

ארנו המחשב ושפת השר

ברק גון

המרכז לחינוך סייבר
CYBER EDUCATION CENTER

ארגון המחשב ומשפט סף

גרסה 2.52 ספטמבר 2020

כתיבה:

ברק גון

מבוסס על מצגות וחומר לימוד מאט 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 המועילות, 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 הצעה"ל.

מהדורה שנייה תש"פ 2020

© כל הזכויות שמורות למרכז לחינוך סייבר של קרן רש"י.

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	השיטה העשרונית
21	חישוב ערך של מספר עשרוני בסיס אחר
21	השיטה הבינארית
22	השיטה הקסדצימלית
25	פעולות החשבון
25	היבור
28	חיסור (הרחבת)
30	כפל (הרחבת)
32	חילוק (הרחבת)
33	יצוג מספרים על-ידי כמות מוגדרת של ביטים
34	יצוג מספרים שליליים
34	שיטת גודל וסימן
35	שיטת המשללים לאחת
36	שיטת המשללים לשתיים
39	או איך מנצחים את חוואן?
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	סגמנטים
56	יחידת היעבוד 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 יחיד – 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	משתני 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 לזכרון
108	региיסטרים חוקיים לגישה לזכרון
109	תרגומ אופרנד לכחובת בזיכרון
111	Little Endian, Big Endian
111	העתקה ממערכיים ואל מערכיים

112	פקודת offset
113	פקודת LEA
113	הנחהיה word ptr / byte ptr
114	ازהרת type override
115	פקודת mov - טעויות של מתחילים
116	שינויי 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	טיור בדיקת הדגלים (הרחבה)
145	פקודת LOOP
146	זיהוי מקרי קצה
147	לולאה בתוך לולאה Nested Loops (הרחבה)
150	קפיצה מחוץ לתchrom
151	סיכום
152	פרק 9 – מהשנית ופראצדרות
152	מבוא
154	המהשנית STACK
154	הגדרת מהשנית
156	פקודת PUSH
158	פקודת POP
160	פראצדרות
160	הגדרה של פראצדרה
162	פקודות CALL, RET
165	פראצדרה NEAR, FAR
167	שימוש במהשנית לשימרת מצב התוכנית
170	העברה פרמטרים לפראצדרה
173	העברה פרמטרים על המהשנית
173	Pass by Value
176	Pass by Reference
178	שימוש ברגיסטר BP
183	שימוש במהשנית להגדרת משתנים מקומיים בפראצדרה (הרחבה)
186	שימוש במהשנית להעברת מערך לפראצדרה
188	גلىית מהשנית - Stack Overflow (הרחבה)

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

240	מבוא
240	פסקות חומרה – Interrupts
243	בקור האינטראפטים – PIC
244	אובדן אינטראפטים
245	זיכרון קלט / פלט – I/O Ports
247	המקלדת
247	הקדמה
248	יצירת Scan Codes ושליחם 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	כתיבה לקובץ
264	סגירת קובץ
265	פקודות נוספות 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	השمعת צלילים
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	תוכנו מוקדם – יצירת תרשימים זרימה
304	חלוקת לפנוטזדורות
305	מעקב אחרי מונימ
305	העתיקות זיכרון
306	הודעות שגיאה של האסמבילר
306	סיכום
307	נספה א' – רשימת פקודות חובה לבגרות בכתב באסמבלי
311	נספה ב' – מדריך לתלמידים: כיצד נכנסים לפורום האסמבלי הארץ-ישראלי
316	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 לתוכנית הלימודים של משרד החינוך עבור יהדות המעלדה. תלמידים המעוניינים להעמק את הידע שלהם, ימצאו בספר את הרקע ההיסטורי לתוכנים שהם מעבר לתוכנית הלימודים. כמו כן, הספר כולל הדרך וכליים לביצוע פרויקטי סיום מורכבים, מעבר לדרישות משרד החינוך. הנושאים שהווים מעבר לתוכנית הלימודים מסוימים בספר כ"הרחבה".

לכתבת שפת אסמבלי קיימות מספר סיבות פיתוח, שפותחו על-ידי חברות שונות. הקוד שבספר כתוב בסביבת העבודה TASM, והספר מליץ על אוסף תוכנות לשימוש בתור סביבת עבודה - גם כדי להסוך לקוראים את הצורך לחפש סביבת עבודה באופן עצמאי, וגם כדי שייהי סטנדרט אחיד לעובדה ולהתרגול בכתיבה. מומלץ להתקין את סביבת העבודה ולהשתמש בה כבר בתחילת הלימוד, וכן לפתח את התרגילים המופיעים בפרקיהם תוך כדי תהליך 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 מהזיכרון ואל הזיכרון.
- בפרק 7 נלמד פקודות אריתמטיות (חיבור, חיסור וכו'), פקודות לוגיות ופקודות הווה.
- בפרק 8 נלמד איך להגדיר תנאים לוגיים ואיך לכתוב פקודות בקרה. בסוף הפרק תוכלנו לכתוב אלגוריתמים שונים, לדוגמה תוכנית שמבצעת מיזון של איברים במערך.
- בפרק 9 נלמד על אוצר בזיכרון שנקרא מהנסית, נלמד לכתוב פרוצדורות, להעביר אליה פרמטרים ולהגדיר משתנים מקומיים. בסיום הפרק נראה איך החומר שלמדנו מאפשר לנו להבין נושאי תוכנה מתקדמים.
- בפרק 10 נקבל רקע וכליים בסיסיים הדורשים להשתתפות בתחרות קודגورو אקסטרים, בין היתר נלמד את העקרון הבסיסי של Reverse Engineering על ידי פיצה "זומביים", הידועה תוכנה שפורסמה 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 ועובדת 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, עובדה עם קבצים ועובדת עם עכבר – ונעמיק את ההבנה שלנו בהם בעזרת תוכניות דוגמה.

הערות לגרסתה 2.0

עם סיום שנת הלימודים התשע"ה, בה שימש הספר בכ-25 כיתות גבאים, עודכן הספר על פי לключи ההוראה ולוקטו אליו חומרים 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 התיאzosות לפקודות אסמבלי שנדרשות על ידי משרד החינוך בבחינות הבגרות.

פורום ארצי לעזרה ושאלות

לຮשות 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 ותרגומים

הספר **Art of assembly** מאת Randall Hyde הוא כלי הנראת המקורי ביחס ללימוד השפה. אמן הקוד שם כתוב בהתאם לחוקי כתיבה מעט שונים מאשר בספר זה (לעומת NASM, למן הדיווק), אך הספר הינו מקור מועלה להבנת השפה ולהסבירים אודוט פקודות אסמבלי. ניתן להוריד אותו בחינם מהאינטרנט.

חוברת התרגילים "הכנה לבגרות 5 יח"ל במדעי המחשב" מאט רוניית גל או רץיאנו מכילה תרגילים רבים בנושאים שנדרשים על ידי משרד החינוך לפרק האסמבלי בבגרות במחשבים. התרגילים הם בהיקף ובסגנון השאלות בבחינות הבגרות.

لتלמידים

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

אתם יוצאים למסע לרכישת ידע. המסע לא צפוי להיות קל, אך במהלךו תלמדו לא מעט ותרכשו יכולת חדשה וחשובה להמשך. יחד עם לימוד 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 המיומנויות שהכי יעוזרו לכם להצליח, גם בסיביר וגם בחיים.

אייקונים

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



פרק 1 – מבוא ללימוד אסמבלי

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

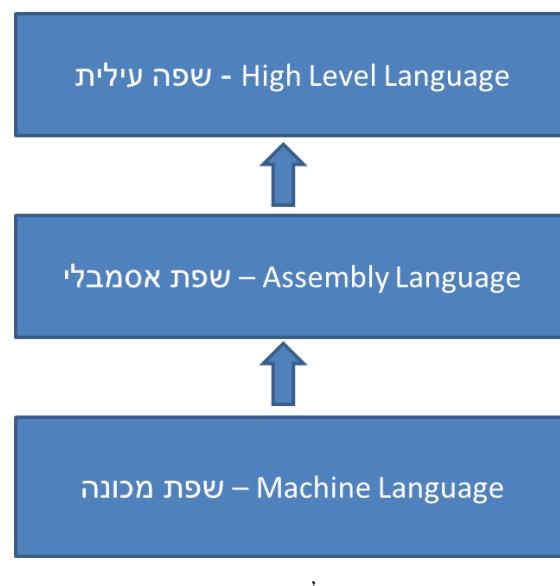
אסמבלי היא שפת התכונות הראשונה בעולם, שפותחה לפני עשרות שנים בתור תחילה פשוט יחסית לשפת מכונה, שבה כותבים באחדות ובאפסים. במילים אחרות, לפני שפותחה שפת אסמבלי, מי שרצה לתוכנת היה צריך להזין למחשב רצף של אחדות ואפסים. ומה קורה אם מתבלבלים? אם מחליפים באיזשהו מקום 0 ב-1? חבל מאוד.

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 לנו, על מנת לבצע פקודה, היה צריך לרשום רצף של אחדות ואפסים, למשל "10111000", שהומצאה 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 קריתו.

אסמבלי נ唤做 Low Level Language. הסיבה היא因为她 שמי שמתכוна באסמבלי כותב פקודות שעובדות ישירות מול החומרה של המחשב – אי אפשר לתוכנת אסמבלי בלי להבין איך עובדים המעבד, הזיכרון ורכיבי החומרה שהשורים אליהם. כל פקודה באסמבלי מתרגם לפקודה אחת בשפת מכונה – לדוגמה 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 למספר פקודות בשפת מכונה.



או למה בעצם ללימוד אסמבלי?

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

- מאוז שאסמבלי פותחה העולם התקדם ויש שפות תוכנה מודרניות.
- אסמבלי היא שפה מורכבת יחסית ללימוד.
- מסובך לכתוב קוד באסמבלי.
- קשה לדבג (למצוא שגיאות ולנפנות 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 לתוכנת בשפות עיליות בדרך כלל לא מתלהב לתוכנת באסמבלי. מתוכנות מנוסים שלומדים אסמבלי, חשים לעיתים קרובות שבשפות אחרות אפשר לעשות הרבה יותר בקלות הפעולות הנלמדות בשיעורי אסמבלי. זה כמובן, עד שמניגעים למוגבלות של שפות תכנות אחרות, מה שambilיאו אותנו לסיבות למה כן ללימוד אסמבלי:

- שפת אסמבלי תורמת להבנה עמוקה של המחשב על חלקיו השונים. העובדה שatoms נחשפים לחלקים הפנימיים ביותר של המחשב וישוום דבר אינו מהו "קופסה שחורה" בשביבכם, תאפשר לכם בעיה לחתמוד עם בעיות תכנות בלחטי שגרתיות, שנדרשות בעולם הסיביר.
- שליטה בשפת אסמבלי היא כלי טכנולוגי חשוב בעולם הסיביר. לדוגמה, כדי לבצע מחקר קוד על ידי **Reverse Engineering** כדי להבין מבנה 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 לנוטרי אפליקציות לשולוט באסמבלי.
- המוקם שתוכנית אסמבלי תופסת בזיכרון הוא קטן למדי יחסית לתוכנות שכותבות בשפות עיליות.

יכול להיות שהשתכנעתם בחשיבות לימוד אסמבלי ויכול להיות שלא. בכל אופן – שימוש לב לדרישות הדרישות שנוסחה על ידי חברת אבטחת 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 לימודי האסטטטי בмагמת גבאים. ראשית, רכישת מיומנויות של סדר, ארגון וחשיבה מתודית. שפת אסטטטי היא מקום טוב במיוחד לרכוש בו יכולות תכונת 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 תידרשו לכתוב קוד בהיקף ממשמעתי ולדבג אותו. זהו אתגר שיכשיר אתכם לקראת הבאות.

שלישית, ידע מקדים באסטטטי 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 ללמידה בעצמנו. נתחיל מהבסיס – שיטות ספירה וייצוג מידע במחשב.



פרק 2 – שיטות ספירה ויצוג מידע במחשב

מבוא – מהן שיטות ספירה?

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

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

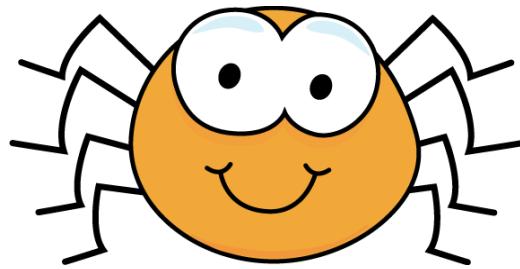
בפרק זה נלמד שיטות לייצג מספרים בסיסי ספירה שונים תוך התמקדות בשיטה הבינארית ובשיטה הקסדצימלית, נבצע את פעולות החשבון הבסיסיות בסיסים שאינם בסיס עשר, נראה איך מייצגים מספרים שליליים בשיטות שונות ונתמקד בשיטה שנקראת "המשלים לשתיים". בואו נתחיל.

שיטת ספירה זו בסך הכל דרכיהם שונות לייצג כמה נתונה של פרטיהם שאפשר לספור אותם. נתכל 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

לפני שנטקדם, 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 "חוואן", השדון בעל השינויים ה策horות. נסו לנצח אותו!



"חוואן". נצחו אותו!

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

השיטה העשרונית

כעת נרחיב את ההסבר לגבי איך מייצגים מספרים בסיסים שאנו לא רגילים אליהם וכדי להקל, נתחיל דוקא מייצוג מספרים בסיס עשר המוכר. בשיטה העשרונית קיימות עשר ספרות: 0,1,2,3,4,5,6,7,8,9. בעזרת הספרות הללו אנו אנו 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.

שים לב – מעכשו 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 מקום, 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

נحوוק 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



נתחילה בחלוקת 5. התוצאה היא 39 והשארית היא 4.

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

חלוקת של 7:5 היא 1, שארית 2.

חלוקת של 1:5 היא 0, שארית 1.

כעה נרשום את כל השאריות, מלמטה למעלה - 1244.

ומכאן ש- $199_{10} = 1244_5$.

ונבדוק 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 חזקתו 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 שמונה ספרות 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

נוקח לדוגמה את המספר 10011_2 . נתרגם את המספר זהה למספר עשרוני. לטובת קלות הציגו נכnis את המספר לטבלה 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 בדיק באותה שיטה שהדגמנו בסעיף "חישוב ערכו של מספר עשרוני בבסיס אחר", רק שהפעם הבסיס الآخر הוא 2. נוקח 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 שנתקראת גם השיטה ההקסדצימלית, קיימות שש עשרה ספרות. כיוון שהכתב שלנו מכיל רק עשר ספרות (מ-0 עד 9), בשיטה זו לוקחים שש אותיות ונותנים להן ערך מספרי. 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 $1A_{16}$, $2B_{16}$, $4C4_{16}$ $C1A_{16}$, $F15_{16}$ $C0DE_{16}$, $DEAD_{16}$ $C0FFEE_{16}$ או $C0DE_{16}$.

בשיטת זו אפשר להשתעשע עם מספרים בעלי ערכי ערכיהם נחמים כמו $C0DE_{16}$ או $C0FFEE_{16}$.

שימושו לב שאנו משתמשים בספרה 0, אות האנגלית ס אין משמעות בייצוג הקסדצימלי.



השיטות 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 $0C0DEh$ (נדגש כי ניתן להוסיף אפסים מצד שמאל בכל מקרה, אבל כאשר המספר מתחילה באות – חיברים להוסיף)

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

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

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

नבעץ 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, התוצאה 0 השארית 12 (בסיס 10), שמיוצגת בסיס 16 על ידי 'C'.

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

כעת נבעץ המורה של מספרים בינאריים להקסדצימליים. להמרה זו יש מאפיין מיוחד, מכיוון ש-16 הוא חזקה של 2. נראה מיד איך תכונה זו בא לידי ביטוי:

הקסדצימלי	בינארי
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 עלי-ידי מספר בינארי בן ארבע ספרות. בעקבות זאת, ההמרה מהקסדיימלי לבינארי יכולה להתבצע פשוט עלי-ידי מעבר של ספרה אחר ספרה, והחלפת הספרה הקסדיימלית באربع ספרות ביןאריות. ניקח לדוגמה את המספר $9B_{16}$:

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

$$B_{16} = 1011_2$$

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

התcona הזו, המעביר הפשט מבינארי 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 ביןארי, 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, ביןארי. בטור הימני מספרים בסיס 16, הקסדיימלי.

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

פעולות חישוב

NELMED AT ARBU' FULLOT HACHSHBON HABSEISIOT - CHIBUR, CHISOR, KFEL V'HILUK - BBETISIM SHAINIM BESIS Usher.

חיבור

NATHIL MATERGOL BESIS Usher. NACHBER AT HAMSPERIM 133 VD 70.

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

ANHANO MATHILIM AT HACHBUR MIZD YMIN V'MAHBRIM KL SPERAH UM HAMSPERAH SHMTAHTA. CASHER MKBLIM TOZAAH SHEIA SHOVA AO GDOLAH - 10, ANHANO COHTBIM LMETHE RAK AT SPERT HAADOT VAILO AT SPERT HAUSROT ANO MOSIFIM ("NOSAIM" AO BANGGLIT "CARRY" NSEA) LZOG HAMSPEROT SHMSMEL.

HACHBUR MAMSPERIM BIINARIM AIINO SHOVA BRMAT HAYIKRON, PRUT LCK SHASHARIT MTKBLAT UM CL TOZAAH HACHBUR SHEIA SHOVA AO GDOLAH - 2. NATHBON BAPESHROVOT HACHBUR SHONOT, HAMROCZOVT BTEBLA LMETHE.

HACHBUR SHL 0 VUD 0 SHOVA 0.

HACHBUR SHL 0 VUD 1 SHOVA 1.

HACHBUR SHL 1 VUD 0 SHOVA 1.

HACHBUR SHL 1 VUD 1 SHOVA 10 (AFS UM NSEA ACHD).

+	0	1
0	0	1
1	1	10

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



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

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 לתוכזאה 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 (הרבה)

כאשר מחסרים שתי ספרות, ישנן 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:



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

הchiaור של 9 פחות 1 שווה 8.

