h

end start
```

אפשר להפוך כל תמונה לתווית ASCII Generator כגון ש�示תנה להורדה מ-[לדוגמה: http://sourceforge.net/projects/ascgen2](http://sourceforge.net/projects/ascgen2)



אתם יכולים להשתמש במגוון המקורות הבאים, או להפץ בעצמכם, לדוגמה:

" how to generate ascii art"

For beginners:

www.en.wikipedia.org/wiki/ASCII_art_converter

ASCII Art Galleries:

<http://www.afn.org/~afn39695/collect.htm>

<http://chris.com/ascii/>

graphic mode ב-Video Graphics Array

עד כה דיברנו על תווים, עכשו נverb על דבר פיקסלים. פיקסל הוא היחידה הקטנה ביותר שנית את הערך שלו בمسך. פיקסל הוא כמו אטום, אבל של גרפיקה. מנות הפיקסלים במסך נקראת רזולוציה. אנחנו נדבר על פורמט VGA, קיצור של Video Graphics Array. סקירה של פורמט זה ניתנת למצוא במקומות רבים, מומלץ כרגע לחת את הכתוב הראשון לוויקיפדיה:

http://en.wikipedia.org/wiki/Video_Graphics_Array

פורמט VGA תומך במספר רזולוציות מסוים, אנחנו נתמקד ברזולוציה של 320X200, כלומר 200 שורות כפול 320 עמודות של פיקסלים.

כדי לעבור לפורט גרפי זה השתמש בפסקית BIOS, int 10h שנעשה חביבה עליינו בפרק זה:

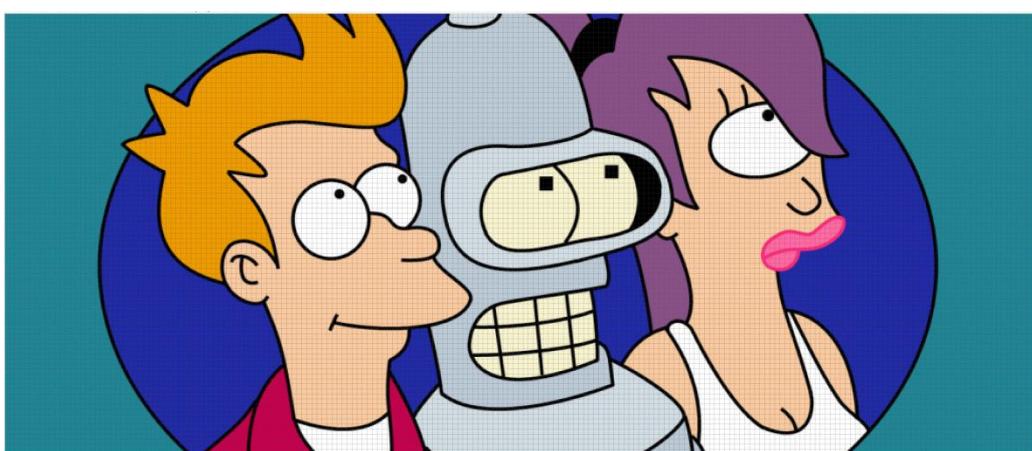
```
mov ax, 13h
```

```
int 10h
```

פסקה זו קובעת את מצב המסך שלנו, למטריצה של 200x320 פיקסלים.

0,0		...		0,319
...				...
199,0		...		199, 319

בתוך הדינמי memory, באוזר שמתחליל ב-A000:0000, שמור כל המידע הקשור לייצוג התמונה. בשביל כל פיקסל אנחנו צריכים לדעת שני דברים: קודם כל, מה המיקום שלו על המסך. אחר כך, מה הצבע של הפיקסל.



לכל פיקסל יש צבע אחר, שילוב של פיקסלים יוצר תמונה

המיקום של הפיקסל על המסך נקבע על ידי המיקום שלו בזיכרון: כל פיקסל מיוצג על ידי בית אחד. כך, הבית 0,0000:A000:0001 הבית 1 קשור לפיקסל 0:0, והבית 0:0000:C000 מציין את מספר השורה 320, מוסיפים את מספר העמודה X ופונים לזכרון במיקום שוחשב.

נעמיך את ההסבר על קביעת הצבע. כאמור, בפורמט VGA שאנו עוסקים בו, מספר הצבע מיוצג על ידי בית אחד, ככלומר 8 ביטים. מכאן שאנו יכולים לבחור $2^8 = 256$ אפשרויות של צבעים. בזיכרון המחשב יש טבלת המרה, שמירה כל מספר לשילוב של RGB, Red, Green, Blue – או בקיצור – RGBC. קיימים הרבה יותר מ-256 שילובים של RGB, אך טבלת memoryvideo ההמרה מכילה רק 256 אפשרויות (כדי שניתן יהיה לייצג צבע באמצעות בית בווד). נניח ששמננו ב-RGBם 0,0,0 – מספר צבע 0. כרטיס המスク יפנה לטבלת ההמרה למיקום 0, שם הוא ימצא – סביר להניח – שערכי ה-RGB הם 0,0,0. ככלומר מייצגים צבע שחור.

בזיכרון המחשב יש טבלת המרה סטנדרטית, שנקראת standard palette. אלו הצבעים שנשמרים ב-*standard palette*:



לעתים יש לנו תמונה, שהגוננים בה די דומים אחד לשני. במקרה זה 256 הצבעים הקיימים ב-*standard palette* לא מספיקים כדי לייצג את השוני בין הגוננים. עם זאת, אפשר לוותר על חלק מ-256 הצבעים ב-*standard palette*, היהו שאינם מיוצגים בתמונה, ולהחלף אותם בגוננים אחרים שיש להם ייצוג בתמונה. מיד נראה איך ניתן לשנות את ה-*palette* בრיתת המחדל, באמצעות טעינה של קוביץ בפורמט bmp שמכיל palette מותאם לקוביץ.

הדפסת פיקסל למסך

לאחר שסקרנו את המנגנון שעומד מאחורי הדפסת פיקסלים למסך, נותר לנו רק לתאר את הפקודות שגורמות לכך בפועל. האפשרות הראשונה היא פשוט לגשת ל-*video memory* ולהכניס לתוכו ערכים גדולים בית באמצעות פקודת *out*. לפני כן צריך רק לחשב את מיקום התא בזיכרון, כדי שנקלע לקוואורדינטות *y, x* הנכונות.

האפשרות השנייה היא לבצע אותה פעולה, אבל בעזרת פסיקת BIOS, קוד ah=0Ch. הפעיקה צריכה לקבל את הפרמטרים הבאים:

ah – צבע

al – עמוד (צריך להיות 0)

cx – קוואורדינטת X

dx – קוואורדינטת Y

```

; -----
; Paint a red pixel in the center of the screen
; Author: Barak Gonen 2014
; -----


IDEAL

MODEL      small

STACK      100h

DATASEG

x      dw      160
y      dw      100
color  db      4

CODESEG

start:

        mov    ax, @data
        mov    ds, ax
        ; Graphic mode
        mov    ax, 13h
        int    10h
        ; Print red dot
        mov    bh,0h
        mov    cx,[x]
        mov    dx,[y]
        mov    al,[color]
        mov    ah,0ch
        int    10h
        ; Wait for key press
        mov    ah,00h

```

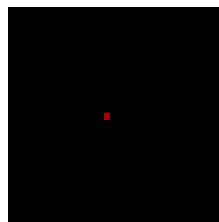
```

int      16h
; Return to text mode
mov     ah, 0
mov     al, 2
int      10h

exit:
mov     ax, 4c00h
int      21h

END start

```



הדפסת פיקסל למסך

קריאה ערך הצבע של פיקסל מהמסך

פונקציית BIOS `int 16h ah=0Dh` מאפשרת לנו גם לקרוא את ערך הצבע של פיקסל מסווג מהמסך, כאשר מבצעים קריאה עם `ah=0Dh`. אנחנו נתייחס בקצת לאפשרות זו, כיוון שהיא יכולה להיות שימושית במקרים רבים. לדוגמה – נניח שאנו כותבים משחק סנייק, ואני רוצח לבודוק האם הנחש שלנו התנגש בקיר. אנחנו יכולים פשוט לחתול לקיר צבע ייחודי ולבדוק את ערך הצבע במקום בו נמצא ראש הנחש. אם ערך הצבע שווה לצבע של הקיר – השחקן נפלט.

```
; Set graphics mode 320x200x256
```

```
mov     ax,13h
```

```
int      10h
```

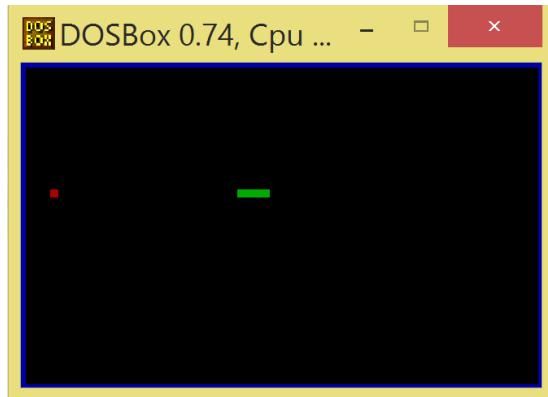
```
; Read dot
```