בשביל החישור 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.chiaור של 0 פחות 1 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 כתוצאה מפעולת החישור.

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

cutet נבצע תרגיל חיסור בסיס 16:



-1 16
 -1 16
- DEAD
CODE
 1DCF

הסביר:

היסור של D פחות E מצריך נשא שלילי. נוריד ייחידה מד A ונפרוט אותה לשש עשרה יחידות שמתווספות לד-D. תוצאה החיסור היא לנ C (בהמרה לעשרוני – 13 ועוד 16 פחות 14, שווה 15).

בטור השני, הורדנו ייחידה מד A וצריך לחסר גם את D.שוב נצטרך להשתמש בנשא שלילי, ניקח ייחידה מהטור השלישי. תוצאה החיסור היא לנ C (בהמרה לעשרוני – 10 פחות 1, ועוד 16 פחות 13, שווה 12).

בטור השלישי, הורדנו ייחידה מד-E, נשארנו עם D.

לבסוף בטור הרביעי החיסור של D פחות C נותן 1.

תרגיל 2.3: חיסור



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	



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

X	0	1
0	0	0
1	0	1

ניקח 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 מכון מוחברים את התוצאות היבור פשוט ומקבלת 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 מתרכחת בסיס 2, כאשר כופלים במספר שהוא שתיים או חזקה של שתיים. על כל חזקה של שתיים, מזוזים בספרה 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 16×16 איברים, כלומר 256 איברים. אך אל דאגה – אין צורך לזכור אותו בעל פה. הדרך הפשוטה יותר היא להמיר את המספרים במסיס בינארי ולהמשיך משם. לדוגמה, כפל של C כפול 5, לפי טבלת ההמרה מתתקבל בקלות 1100 כפול 101 והדרך 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 BINARIE 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.

nbz דוגמה – $10110_2 / 101_2$:



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

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		

ייצוג מספרים על-ידי כמות מוגדרת של ביטים

זכרנו המחשב בניו מתחאים, שבתוכם שמורות 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 בשביל לייצג מספר. במחשבים זהה מגבלה מאוד מעשית, כיון שהמחשבים שמורות מספרים בהתאם בגודל קבוע. הגודלים המקובלים הם 8, 16, 32 או 64 ביטים. ככלומר המגבלה על כמות הביטים לייצוג 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, שמרכזות את המספר הכי גדול שאפשר לייצוג על-ידי כמות ביטים שבין 1 ל-8:

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

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

מה קורה אם אנחנו מנסים לייצג מספר שהוא יותר גדול מאשר יכול שארט לייצוג 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, ומתקיים נשא. הדרך שבה המחשב מטפל במקרים כאלה היא כזו: הזיכרונו 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 בזכרונו שלנו יש 00000000. ככלומר, מבחינת המחשב, כמשמעותו לייצוג של 8 ביטים, אז המשווה $255+1=0$ היא נcona. עליינו להבין כי כך המחשב עובד, ונctrיך להתחאים את עצמנו כדי שלא יקרו לנו טעויות כאלה.

ייצוג מספרים שליליים

עד כה חשבנו על מספרים בינאריים רק בתור מספרים עם ערכים חיוביים. המספר הבינארי 0000 מייצג אפס, 0001 מייצג אחד, 0010 מייצג שתים וכך הלאה עד אינסוף. לייצוג מספרים בדרך זו קוראים **unsigned** – קלומר חסרי סימן. אבל, מה אם נרצה לייצג מספרים שליליים?

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

שיטת גודל וסימן

נניח שעומדים לרשותנו N ביטים בכל תא, ואנחנו רוצים לייצג 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 מתרגמים אותו למספר כמו שעשינו עד עכשיו עם מספרים **unsigned**, וזאת לפי הבית השמאלי כתובים את הסימן לפני המספר.

נוקח לדוגמה את המספר: 0011. החבו, 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 לייצג עלי-ידי 4 ביטים הוא 0111, קלומר 7, ואילו המספר היכי נמוך הוא 1111, קלומר מינוס 7.

שים 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 שלה. קל יחסית להסתכל על מספר ולקבוע מה הערך שלו, וכן קל להשוות בין שני מספרים.

היתרון המרכזי של השיטה הוא שימושי לפעולות תרגילי החשבון. נסתכל לדוגמה על 3 ועוד (-3):

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

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

שיטה המשלים לאחת

שיטה המשלים לאחת (One's complement) אמורה להתגבר על החיסרון העיקרי של שיטת הגודל והסימן, ביצוע פעולות חשבון 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 המשלים ל-1	מספר עשרוני	ייצוג בשיטת המשלים 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 מלמעלה - 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, שבסיטה זו שקול לערך 0. ככלمر התגברנו על מכשול אחד.



בשיטת המשלים לאחת, כדי שוגם יתר פעולות החשבון יתנו תוצאות נכונות, משתמשים בחוק הבא: אם לתוצאה הפולה יש נשא, מחברים אותו לבית הימני ביותר.

נסתכל על דוגמה, התרגיל 5 פחות 2. ניתן לכתוב את התרגיל הזה 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 ללמוד את המחשב ש- $1111 - 0000 = 1111$ צריך להשיקע עוד זמן ומאמץ, ועדייף פשוט לעבור לשיטת המשלים לשתיים.

שיטת המשלים לשתיים

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

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

כדוגמה, נמצא את היצוג של מינוס 6, ביצוג שלו עליידי 8 ביטים:



היצוג של 6 הוא 00000110.

לאחר הפיכת הביטים תוצאה הביניים היא 11111001.

לאחר הוספת 1, התוצאה היא 11111010. זהו היצוג של מינוס 6 בשיטת המשלים לשתיים, ביצוג עליידי 8 ביטים. אם היינו רוצים לייצג מינוס 6 עליידי 16 ביטים, היינו צריכים להוסיף להוספה אחדות בהתחלה – 1010 1111 1111 1111 (מעכשיו, כשנרצה לרשום מספרים ארוכים בבינארי, נפריד אותם לקבוצות של ארבע ספרות מסוימות של נוחות קראיה). 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 שראים, כל שמונה הביטים אכן התאפסו (זכרו – הד' שמאך הוא נשא והוא אינו נכון בשמונה הביטים).

נסתכל על הייצוג של כמה מספרים בינאריים בשיטת המשלים לשתיים. לטובת המשך, אנחנו נסתכל על הייצוג ב-8 ביטים, למרות שבשביל לייצג את המספרים שבבלה 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. זה יותרו משמעותי 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 $(-1)^{N-1}$ ל- 2^{N-1} . כך לדוגמה, בעזרת 8 ביטים אפשר לייצג את המספרים שבין 127 למינוס 128. בעזרת 16 ביטים אפשר לייצג מספרים בתחום שבין 32,767 למינוס 32,768.

לטווח הערכים הזה יש חשיבות מעשית גדולה – כל פעולה חשבונית שהתוצאה שלה הורגת מתוך הערכים שניתן לייצוג, תגרום להחזרת תשובה שגויה.

כדי לסקם את הדיוון 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 שmaguiim 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 לשוכוח שהמספר הוא שלילי – נוסף סימן מינוס ונגיעה 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 יוצג 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 מונצחים את חואן?

... ומה הקשור 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>)

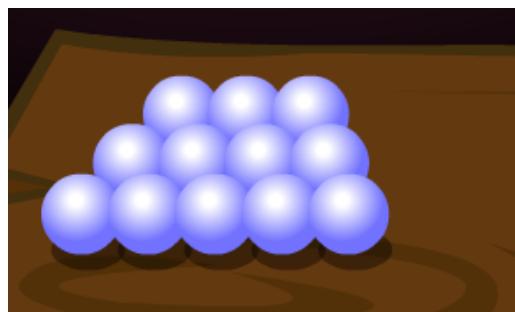
קיימת טכניקה שmbטיחה את הנצחון במשחק, אנחנו נפרט עליה כאן. אם אתם סקרנים מדוע הטכניקה זו עובדת, ישנו הסבר מתמטי, שהינו מחוץ להיקף של ספר זה, אך הוא מופיע בוויקיפדיה ובמקומות אחרים באינטרנט.

כדי להבטיח ניצחון במשחק, יש לבצע את הצעדים הבאים:

1. יש לתרגם כל מספר של "פנינים" לייצוג הבינארי שלו (לדוגמה 7 פנינים - 111 ביבاري).
2. לרשום את כל הייצוגים הבינאריים אחד מעלה 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 מהחת השורוט כך שהסכום יהיה אפס.

דוגמאות:

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



ייצוג בינארי (נשתמש בארכע ספרות בינאריות, 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 שנגיעו למספר בו אנו יכולים להשאיר רק פניה אחת ולנצח.

תרגיל 2.7: נצחו עשרה משחקים

נצחו את חואן בעשרת השלבים הראשונים! אתגר - נסו לא להפסיק ב אף משחק ולשמור על מזון נצחותנו מושלם.



ייצוג מידע במחשב

בתחילת הפרק, כשניסינו להסביר למה כדאי ללמידה בסיסי ספירה, נתנו הבטחה מעורפלת שבזורה בסיס בינהרי והקסדצימלי קל יותר לעבוד עם מחשבים. עכשיו הגיע הזמן לפרק את השטר – נעבור לדבר על מה קורה בתוך המחשב, על הדרך בה מידע מיוצג במחשב. כך, נוכל לראות את השימוש 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 שני ערוצים שונים זה מזה. לדוגמה, אפס או אחד,אמת או שקר, למעלה או למטה, יורך או אורך, או 519. כל שני ערוצים שונים זה מזה ניתנים לייצוג על-ידי בית בודד, אך ניתן לייצג באמצעות רק שני ערוצים שונים.

אם כך, כשסתכלים על בית, איך יודעים אילו שני ערוצים 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 עקייבי 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-4 (היזכרו 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, אגור תוך שימוש באחדות ואפסים בלבד. בזכרון המחשב לא קיימת הספרה 2, לא קיימים 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 מוחשיים מתמטיים וכלה בסרטוי וידואן.

כוון שבאמצעות בית בודד ניתן לייצג רק שני ערוצים, השימוש בביטים בודדים הוא בעל שימושים מוגבלים. במקרה זה, לרוב השימושים משתמשים באוספים של ביטים.

Nibble – ניבול

 **Nibble** הוא אוסף של ארבעה ביטים. Nibble יכול לייצג 2 בחזקת 4 ערוצים שונים, כלומר 16 ערוצים. במקרה של מספר הקסדצימלי, Nibble יכול לייצג את הערוצים 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 לספרה 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

訳すと、このビット列は、16進数の値を表すのに使われる4ビットの組合せであることを示す。つまり、このビット列は、1101, 1110, 1010, 1101, 1100, 0000, 1101, 1110の8つの4ビットの組合せである。

הקסדצימלי	בינארי
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

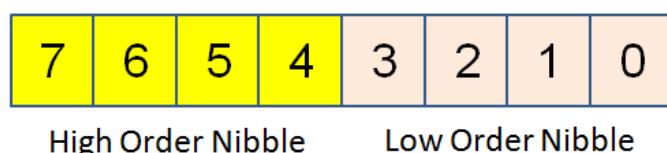
בית (Byte) הוא אוסף של שמונה ביטים, והוא גם היחידה הקטנה ביותר שיש לה כתובות משלها – הזיכרון מחולק לכתובות, וכל כתובות היא של בית יחיד. משמעות הדבר היא שהיחיד הזיכרון הקטנה ביותר שניתן לגשת אליה היא בית. כדי לגשת ליחידה קטנה יותר, נניח ביט, נדרש לקרוא את כל הבית שמכיל את הבית הרצוי, ואז על-ידי פעולה שנתקראת מיסוך ביטים (NELMD ULIA BEMASH, BHALK SHMSBIR FKODOT LOGIOT) 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 ליצג שתיים בחזקת 16, או 65,536, ערכים שונים. באסםביי 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:



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-**DWORD** 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 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 בקוד 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 (בהמשך 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 שאנו רגילים אליה. התמקדנו בשתי שיטות: השיטה הבינארית, שמשמשת לייצוג של ערכיהם בזיכרון 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 החשבון בסיסיות- חיבור, חיסור, כפל וחילוק- לבסיסים 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.

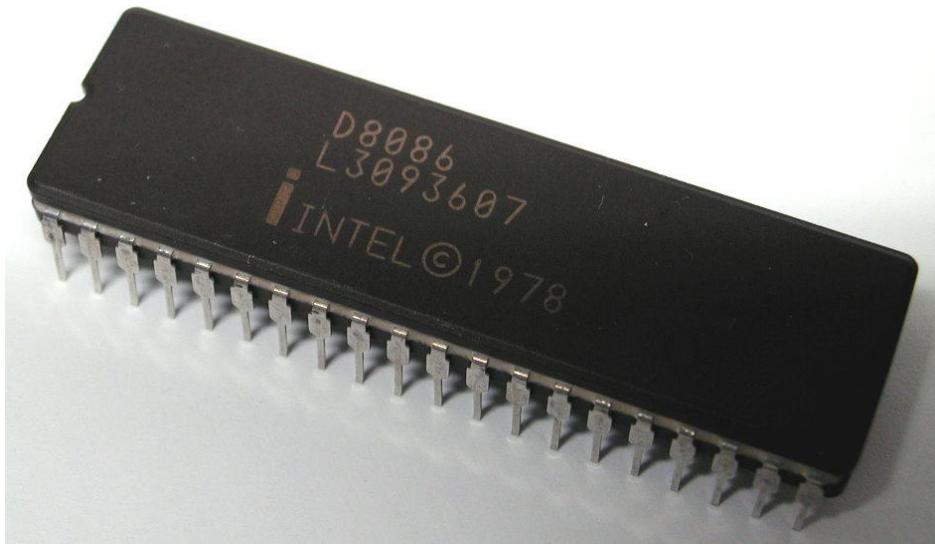
בסוף פרק זה, סקרנו איך מחשב שומר ערכים שונים בזיכרון וראינו את היתרונות של שימוש בסיס בינארי והקסדצימלי. כתע אנחנו מבין איך מספרים מסוימים מיוצגים במחשב, ואנחנו בשלים לעבור לפרק הבא בו נרחיב ונתעמק במבנה המחשב.

פרק 3 – ארגון המחשב

מבוא

אסambilי היא שפת התוכנה שעובדת בצדקה הקדומה ביותר מול החומרה של המחשב. רק אסambilר קטן מפ raid בין הקוד שאותם כותבים באסambilי לבין שפת מכונה, ובהמשך גם מתנסו בתרגום מאסambilי לשפת מכונה. לכן כדי לכתוב תוכנית, אפילו בסיסית, בשפת אסambilי, נדרש הבנה של ארגון המחשב והחומרה. כדי לעסוק בסיביר באופן מקצועי צריך כישורים לביצוע פעולות מתקדמות כמו מציאת אגים 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 ה-80x86 כוללת מספר מעבדים, כאשר במקומם 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.



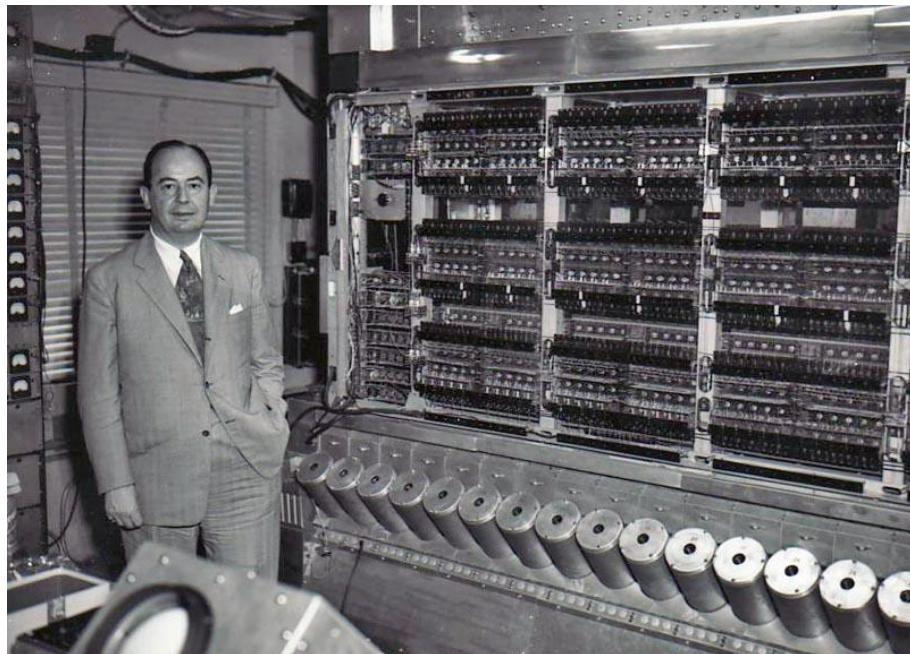
ראשית, ראוי שנשאול את עצמנו – איזו תועלתה יכולה להיות ללימוד מעבד כל כך מיושן? הרי חומרת המחשבים משתנה כל מספר שנים בזדמנות. איך אפשר ללמוד משהו מעשי לעולם הסיביר באמצעות מעבד שמקורו בדף ההיסטוריה?

ובכן, ישן כמה תשובות לשאלת זו:

- מעבד ה-80x86 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 שומרה על תאמיות לאחר עם מעבדי ה-80x80. כמובן, אתם יכולים לכתוב פקודת אסambilי שתሂרץ על מעבד 8086, והמעבד החדש ביותר של אינטל יוכל להריץ אותה.
- בשוביל ללמידה את ארגון המחשב צריך להתחיל ממקום כלשהו, ומעבד ה-80x86 הוא מקום טוב להתחילה ממנו.