```
mov     bh,0h
```

```
mov     cx,[x]
```

```
mov     dx,[y]
```

```
mov ah,0Dh
int 10h ; return al the pixel value read
```



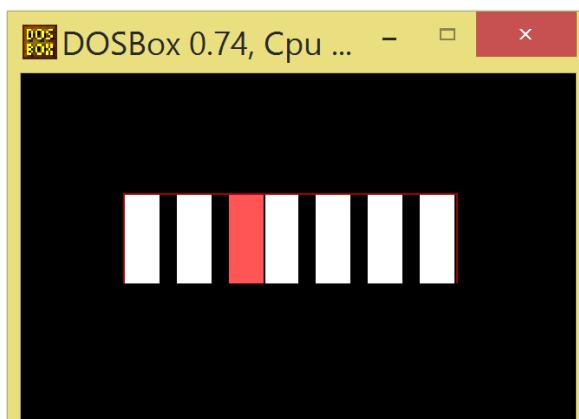
(משחק סנייק שבתוכנת באסמבלי (เครดיט: נעם אולאי)

יצירת קווים ומלבנים על המסך

ברגע שאנו יודעים ליצור פיקסל על המסך, אנחנו יכולים ליצור גם קווים ומלבנים (למעשה אנחנו יכולים ליצור כל שורה, אבל נתמקד בקווים ומלבנים לצורך הסביר).

הרענון הבסיסי מהורי יצירה קו, הוא שאנו קוראים לפרוזדורה שמדפסה פיקסל למסך, כל פעם עם ערך X גדול ב-1 (אם אנחנו רוצים קו אופקי) או עם ערך Y גדול ב-1 (כדי ליצור קו אנכי). יצירה קו אלכסוני דורשת שינוי גם בערך X וגם בערך Y . אם אתם רוצים ליצור קו אלכסוני, או מעגל, חפשו בגוגל “Bresenham algorithm” – נושא זה הוא ללימוד עצמי, מעוניינים בכך.

יצירת מלבן מتبוצעת באמצעות אוסף קריאות לפרוזדורה שיוצרת קו אופקי, כאשר בכל פעם נקודת התחלה בziej'ר ה- Y קטנה ב-1. כדי ליציג כפתור מלבן לחוץ ניתן לקבוע לו צבע אחר, או להקיף אותו במסגרת.



סימולציה של קלידי פסנתר, אחד הקלידים לחוץ (เครดיט: אליזבת לנגרמן)

קריאת תמונה בפורמט BMP

לעתים, במקרה לציר תמונה בכוחות עצמוו, נרצה לשלב בתוכנה שלנו תמונה נאה שמצאנו. קיימים פורמטים (הגדroots מקובלות לייצוג מידע בקבצים) רבים של תמונות, אנחנו נתמקד בפורט BMP, קיצור של Bit Map, משתי סיבות. הראשונה היא שהוא פורט נפוץ, והשנייה היא שהוא פשוט. את הנתונים שיש בקובץ BMP אפשר לטען בלי הרבה שינויים ומשחקים לתוך הדעת memory video, זאת בניגוד לפורמטים אחרים, כמו JPG הנפוץ, שמצריך אלגוריתמיקה מורכבת לפתחה.

נתאר בקיצור רב את פורט BMP. לפני כן, נקדים ונאמר שאת הלימוד של פורט BMP מומלץ وكل לעשות עליידי היפוש באינטרנט. הסברים מפורטים ניתן למצוא באמצעות היפוש **in assembly** "read bmp file". קישורים לדוגמה:

www.brackeen.com/vga/bitmaps/html

www.ragestorm.net/tutorial?id=7

פורט BMP מרכיב מהשדות הבאים:

1. Header – פtile בן 54 בתים. בתחילת header נמצאים התווים 'BM' שמצוינים שהקובץ הוא בפורט .BMP.

2. Palette – 256 צבעים, כל צבע תופס ארבעה בתים (בסך הכל 1,024 בתים). שימוש לב, שבעוד שכרטיס המסך מקבל את הצבעים בפורט RGB (כלומר, אדום–ירוק–כחול), הצבעים שבpalette של ה-BMP שמורים בפורט BGR (כחול–ירוק–אדום), ולכן נדרש להפוך את סדר הצבעים לפני שנטען אתpalette לכרטיס המסך.

3. Data – המידע על כל פיקסל ופיקסל שמור בבית יחיד. הבית הזה מכיל מספר בין 0–255, שהוא מספר הצבע干部队伍. שימוש לב, שהשורות שמורות מלמטה למעלה. ככלומר, אם ניקח את data ונווטיק אותו כמו שהוא לתוך ה-video memory, נקבל תמונה הפוכה. קבצי BMP יכולים להכיל מספר פיקסלים הרבה יותר גדול ממה שה-video memory של VGA יכול לקבל. אנחנו עוסקים בקבצים שהגודל המקסימלי שלהם הוא 320 עמודות כפול 200 שורות, או 64,000 פיקסלים.

נחבר את כל הדברים זהה עתה למדנו כדי לראות איך הצבעים מועברים למסך. נניח שיש לנו תמונה בגודל 200x320 פיקסלים. ניקח את הפיקסל הראשון בשורה התחתונה (שמוצג במאט למטה מצד שמאל). כיוון שי-BMP שומר את השורות הפוך, בתוך קובץ ה-BMP, זה הפיקסל שנמצא במקום הראשון ב-data. בambilים אחרות, הוא נמצא בבייה 1079 בתוך קובץ ה-BMP (1079=54+256x4).

נניח שהערך בבייה 1079 הוא '0'. המשמעות של ערך זה, היא שהצבע של הפיקסל השמאלי בשורה התחתונה הוא מה שנמצא באינדקס 0 ב-palette. אנחנו נגישים ל-palette, ערכי ה-BGR של הצבע באינדקס 0 נמצאים בתחילת ה-palette, בתים 54:57.

התוכנית הבאה קוראת ומדפסה למסך קובץ BMP בגודל 320 על 200. כדי ליצור קובץ זה, אתם יכולים להיכנס ל-*Microsoft Paint* ולשנות את גודל התמונה כך שהיא תהיה בגודל המתאים. אתם יכולים גם ליצור ולטען קבצים יותר קטנים, אך תצרכו לשנות את הגודלים שבתוכנה.

הסבר כללי על התוכנית:

- בתוכנית מוגדר קובץ בשם `test.bmp`. אתם יכולים לשנות את שם הקובץ. את הקובץ צריך לשים בספריה `.tasm/bin`.
- התוכנית פותחת את הקובץ לקרואיה.
- לאחר מכן נקרא ה-`header`.
- לאחר מכן נקרא ה-`palette`.
- ה-`palette` נתען לרטיס המסך, בפורטים `3C9h`, `3C8h`, `3CCh`. התוכנית משנה את סדר הערכיהם של ה-BGR כדי להפוך אותו ל-`RGB`.
- לאחר מכן התוכנית קוראת את ה-`data` שורה אחרי שורה, ומעתיקה ל-`video memory` בסדר הפוך כדי שהתמונה לא תהיה הפוכה.
- שימושו לב לחלוקת התוכנית לפrozדורות, דבר שמאפשר לדבג את התוכנית באופן פשוט יחסית.

```
; -----
; Read a BMP file 320x200 and print it to screen
; Author: Barak Gonen, 2014
; Credit: Diego Escala, www.ece.msstate.edu/~reese/EE3724/labs/lab9(bitmap.asm)
;
```

IDEAL

```
MODEL      small
STACK      100h
```

DATASEG

```
filename    db 'test.bmp',0
filehandle   dw ?
Header      db 54 dup (0)
Palette     db 256*4 dup (0)
ScrLine     db 320 dup (0)
ErrorMsg    db 'Error', 13, 10,'$'
```

CODESEG

```
proc OpenFile
; Open file
    mov ah, 3Dh
    xor al, al
    mov dx, offset filename
    int 21h
    jc openerror
    mov [filehandle], ax
```

```
ret
```

```
openerror:
```

```
    mov    dx, offset ErrorMsg
    mov    ah, 9h
    int    21h
    ret
```

```
endp OpenFile
```

```
proc ReadHeader
```

```
; Read BMP file header, 54 bytes
```

```
    mov    ah,3fh
    mov    bx, [filehandle]
    mov    cx,54
    mov    dx,offset Header
    int    21h
    ret
```

```
endp ReadHeader
```

```
proc ReadPalette
```

```
; Read BMP file color palette, 256 colors * 4 bytes (400h)
```

```
    mov    ah,3fh
    mov    cx,400h
    mov    dx,offset Palette
    int    21h
    ret
```

```
endp ReadPalette
```

```
proc CopyPal
```

```
; Copy the colors palette to the video memory
; The number of the first color should be sent to port 3C8h
; The palette is sent to port 3C9h

    mov    si,offset Palette
    mov    cx,256
    mov    dx,3C8h
    mov    al,0
    ; Copy starting color to port 3C8h
    out    dx,al
    ; Copy palette itself to port 3C9h
    inc    dx
```

PalLoop:

```
; Note: Colors in a BMP file are saved as BGR values rather than RGB.
        mov    al,[si+2]          ; Get red value.
        shr    al,2              ; Max. is 255, but video palette maximal
                                ; value is 63. Therefore dividing by 4.
        out    dx,al             ; Send it.
        mov    al,[si+1]          ; Get green value.
        shr    al,2
        out    dx,al             ; Send it.
        mov    al,[si]             ; Get blue value.
        shr    al,2
        out    dx,al             ; Send it.
        add    si,4              ; Point to next color.
                                ; (There is a null chr. after every color.)
```

```

loop  PalLoop

ret

endp CopyPal

proc CopyBitmap

; BMP graphics are saved upside-down.