מכונה פון נוימן – Von Neumann Machine

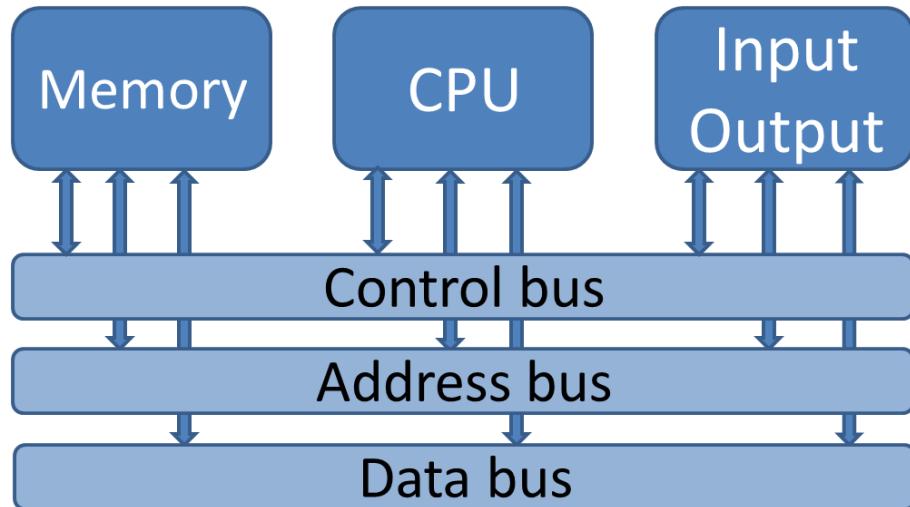
התכנון הבסיסי של מחשב נקרא ארכיטקטורה. ג'ון פון נוימן היה חלוץ בתכנון מחשבים, וניתן לו הเครดיט עבור הארכיטקטורה של רוב המחשבים שאנחנו משתמשים בהם היום. **ארכיטקטורת פון נוימן** (או באנגלית **Von Neumann Architecture – VNA**) כוללת שלוש אבני בניין מרכזיות: יחידת העיבוד המרכזית (Memory) זיכרון (Memory)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



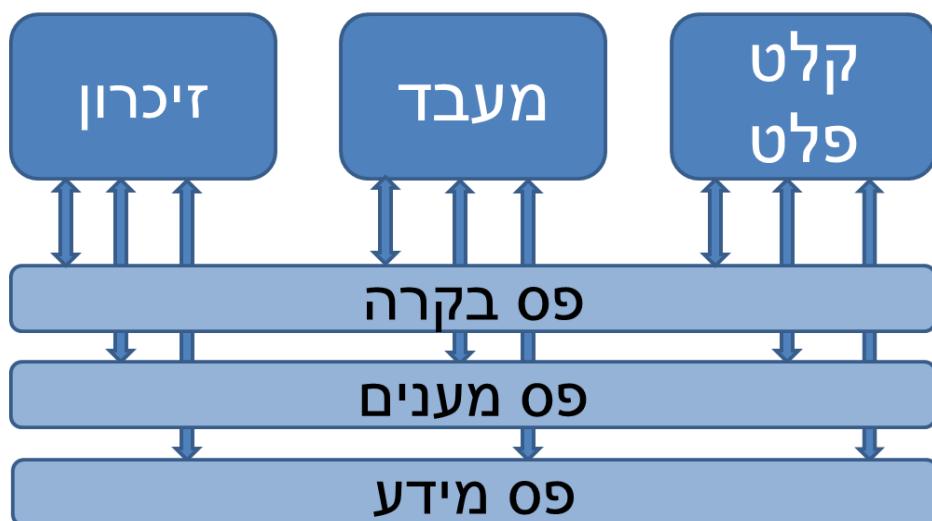
ג'ון פון נוימן (1903–1957) והמחשב הדיגיטלי הראשון

במעבדים מבוססי ארכיטקטורת VNA, כמו משפחת ה-i86x80, ייחדית העיבוד המרכזית מבצעת את כל החישובים. מידע והוראות למעבד (מה שנקרא קוד התוכנית) נמצאים בזיכרון עד שהם נדרשים על-ידי המעבד. מבחינות המעבד, ייחדות 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).

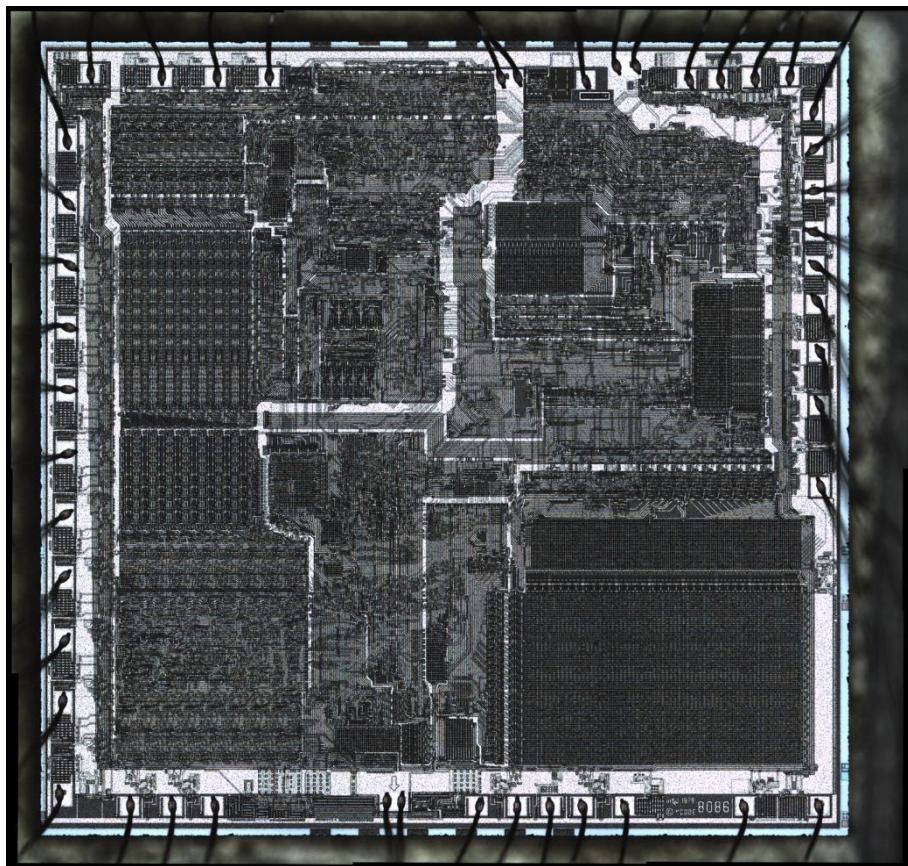
יחידת העיבוד המרכזית של המעבד מתחברת עם הזיכרון ועם ייחדות הקלט / פלט באמצעות קווי תקשורת שנקראים **Bus**, או בעברית **פסים**. יש סוגים שונים של קווי תקשורת, שלכל אחד יש תפקיד ייחודי.



שמות הרכיבים ב-**VNA**, במשמעות המקובלם באנגלית



המשמעות המקובלם בעברית



תמונה רנטגן של מעבד ה-8086 (התמונה בהגדלה, הגדל האמתי הוא 33 מ"מ מרובעים). הקווים הביררים הם הפסים השוניים. המרובע מצד ימין למטה הוא זיכרון פנימי של המעבד. החוטים הכהים המוחברים להיקף הם קווי התקשרות לרכיבי ה-I/O שהזוזן 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 של קווי חשמל שימושיים אותן בין הרכיבים השונים. האותות המועברים הם רמות מתח צדדיות – יש מתח שמייצג 0 ויש מתח שמייצג 1. כיוון שرمות המתח שונות בין מעבד למעבד, נתיחס אליהן 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? תפקידם של הפסים השונים הוא לאפשר לתת פרשנות ברורה לכל נתון והוראה שעוברים.

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 בבת אחת. אין זה אומר שמעבד שמהיר 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 בבת אחת, מסוגל לגשת גם לכמה 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 ולקרוא מהזיכרון בקצב מהיר יותר ולכן יעבד 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 (כוכור, בית הוא היחידה הקטנה ביותר של זיכרון שיש לה כתובת משלها).

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

כמה 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 כתובות שונות. לדוגמה, 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 מעלה ומעבר. מעבדים מתוקדים כוללים פס מענים של 32 ביט, ככלומר הם מוגבלים ב-2³² – 4,294,976,296 בתיים – ארבעה ג'יגובייט. בשלב מסוים, גם ארבעה ג'יגה של מרחב כתובות זיכרון הפכו להיות מגבלות, וכיוום מעבדים ומערכות הפעלה כגון Windows 7 תומכים במרחב כתובות בגודל 64 ביט.

פס בקרה – CONTROL BUS

פס הבקרה מכיל קווים חשמליים שתפקידם לעשوت סדר בדרך שבה המעבד מתקשך 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 מהזיכרון?

פס הבקרה מכיל שני קווים, קו קריאה (read) וקו כתיבה (write), אשר קובעים את כיוון העברת המידע. כאשר קוו 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 הזכירנו שמעבד בעל פס מידע של 16, 32 או 64 ביטים, יכול לקרוא מידע של בית בודד. קווי בקרה שננקאים byte enable מאפשרים פעולה זו.

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

הזיכרון

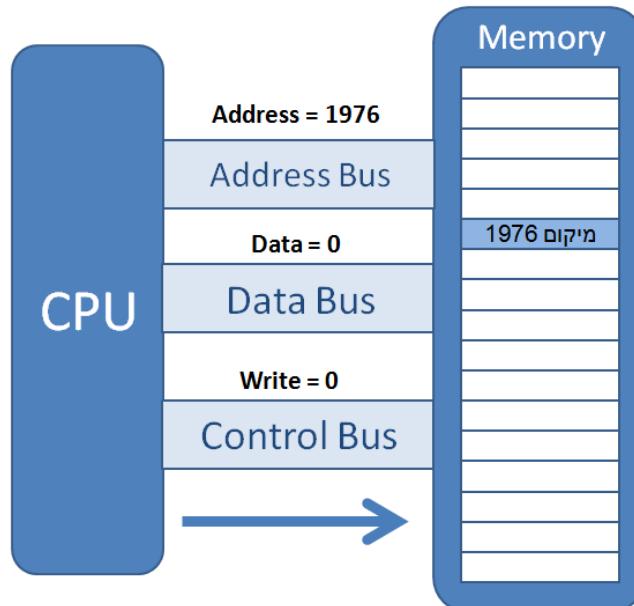
בסעיפים הקודמים הזכירנו, שכמויות המיקומות בזכרון היא 2^n , כאשר n כמות הביטים בפס המענים. בפרק זה נדון בהרחבה בנושא הגישה לכתובות שונות בזכרון.

אפשר לחשב על הזיכרון בתור מערך של בתים. 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 בתים.

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



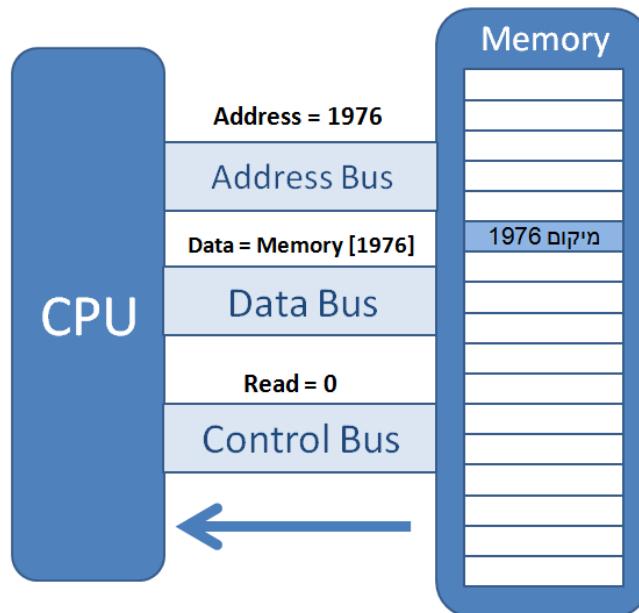
1. המעבד שם את הערך "0" בפס המידע.
2. המעבד שם את הכתובת 1976 בפס המידע.
3. המעבד משנה את קו ה-`write` בפס הבקרה וקובע את ערכו ל-0 (הקו "פועל").



פעולות כתיבה לזכורן

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

1. המעבד שם את כתובת 1976 בפס המיעון.
2. המעבד משנה את קו ה-**read** בפס הקריאה וקובע את ערכו ל-0 (הקו "פועל").
3. המעבד קורא את הערך שבפס המידע.



פעולות קריאה מזיכרון

כעת נבחן איך ערכאים בגודלים שונים שמורים בזיכרון.

נניח שהגדינו 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, שניתן לרשום כ-hah, אך כפי שנלמד בהמשך יש לה ah גם משמעות אחרת.

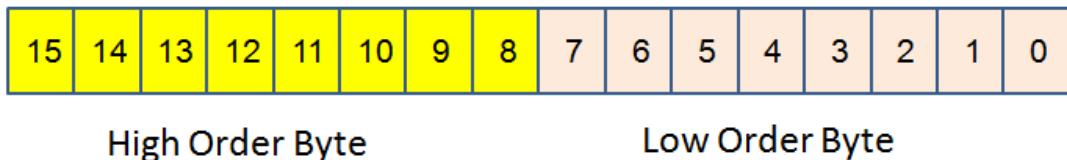


לאחר שמירת הערכים, הזיכרון שלנו ייראה כך:

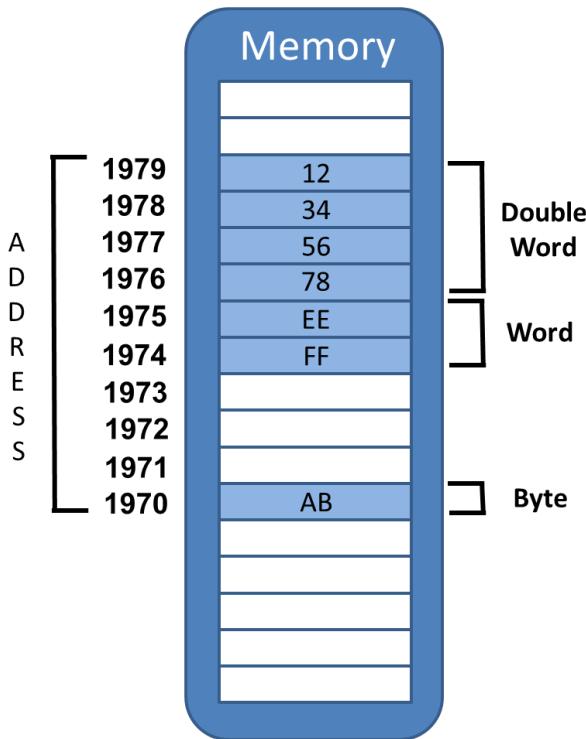
1. במקום 1970 בזיכרון שומר הערך 0ABh.
2. במקום 1974 בזיכרון שומר הערך 0FFh, במקום 1975 בזיכרון שומר הערך 0EEh.
3. במקום 1976 בזיכרון שומר הערך h78, במקום 1977 שומר h56, במקום 1978 שומר h34 ובקום 1979 שומר h12.



שימוש לב – הבית ה-H.O.L. שומר בכתובת נמוכה יותר בזיכרון, הבית ה-L.O.H. שומר בכתובת גבוהה יותר בזיכרון.



L.O. Byte, H.O. Byte – תוכרת



אופן השמירה של גודלים 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 05678EEFFh.

.4. Word ממיקום 1970, ולקבל את ה-byte שהשמרנו במיקום 1970 בתוספת ערך כלשהו שקיים במיקום 1971
– בזיכרון תמיד קיים ערך כלשהו, זיכרון אף פעם אינו ריק!

סגמנטים

כזכור, למעבד ה-8086 יש 2^{20} כתובות זיכרון. מצד שני, כפי שנראה בהמשך, כדי לגשת לזיכרון המעבד עשוה שימוש ביחידות בגודל 16 ביטים, שנקראות **רייסטרים (registers)**. כל רגיסטר של 16 ביטים מסוגל להזניק מרחב כתובות מ-0 עד 65,535 (או 0FFFFh). כמובן, צריך למצוא שיטה לחבר שבין מרחב הזיכרון הקיים לרשות המעבד לבין מרחב הזיכרון, המצויץ יותר, שרגיסטר מסוגל לפניו אלו. השיטה שנבחרה היא לחלק את הזיכרון למקטעים קטנים יותר – **סגמנטים (segments)**. כל כתובה בזיכרון ניתנת לביטוי על-ידי מספר ה-**segment offset** (offset) מתחילה הסגמנט. הזרה המקובלת לרשום של כתובה בזיכרון היא:

Segment:offset

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

הגודל המקורי של האופסט קובע את הגודל המקורי של הסגמנט. מעבדים בהם אנו נדונם, בעלי אופסט של 16 ביטים, גודל של סגמנט לא יכול להיות יותר מאשר 2^{16} , 64K כתובות, כאשר הכתובות בכל סגמנט הן בין 0000h לבין 0FFFFh. מעבד ה-8086 גודל כל הסגמנטים הוא בדיק K, 64K, מעבדים מתקדמים יותר ניתנים להגדיר גם סגמנטים קטנים יותר.

כדי להגיע לכתובת של מיקום תחילת סגמנט, כופלים את הסגמנט ב-16. כתוצאה לכך יוצא שסגמנטים תמיד מתחילה בכתובת שהיא כפולה של 16 בתים. לדוגמה, סגמנט מספר 0002h מתחיל $(2^{*}16)$ 32 בתים לאחר תחילת הזיכרון. סגמנט מספר h 0011 מתחיל 272 בתים לאחר תחילת הזיכרון (11 בבסיס 16 הוא 17, 17 כפול 16 בתים שווה 272 בתים).

כדי להגיע לכתובת כלשהו בזיכרון, מוסיפים לכתובת תחילת הסגמנט את האופסט. לדוגמה, נניח שהסגמנט הוא 3DD6h והאופסט הוא h 12. הכתובת בזיכרון תרשם כך – 3DD6h:0012h.

הכתובת בזיכרון תחשיב כך:

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

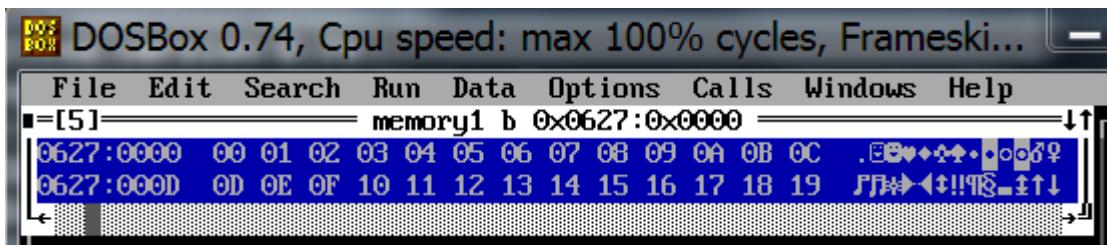
שימוש לב לכך שתווספה ה-0 מצד ימין באבר הראשון נועדה לכפול את כתובת הסגמנט ב-16, כדי להגיע למיקום תחילת הסגמנט.



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-w רק בסעיף זהה כיוון שהוא ממחישה היטב את נושא הסגמנט והאופסט.

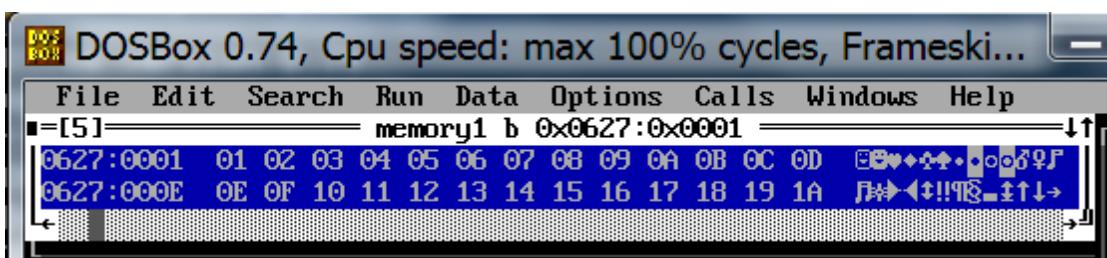
מצד שמאל, רשום מרחב הכתובות בזיכרון שאנו מסתכלים עליו. בדוגמה שלנו, השורה הראשונה מתחילה במיקום בזיכרון שכתובתו היא 0000:0627. בשורה זו יש 13 בתים, ולכן הכתובת الأخيرة בשורה הראשונה היא C000:0627. השורה השנייה מתחילה בכתובת של הבית הבא אחריו: D000:0627.

במרכזו רשומים הערכים השמורים בזיכרון. בדוגמה זו, יש בזיכרון ערכים עליים – 0, 1, 2 וכו'. באוטה מידת הינו יכולים לשומר בזיכרון אוסף אחר של ערכים בגודל בית אחד. בצד ימין מוצג התרגום של הערכים בזיכרון לתחומים טקסטואליים לפי קוד ASCII. אתם מוזמנים לחזור לסעיף שמספר על קוד ASCII ולהיווכח שהתחומים הרשומים הם אכן התרגום של הערכים שבזיכרון.

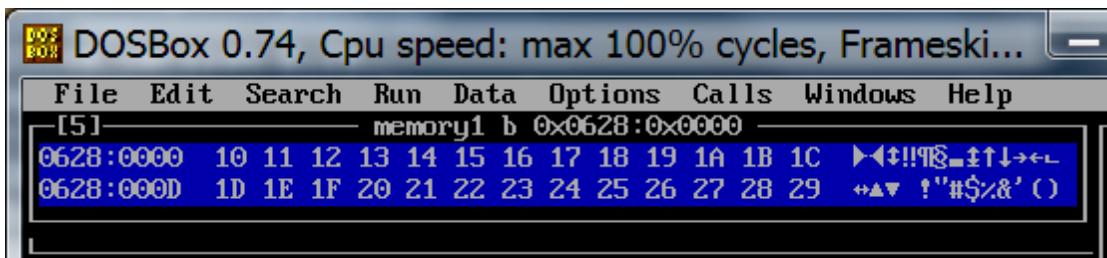


כעת נדגים את מושג האופסט. נעשה זאת על ידי הוספה נוספת למרחב הכתובות בזיכרון. צעת השורה הראשונה מתחילה בכתובת .0627:0001.

שיםו לב שגם הערכים בזיכרון וגם ערכי ה-ASCII מוצעים כעת בבית בודד.



להמשך מושג הסגמנט, נapse את האופסט ובמקומו נקדם את הסגמנט ביחידת אחת. צעת השורה הראשונה מתחילה בכתובת .0628:0000. שימו לב לכך שפעולה זו גורמת להסתה של 16 בתים בזיכרון:



לסייע, בדוגמה אנחנו רואים بصورة מוחשית שSEGMENT תמיד מתחילה במיקום שהוא כפולה של 16 בתים.

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 נמשיך ליחידה האריתמטית לוגית (Arithmetic & Logical Unit), שאחראית על ביצוע פעולות חישוב כגון חיבור, חיסור, השוואה ופעולות לוגיות שונות, ונסיים ביחידה הבקרה (Control Unit), שאחראית על פענוח פקודות המכונה, ניהול רצף התוכנית, הוצאה לפועל של פקודות ואחסון התוצאות המתתקבלות.

רגיסטרים – 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 יותר – במקום לפנות לזכרון ולהחזיר ל渴בלת הנתונים, המעבד אינו מחייב כלל. מצב זה נקרא *zero wait*. יש כמה מצומצמת של רגיסטרים ולחילוקם יש תפקידים מיוחדים, שמכבילים 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 "אוגר"

קייםים דורות שונים של מעבדים במשפחה ה-80x86. כמה רגיסטרים וגם הגודל שלהם בביטים עשוי להיות שונה בין דור אחד למשנהו. אנחנו נדונו 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. למרות שנייה 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 לסתירת כמות הפעמים שהרצינו לו לא, לכמות התווים בקובץ או במחוזת.
dx	Data register	מידע	משמש לשתי פעולות מיוחדות: ראשית, ישן פעולות אריתמטיות דודשות מיקום נוספת לשימירת התוצאה. שנית, כשפונים 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 שנייה לעשوت עם ax). כמו כן הם שימושיים 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.

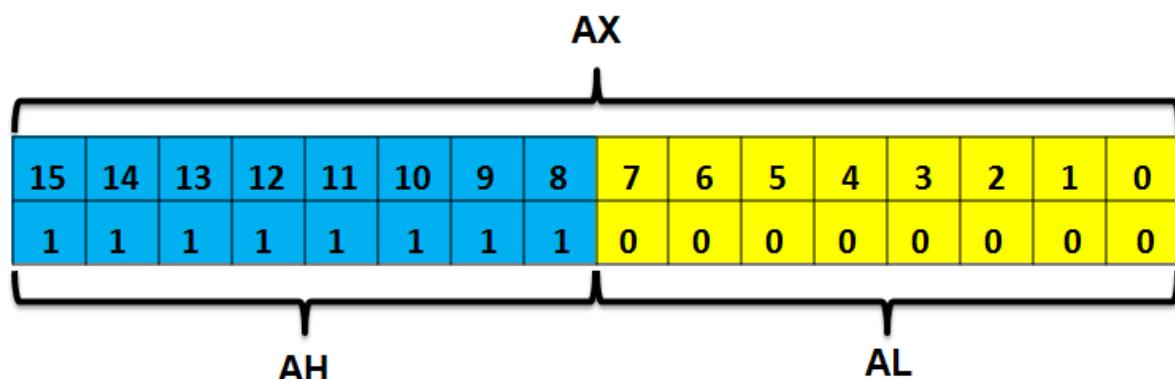
כאמור כל אחד מהרגיסטרים הנ"ל הוא בעל 16 ביטים, כולל שני בתים. לעיתים אנחנו צריכים לעסוק רק ב-8 ביטים. לדוגמה, כשהאנחנו רוצים להעתיק מידע לבית כלשהו זיכרונו. אם נתיק לזכור את ah כולו, הזיכרונו ישנה בשני בתים – שינינו את הבית שרצינו לשנות בזיכרון, אבל כתוצאה לוואי לא רצואה שינינו גם את ערכו של הזיכרונו בכתובת הבאה וייתכן שדרסנו ערך חשוב.

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

נלמד כעת פקודת אSEMBLY ראשונה – הפקודה mov (קייזר של move) מבצעת העתקה של הערך שנמצא מצד הימני אל הצד השמאלי. לדוגמה:

```
mov    ax, 0FF00h
```

לאחר ביצוע הפקודה, רגיסטר ax ייראה כך:



ניתן להגיע לאותה תוצאה גם על-ידי הקוד הבא:

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

שימושם לב שריגיסטרים של 8 ביטים אינם רגיסטרים עצמאיים. שינוי של al ישנה גם את ax, ולהיפך.

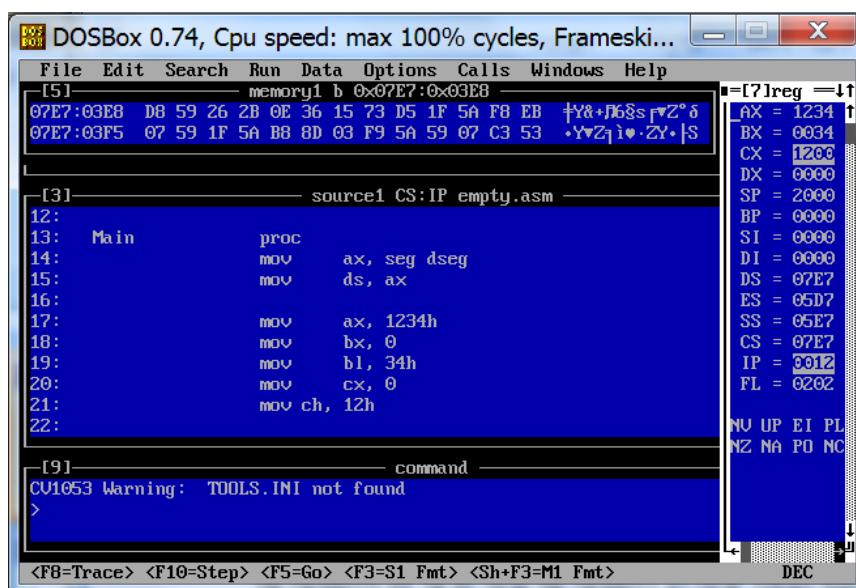


באופן דומה, ניתן לעשות שימוש ברגיסטרים bh, cl, ch, bl, dl, dh.

להלן טבלה המסכםת את הרגיסטרים הכלליים שגודלם 8 ביט:

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 **bl** על **ax**: שינוי של **bl** 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 לפנוטם באופן ישיר אל הבית העליון או התחתיו שלהם כמו שעשינו עם **ax**, **bx**, **cx** ו-**ds**.

לתרגול נושא הרегистרים הכלליים, להלן טבלה הכוללת פקודות העתקה ורגיסטרים. כל פקודה העתקה היא של ערך או רגיסטר לתוכו 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	לרגיסטר
								AX	הערך h1234
								AL	0ABh
								BX	0ABCDh
								CH	0EEh
								DL	0BBh
								DH	הריגיסטר CH
								AH	הריגיסטר DL
								CX	הריגיסטר BX

הפתרון 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	

רегистרי סגמנט – Segment Registers

במעבד ה-8086 יש ארבעה רегистרים מיוחדים (בוגדים מאחרים יותר במשפחה ה-80x86 נוסף רегистר חמישי - FS):

CS - Code Segment

DS - Data Segment

SS - Stack Segment

ES - Extra Segment

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

הрегион CS שומר את מיקום תחילת סגמנט הקוד. סגמנט זה מכיל את פקודות המכונה שהובוצעות 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 `model small` וכאן ישנו רק סגמנט קוד אחד). באופן מעשי לא סביר שתכתבו תוכנית אסטטטי שתדרוש יותר מסגמנט קוד אחד.

לגביו הрегион 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, נותן לנו גמישות נוספת – אם אנחנו רוצים לפנות לSEGMENT נוסף, זה בדיקת מה שהוא אפשר לנו. רגיסטר זה שימושי, לדוגמה, בהעתקות של רצפי נתונים אל הזיכרון.

רגיסטרים ייעודיים – Special Purpose Registers

ישנם שני רגיסטרים ייעודיים:

IP - Instruction Pointer- מצביע הוראה

FLAGS - דגליים

רגיסטר ה-IP מחזיק את הכתובת של הפקודה הבאה לביצוע. זהו רגיסטר של 16 בית שŁemañsa משמש כמצביע (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 ביט אחד. רגיסטר זה משתנה בעקבות הרצת פעולות אריתמטיות שונות וכאן הוא שימושי לפעולות בקרה על התוכנה. כמו כן יכול רגיסטר הדגליים לשנות 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, הערך של רגיסטר ה-IP והדגליים ישתנו על-ידי המעבד בזמן ריצת התוכנית. נציין, כי קיימות פקודות שמאפשרות לתוכנת לשנות את רגיסטר הדגליים, אולם הן מחוץ להיקף של ספר לימוד זה.

אנחנו נקדים לרגיסטרים 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 (ALU)

שנתרחשת בו רוב הפעולות. ה-ALU טוען את המידע שהוא צריך מהרגיסטרים, לאחר מכן יחידת הבקרה קובעת



ל-ALU איזו פעולה צריך לעשות עם המידע, ובסיוף ה-ALU מבצע את הפעולה ושומר את התוצאה ברגיסטר שנקבע כרגיסטר יעד.

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

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

יחידת הבקרה – Control Unit

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

למחשבים הראשונים היה פאנל עם שורות של מנגלים魑魅魍魎, שנitin היה לחוטם אותם החשמליים אפשר היה לקבוע איזו פקודה המעבד יירץ. בשיטה זו, כמות הפקודות שנitin היה להריצ' בתוכנית אחת הייתה שווה למספר השורות בפאנל. אחת הפתוחיות המשמעותית במדעי המחשב הייתה המצאת הרעיון לשמר את התוכנה בזיכרון המחשב ולהביא אותה ליחידת העיבוד פקודה אחרי פקודה. בשיטה זו כל פקודה מתורגמת 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, IP, או Instruction register, אותו זכרנו בסעיפים הקודמים. ה-IP מחזק את הכתובת בזיכרון של הפקודה הבאה שיש להעתיק. לאחר שייחידת הבקרה מעתקה את ה-OpCode לתוך רגיסטר הפענוח, היא מקדמת את ה-IP אל כתובת הפקודה הבאה וכך הלאה.

שעון המערכת (הרחבה)



מכונות פון נויין מעבדות פקודה אחרי פקודה, באופן טורי.

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

mov ax, bx

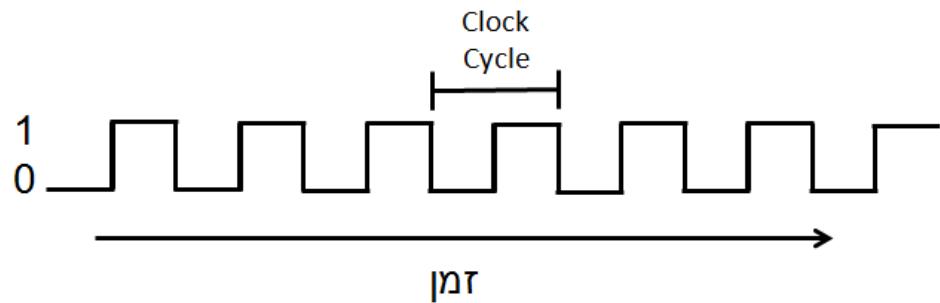
add ax, 5

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

כל פעולה דורשת זמן לביצוע. זמן זה כולל פעונה ה-OpCode, פניה לזכרון (בדרך כלל), טיענת ערך כלשהו לריגיסטר, ביצוע פעולה אריתמטית ועוד. כדי שפעולות יתבצעו לפי סדר, המעבד צריך מגננון שיתזמן את הפעולות – שעון המערכת.

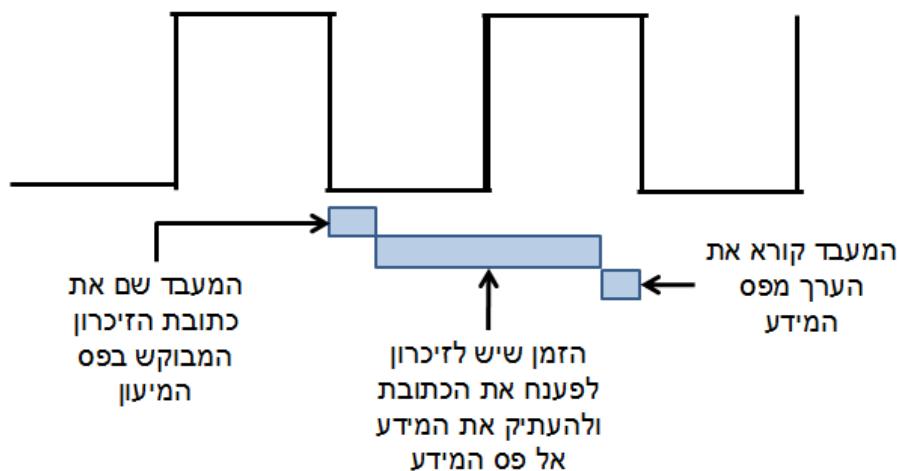
שעון המערכת הוא אחד החשיוני על פס הבקרה Control Bus, אשר משנה את ערכו כל פרק זמן קבוע בין 0 ל-1. התיירות שבה האות החשמי משתנה בין 0 ל-1 היא תדיםות שעון המערכת. הזמן שלוקח לאות להשתנות מ-0 ל-1 וחזרה ל-0 נקרא **clock cycle**. הזמן של clock cycle הוא אחד חלק תדיםות השעון. לדוגמה, מעבד בעל תדר שעון של 1MHz הוא בעלי clock cycle של 1 מיקרו שנייה (1/1,000,000 של שנייה).