; Read the graphic line by line (200 lines in VGA format),
; displaying the lines from bottom to top.

    mov    ax, 0A000h

    mov    es, ax

    mov    cx,200

PrintBMPLoop:

    push   cx

    ; di = cx*320, point to the correct screen line

    mov    di,cx

    shl    cx,6

    shl    di,8

    add    di,cx

    ; Read one line

    mov    ah,3fh

    mov    cx,320

    mov    dx,offset ScrLine

    int    21h

    ; Copy one line into video memory

    cld                                ; Clear direction flag, for movsb

    mov    cx,320

    mov    si,offset ScrLine

```

```

rep    movsb           ; Copy line to the screen
;rep movsb is same as the following code:
;mov  es:di, ds:si
;inc  si
;inc  di
;dec  cx
... ;loop until cx=0

pop   cx
loop  PrintBMPLoop
ret
endp CopyBitmap

```

start:

```

mov   ax, @data
mov   ds, ax
; Graphic mode
mov   ax, 13h
int   10h
; Process BMP file
call  OpenFile
call  ReadHeader
call  ReadPalette
call  CopyPal
call  CopyBitmap
; Wait for key press
mov   ah,1

```

```

int      21h
; Back to text mode
mov     ah, 0
mov     al, 2
int     10h

exit:
mov     ax, 4c00h
int     21h

END start

```

קטעי קוד וטייפים בנושא גרפיקה

פורום הפיאצה מכיל מידע איקוטי בנושא שימוש בגרפיקה, כולל קטעי קוד וטייפים. בחלון החיפוש אתם יכולים לחפש כל ביטוי. לדוגמה bmp או graphics. הדיוון הבא מומלץ במינוח:

<https://piazza.com/class/i98qbkdp1mg15m?cid=20>

דיוון זה נקרא "Graphics- advanced" והוא מרכז מספר נושאים. להלן הודעה שפותחת את הדיוון:

"שלום לכלום"

אני רוצה שדיוון זה ירכו פרקטיקות טובות לטיפול בגרפיקה, שלא ספק זהו נושא שמעסיק תלמידים רבים. לדיוון זה מוזמנים מאד לענות תלמידים שעשו פרויקטים עם גרפיקה איקוטית, והידע שלהם גם יסייע לאחרים וגם ישמר לשנתונם הבאים.

נא להעלות הסברים וקטעי קוד לנושאים הבאים:

1. איך מעלים תמונה BMP למקום מוגדר במסך? לדוגמה, ציר של מטרה שגודלה 10x10 פיקסלים במקומות 80,100.

2. איך מזיזים תמונה על המסך בצורה חלקה בלי ריצודים? לדוגמה, הליקופטר נע מצד לצד.

3. איך מזיזים תמונה על גבי תמונה רקע קבועה? לדוגמה, דמות של מריו קופצת מעלה ומטה כאשר הרקע מאחוריה נשאר קבוע.

4. איך יוצרים רקע דינמי? לדוגמה, תמונה של עננים שמכסה את כל המסך וכל התמונה נעה מימין לשמאל (המסוק נשאר במקום והרקע נע מאחוריו).

אין פרסים על תשובה, אבל מי שיעלו הסברים טובים, אשמה לשלב אותם בגרסת הבאה של ספר הלימוד + קרדיטים (:

ברכה

ברק"

เครดיטים על תשובה:

סעיף 1 - **יואב שטרנברג**, אהל שם רמת גן

סעיפים 2,3 – קורן ברנד, אהל שם רמת גן

סעיף 4 – שני שורץ, אורט בנימינה

השמעת צלילים

כפתיה לנושא השמעת צלילים, מומלץ לקרוא קודם כל תיאוריה על צלילים, גלי קול ותדריות. אתם יכולים למצוא מידע עצום של מקורות באינטרנט. מומלץ במיוחד להוריד את גרסה ה-*pdf* החינמית של הספר *Art of Assembly* ולקראן אודות הפיזיקה של גלי קול (*The Physics of Sound*).

נתמקד בחלוקת הטכנולוגיות של השמעת צלילים מהמחשב.

למבחן יש רכיב חומרה שצמוד אליו – טיימר, שעון, שלמדנו עליו בפרקם הקודמים. הטיימר מחובר לכרטיס הקול של המחשב ומעבר לו תקتوقي שעון, או "טייקם". כל "טייק" של הטיימרגורם לסגירת מעגל השגלי בכרטיס הקול, שמתרגם אותו לקול שאנחנו מסוגלים לשמוע. כמוות ה"טייקם" שmaguiim כל שנייה מהטיימר אל כרטיס הקול הם התדר. ככל שמתגברים יותר "טייקם" – התדר יותר גבוה. בספר המומלץ קיימת טבלה שמיירה בין תדרים לצלילים שנוכרים לנו. בעיקרונו יש חוקיות מתמטית פשוטה – כל אוקטבה מורכבת מ-12 צלילים. האוקטבה הבאה היא בתدر כפול. בתחום אוקטבה, הצלילים רוחקים זה מזה במכפלות של שורש 2 (בערך 1.06), כלומר אם התדריות של TWO מסויים היא 110 הרץ, אז התדריות של ה-TWO הבא אחריו היא $1.06 \times 110 = 122$, וכך הלאה.

על מנת לנגןתו כלשהו יש להפעיל קודם את הרמקול של המחשב (speaker). אנחנו עושים זאת עליידי פורט 61h, המקשר לטיימר. צריך לקרוא את הسطטוס שלו, לשנות את שני הביטים האחרונים ל-00, ולכתוב חזורה לפורט 61h. כך:

in al, 61h

or al, 00000011b

out 61h, al

הפסקת פעולה הרמקול:

in al, 61h

and al, 11111100b

out 61h, al

כעת, כאשר הרמקול פועל, עלינו לשולח את תדר התו שנרצה להשמי. על מנת לעשות זאת יש להשתמש בפורט 43h ובפורט 42h.

ראשית, אנחנו צריכים לקבל גישה לשינוי התדר ברמקול. יש להכניס את הערך הקבוע 0B6h לפורט 43h:

```
mov al, 0B6h
```

```
out 43h, al
```

לאחר מכן יש לשולח לדרכו של port 42h "מחלק" (divisor) בגודל 16 ביט, עבור התו שאנו רוצים להشمיע. המחלק יסמל את תדר התו הרצוי.

כדי לקבל את המחלק יש לחלק את הקבוע 1193180 בתדר של התו הרצוי. ככלומר:

$$\text{Divisor} = \frac{1193180}{\text{Frequency}}$$

תדר port 42h הוא בגודל 8 ביט בלבד, ולכן יש לשולח את המחלק הרצוי בשני חלקים – ראשית את הבית הפחות משמעותית, ולאחריו המשמעותי.

למשל, עבור התו "לה" של האוקטבה הראשונה, שהתדר שלו הוא 440 הרץ, המחלק יהיה 2712, ובבסיס הקסדצימלי 0A98h. לנוכח השילוח מתבצע כך:

```
mov al, 98h
out 42h, al      ; Sending lower byte
mov al, 0Ah
out 42h, al      ; Sending upper byte
```

לסיכום, תוכנית דוגמה, המשמיעה צליל בתדר 131 הרץ.

```
; -----
```

```
; Play a note from the speaker
```

```
; Author: Barak Gonen 2014
```

```
; -----
```

```
IDEAL
```

```
MODEL     small
```

STACK 100h

DATASEG

note dw 2394h ; 1193180 / 131 -> (hex)

message db 'Press any key to exit',13,10,'\$'

CODESEG

start:

mov ax, @data

mov ds, ax

; open speaker

in al, 61h

or al, 00000011b

out 61h, al

; send control word to change frequency

mov al, 0B6h

out 43h, al

; play frequency 131Hz

mov ax, [note]

out 42h, al ; Sending lower byte

mov al, ah

out 42h, al ; Sending upper byte

; wait for any key

mov dx, offset message

mov ah, 9h

int 21h

```
mov ah, 1h
int 21h
; close the speaker
in al, 61h
and al, 11111100b
out 61h, al
exit:
mov ax, 4C00h
int 21h
END start
```

שעון

הקדשו לנושא הטימר סעיף משלו בפרק על הפסיקות – חיזרו על סעיף זה ובידקו שאתם מבינים את התיאוריה לפני שאתם מתקדמים הלאה. כזכור, אפשר לקרוא את השעה באמצעות פסיקה `h 21h int ah=2Ch`. במקרה זה אנחנו עוסקים בכלים לפרויקטים ולכון נתמך בשני כלים מעשיים שונים לנו השעון.

הכלי הראשון הוא יכולת למדוד פרקי זמן קצובים שקבענו מראש – נניח שאנו רוצים שפעולה כלשהי תבוצע כל זמן קצוב. לדוגמה:

- במשחק סנייק, קבוע כל כמה זמן הנחש יתקדם צעד אחד.
- כשהמנגנים צליל כלשהו, קבוע למשך כמה זמן יונגן הצליל.
- במשחק שחמט, קבוע פרק זמן מסוימלי לצעד.

הכלי השני הוא יכולה ליצור מספרים אקראים. זהו כלי חשוב אם אנחנו רוצים ליצור משחקים מעניינים, שלא יחוירו על עצם. לשם כך, אנו צריכים מגננון שנותן לנו ערכים משתנים בכל פעם שאנו מריםים את המשחק, ואת הערכים האלה אנחנו יכולים אחר כך לתרגם לדברים אחרים שאנו צריכים. לדוגמה:

- במשחק סנייק, הצבה של אוכל במקום אקרי בمسך.
- במשחקי קובייה, כמו סולמות וחלבים, קבוע את הערך של הקובייה.
- במשחקי פעולה, תזוזה אקראית של דברים על המסך.

מדידת זמן

דרך אפשרית אחת למדוד זמן שעובר היא על-ידי הפסיקה שקוראת את השעון – מבצעים לולאה של קריאה חוזרת לפסיקה, ובכל פעם משווים את השעה עם השעה הקודמת (מספיק להשוות את ערך מאות השניה, שנשמר לתוך `df` – אם הוא לא השתנה אזשאר הערכים בהכרח לא השתנו). הדרך זו נראה מורה מלאה ולכון לא עמוקה בה. הנקודה היחידה שצריך לשים לב אליה, היא שלמרות שהשעה משתנה כל 55 מילישניות, השינוי הראשון של השעה לא יהיה בהכרח אחרי 55 מילישניות. זאת מכיוון שהקריאה הראשונה של השעה בוצעה לא בדיק ברגע השנתונות השעה ולכון ייקח פחות מ-55 מילישניות עד שהשעה תשתנה. רק מהמודידה השנייה והלאה אנחנו יכולים להניח שהפרש הזמן הוא מדויק.

הדרך השנייה לדעת ש-55 מילישניות עברו היא להשתמש בעובדה שכל פסיקה של הטימר גורמת לעדכון של מונה שנמצא בכתובת `0040:0006Ch`. אנחנו לא נדע מה השעה, אבל כל שינוי בערך שנמצא בכתובת זו בזיכרון יציביע על כך שעברו 55 מילישניות. היתרון של השיטה זו הוא מהירות. קריאה ישירה תמיד תהיה מהירה יותר מפסיקה (עם כל הפעולות

הנוספות הייתה גוררת, כגון שמירה על נתונים במהלך קפיצות). באופן כללי, אם יש קוד שאנהנו מರיצים לעיתים קרובות או עדיף לשימוש בצד שיקולי הנוחות ולכתוב אותו בדרך היעילה יותר.

להלן דוגמה לתוכנית שמודדת פרק זמן של עשר שניות (בקרוב רב). הרעיון הוא כזה: בודקים את מצב המונה בכתובת 0040:006Ch מהכים שהוא השתנה פעמי אחת – כך אנחנו יודעיםמתי מתחילה פרק הזמן שאנו רוצים למדוד. אנחנו סופרים 182 שינויים של המונה (sec=10.01 sec) וכך אנחנו מודדים פרק זמן של עשר שניות.

; ; Produce a delay of 10 seconds (182 clock ticks)

; ; Author: Barak Gonen 2014

IDEAL

MODEL small

STACK 100h

DATASEG

Clock equ es:6Ch

StartMessage db 'Counting 10 seconds. Start...',13,10,'\$'

EndMessage db '...Stop.',13,10,'\$'

CODESEG

start:

mov ax, @data

mov ds, ax

; wait for first change in timer

mov ax, 40h

mov es, ax

mov ax, [Clock]

FirstTick:

cmp ax, [Clock]