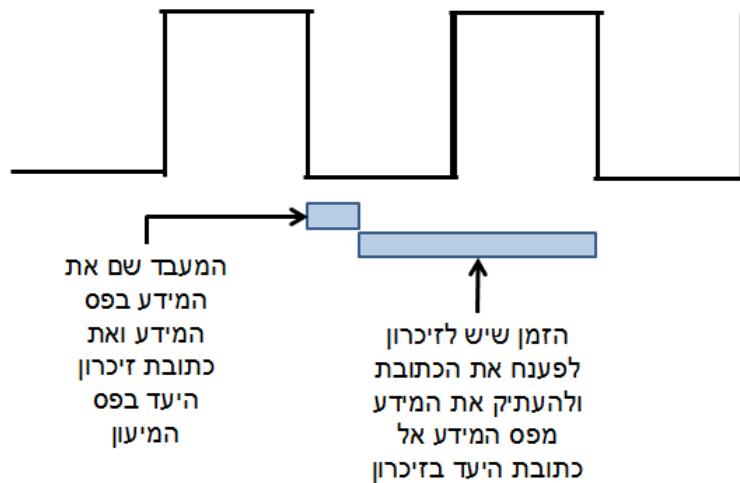


כדי להבטיח סync'רוניזציה, המעבד יתחיל לבצע פעולה או בנקודות הזמן בה מתרחשת עלייה את השעון (זמן שהאות משתנה מ-0 ל-1) או עם ירידת האות השעון (זמן בו האות משתנה מ-1 ל-0). כיוון שככל המשימות של המעבד מסונכרנות על ידי שעון המערכת, משימה לא יכולה להימשך פחות מ-*clock cycle* אחד, ככלור תדרות שעון המעבד מגבילת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

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



סיכום של פעולה קרייה מהזיכרון על ציר הזמן של שעון המערכת



סיכום

בפרק זה למדנו אודוט ארגון המחשב, בדגש על מעבדי משפחת ה- $80x86$: התחלנו מפירוט הסיבות שבגלאן רלבנטי ללמידה על משפחת מעבדי ה- $80x86$ למרות 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- $80x80$ והכרנו את החלוקה לsegments ואופסטים, שמשמשת להגעה לכל כתובות בזכרון. לאחר מכן, עברנו לדון ביחידת העיבוד המרכזית על חלקיה השונים. התחלנו מהרגיסטרים השונים ועברו מה משמש בדרך כלל כל רגיסטר. ממש עברנו ליחידה האריתמטית, האחראית על ביצוע הפעולות, וליחידה הבדיקה, שהחראית להביא את פקודות המכונה לפעונה. 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 בטוחים WHATSOEVER שמדוברם בנושאים אלו – חיזרו וודאו שגם מבניים את התיאוריה. בפרק הבא נתקן את סכיבת העבודה שתאפשר לנו להתחליל לתכנתה.

פרק 4 – סביבת עבודה לתוכנות באסמבי

מבוא

בפרק זה נלמד על כל הפעולות בהם נשתמש בשביל לתוכנה בשפה אסמבי. ראשית, נציין כי יש מגוון של כלים לעובדה באסמבי, ופרק זה אינו מתיימר להציג את כולם. קיימים אוסף של אסמבלרים (תוכנות הממירות אסמבי לשפת מכונה) – הבודלים ביניהם קטנים, אבל קוד אסמבי שנכתב לאסמבלר כלשהו לא יתאים בהכרח לאסמבלר אחר. למעשה, אפשר להחליף כל אחת מהתוכנות שנציג כאן בתוכנה אחרת, אבל משיקולים של נוחות ורצון להשיג בסיס משותף בין כלל לומדי האסמבל, בחרנו להציג אפשרויות אחת בלבד לכל תוכנה. היעד אליו אנו שואפים להגיע בסיום פרק זה הוא ליצור קובץ ראשון בשפה אסמבל, קובץ שנקרא לו `base.asm`, להפוך אותו לקובץ בשפה מכונה ולהיות מסוגלים להריץ את התוצאה באמצעות כלי שנקרא **דיבאג'ר (Debugger)**. בסיום הפרק יש תרגיל מחקר קטן, לתלמידים המעניינים להרחב את הידע שלהם. בפרק הקודם למדנו באופן תיאורטי שהאסמבלר מתרגם פקודות בשפה אסמבל לשפה מכונה – `Opcodes`. תרגיל המחקר משלב את הכלים שנלמדו בפרק זה על מנת להבין יותר לעומק את הדרך בה מתורגמות פקודות האסמבל `לـ-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 בעיתי. אנחנו יכולים לכתוב ב-`Notepad++`, `Visual Studio` או כל `Editor` אחר. השימוש ב-`Notepad++` מומלץ במספר סיבות:

- התוכנה חינמית
- פשוטות התקנה ושימוש
- טקסט מוארocabularies – פקודות בכחול, הערות בירוק, רגיסטרים בשחור וכו'. דבר זה מקל על ההתחמזהות בקוד.

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



- לכל קובץ יש שם ו הסיומת. לדוגמה hello.doc, doc היא הסיומת של הקובץ שנקרא hello.hello. 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 וקללה 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 וההגדירות צריכות להיות ברורות לכם – הן יתבררו בהמשך. לטובת המשך העבודה, פיתחו קובץ חדש ב-Notepad++ והעתיקו לתוכו את הקוד הבא (האזורים הירוקים הם הערות ואין צורך להעתיק אותם, כמו כן מספרי השורה נוצרים אוטומטית ואין להעתיק אותם). לאחר מכן, שמרו את הקובץ בשם base.asm.

[הקובץ נמצא גם בקישור](http://data.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 לתוכנה בשפת אסמבלי, והסביר שני מתקדם. בשלב זה עדין 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 באזוריים 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 שמו של משתנים בזיכרון המחשב. נזכיר כי משתנים הם שמות שנחנו נתונים כתוכות בזיכרון של המחשב. בטור **DATASEG** לא נהוג לשים קו.

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

```
var1    db      5
```

ובכך הגדרנו משתנה בשם **var1** שמקבל את הערך 5. האסמלר (נגיע 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 שנחנו רוצחים שהמעבד יירץ. בתחילת הסגמנט יש כמה פקודות עוז, 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 הריגיטר **al**.
זה כל מה שצרכי לדעת בשבייל להתחיל לעורך שינויים בקובץ **base.asm**.

הסבר מפורט:

מעבר שורה שורה על הקובי:

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

- **MODEL small** – מודל זיכרון **small**, קבוע לאסמלר שהתכוונית מכילה שלושה סגמנטים – **Data**, **Code**, – 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 יכול לבוא במקומה – לדוגמה – “**main**” – העיקר להיות עקביים עם הוראת **end** שבשורה האחרונה.

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

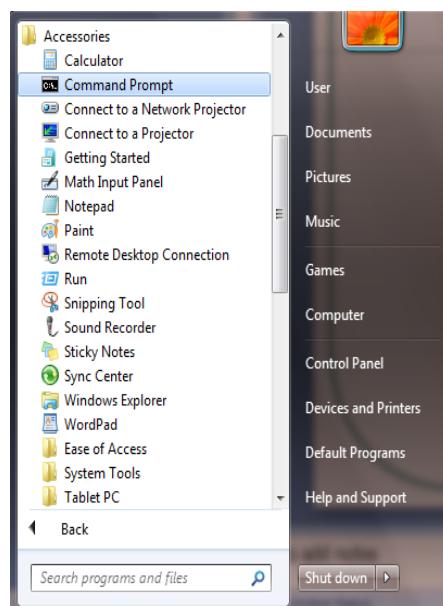
- **exit** – יציאה מסודרת מהתוכנית. שתי השרוות שמתוחת למילה **exit** הן קוד שגורם ליציאה מסודרת של התוכנית ושחרור הזיכרון שהיא תפסה, כך שמערכת הפעלה תוכל להשתמש בו לצרכים אחרים. הסבר מפורט תמצאו בחלק על **פקודות DOS**.

- **end start** – המילה **end** אומרת לאסמלבל להפסיק את הקומpileציה, ככלומר פקודות שיופיעו אחרי מילה זו לא יתורגם לשפת מכונה. לאחר **end** מופיעה תווית שאומרת לאסמלבל מאיפה המעבד צריך לתחילה 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 **על-ידי Command Line**. ב-**Command Line** אין אפשרות להשתמש בעכבר, כגון כדי להריץ קובץ, לדוגמה, במקרים לעשו עליון דאבל קליק עם העכבר צריך לכתוב את שם הקובץ.

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



לאחר מכן יפתח חלון הדוגמת החלון הבא:



קייםים אתרים שונים שמסבירים את הפוקודות 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 למספרה כלשהי יש לרשום:

CDDirectoryName

כאשר שם הספריה בא במקום "DirectoryName". לדוגמה:

CD Games

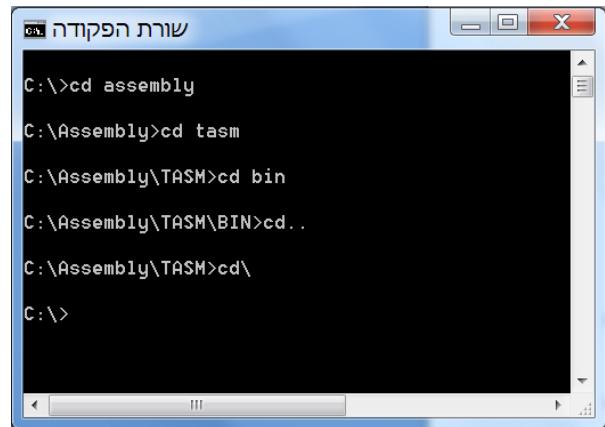
כדי לחזורספריה אחת אחורה בעז הספריות יש לרשום:

CD ..

וכדי לחזור לתחילה העז יש לרשום:

**CD **

לדוגמה:



DIR ♦

פירוט כל הקבצים וחתרי 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 אחרி עמוד:

```
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 שנקרأت 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.

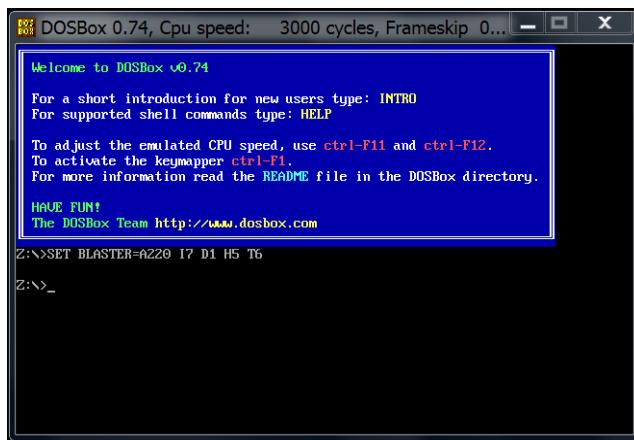
אמולטור של מחשב קלטי, **Commodore 64**, שיצא בשנת 1982, מאפשר לאייפון להרים משחקים שנכתבו ל-**Commodore 64**



כדי להריץ את סביבת העבודה של אסambilי, נדרש מערכת הפעלה מוקדמת של מיקרוסופט. מערכת הפעלה זו קרובה DOS, קיצור 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: <http://data.cyber.org.il/assembly/dosbox.exe>

הקלקה על דוסבוקס תפתח מסך כדוגמת המסך הבא:



הפעולה הראשונה שאנו רוצים לעשות היא להגיע לכונן בו נמצאים הפרויקטים 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 נכתוב:

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.



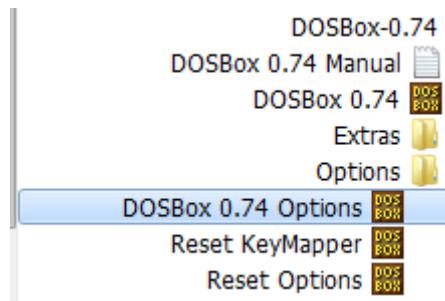
כיוון שאנו מביצים אמולציה של מעבד ישן, מהירות הריצה של הדוסבוקס הופחתה בהתאם ובירית המהדר היא ריצה בקצב של 3000 cycles לשניה בלבד (שימוש לב כתוב בכוורת למעלה). ניתן לשנות את הקצב הזה באמצעות כתיבת הפקודה הבאה במסך ה-Command Line: DosBox

Cycles = max

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

טיפ לעובודה קלה עם Dosbox: אפשר לקבוע לדוסבוקס אוסף של פקודות שיירצטו באופן אוטומטי עם עלית התוכנה. קביעת פקודות אלו יכולה להשוך לנו הקלדה רבה בעתייד. לדוגמה, נוכל לנפג 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 הלוונת. יש לבחור ב-**DOSBox 0.74 Options** בתוך תפריט הדוסבוקס:



לאחר מכן הקלקה על לחץן שמאלי של העכבר תפתח 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 כלשחי לשפת מוכנה. קיימים מגוון אסמבליים לשפת אסמבלי, אנחנו בחרנו לעבוד עם **TASM**, קיצור של Turbo Assembler. הגרסת الأخيرة 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 שימושו במקרים שבהם התוכנה מוחולקת בין מספר קבצים. לדוגמה, קובץ אחד כולל את התוכנית הראשית, SMBצעת קריאה לקטעי קוד שנמצאים בקבצים אחרים. הלינקר יודע לחבר בין הקריאה לקטעי הקוד לבין קטעי 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 **tasm\bin**.

- פיתחו את הקובץ **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**.

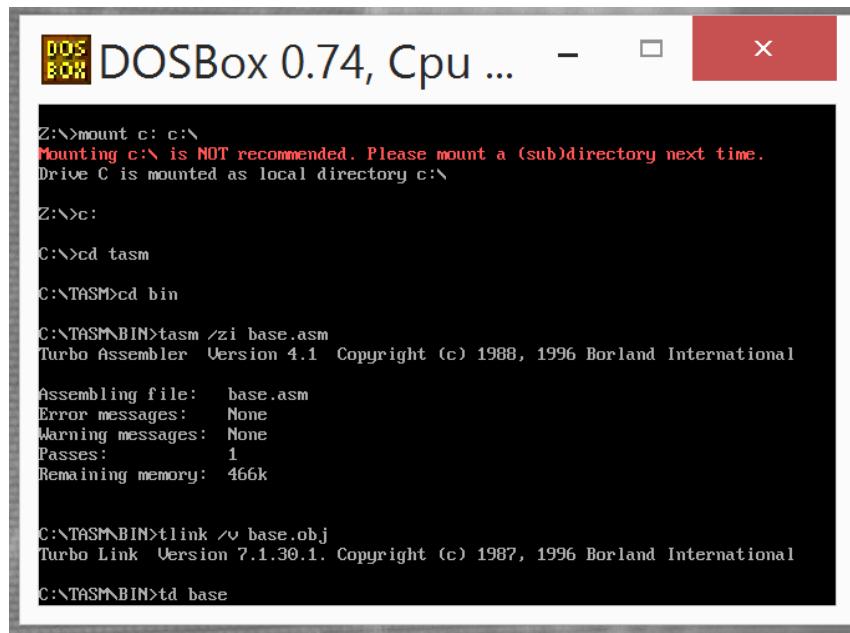
- הפקודה הבאה – SMBוצעת על ידי תוכנת הלינקר **tlink** – מmirah את הקובץ בשפת המוכנה לקובץ הרצה:

tlink /v base.obj

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

סיימנו!

כדי להריץ את הקובץ שנוצר, כתבו base בשורט הפקודה והקישו enter, או כתבו td base כדי להריץ אותו מהדיבאגר.



```

DOSBox 0.74, Cpu ... - X

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

```

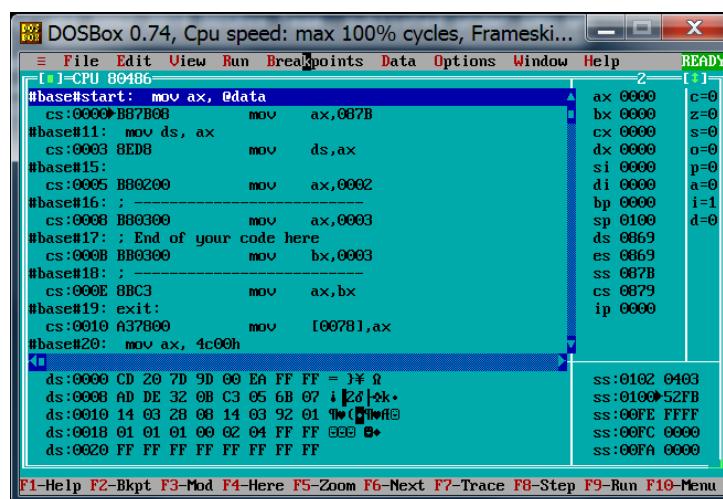
כעת, נריץ את הקובץ base.exe באמצעות תוכנת דיבוג בשם Turbo Debugger.

Turbo Debugger – TD

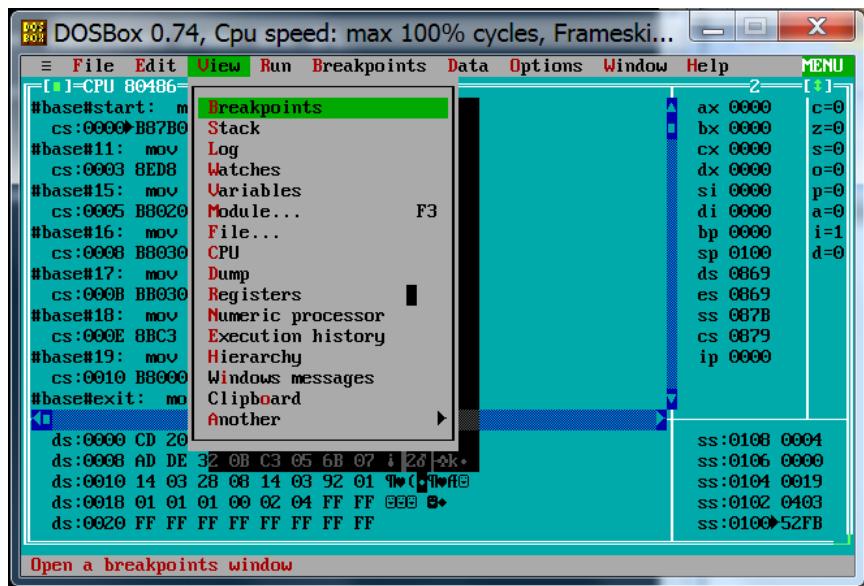
כדי להפעיל את תוכנת הדיבוג שלנו, נכתב בדוסבוקם:

td base

וופיע המסך הבא:



אם במקומות מסוימים הוזה מופעים דברים אחרים, היכנסו לתחפיטים למעלה (לחיצה על מקש view וכתפריט שנספתח, בחרו את האפשרות CPU



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 דרכים. מצב **Step**, על-ידי לחיצה על F8, עבור לשורת הקוד הבאה. אם לשורת הקוד הנוכחית היא פרוצדורה, בהרצה של שורה יחידה נעבור את כל הפרוצדורה ונצא ממנה. כדי לדבג פרוצדורה, השתמש במצב **Trace**, על-ידי לחיצה על F7. במצב זה, כאשרנו מגיעים לתחילתה 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, שמייצג לנו את שורות הקוד ותרגםן לשפת מכונה, כמו כן את סגמנט DATA: -

The screenshot shows assembly code for CPU 80486. The code includes instructions like add, mov, and int. A specific instruction at address cs:000C CD21 is highlighted in blue. Below the assembly code, there is a memory dump window showing hex values for memory locations ds:0000 to ds:0020.