```

je      FirstTick

; print start message

mov    dx, offset StartMessage

mov    ah, 9h

int    21h

; count 10 sec

mov    cx, 182      ; 182x0.055sec = ~10sec

```

DelayLoop:

```
mov    ax, [Clock]
```

Tick:

```

cmp    ax, [Clock]

je      Tick

loop   DelayLoop

; print end message

mov    dx, offset EndMessage

mov    ah, 9h

int    21h

```

quit:

```

mov    ax, 4c00h

int    21h

```

END start

יצירת מספרים אקראיים – Random Numbers

יצירת מספרים אקראיים באמצעות המחשב היא בעיה מעניינת מאוד עם השלכות פילוסופיות. בערךון, הטענה היא שאפשר ליצור באמצעות המחשב – שהוא מכונה דטרמיניסטית (לכל פעולה יש תוצאה אחת הניתנת לחיזוי), המחשב אינו מפעיל שיקול דעת או בוחר תוצאה באופן אקראי) – מספרים אקראיים באמצעות. כל מספר אקראי שהמחשב יוצר הוא למעשה תוצאה של חישוב שניית לשחזר אותו. למחשה פילוסופית זו יש השלכות מעשיות רבות, לדוגמה בעולם ההימורים אונליין.

לכן אנחנו נזכיר מושג שנקרא מספרים **פְּסָאוֹדוּ-אֲקָרְאיִים** (Pseudo-random). אלו מספרים שניית לשחזר אותם באמצעות ידיעת האלגוריתם ותנאי ההתחלה, אבל הם עדין עוניים לחוקים סטטיסטיים של אקראיות (הערכים מתפלגים בצורה אחידה, לא חוזרים על עצם וכו'). אין צורך שנזכיר יותר לעומק את המשמעות של מספרים פסאדו אקראיים, תלמידים המתעניינים בנושא מוזמנים לפנות ללמידה עצמי). מעכשו, בכל פעם שנכתוב "אקראי" נתכוון בעצם "פסאדו אקראי".

קיימות שיטות שונות ליצירת מספרים אקראיים. לא נוכל להתייחס במסגרת ספר זה לכלן ואנו לא למקצתן. נתאר אפוא שיטה אפשרית אחת, ותלמידים המעוניינים להרחב את הידע בנושא מוזמנים להמשיך את הלימוד באופן עצמאי.

השיטה מבוססת על מונה הטימר שהכרנו בסעיף הקודם – הכתובת Ch006:0040. אנחנו יכולים לחתך חלק מהביטים שלו וכך לקבל מספר אקראי בתחום שאנו רוצים.

לדוגמה, אם ניקח את הביט האחרון נקבל מספר אקראי: 0 או 1. קטע קוד שם מספר אקראי (0 או 1) בתוך al:

```
mov ax, 40h
```

```
mov es, ax
```

```
mov ax, es:6Ch
```

```
and al, 00000001b
```

אם ניקח את שני הביטים האחרונים נקבל מספר אקראי מבין: 00, 01, 10, 11. כלומר, מספר בין 3–0. נבצע זאת באמצעות שינוי קטן בשורה الأخيرة:

```
and al, 00000011b
```

וכך הלאה. אנחנו יכולים לקבל מספר אקראי בתחום 0–15, 7–0 וכו'.

שימוש לבכך שבשיטה זו יוצרת מספר אקראי בין 0–9, או בכל תחום אחר שאינו חזקה של 2, היא אינה פשוטה. לעומת זאת אפשר לקבל את התחום בין 0–9 על ידי חיבור של שלוש פעולות ליצירת מספרים אקראיים:

- יצירה מספר אקראי בתחום 0–1.

- יצירה מספר אקראי נוסף נוסף בתחום 0–1.

- יצירה מספר אקראי נוסף נוסף בתחום 0–7.

הבעיה הראשונה בשיטה זו היא שההתפלגות של המספרים כבר אינה אחידה. לדוגמה, יש רק אפשרות אחת לקבל תוצאה 0 (המספרים שנוצרו הם 0,0,0) אבל יש יותר מכך אחד שモוביל לתוצאה 7 (0,0,7) או 1,1,5 או 1,1,6 או 1,5,1 או 6,1,0 ועוד...).

פתרון פשוט הוא ליצור מספר אקראי בתחום גודל יותר מהתחום שאנו צריכים, ואם המספר שנוצר לנו הוא מחוץ לתחום המבוקש – לזרוק אותו ולבחרו במספר אחר. לדוגמה, כדי ליצור מספר אקראי בתחום 0–9, ניתן ליצור מספר אקראי בתחום 0–15 ונזרוק את כל המספרים שגודלם מ-9.

הבעיה השנייה בשיטה זו, היא קצב הייצור של מספרים אקראיים. קצב זה תלוי בקצב השינוי במונח שככובתה Ch 0040:006, שכאמור מתעדכן כל 55 מילישניות. לעומת, אם נרצה ליצור כמה מספרים אקראיים בפרק זמן קצר, לא נקבל גיון בתוצאות, או שנקבל מספרים שיש ביניהם קשר ברור. לעומת, השיטה הזו טובה בעיקר אם רוצים ליצור מספר אקראי פעם אחת לפרק זמן ארוך יחסית לפרק הזמן של עדכון הטימר.

פתרון אפשרי הוא לבצע פעולה מתמטית כלשהי על הביטים שאנו קוראים מהטימר, כך שגם אם קראנו אותם הביטים מהטימר במספר קריאות נפרדות, אכן נפיק מהם ביטים אחרים בכל פעם. אנחנו יכולים, לדוגמה, לעשות לביטים שקראנו מהטימר xor עם ביטים במקום כלשהו בזיכרון – בכל פעם עם מקום אחר. להלן, ניתן לחתות קובץ כלשהו, לקרוא ממנו בית ולעשות אותו xor. בכל פעם שנקרא מהקובץ, נקרא בית אחר.

להלן דוגמה לתוכנית שיזכרת מספרים אקראיים באמצעות שילוב של קריאת הטימר וxor עם ביטים שנמצאים ב-GCODESEG. נסכם את הסעיף על יצירה מספרים אקראיים בכך שקיימות שיטות רבות וטובות מזו ליצירת מספרים אקראיים, וכי שמתעניין בנושא יוכל למצוא עוזר של מידע באינטרנט.

; -----;

; Generate 10 random numbers between 0–15

; The method is by doing xor between the timer counter and some bits in GCODESEG

; Author: Barak Gonen 2014

; -----;

IDEAL

MODEL small

STACK 100h

DATASEG

Clock equ es:6Ch

EndMessage db 'Done',13,10,'\$'

divisorTable db 10,1,0

CODESEG

proc printNumber

push ax

push bx

push dx

mov bx,offset divisorTable

nextDigit:

xor ah,ah ; dx:ax = number

div [byte ptr bx] ; al = quotient, ah = remainder

add al,'0'

call printCharacter ; Display the quotient

mov al,ah ; ah = remainder

add bx,1 ; bx = address of next divisor

cmp [byte ptr bx],0 ; Have all divisors been done?

jne nextDigit

mov ah,2

mov dl,13

int 21h

mov dl,10

int 21h

```

pop    dx
pop    bx
pop    ax
ret
endp  printNumber

```

```

proc  printCharacter
      push   ax
      push   dx
      mov    ah,2
      mov    dl, al
      int    21h
      pop    dx
      pop    ax
      ret
endp  printCharacter

```

start:

```

mov    ax, @data
mov    ds, ax
; initialize
mov    ax, 40h
mov    es, ax
mov    cx, 10
mov    bx, 0

```

RandLoop:

```
; generate random number, cx number of times

mov ax, [Clock]           ; read timer counter
mov ah, [byte cs:bx]       ; read one byte from memory
xor al, ah                ; xor memory and counter
and al, 00001111b         ; leave result between 0-15
inc bx
call printNumber
loop RandLoop