```

[CPU 80486]
cs:FFFB 0000      add    [bx+si],al
cs:FFFD 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 = }¥ ¥
ds:0008 AD DE 32 0B C3 05 6B 07 4 [♂]ak.
ds:0010 14 03 28 08 14 03 92 01 9 [♀]lf
ds:0018 01 01 01 00 02 04 FF FF 0000 00
ds:0020 FF FF FF FF FF FF FF FF FF FF

```

- הרגיסטרים והדגלים:

The screenshot shows the register status for CPU 80486. It lists registers with their current values and flags (c, z, s, o, p, a, i, d) set to 0 or 1.

Register	Value	c	z	s	o	p	a	i	d
ax	0000	0							
bx	0000	0							
cx	0000	0							
dx	0000	0							
si	0000	0							
di	0000	0							
bp	0000	0							
sp	0100	0							
ds	0869								
es	0869								
ss	087A								
cs	0879								
ip	0004								

ניתן לשנות את ערכו של כל רגיסטר על-ידי סימונו והקלדת ערך כלשהו:

- ה-STACK, שמציג לנו את סגמנט ה-STACK – כרגע אנחנו לא יודעים עליו הרבה, אבל נשתמש בו בהמשך:

The screenshot shows the stack segment (ss) starting at address 0102. It displays memory values: 0403, 52FB, FFFF, 0000, and 00FA. The value 52FB is highlighted in yellow.

ss:0102 0403
ss:0103 52FB
ss:00FE FFFF
ss:00FC 0000
ss:00FA 0000

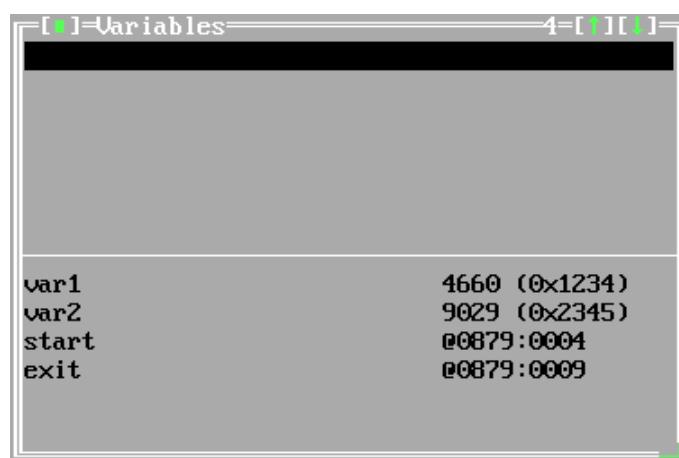
- אנחנו יכולים לנקה משתנה או ביטוי כלשהו ולהכנסו אותו בתור watch. ערך הביטוי יעדכן באופן דינמי. לדוגמה, ייצורנו שני משתנים ב-DATASEG:

Var1	dw	1234h
Var2	dw	2345h

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

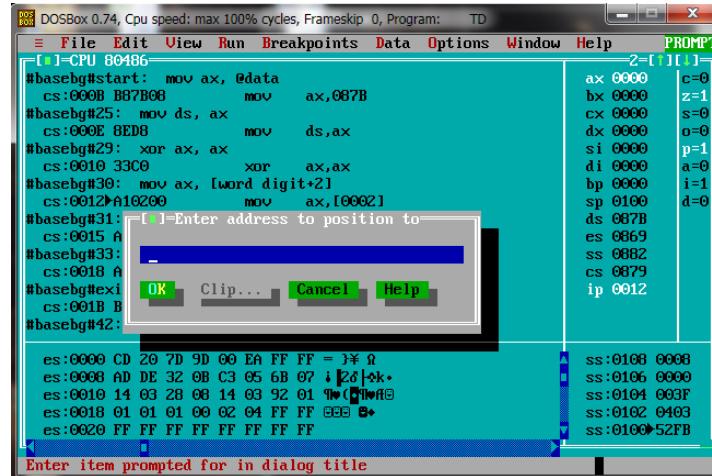


– מציג לנו את רשימת כל המשתנים והתוויות בתוכנית. לדוגמה:

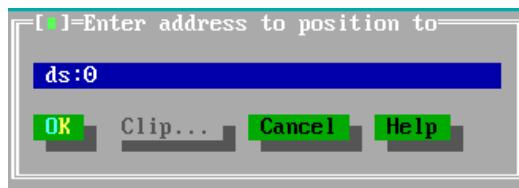


מעבר למקום מבוקש בזיכרון

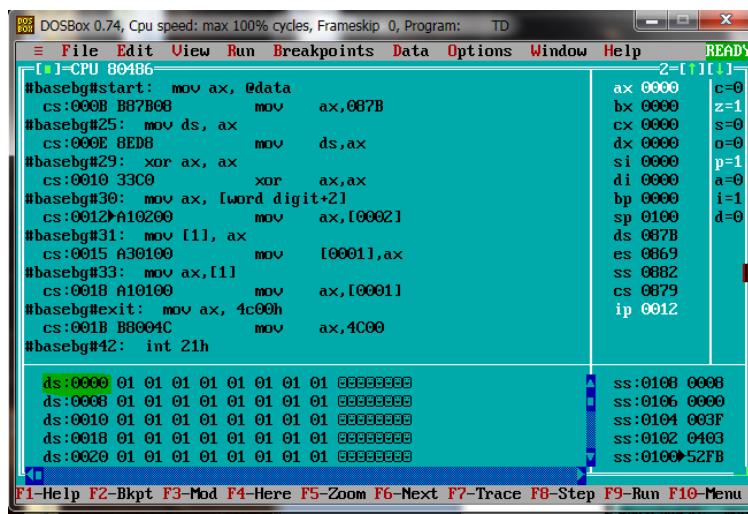
עמדו על החלק התההון של המסר, היכן שכותבים הערכים שנמצאים בזיכרון. לחיצה על CTRL+G תפתח את המסר הבא:



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



ופעולה זו "תקפי" אותו לאוצר המבוקש:

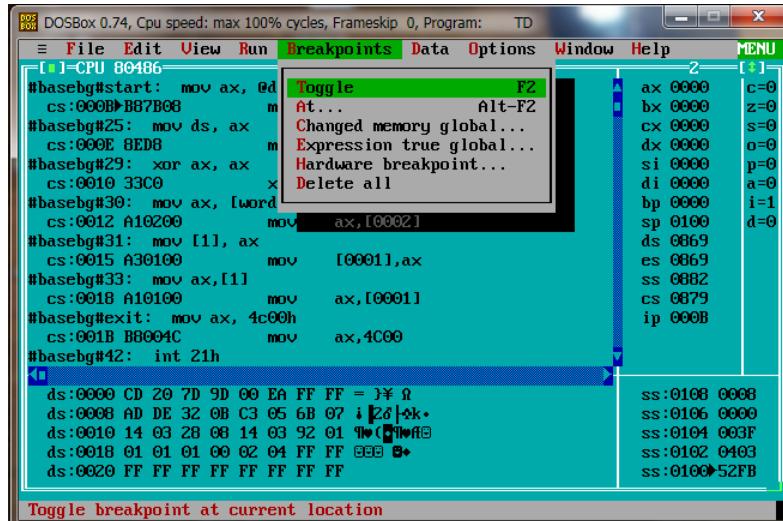


שימוש לב לנק שבתחלת התוכנית ds עדין אינו מוגדר להציג על המיקום של DATASEG, רק אחרי שיופיעו ה الأوامر הראשונות ds מקבל את הערך הנכון. לכן, עלינו ללחוץ פעמיים על F8 לפני שנבצע את הפעולות האחרונות.



שימוש ב-breakpoints

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



לאחר שקבענו breakpoint, אם נרים את התוכנית בעזרת פקודה Run (F9) היא תעצור בשורת הקוד שיש עליה breakpoint. זה יאפשר לנו להגעה מהר לנקודות בקוד שאנו חושדים שיש בהןบางים.

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



טיפים לעובדה עם TD

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

- הריצה מחדש של התוכנית: CTRL+F2

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

- מעבר למסך אחר (ויציאה ממנו): לחיצה על ALT+Enter

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

תרגיל מחקר (הרחבה) – קידוד פקודות אסמלרי ל-**Opcodes**



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

השאלה שאנחנו מעוניינים להחקור, אם כך, היא איך פקודת **mov** מיתרגמת לאחדות ואפסים בסגמנט הקוד שבזיכרנו. נחקור את פקודת **mov** כיון שכבר נתקלנו בה בעבר. פקודות אסמלרי אחרות מיתרגמות באופן דומה. עובdot החקר היא **למצוא את ה-Opcode של הפקודה mov ax, dx בלי להריץ את הפקודה הנ"ל, אלא רק באמצעות התבוננות בפקודות אחרות.**

הדרך וכליים לביצוע המשימה:

1. הקובץ מכיל שלד של תכנית. הכניסו את הקוד שאותם רוצים להריץ לאחר הערה `"Your code"`. שמופיעה לאחר שורת הקוד `.here`

```
mov ds, ax
```

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

ax	0000	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	0869	
es	0869	
ss	087B	
cs	0879	
ip	0000	

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`

חזרה למשימת המחבר: נתונה הפקודה `mov ax, dx`. 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, שנעבוד אליה מעכשו וайлד. קיימות סביבות עבודה שונות ומגוונות, אך למען האיחדות הتمקדנו בסביבת עבודה אחת. הכלים שקיבלנו בפרק זה הם:

Editor Notepad++ -

הרצה דרך Command line, DOS, כולל פקודות DOS -

אמולטור DOSBOX -

אסמלבר Turbo Assembly -

لينקר Tlink -

דייבאג'ר 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

מבוא

חלקים 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 ותוכנו, הזכרנו בקצרה שני רגיסטרים ייעודיים, **FLAGS**.
רגיסטר IP ורגיסטר

IP - Instruction Pointer- מצביע ההוראה

FLAGS – דגליים

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

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

IP – Instruction Pointer

רגיסטר ה-IP מholding את הכתובת של הפקודה הבאה לביצוע. זהו רגיסטר של 16 ביט שלמעשה משמש כמצביע (pointer) לתוך ה-.code segment:

העתיקו והריצו את התוכנית הבאה, המבוססת על **base.asm** בתוספת מספר שורות קוד (מודגשות):



```

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:

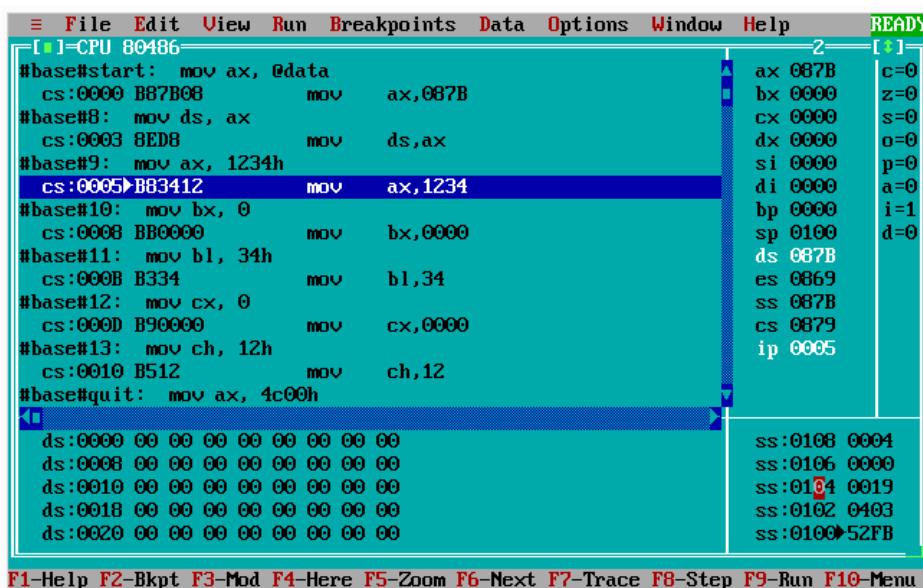
```

```

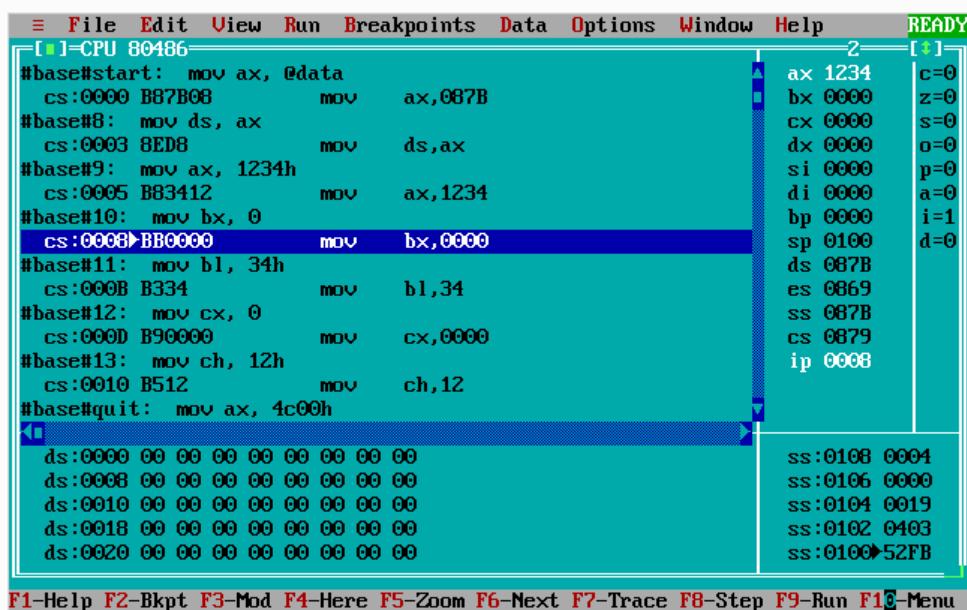
    mov  ax, 4c00h
    int   21h
END start

```

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



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



השינויים שהתרחשו הם:

- .1. הפס הכהול בפסק, שמציג את השורה הבאה שהמעבד ירץ, 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`.

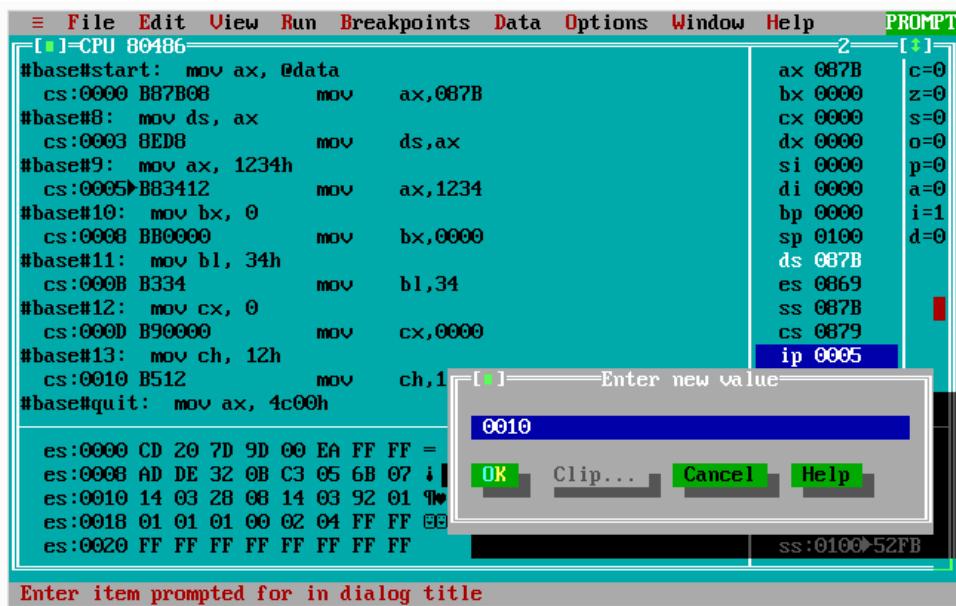
הסיבה שהתקדמות הייתה בשלושה בתים, היא משומם שהפקודה `mov ax, 1234h` תופסת שלושה בתים בזיכרון התוכנית. לא כל הפקודות תופסות שלושה בתים בזיכרון התוכנית – במקרים אלו `IP` יכול להשתנות בבית אחד או בשניים. בנוסף, נראה בהמשך מצבים בהם הערך של `IP` יקפוץ קדימה או אחורה, לפי הצורך.

תרגיל 5.1



.א. המשיכו להריץ את שורות הקוד של התוכנית ויעקבו אחרי השימוש של `IP` בין פקודה לפקודה. האם אתם מזמינים שבו `IP` אינו משתנה בשלושה בתים?

.ב. הריצו את התוכנית מההתחלת, ועיצרו כאשר `IP=0005h` ב開啟 ידני, עימדו עליו עם העכבר והכניסו ערך כלשהו. יפתח לכם חלון דוגמת החלון הבא:

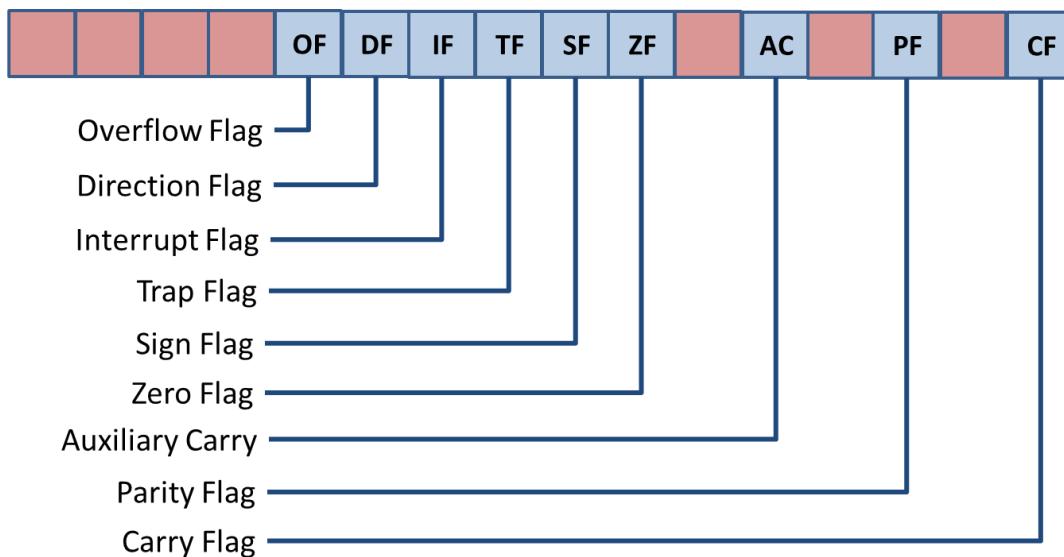


.ג. שנו את ערכו של `IP` ל-`0010h`, ללחוץ על `enter` וודאו שגם `IP` מקבל את הערך שהזנו. עברו לפקודה הבאה על-ידי לחיצה על `F8`. לאיזו פקודה הגעתם? האם הפקודות שבין `0005` לבין `0010` במקטע הקוד אכן ה貼צעו (שימו לב לערכים של הרגיסטרים `ax, bx, cx`)?

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 חשמליים בני בית אחד, שעוזרים לקבוע את מצב המעבד. למרות שיש ב-16 ביטים, רק לחלקם יש שימוש.

להלן שמות הדגלים והמקום שלהם בתחום רגיסטר FLAGS:



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

Zero Flag -

Overflow Flag -

Carry Flag -

Sign Flag -

לדגלי התנאים החשובים מיוחדת, מכיוון שהמעבד משתמש בהם כדי לבצע פקודות הקשורות 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 האחרונה 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?). כתע 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. תוצאת הרגיל זה הייתה אמורה להיות 100h, אך כיוון שתוצאת החיבור נשמרה ב-al, רגיסטר שגודלו בית, הספרה השמאלית "גולשת" ולרגיסטר נשמר רק 00h, לכן גם במקרה זה דגל האפס מקבל ערך 1.

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

תרגיל 5.2



כתבו קוד שמודליק את דגל האפס בעזרת **פועלות חיסור**, 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 שהתוכנאה שלHon "גולשת" מתחום ערכיהם 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 מושתנים או רגיטרים בגודל של 8 ביטים, שכזוכר יכולם לשמר מספרים signed בתחום מ -128 עד +127, דגל הגלישה יידלק אם התוצאה הנשמרת באופrnd היעד חורגת מתחום (-128) עד +127. אם תוצאה הפוללה 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	1	0	0	1	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 שמוגדר 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 השני חורגת מתחום 0 עד +255. עבור אופרנד יעד בגודל 16 ביטים התוחם הוא מ-0 ועד +65,535.

לדוגמה:

```
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 מחסירים) גדול מהמחוסר (המספר ממנו מתבצע החיסור) דורש שימוש בנשא שלילי (היזכרו בהסבר על פעולות חיסור בפרק על שיטות ספירה) ולכן ידליק את דגל הנשא.

לדוגמה:



```
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" בית שערכו 256) ולכן נדלק דגל הנשא.

נסתכל על דוגמה אחרת, שאינה מדliquה את דגל הנשא, ונבין 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`, ולהدلיק את דגל הנשא. אולם, בפועל זה לא הינו עקביים - 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 שהכנסנו -128 לתוך `al`, ערכו של `al` הוא `80h`. זה גם הייצוג של $+128$, כאשר מפרשם את הערך של `al` בתור מספר `unsigned`. כעת אם נחבר $+128$ ועוד 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
 ביוור (השמאלי ביותר) של התוצאה נשמרת באופרנד היעד הוא 1. אחרת ערכו של דגל הסימן יהיה 0. דרכים פשוטות לראות אם מספר הוא שלילי, כאשר מתייחסים אליו כמספר מסומן `Signed`:

- ביצוג בינארי – הביט השמאלי הוא בעל ערך 1.
- ביצוג הקסדצימלי – ה
יבול nibble השמאלי נמצא בתחום שבין 8 ל-F. לדוגמה: פועלה שההתוצאה שלה היא 0F100h, 0A3h, 088h (משתנים או רגיסטרים בגודל בית), או פועלה שההתוצאה 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 עבר 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 שחייב לזרוץ בראץ' ולא הפרעה של מאורעות חיצוניים. לפני הרצת קטעי קוד אלו, מושנים את דגל הפסיקות ל-0, וכך מבטיחים שהקוד יזרוץ ללא הפרעה. בסיום ריצת הקוד מאפשרים חזרה את הפסיקות, על-ידי קביעת ערך דגל הפסיקות ל-1.



דגל צעד יחיד – Trace Flag

דגל צעד יחיד מכניס את המעבד לצב Trace. במצב זה, המעבד מפסיק את פעולה העיבוד לאחר כל שורת קוד ו מעביר את השליטה לתוכנה החיצונית. פוללה זו מאפשרת את פעולה של תוכנות debugger כגון turbo debugger. אם ערכו של הדגל שווה ל-0, המעבד מריז את התוכנה ללא עצירה.



דגל זוגיות – Parity Flag

זהו אינו דגל שנשאמש בו. ערכו של דגל הזוגיות נקבע לפי כמות האחדות במסונת הביטים התחתיונים של כל פעולה חישוב. אם בסופה של פעולה החישוב, במסונת הביטים התחתיונים יש מספר זוגי של '1' (0,2,4,8), דגל הזוגיות קיבל 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 שעלה ידי פעולה אחת של חיבור או חיסור, מدلיק בו זמנית את דגל הגלישה, את דגל הנשא ואת דגל הסימן.

תרגיל 5.9 (אתגר)

תנו דוגמה לקטע קוד שעלה ידי פעולה אחת של חיבור או חיסור, מدلיק 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 שישמשו אותנו 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, שהן הבסיס לכתיבת אלגוריתמים בתוכנה.

פרק 6 – הגדרת משתנים ופקודת mov

מבוא

את הפקודות הבסיסיות בשפת אסםביי אנחנו נלמד בשלושה חלקים:

- בחילק 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.

- בחילק השני נלמד פקודות אריתמטיות (פעולות חיבור), פקודות לוגיות ופקודות הזזה.

- בחילק השלישי נלמד איך יוצרים תוכנית עם תנאים לוגיים ("אם מתקיים תנאי זה, בצע פעולה זו..."), באמצעות פקודות השוואה, קפיצות ולולאות.

עם סיום הפרקים הללו נוכל לכתוב תוכניות צנויות. לדוגמה, מציאה של האיבר הגדול ביותר מתוך רשימה איברים, מיוון של איברים מהגודל הקטן, ספירת מספר איברים ברשימה וכדומה.

או קדימה, בוואו נראה איך מגדירים משתנים באסםביי.

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). הסוגרים המרובעים אוומרים לאסםבלר שצריך להתיחס לערך שנמצא בכתובת שבתוך הסוגרים. כלומר אם בכתובת `ds:1h` נמצא הערך 5, או הערך 5 יועתק לתוך al. הבעיה היא שם התוכנית שלנו מלאה בהצבות זיכרון בשיטה זו, יהיה לנו קשה לdebug אותה. נצטרך 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 נצטרך לשנות את הקוד של התוכנית.

אחד מתפקידיו החשובים של האסםבלר הוא לאפשר למתכנת להשתמש בשמותמשמעותיים בשביב מקומות בזיכרון. לדוגמה, אם הכתובת `ds:1h` תיקרא age, אז כל פעם שנפנה לכתובת `age` נגיע לכתובת `ds:1h`. כתע נתנו יהיה לכתוב את הקוד שלנו כך:

`mov al, [age]`

הasmBLDR לא רק נותן תווית שם למקומות בזיכרון, הוא גם דואג בשביבינו להקצתת המיקום ואפלו – אם נרצה בכך – לקביעת ערך התחלתי למיקום 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 בתורו 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. היזכרו 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, האsmBLDR 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 לטיעינת ערך עם אופסט נכון, אך מסגמנט אקראי כלשהו, ולכן תוקן age יועתקו ערכיו "זבל".

הקצתת מקום בזיכרון

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

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

```
ByteVarName db ?
```

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

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

זו לא תהיה שגיאה; יעתיק לתוך הרגייסטר al ערך כלשהו, שהיה בזיכרון בזמן הקריאה, אך סביר שהערך זה יהיה "זבל" – אוסף סטמי 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 בכתבota 0:ds. מיד לאחריו יוקצו יתר המשתנים. במקרה זה ByteVarName הוא בגודל בית אחד ולכן המשתנה הבא אחריו, ByteVar2, יהיה בכתבota 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)) וריצו את התוכנית. בידקו 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. לפני כן הסבירנו על חשיבות הטעינה של כתובות סמנט הנתוניות לתוך DATASEG. הכניסו את הthora mov ax, ds, ax להערכה בעזרת נקודה פסיק בתחילת השורה. כת קמפלו שנייה, היכנסו ל-(Turbo Debugger (TD)) וריצו את התוכנית. האם קיבלתם את הערך שציפיתם לו?

משתני Signed, Unsigned

שים לב שהגדירות אלו אינן קבועות אם המשתנה שהגדרנו הוא signed או unsigned – ככלומר מה טווה הערכים שניתן להכניס במשתנים. כל מה שהגדירות אלו קבועות, הוא כמה בתים Bytes בזיכרונו יוקצו לטובה המשתנים.



נראה את הטענה האחורונה שלנו בעזרת דוגמה. נגיד רשותם בגודל Byte אחד:

DATASEG

Var1 db ?

Var2 db ?

בתוך CODESEG נניס למשתנים אלו ערכיהם:

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.88.

```
[ ]=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 לפלוס 136? יכול להיות שמצאנו באג במעבד?

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

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

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

קבעת ערכים תחילה למשתנים

לא חייבים להזכיר CODESEG בשביל לטעון ערכים תחילה למשתנים. כבר בזמן ההגדרה, אנחנו יכולים לקובע למשתנים ערכים תחילה.

שימוש לב לכ"ק שניתן להגדיר כל ערך שניתן 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 הפקחה:

```
[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 לב מיוחד לכך שהגדרכנו שני משתנים שקיבלו את הערך מינוס 5. הראשון בגודל בית, השני בגודל מילה כפולה (ארבעה בתים). למרות שנראה לנו שלשניים אותם ערך – כל אחד מהם מיוצג בזיכרון בצורה אחרת.



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

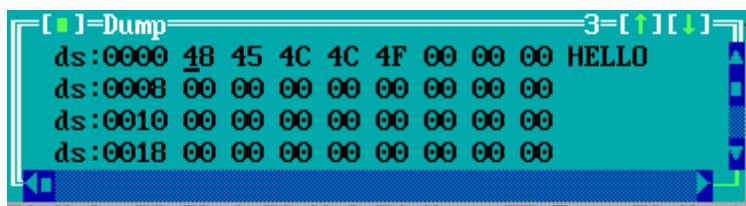
DATASEG

```
ByteVarName db 'HELLO'
```

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

DATASEG

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



יכרנו 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)**. כתוב אנו בשילם- לדין מפורט יותר על מערכים.



תרגיל 6.2: הקצת זיכרון



באמצעות התוכנית **base.asm**, הגדרו ב-SEG **DATASEG** משתנים בגודלים שונים – משתנה בגודל בית, משתנה בגודל מילה, משתנה בגודל מילה כפולה, וכן לחלקם ערכים ההתחלתיים, הוסיפו משתנה ששומר אוסף של תוכן ASCII. בסיום, עיברו על הזיכרון ב-SEG ומיצאו כל אחד מהמערכים שהגדתם.

הגדרת מערכים

מערכותם הם צורה נפוצה מאוד לשימוש מידע. מה שמייחד מערכת, נניח, שטירה של משתנים שונים בזיכרון, הוא שבמרכז כל האיברים הם בעלי אותו גודל. כל משתנה שהוא חלק ממכלול 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, ששויה לבדוק לכתובת של האלמנט הראשון במערך. אפשר לדעת מה הכתובת של כל אלמנט במערך, בעזרת כתובת הבסיס של המערך, אינדקס האלמנט וגודלו האלמנט, באמצעות חישוב פשוט:

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

לדוגמה, אם יש לנו מערך של מילים (words), והמערך מתחילה בכתובת 0200h בזיכרון האלמנט באינדקס 0 בזיכרון 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 (?)

הו ArrayName שם המערך, שניתן לקבוע לפי רצוננו.

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

N הוא כМОון כמהו האיברים בזיכרון. N חייב להיות מספר שלם וחובי.

Dup הוא קיצור של duplicate, שכפול.

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

לדוגמה:

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 *****
ds:0008 05 05 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

```

האסמבלר ישבפל לתוך המערך 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 פעים 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-**TD** מעריכים בגדים 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 פעים 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 ברperfօף, רק כדי לאפשר דיוון במספר נושאים חשובים. כעת הגיע הזמן להסביר מפורט יותר>About 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 המקור לתוך אופרנד היעד.



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

mov ax, 22

אנו יכולים להכניס את הערך 22 באמצעות שימוש בהצגה 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 תעתק **ל-ax** את הערך שנמצא **ב-bx**. שימו לב לכך ש-**bx** לא ישנה בעקבות פעולת העתקה. הערך שבתוכו פשוט **ישוכפל** **ל-ax**. באופן דומה ניתן גם לכתוב:

```
mov ax, cx
```

```
mov ax, dx
```

```
mov ax, ax
```

הפקודה الأخيرة היא חוקית, למרות שלא תנסה כלום. היא פשוט תעתק את ערכו של **ax** חוזה לתוכו **ax**.

בתתייחסויות הבאים נסקור את הצורות המותרות לשימוש בפקודת **mov**:

```
mov register, register
```

```
mov register, constant
```

```
mov register, memory
```

```
mov memory, register
```

```
mov memory, constant
```

כפי שראתם רואים, לא ניתן לבצע העתקה **memory** אל **memory**. במלילם אחרות, כל העתקה מהזיכרון אל הזיכרון צריכה לעבור דרך רגיסטר. לאחר שקראותם את הפרק על מבנה המחשב והזיכרון אתם ודאי מביןם 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
```

הורה זו מעתקה רגיסטר בן 8 ביט, 16 ביט (או יותר, במקרים מתקדמים יותר) לתוך רגיסטר אחר, שהיב 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 יעד. לדוגמה: `mov cs,ax` שגויה. ההפחה `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 לגודל הקבוע – ניסיון להעתיק ערך שנייתן לשומר רק ב-16 ביט (לדוגמה 257) לתוך רגיסטר של 8 ביט, יוביל לשגיאת קומפילציה.



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 שיטות ספירה: בייצוג העשרוני שלו, בייצוג הקסדצימלי וביצוג הבינארי (הערה: ייצוג בינארי מסוימים באות b, לדוגמה b1111b). הריצו את התוכנית ב-DS ובדקו שה-ax קיבל את הערכים הרצויים.

העתקה של רגיסטר אל תא זיכרון

`mov memory, register`

פקודה זו מעתקה את ערכו של הרגיסטר לתוך הכתובת בזיכרון שמוחזקת על ידי אופרנד היעד. אופרנד היעד יכול להיות ערך קבוע, ששווה לכתובת אליה אנחנו רוצים לפנות (Direct addressing (שיטת (Direct addressing (שיטת indexed addressing (Indexed 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
```

שcolaה לפקודות הבאות:

```
mov bx, 1
```

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

בדוגמה הרביעית, ערכו של ax יועתק לתוך הזיכרון כתובות 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) ולתוכך 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 ארבעה הבטים הראשונים ב-DATASEG מקבלים את הערך 100 (בייצוג הקסדצימלי שלו, כמובן).