; print exit message

mov dx, offset EndMessage
mov ah, 9h
int 21h

exit:
mov ax, 4c00h
int 21h

END start
```

ממשק משתמש

בסעיף זה עוסוק בשני כלים שכלי תוכנית שעובדת עם ממשק משתמש צריכה – מקלדת ו/או עכבר.

קליטת פקודות מהמקלדת

בפרק על הפקות כיסינו את התיאוריה והפרקטיקה של פעולה המקלדת ואת הגורמים הקשורים לפעולה של המקלדות:

Scan Codes -

- פורטים שקשורים לעובדה עם המקלדת

- שימוש בפסיקות BIOS לקליטת תווים וניקוי באפר המקלדת

- שימוש בפסיקות DOS לקליטת תווים וניקוי באפר המקלדת

עקרונית, עברנו על כל הטכניקות שצורך בשבייל לשלב מקלדת בפרויקט הסיום. בסעיף זה ניתן דגשים לעובדה עם המקלדות:

.1. כאשר אנחנו צריכים להכנת פרויקט שמשלב קליטת פקודות מקלדת, אנחנו יכולים להחליט באיזו גישה

לעבוד (פורטים / BIOS / DOS). כמו שראינו, פסיקות BIOS מאפשרות לנו לקבל קלט מבלי לעזר

את ריצת התוכנה, ופסיקות DOS מאפשרות לנו לחכות למשתמש. כמובן, המימוש המומלץ תלוי בזורה

בה אנו רוצים שהתוכנה תפעל.

.2. שימוש בפסיקת ה-int 21h DOS מוחיר את קוד ה-ASCII של המקש שהוקלד. ישנו מקשיים –

ודווקא אולי שצורך להשתמש בהם במקרים (מקשי חיצים לדוגמה) – שיש להם קוד ASCII מורחב.

בקוד ASCII מורחב, לטור באפר המחסנית מועתק קוד ה-ASCII במקום של ה-int scan code, ובמקום

של קוד ה-ASCII מועתק אפס. מבלבל? נכון. מסיבה זו, עדיף תמיד לבצע את פעולות הבדיקה

וההשוואה עם ה-int DOS. גם אם משתמשים בפסיקת DOS, עדיף להוסיף אחרת:

in al, 60h

קליטת פקודות מהעכבר

קליטת פקודות מהעכבר מתבצעת על-ידי הפסיקה int 33h. במסגרת סעיף זה נסביר רק את התמצית של העבודה עם העכבר בסביבת DOS, תוכלו להגיע بكلות לחומרים נוספים על-ידי חיפוש "int 33h mouse function calls" בגוגל. בדרך זו תוכלו למצוא קודים נוספים שיאפשרו לכם יכולות נוספות, שאינן מכוסות בסעיף זה.

לפניהם שפעיל את העכבר, נבעור למוד גרפי, כדי שלמדנו לעשות מוקדם יותר בפרק זה:

mov ax,13h

```
int    10h
```

העכבר נשלט בידי פרוצדורות שונות שפעילה פסיקה 33h. הקודים נשלחים על גבי ax (שים לב, ax ולא ah כמו שאנו רגילים עם int 21h). כדי להפעיל את העכבר, ראשית יש לאתחל אותו ואת החומרה שלו. הקוד ax=0h מבצע את פעולה האתחול:

```
mov  ax,0h
```

```
int  33h
```

כעת נציג את העכבר על המסך, קוד h=1ax=0:

```
mov  ax,1h
```

```
int  33h
```

השלב הבא הוא לקלוט את מיקום העכבר ואת סטטוס הלחיצה עליו:

```
mov  ax,3h
```

```
int  33h
```

הפסקה מוחירה את הערכים הבאים:

- ax – מצב הלחיצה על כפתורי העכבר

○ xxxx xxxx – הביט במקום 0 שווה '1' – כפתור שמאלית לחוץ.

○ xx xxxx – הביט במקום 1 שווה '1' – כפתור ימני לחוץ.

כלומר, כל ערך של ax שהביטים הימניים שלו הם '00' – משמעתו שאין כפתור לחוץ.

- cx – מיקום העכבר, שורה בין 0–639 (שים לב, כאשרנו עובדים במוד גרפי כמוות השורות שיש לנו היא 320 בלבד, לכן צריך לבצע התאמת ולחילק את cx בשתיים כדי להגיע למיקום הנכון).

- dx – מיקום העכבר, עמודה בין 0–199

התוכנית הבאה משלבת את האלמנטים שסקרנו. לחיצה על המקש השמאלי של העכבר תוביל להופעת נקודה אדומה במיקום העכבר. לאחר מכן, לחיצה על המקלדת תגרום ליציאה מהתוכנית.

```
; -----  
; PURPOSE : Paint a point on mouse location, upon left mouse click  
; AUTHOR : Barak Gonen 2014  
;  
IDEAL  
MODEL small  
STACK 100h  
DATASEG  
color db 12  
  
  
CODESEG  
  
start:  
    mov ax,@data  
    mov ds,ax  
    ; Graphics mode  
    mov ax,13h  
    int 10h  
    ; Initializes the mouse  
    mov ax,0h  
    int 33h  
    ; Show mouse  
    mov ax,1h  
    int 33h  
    ; Loop until mouse click
```

MouseLP:

```

mov    ax,3h
int    33h
cmp    bx, 01h      ; check left mouse click
jne    MouseLP
; Print dot near mouse location
shr    cx,1          ; adjust cx to range 0-319, to fit screen
sub    dx, 1          ; move one pixel, so the pixel will not be hidden by mouse
mov    bh,0h
mov    al,[color]
mov    ah,0Ch
int    10h
; Press any key to continue
mov    ah,00h
int    16h
; Text mode
mov    ax,3h
int    10h
exit:
mov    ax,4C00h
int    21h
END start

```

Debug ניפוי

עד עכשו עסקנו בתוכניות לא ארוכות במיווחד – את כל התרגילים שבספר הלימוד אפשר לבצע עם תוכניות מסדר גודל של מהה שורות קוד, ובדרך כלל הרבה פחות. לעומת זאת, בפרויקט סיום יש לעלה לפחות שורות קוד. אפילו אם אתם מתכוונים לעובדים ויש לכם באג רק אחת למאה שורות קוד, זה עדין אומר שתיקלו בעשרה באגים לפני שהתוכנית שכתבתם תעבור כמו שחכנתם. אנחנו עוד לא מדברים על באגים שנובעים מבעיה בתוכנון, כזו שתאלץ אתכם להבהיר חלקן קוד מקום למקום, או באג באлогריתם, שעלול לגרום לכם לבנות ערבות מול מסך המחשב בעודכם כוסים ציפורניים, נעימים בין ייאוש לזעם.

אין הכוונה לייאש אתכם, רק לציין את המובן מאליו – כתיבת תוכנית גדולה דורשת מילונות תכנות ושימוש בטכניקות דיבוג, שאינן נחוצות בשבייל כתיבת תוכנית קטנה כמו התוכניות שנתקלנו בהן עד עכשו.

אנו נסוק עכשו בטכניקות שונות שיכלו, אולי, לעזור לכם לצלוח יותר בבעיות את אותן ערבים בהם התוכנה לא עובדת ואין לכם מושג למה.

тиיעוד

כתבת תוכנית קטנה אורך מעט זמן, אז כשאתם מגיעים לסוף התוכנית אתם עדין זוכרים מה כתבתם בהתחלה. כתיבת תוכנית גדולה אורך הרבה זמן. אל תניחו שאחרי שבועיים של כתיבה תזכרו מה הפרמטרים שמקבלת פרוצדורה שכתבتم. תתעדו. רמה נכונה של תיעוד היא זו:

- בתחילת כל פרוצדורה, הסבר מי נגד מי: ציון פרמטרים ותוצריים.
- אחת לפחות שורות קוד, לרשות הערכה מה עשו קטע הקוד הבא.
- אם יש שורה מסובכת, כזו שלא מובן מליין מה היא צריכה להיות שם, הוסיפו הערת הסבר לצד. לדוגמה, בתוכנית לדוגמה על הפעלת העבר, איך נזכר למה חילקנו את ax ב-2 אם לא נסביר הצד?

```
shr    cx,1      ; adjust cx to range 0-319, to fit screen
```

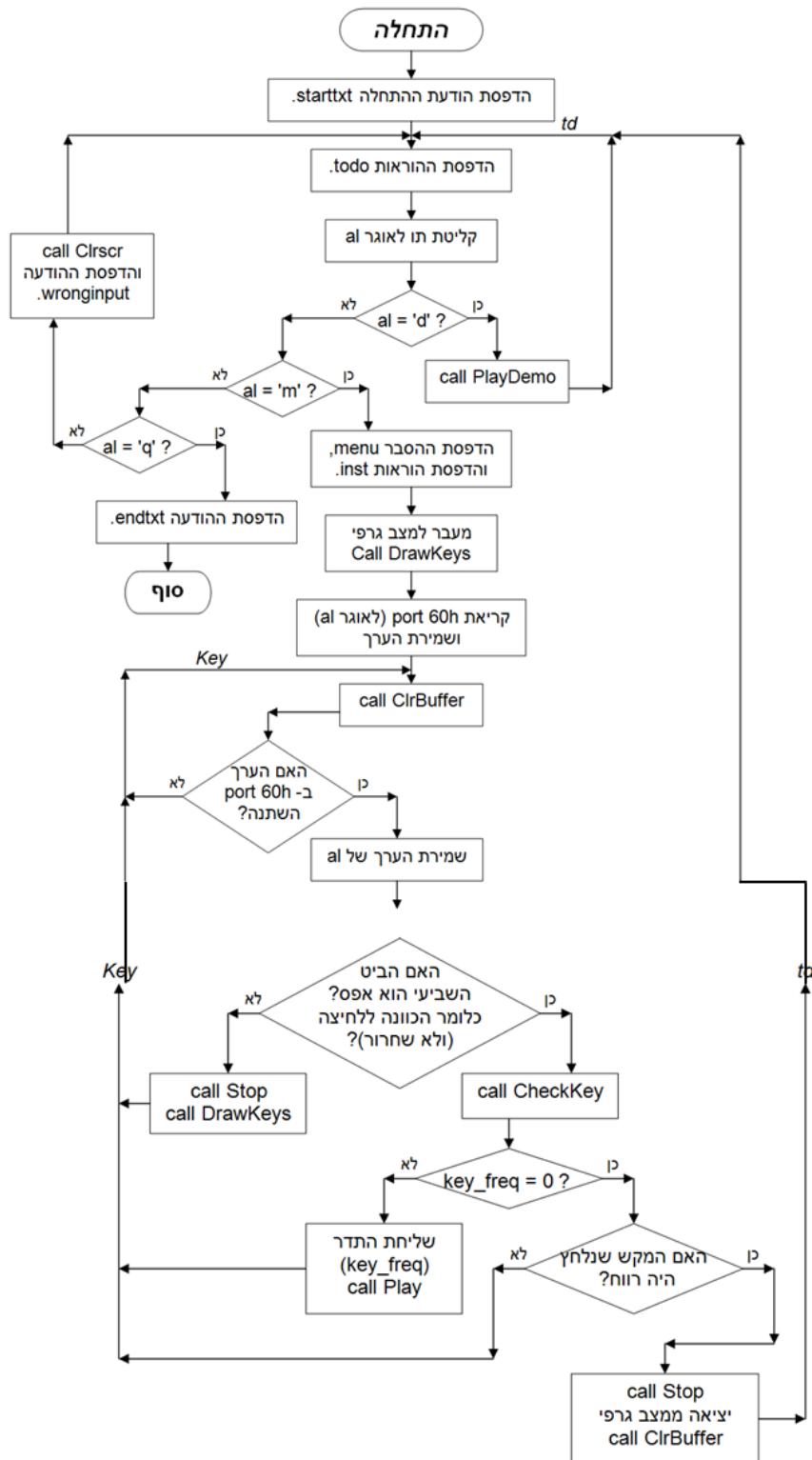
עם זאת, חשוב לא להציגם. מתקנים מתחילה נוטים לעתים להוסיף תיעוד רב, שרומו מיותר, או שאינו מסביר למה שעשינו את מה שעשינו. דוגמה לתיעוד שלא משרת שום מטרה טובה:

```
mov    ax, 5      ; copy '5' into ax
```

תוכנון מוקדם – יצירה תרשימים זרימה

תרשימים זרימה לכל התוכנית יכול להקל עליכם לעשות סדר בתוכנית המורכבת שלכם ולמנוע טעויות תכנון שיגולו מכמ זמן יקר בהמשך. אל תוותו על תיאור מסויר של התוכנית שלכם לפני שאתם מתישבים לתוכנת. בכל אופן, גם אם תוותו לעצמכם, משרד החינוך דורש שלכל תוכנה יתלווה תרשימים זרימה, כך שבעל מקרה עלייכם להכין אחד... עדיף שתתחלו מתרשימים זרימה טוב שיעשה לכם סדר بما שאתם מתכוונים לתוכנת.

لتרשימי זרימה יש שפה מיוחדת. כדי להקל על הקראיה וההבנה יש טקסט ששמות במלבן, טקסט ששמות בתוך מעוין, טקסט ששמות בעיגול וכו'. תוכלו ללמידה בקלות את שפת תרשימי זרימה באינטרנט (חפשו "איך לכתוב תרשימים זרימה").



וונגה לתרשים זרימה של פסנתר (אליגצת לנגרמן)

חלוקת לפרויקטים

קרוב לוודאי שאחרי שיצרתם תרשימים זרימה, זההיתם מספר קודי שצרכים לרוץ מספר רב של פעמים. אלו הן הפרוצדורות שאתם מיעדים לכתוב. אם אתם רוצים שהיה לכם סיכוי סביר למצוא באגים, אתם צריכים לחשוף היבט על החלוקה לפרויקטים. היתרונו העיקרי של פרוցדורות הוא בשלב הדיבוג – אם כתבתם פרוցדורה טובה ובדקתם אותה, אתם לא צריכים להיות מודאגים בקשר אליה. אנחנו נציג כרגע כמה דרכים לדבג תוכנית שכוללת פרוցדורות:

- לחפש בשיטת "אריה במדבר" – אם אנחנו רוצים למצוא אריה שמסתתר במדבר, אנחנו נחלק את המדבר לשטחים, נקייף כל שטח בגדיר, ואחרי שנסיים לסרוק את השטח המוגדר נמשיך לשטח הבא. ניקח בתור דוגמה את הקוד העיקרי של התוכנית שקוראת לתמונה מקובץ bmp:

```
; Process BMP file
```

```
call    OpenFile
call    ReadHeader
call    ReadPalette
call    CopyPal
call    CopyBitmap
```

נניח שהתוכנית שלנו פשוט לא פועלת. אנחנו מריםים אותה והתמונה לא עולה. פשוט כלום לא קורה. מה עושים? בשיטת "אריה במדבר" נכניס את הפרויקטים שלנו להערכה, ו"נשחרר" אותן מההערכה אחת אחת, אחרי שווידאנו שהן עושות מה שצריך. לדוגמה, הצעד הראשון יהיה:

```
; Process BMP file
```

```
call    OpenFile
; call  ReadHeader
; call  ReadPalette
; call  CopyPal
; call  CopyBitmap
```

נՐץ את התוכנית, שכרגע כל מה שהיא עושה זה רק לפתוח קובץ. נבדוק שהקובץ נפתח בצורה תקינה (לא מוחזר קוד שגיאה). הכל תקין? המשכנו הלאה.

- לקרוא זיכרון תוך כדי ריצה. ה-DT מספק לנו תמייה טובה בבדיקה מצב הזיכרון תוך כדי ריצה. נניח שהוצאנו את הקראיה ל-*ReadHeader* מהערכה, אנחנו רוצים לבדוק אם הפרויקטורה עובדת בצורה תקינה. בתוך *header*.*header* נוכל לראות איך *header* נראה בזיכרון מיד בסיום DATASEG הגדרנו מערך ששומר את ה-

הקריאה. אנחנו יודעים שני הบทים הראשונים שלו צריכים להיות 'BM' – אם קיבלנו ערך אחר, יש לנו בעיה בקריאה.

להחליפ פרוצדורה בקטעי קוד קבועים, שאחנו יודעים מה הם צפויים לבצע. נניח שיש לנו פרוצדורה שבודקת איזה מקש נלחץ על-ידי המשתמש, והיא מעבירה את המידע לפרוצדורה אחרת ש羞שה שימוש במידע זהה, נניח כדי להזין אלמנט גրפי על המסק. נניח כי הצירוף של הפרוצדורות לא עובד היטב, אבל מסובך לנו לבדוק באיזו פרוצדורה נמצאת הבעיה. אנחנו יכולים להוסיף קטע קוד, שאומר שלא משנה איזה מקש נלחץ על-ידי המשתמש, התוצאה תידرس ובמקרה יוותק ערך קבוע. אם הבעיה ממשיכה, כנראה שיש בעיה בפרוצדורה שמזינה את הגרפיקה על המסק.

שים לב, שכדי שתוכלו לדבג בצורה ייעלה, החלוקה לפרוצדורות צריכה להיות של קטעי קוד SMBצעים מישימות קטנות ומוגדרות היטב. חישבו תמיד אם אפשר לפצל את המשימה שהפרוצדורה שלכם מבצעת ליותר פרוצדורה אחת, אם יש קטעי קוד שהווים על עצם בתוך הפרוצדורה שלכם, ואם הפרוצדורה לא אורך מדי.



מעקב אחרי מונחים

مونחים, כאלה שסופרים בשביבנו כמה פעמים לולאה שכתבנו צריכה לרוץ, הם מקור נפוץ לשגיאות ובעיות. לכן, אם יש בעיה בקוד שלכם, שווה לבדוק אם אפשר לפצל את המשימה שהפרוצדורה שלכם מבצעת ליותר פרוצדורה (לדוגמה, הרצתם לולאה או קראתם לפרוצדורה שמשנה את CX).

אם אתם לא שמים לב בעיה, מומלץ להכנס לתוכנית שלכם קטעי קוד SMBدافים את הערכיהם של המונחים למסק או אפילו לקובץ. כך תוכלו לוודא שהמונחים שלכם מקבלים רק ערכים בתחום הצפוי והגיוני. יוזר לכם אם כתבו פרוצדורה או מקרו, שמקבלים כפרמטר רегистר ומדפסים את הערך שלו למסק או לקובץ, במקרה למש את הלוגיקה הזה בכל פעם חדש.

העתיקות זיכרון

עוד באג נפוץ מאוד הוא העתקה למקום לא נכון בזיכרון. אין הכוונה להעתיקה למקום לא נכון בזיכרון, בעיה נפוצה בפני עצמה, אלא להעתיקה של מידע מוחוץ למערך שהגדרתם. העתקה זו עלולה לדורס לכם ערכים אחרים ב-SEG, או – אם ממש יש לכם מזל – לדורס ערכים ב-SEGCODE וליים ברישוק התוכנית.

נקודה נוספת שקשורה להעתיקות לזכרון, היא שלעיתים אנחנו צריכים לשנות את הערכים של רегистרי הסגמנט. לאחר השינוי צריך להחזיר אותם לUMB, אחרת אנחנו משתמשים בתוצאות בלתי צפויות בכל פניה לזכרון.

הודעות שגיאה של האסמבולר

לעתים האסמבולר יגרום לכם להרגיש חסרי מזל במיוחד, כשהתתקלו בהודעת שגיאה שלא נתקלתם בה בעבר, לדוגמה: A2034: must be in segment block העתיקו את קוד השגיאה למסך החישוף, קרוב לדאי שתגיעו לפורום מתכנתים (לדוגמה StackOverflow) ותמצאו שמשיחו כבר נתקל בבעיה שלכם וקיבל עזרה.

תרגיל הכנה לפרויקט הסיום



בתרגיל זה נתרגל שימוש במספר אבני בניין שנחוצות לצורך פרויקטי הסיום

א. כתבו תוכנית, שעם לחיצה על מקש כלשהו במקלדת תדפיס למסך הודעה "Key pressed" ועם שחרור המKeySpec תדפיס הודעה "Key released". לחיצה על ESC תסימן את ריצת התוכנית. שימו לב - לחיצה ארוכה נחשבת לחיצה אחת. לכן, הדרך הנכונה לבצע את הבדיקה היא לשמר את ה-`code`-`scan` לאחרון שהגיע ממקלדת ורק אם הוא השתנה אז להדפיס הודעה בהתאם.

ב. שדרגו את התוכנית, כך שללחיצה על המקלדת גם תשמע צליל כלשהו. הצליל יופסק רק עם שחרור המקלדת.

ג. שדרגו את התוכנית, כך שבמוקום להדפיס הודעה למסך יצירר ריבוע אדום 5X5 פיקסלים במרכז המסך. הריבוע יופיע עם לחיצה על מקש ויעלם עם שחרור המKeySpec. טיפים:

א. מומלץ לצייר את הריבוע באמצעות לולאה ופrozדורות, ולא באמצעות שכפול קוד של ציור פיקסל 25

פעמים

ב. "מחיקת" הריבוע האדום מתבצעת על ידי ציור ריבוע בצד הרקע במוקום הריבוע האדום

ג. שדרגו את התוכנית, כך שמיוקם הריבוע במסך יהיה אקריאי, כל פעם יבחר מיקום חדש

סיכום

בפרק זה עשינו במאוגן נושאים רלבנטיים לכתיבת פרויקט סיום. למדנו איך לשלב אלמנטים גרפיים בתוכנית שלנו – ראיינו איך אפשר בקלות להוסיף גרפיקת ASCII המשדרגת את הפרויקט ו איך משלבים תМОנות בפורמט bmp הנפוץ. למדנו באמצעות קבצים, העבודה עם עכבר, ייצירת מספרים אקראיים. סיימנו בטיפים לכתיבת פרויקט ובאופן כללי כיצד יש לגשת לפרויקט כתיבת תוכנה בהיקף ממשמעותי.

סיימנו את לימוד שפת האסמבלי ומבנה המחשב, אנו מוכנים לאתגרים הבאים שלנו.

لتלמידים: זיכרו, הودות לאינטרנט, ידע אינטובי נמצא במרקח כמה הקשות מקלדת. זהה מתנה שלא הייתה קיימת בדורות קודמים. כשתלמדו למצוא לבד מקורות לימוד, תרתו את כה הידע של אחרים לטובתכם. מכאן והלאה, דבר לא יכול לעזור את הסקרנות שלכם. בהצלחה!

נספח א' – רשימת פקודות חובה לבגרות בכתב באסמלוי

לפניכם רשימת פקודות החובה לבגרות בכתב באסמלוי. שימו לב – אלו אינן פקודות חובה עבור תלמידים שלומדים אסמלוי במסגרת יחידת המעבדה (כמו ב"גבהים") אלא רק עבור תלמידים שניגשים לבגרות במחשבים בפרק בחירה באסמלוי.

את רוב הפקודות הכרנו כבר, בפרק זה נפרט אודות הפקודות שטרם הכרנו.

ADD	JNA	NEG
AND	JNAE	NOP
CALL	JNB	NOT
CLC	JNBE	OR
CLI	JNGE	OUT
CMP	JNL	POP
DEC	JNLE	POPF
DIV	JNO	PUSH
IDIV	JNP	PUSHF
IMUL	JNS	RCL
IN	JNZ, JNE	RCR
INC	JO	RET
INT	JP	ROL
IRET	JPE	ROR
JA	JPO	SAL
JAE	JS	SAR
JB	JZ	SBB
JBE	LAHF	SHL
JC	LEA	
JCXZ	LOOP	
JE	LOOPE	
JG	LOOPNE	
JGE	LOOPNZ	
JL	LOOPZ	
JLE	MOV	
JMP	MUL	

פקודות שקובעות מצב דגלים:

CLI- הורדת דגל הנשא- CLC

CLI- הדלקת דגל הנשא- STC

CLI- הורדת דגל הפסיקות

פקודות קפיצה:

תיאור התנאי	Signed	Unsigned	מספרים
קפוֹץ אֶם גָדוֹל מִמְשָׁ	JG (JNLE)	JA (JNBE)	
קפוֹץ אֶם קָטָן מִמְשָׁ	JL (JNGE)	JB (JNAE)	
קפוֹץ אֶם גָדוֹל או שוֹוָה	JGE (JNL)	JAE (JNB)	
קפוֹץ אֶם קָטָן או שוֹוָה	JLE (JNG)	JBE (JNA)	
קפוֹץ אֶם שוֹוָה	JE	JE	
קְרוֹז אֶם שׂוֹנוֹה	JNE	JNE	

cx=0 JCXZ - קפוץ אם cx=0

פקודות קפיצה שבודקות ישירות את מצב הדגלים:

תיאור	הוראה
קְרוֹז אֶם דָגֵל הַנְשָׁא דָלוֹק	JC
קְרוֹז אֶם דָגֵל הַנְשָׁא כָבּוֹי	JNC

קפוץ אם דגל האפס דלוק	JZ
קפוץ אם דגל האפס כבוי	JNZ
קפוץ אם דגל הסימן דלוק	JS
קפוץ אם דגל הסימן כבוי	JNS
קפוץ אם דגל הגלישה דלוק	JO
קפוץ אם דגל הגלישה כבוי	JNO
קפוץ אם דגל הזוגיות דלוק	JP / JPO
קפוץ אם דגל הזוגיות כבוי	JNP / JPE

פקודות רегистר דגלים:

-LAHF – מעתיק את 8 הביטים הנמוכים של רегистר הדגלים ל-ah

-PUSHF – מעתיק את רегистר הדגלים למחסנית

-POPF – מעתיק את תוכן ראש המחסנית אל רегистר הדגלים

פקודות לולאות נוספות:

-LOOP – שאנחנו מכירים, ישנן הפקודות הבאות-

-LOOP, אם cx לא שווה לאפס מתרכעuta קפיצה לתווית. אך יש עוד תנאי לקפיצה-נדרש שדגל האפס יהיה דולק. לכן שימושי לבצע פקודת השוואה cmp לפני ביצוע LOOP או LOOPZ.

-LOOPNE (LOOPNZ) – כמו LOOP, אם cx לא שווה לאפס מתרכעuta קפיצה לתווית. אך יש עוד תנאי לקפיצה-נדרש שדגל האפס יהיה כבוי.

פקודות הזזה נוספת:

נוסף על SHR, SHL, SHR שאנו מכירים, ישנן הפקודות הבאות-

ROL- כמו SHL, כל הביטים זזים שמאליה והבית השמאלי ביותר מועתק לדגל הנשא, בהבדל אחד- הבית הכוי שמאליה מועתק גם אל הבית הכוי ימני. כלומר אם ניקח את ah לדוגמה ונבצע לו ROL שמונה פעמים, הוא יחזיר למצבו המקורי.

ROR- כמו SHR, כל הביטים זזים ימינה והבית הימני ביותר מועתק לדגל הנשא, בהבדל אחד- הבית הכוי ימני מועתק גם אל הבית הכוי שמאליה.

RCL-SHL, אך דגל הנשא מועתק אל הבית הכוי ימני.

RCR-SHR, אך דגל הנשא מועתק אל הבית הכוי שמאליה.

SAL-SHL- זהה ל-

SAR-SHR- זהה ל-

פקודת חיסור מיוחדת:

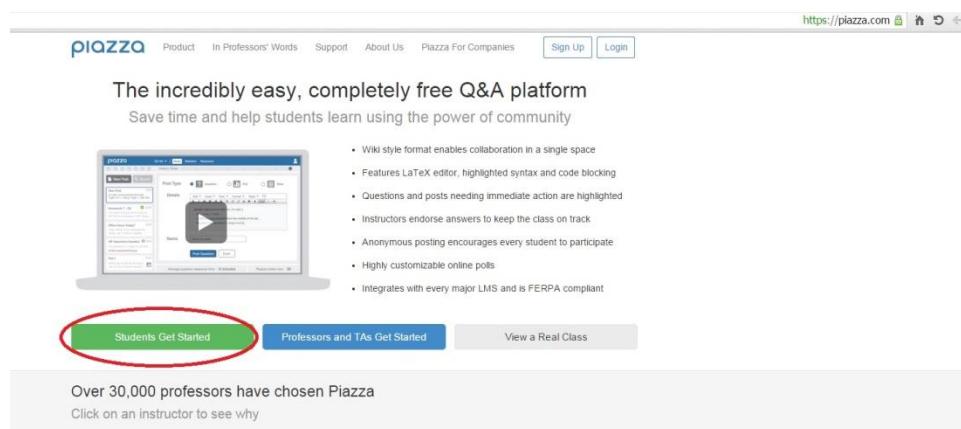
SBB-SUB, אבל מוסיפה לתוכאה את ערכו של דגל הנשא.

נספח ב' – מדריך לתלמידים: כיצד נכנסים לפורום האסמלבי הארץ'

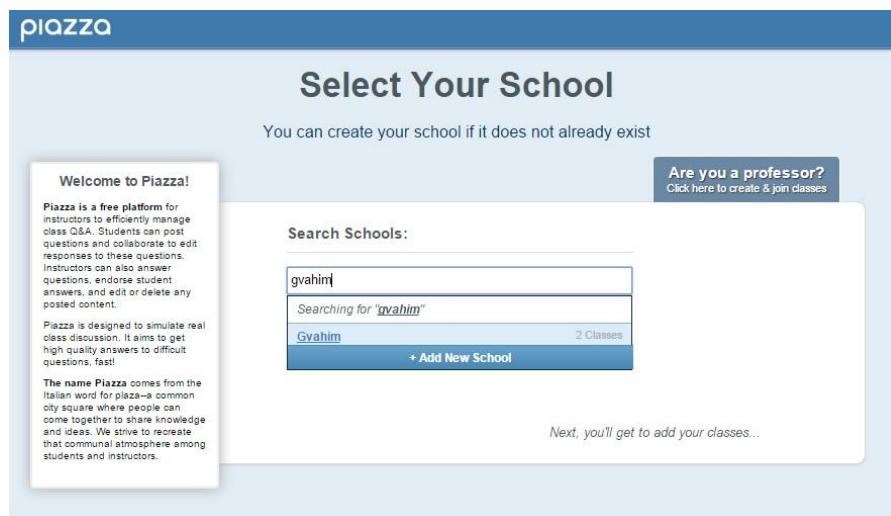
היא פורום שאלות ותשובות שמיועד לתלמידים בכל הארץ. ניתן לפתח פורום בכל נושא, לכתוב שאלה ולענות תשובות. כיוון שתלמידים בכיתות גבהים בכל הארץ נתקלים בעיות דומות, עמדת לרשותכם "אוניברסיטה" וירטואלית, שם תוכלו לסייע אחד לשני. בנוסף, התשובות שלכם יסייעו לתלמידים בשנים הבאות!

לכניסה ל-Piazza:

1. היכנסו לאתר www.piazza.com ובחרו "students get started"



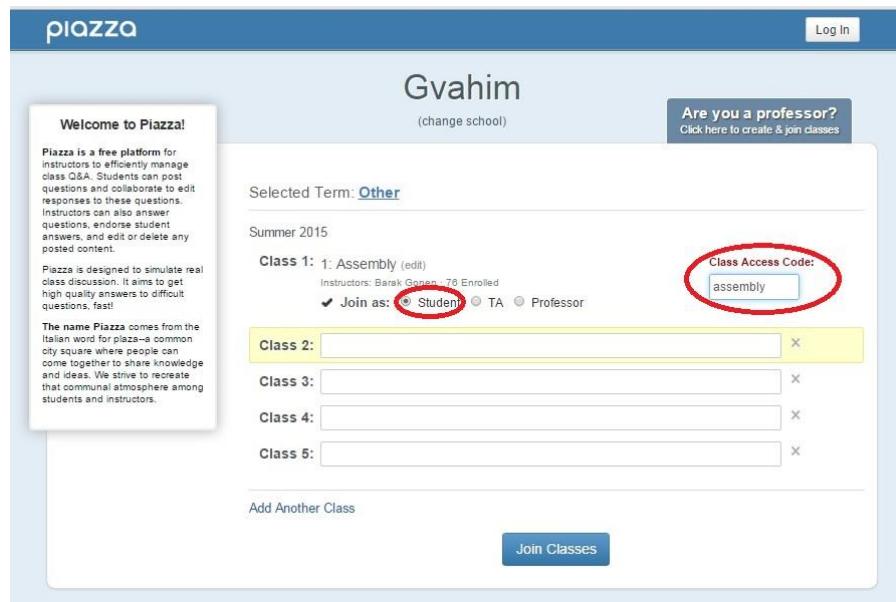
2. בחרו באוניברסיטה "gvahim"



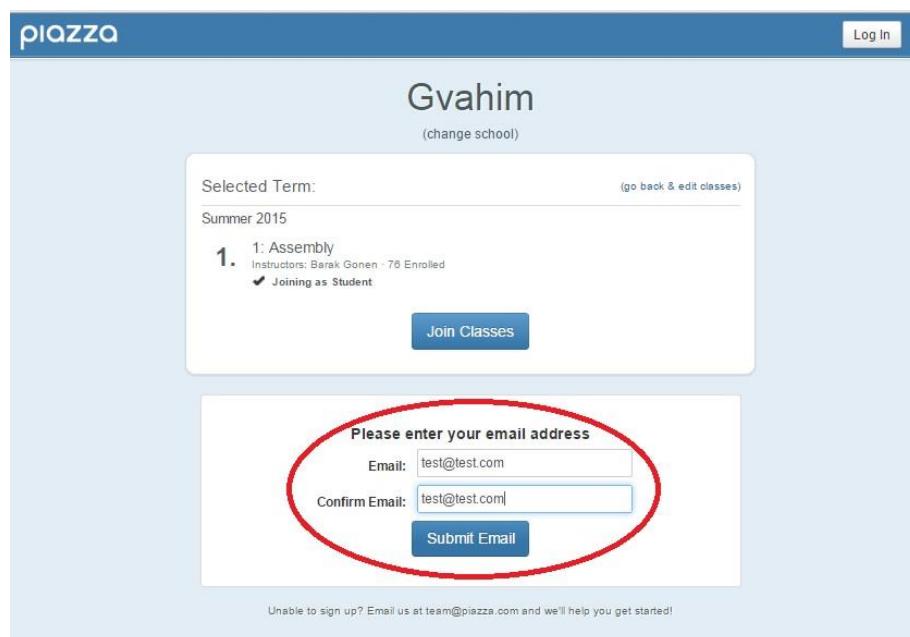
.3 בחרו סטטוס "other".

.4 בחרו קורס assembly.

.5. בחרו להצטרף כ-student והכניסו סיסמה "assembly".



.6. הכניסו כתובת מייל, אליה ישלח קוד הפעלה של האתר, וליהצטו על submit email



.7. הכניסו את הקוד שקיבלתם ולחצו על submit .

The screenshot shows the Piazza sign-up confirmation page. At the top, it says "Selected Term: Summer 2015" and "1: Assembly". Below that is a "Join Classes" button. In the center, there's a message: "We see you're new to Piazza! Check your inbox for your confirmation email. Enter the validation code below so you can access your classes!" A red circle highlights the "Validation Code:" input field, which contains the code "iz9TeY9ShhD". Below it is a "Submit Code" button. To the right, there's a "Not Getting Our Email?" section with instructions and a link to resend the email.

.8. בחרו שם משתמש וסיסמה. למטה בחרו באופציה I am not pursuing a degree ואשרו שקראותם את תנאי השימוש. לאחר מכן לחצו על Continue - Continue . זהו, סימתם. ברוכים הבאים.

The screenshot shows the Piazza account setup page. It starts with the heading "Finish setting up your Piazza account:". There's a "Account Information (required)" section with fields for "Full Name" (containing "test test"), "Choose Password", and "Confirm Password". To the right, there's a contact box: "Contact us at team@piazza.com with any questions." Below that is an "Academic Information (required)" section. It includes dropdowns for "Graduate Program" (with "Select current program...") and "Major" (with "Enter current major..."), and dropdowns for "Anticipated Completion" (Month and Year). Two checkboxes are present: "I have two majors" (unchecked) and "I'm not pursuing a degree" (checked). A note below states: "This information will be used for collaborative features on Piazza. We will never share your information without your permission." At the bottom, a red circle highlights the "I've read and accept the terms of service" checkbox. A "Continue to Piazza" button is at the very bottom.

הכנסת שאלה חדשה

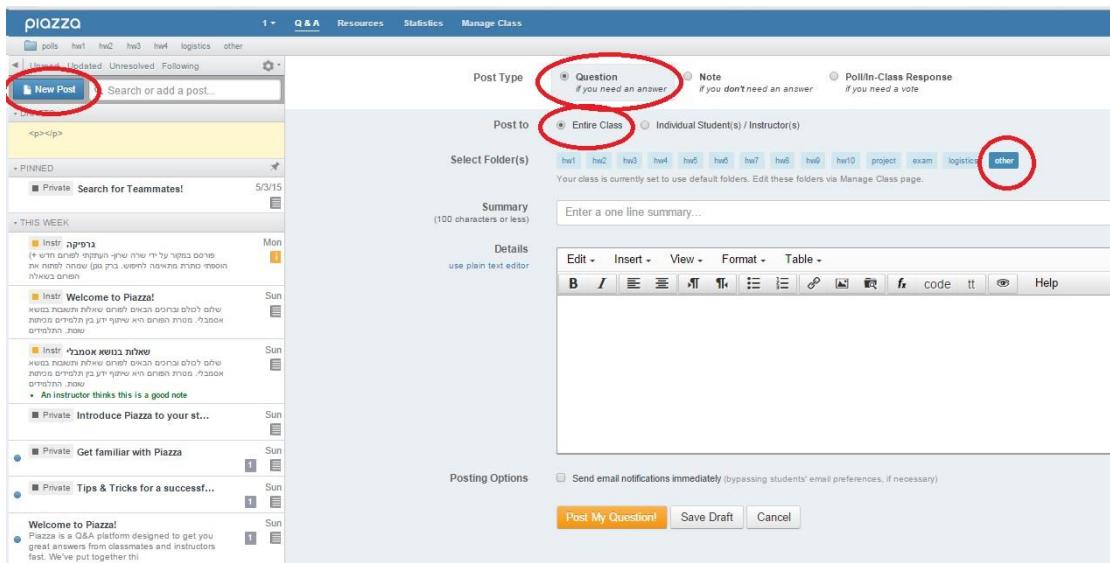
על מנת לשאול שאלה, הקישו על **new post**. בחלון שייפתח בחרו

- Post type = Question

- Post to = Entire class

- Folder = Other

Post my question אל תשחחו تحت כוורתה לשאלתך, ולבסוף הקישו על



זכויות יוצרים – מקורות חיצוניים

http://edjudo.com/wordpress_livedec10/wp-content/uploads/slider/digital.jpg

http://visual6502.org/images/pages/Intel_8086_die_shots.html

<http://www.ousob.com>

http://en.wikipedia.org/wiki/MS-DOS_API

<http://iitestudent.blogspot.co.il/>