העתקה של תא בזיכרון אל רגיסטר

השיטה להעתיק תא בזיכרון לתוך רגיסטר הוא לבדוק כמו השיטות להעתיק רגיסטר לתוך תא בזיכרון, בהבדל אחד – האופרנדים הפוכים:

```
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]

```

הristol את התוכנית ב-IDT ועיקבו אחרי השינויים ב-*al* בזמן ריצת התוכנית. וודאו שבכל אחד מארבעת הקטעים *al* מקבל את הערך 100 (ביצוג הhexadecimal 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 הפקודה:

`mov cx, [ax]`

תזכיר שגיאה:

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

לכן, תמיד כשרצזה לגשת אל כתובת בזיכרון, נבעוד 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 `bx`, נוסיף ל-`ax` ערך קבוע. דוגמאות:

`mov ax, [bx+2]`

`mov [bx+2], ax`

- אם נרצה להציג כתובות בהיסט משתנה מהכתובת ששמורה עלי-ידי **ax**, נוכל להוסיף לו את **si** או את **di**, 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 ל**ax** יחד את **si** או **di**, 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
```

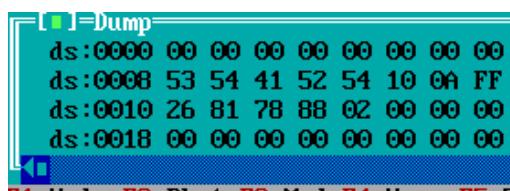
השאלה היא צו: אנו יודעים שככל כתובות בזיכרון מוצגת באמצעות 20 ביט. איך בדיק האסמלבל ממיר את הקבוע "1", את המשתנה **var** או את הרגיסטר **ax** (שכוור, מכיל 16 ביטים), לכתובת בת 20 ביט?

התשובה מתחולקת לשניים. הדבר הראשון שאסמלבל עושה, הוא לתרגם את הביטוי שבסוגרים לכתובת בת 16 ביט – זהו האופסט בתחום הסגמנט. קלומר 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 – שהSEGMENT הוא סגמנט הנתונים, .DATASEG. لكن המעבד יעתיק את **ax** לתוך הכתובת שבאופסט 1 בתחום SEG.

השאלה הבאה שעליה היא: איך המעבד מכניס את **ax**, בגודל 16 ביט, לתוך כתובות בזיכרון בגודל בית אחד? התשובה היא, שהוא יודע להשתמש גם בבית הבא בזיכרון. נסתכל איך נראה זיכרון המחשב לאחר פקודה ה-**mov**. רק בשביל להפוך את הקריאה ב-SEG לברורה יותר, נגיד בתחילת DATASEG מערך בן שמונה בתים, שמאוחתל לאפסים:

DATASEG:

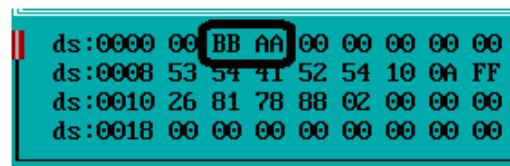
```
ZeroArray db 8 dup (0)
```



עכשו בתוך הקוד נריץ את השורות הבאות:

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

ושינויו את DATASEG:



זוכרים שאמרנו שהאSEMBLER תמיד יתייחס ל זיכרון כאילו הוא ב-DATASEG, אלא אם כתבנו אחרת?

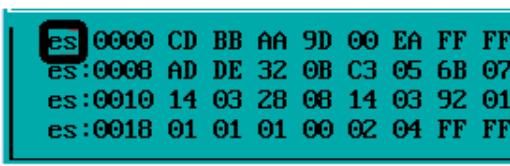
על מנת להתייחס ל זיכרון שלא כאילו הוא ב-DATASEG, נחליף את הפקודה:

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

בפקודה:

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

ולאחר ההרצה, ניתן לראות את הערך 'AABBh' בכתובת המתאימה, הפעם בתוך הסגמנט Extended Segment אשר הגדיר ES מצביע עליו:



Little Endian, Big Endian

ראינו שכשאנו מעתיקים רגיסטר ל זיכרון, הרגיסטר נראה "הפוך" – לדוגמה כשהעתקנו ל זיכרון את ax, המעבד שומר את ah בטור הכתובת הגבוהה יותר בזיכרון. למה דואק בכתובת הגבוהה ולא בכתובת הנמוכה? הסיבה היא שכ מוגדר למעבד לבצע. אפשר גם היה להגדיר הפוך ולשמור את ah בכתובת הנמוכה.

כאשר יש רגיסטר או משתנה כלשהו, בגודל של יותר מבית אחד (לדוגמה ax, bx שגודלם שני בתים וכו'), ואנו מעתיקים אותו ל זיכרון, הפעתקה ל זיכרון יכולה להיות באחת משתי שיטות:

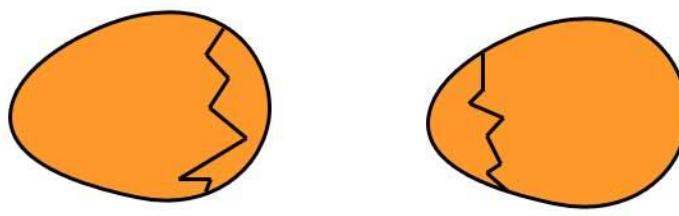
- הבית ה-High Order (כלומר bh, ah וכו') נשמר בכתובת הנמוכה יותר בזיכרון. שיטה זו נקראת **BigEndian**.

- הבית ה-Low Order (כלומר bl, al וכו') נשמר בכתובת הנמוכה יותר בזיכרון. שיטה זו נקראת **LittleEndian**.

נצין שהשיטות הללו תקפות עבור כל רגיסטרים או משתנה שגודלם יותר מבית אחד.

יש מעבדים שעובדים בשיטה כזו ויש בשיטה אחרת. מהדוגמה האחרונה אנחנו מבינים, שמעבד ה-8086 עובד בשיטה 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 לקוביות מהספר " מסעות גוליבר "

מלך ליליפוט חבק חוק שני במלחוקת לובי האופן בו יש לשבור ביצה

העתקה ממערכיים ואל מערכיים

לעתים אחד האופרנדים בפקודת mov הוא איבר במערך. זה מקרה פרטי של העתקה מזיכרון לרגיסטר או של העתקה מרגיסטר ל זיכרון. לדוגמה 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). פקודה:
ההעתקה תיכתב:

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

האSEMBLER יתרגם את הפקודה זו לכתובת בזיכרון. אם במקרה Array מוגדר ממש בתחום DATASEG, האינדקס השני של Array נמצא אופסט 0002 מתחילה DATASEG ותוכנת 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 00CCh, 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 מרגייסטר לערך, 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.

פקודה offset

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

```
mov bx, offset Array
```

ואז פקודת ההעתקה ממערך לרגייסטר מקבלת את הצורה:

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

במקרה זה אופרנד היעד al, שגודלו בית יחיד, כיוון שה-Array מוגדר כערך של בתים. אילו Array היה מוגדר כערך של מילים words, אופרנד היעד היה ax או כל רגייסטר כלשהו 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 פעולות על כל איברי המערך, אז נוכל לעשות זאת ביתר קלות כאשר ax שומר את כתובות התחלה של המערך וכל פעם מקדמים את ax כך שהוא מצביע על האיבר הבא במערך.

פקודה LEA

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

```
mov bx, offset Array
```

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

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

```
#base#10: mov bx, offset Array
    cs:0005 BB0000          mov     bx,0000
#base#11: lea bx, [Array]
    cs:0008 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]
```

חישבו 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 (שכן הריגיסטר הוא באורך 16 בית).

אי לכך, עלינו לידע את האסמלבלר שאנו מנסים לבצע העתקה של שני בתים – ובכך להמנע מהשגיאיה. שימוש לביצירת כתיבת הברה:

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

מה בעצם עשינו? הודיענו לאסמלבלר להתייחס אל המיקום בזיכרון לא בתוර byte בלבד, אלא בתוර 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 2.Array+2. התוצאה הסופית תהיה:

2
ax DDCC
bx 0000
cx 0000
dx 0000
si 0000

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

הערה: במקרים לכתובת word ptr או byte ptr אפשר לקצץ 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 יחויר לנו אזהרת type override. ניקח את צורת העברת החוקית הבאה:

mov memory, constant

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

mov [bx], 5

האסמלבר היה מחויר 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 לטובת שמירת הערך 5.

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

mov [byte ptr bx], 5

mov [word ptr bx], 5

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

נשאף להיות מפורשים על מנת למנוע משגיאות. אי לכך, בכל פעם שנחנו מעתיקם משתנה באופן שגודל העתקה נתון לפירושות, נכוון את האסמלר באמצעות הסימון .word ptr byte ptr ptr או word ptr word ptr.

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. שימוש לב שבפקודה השניה גודל המקור יותר קטן מהיעד. לכארה יש ביעד מקום להכניס את תוכן המקור, אך הוראה זו אינה מקובלת על האסמלר.

2. הכנסת קבוע לתוך רегистר סגמנט. לדוגמה:

```
mov ds, 1234h
```

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

```
mov ax, 1234h
```

```
mov ds, ax
```

3. העתקה מהזיכרון ישירות אל הזיכרון. לדוגמה:

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

אי אפשר להעתיק מהזיכרון אל הזיכרון באמצעות אופן ישיר. לכן, כדי להעתיק את var2(var1 לתוכה, נשתמש ברגיסטר עזר:

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

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

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

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

פקודה זו תגרום אף היא להודעת שגיאה של האסמלר. רק ax, bx, cx ו-dx 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 (במעבדים 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 קצר:

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

`mov [1], al`

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

היווצרו בהסביר על שיטת הסגמנט והאופסת. הפעולה הראשונה שהאסמבלי עשו היא להניח שאתם מתכוונים לפנות למקום שנמצא בסגמנט הנתוני. כיוון ש-`ds` מכיל את כתובות סגמנט הנתוני, הפעולה שהאסמבלי יבצע היא לחת את `ds`, להכפיל ב-16 (הזכירנו 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 בפניינו הזדמנויות מעניינות. מה אם נשנה ערכיהם בסגמנט הקוד, כך שהערכים החדשים יהיו זרים לפקודות בשפת מוכנה? ומה אם נשנה פקודות שהמעבד צפוי להריץ?

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

תרגיל 6.8: מרגיל אתגר - שינוי תוכנית תוך כדי ריצה



לפניכם קטע קוד. העתיקו אותו לתוך התוכנית שלכם בשלמות:

`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 מגבלות:

- יש להעתיק את קטע הקוד בשלמותו 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 הכלליםensusus المسעיף א' תקפים גם כתע.

סיכום

בפרק זה למדנו:

- להגדיר משתנים מסוגים שונים ולתת להם ערכים התחלתיים
- לעבוד עם מערכים
- לבצע העתקות מידע שונות, כוללות רגיסטרים ומקומות בזכרון, בעזרת שימוש בפקודה `mov`:

 - העתקה מרגייסטר לרגייסטר
 - העתקה של קבוע לרגייסטר
 - העתקה מהזיכרון לרגייסטר
 - העתקה רגייסטר לזיכרון
 - העתקה של קבוע לזיכרון

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

פרק 7 – פקודות אРИתמטיות, לוגיות ופקודות הזזה

מבוא

בפרק הקודם למדנו פקודה יחידה – פקודה `MOV`. בפרק זה נוסיף ליכולת התוכנות 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`, `INC`, `DEC`, `IDIV`, `DIV`, `IMUL`, `MUL`, `SUB`. הזרה הכללית שכל 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 הדיבאגר 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. הכניסו לתוך al את סכום כל האיברים במערך. הרצאו 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:

- var1 בגודל בית

- 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 שואנו מכך, 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



א. הגדרו שלושה משתנים:

- $var1$ בגודל בית

- $var2$ בגודל בית

- **diff**

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

ב. צרו שלושה מערכים בני 4 בתים. אתחלו את שני המערכים הראשונים עם ערכים כלשהם. הכניסו לתוך המערך השלישי את החיסור של שני המערכים הראשונים (לדוגמה המערך הראשון 9,8,7,6 המערך השני 6,7,8,9 (3,1,-1,-3).

פקודות INC / DEC

פקודת inc (קייזור של increase) מעלה את ערכו של אופרנד היעד ב-1. פקודת dec (קייזור של decrease) מבצעת את הפעולה ההפוכה ומורידה ב-1 את ערכו של אופרנד היעד. אפשר, כמובן, לבצע הוראות אלו באמצעות פקודות add או sub, אבל כיוון שהוספה 1 או חיסור 1 הן פעולות נפרוצות ביותר, הוקדשו להן פקודות נפרדות. הפקודות הבאות הן חוקיות:

תוצאה	דוגמאות	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

פקודת mul (קייזור 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 0.AB00h. האסמלר דואג שההתוצאה תועתק לתוך ax. ככלומר .ah=0Ah, al=0B0h.

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

להלן דוגמאות לפקודות חוקיות:

תוצאה	דוגמאות	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 למכביד אם אנחנו עוסקים במספרים **signed** או **unsigned**, כיון שהתווצה משנה. ניקח לדוגמה את המספר מינוס חמיש. מספר זה מיוצג באמצעות הרץ' 11111011 בשיטת המשלים ל-2. זה יציג זהה לייצוג של 251. אבל אם נכפול כל אחד מהמספרים בשתיים, לדוגמה, התוצאות חייבות להיות שונות.

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

הקוד הבא טוען לתוך **al** את 11111011 (ניתן לפיריש כ 5- או +251), לתוך **bl** את 00000010. הקוד מבצע בינייהם שתי הפעולות. פעם אחת **unsigned** על-ידי הפקודה **mul** ופעם נוספת **signed** על-ידי הפקודה **imul**. הפקודה **imul** (כזכור שבין הפעולות מאפסים את **ax**). שימוש לב לערך של **ax** לאחר כל אחת מהפעולות:

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	

הרגיסטרים **ax**, **bx**, **al** לפני פעולות ההפול

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	

בהכפלת על-ידי פקודה *mul*, האסמבילר מתייחס לערכו של *ai* בתווך 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	

בהכפלת על-ידי פקודה *imul*, האסמבילר מתייחס לערכו של *ai* בתווך 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 בזזה 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 בזזה 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 ההתחלתיים. בצעו כפל של המערכים והכניסו את התוצאה לתוך sum. לטובת הפשטות, הניחו שההתוצאה נכנסת לתוך word. הדרשה: אם שמות המערכים הם a ו-b, 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 dx:ax, 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 (המחליק) 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 מספיקה טובה. רק צריך להזכיר לשם 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 שנגរום 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)

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

פקודה NEG

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

`neg dest`

tab'utz 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 השימוש. פקודות לוגיות הן שימושיות מאוד כשאנו רוצים לשנות את ערכו של בית, או מספר ביטים, ובאותו זמן להשאיר ערכים של ביטים אחרים בלי שינוי. פעולה זו נקראת **MASKING**, ונראה בהמשך הפרק.

למה בכלל נרצה להתעסק עם ביטים בודדים? למשל הצפנות, לדוגמה. כמשמעותם מידע, המידע שמיועד להצפנה נשמר בצורה "דחוסה" – **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 שנitin להציג כשבודדים בצורה לא דחוסה בה מוקצת בית נפרד לשימורת 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. בשביל לביצוע פעולה כזו, צריך לדעת "**לשחק**" עם ביטים בודדים. כתה נראה איך עושים את זה.

קיימות ארבע פקודות לוגיות – **not**, **or**, **xor**, **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 מbyn 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 האסמלוי **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

נדגמים 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 **the-and** שלהם תהיה 0 בכל מקרה. אי לכך, אנו מתעלמים למעשה מהערך של שבעת הביטים האלה. כתוצאה לכך, תוצאה **the-and** של הבית השמיני תלוי רק במספר שתואת הזוגיות שלו אנחנו בודקים – אם הבית השמיני שלו הוא 1, **the-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, כשהנלמד על פסיקות, נלמד על רכיב כזה). 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 רוצחים לאפשר תקשורת עם הtekן מס' 4 (הtekנים ממושפרים מ-0 עד 7). נקרא את זיכרונו הרכיב. נניח שקיבלנו:

1100 0100

בית מס' 4, אשר מייצג את הtekן מס' 4, כבוי ומוסמן בצהוב. כתע, ברצוננו להדליך את הבית זהה.

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 והתוצאה 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באים הצפנה. הסיבה היא התוכנה הבאה שלה:

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

התוכנה המינימלית זו תשרת אותנו ביצירת הצפנה. נניח שאנו רוצים להצפין את המידע הבא:



1001 0011

נגדיר מפתח הצפנה, שידוע רק לנו ולמי 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 מישחו 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.

אבל, ה-*opcode* של xor קוצר יותר משל mov, וכך מעתה ואילך כשרצחה לאפס רגיסטר נעשה זאת על-ידי xor של ערך הרגיסטר עם עצמו. מכאן שבמוקום לבצע:

```
mov ax, 0
```

ונכתוב:

```
xor ax, ax
```

שתי הפעולות גורמות לכך שב-ax יהיה הערך 0, אך הפעולה השנייה לוקחת פחות מקום מהראשונה, וכך מועד 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 תוכי ASCII LIKE ASSEMBLY\$'. לדוגמה \$'ABCDEF'. הגדרו מפתח הצפנה בן 8 ביטים, לבחירתכם. הצפינו את המידע באמצעות מפתח ההצפנה (תו ה-\$ ישיע לכם לדעת שהאגותם לסוף המערך אותו יש להציגן, וכן להדפסה).lico ל-DATASEG במקומם שהגדרתם את המערך והסתכלו איך המחשב מתרגם את המערך לתוכי ASCII לפניה ואחריו פועלות ההצפנה שלכם. פענוו את המידע שהצפנתם באמצעות מפתח ההצפנה. חיזרו על הבדיקה ב-DATASEG ועודאו שקיבלתם זהה 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 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 בעבודה עם ביטים בודדים.

פקודות הזהה

פקודות הזהה מקובלות אופרנד ו"מצוות" 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 השימושות 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 הרגיסטר cl. הוצאות הבאות תקינות:

shl register, const

shl register, cl

shl memory, const

shl memory, cl

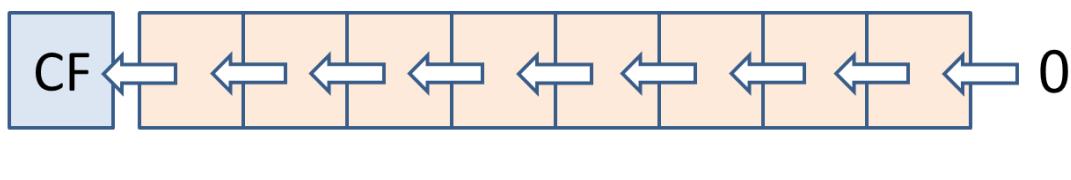
פעולות ה-shl משפיעה באופן הבא:

- על כל הזהה, הביטים שבאופרנד היחיד זרים מוקם 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 שייצא מצד שמאל נכנס אל דגל הנשא CF.
- דגל הגלישה OF קיבל ערך 1 במקרה שהבית הcy שמאליה השתנה. כלל זה נכון רק עבור הזהה של בית יחיד.
- דגל האפס ZF קיבל ערך 1 אם לאחר ההזהות ערכו של האופרנד הוא אפס.

- דגל הסימן יהיה שווה לערך של הביט הcy שמאל.

- 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:



פקודה SHR

פקודת הזזה ימינה, `shr`, כMOVן דומה בכתיבתה שלה ובפעולתה שלה ל-`shl`. הפוקודה מקבלת אופרנד יעד וכמות ביטים להזזה. האופרנד יכול להיות משתנה בזיכרון או רегистר, וכמות הביטים להזזה יכולה להיות מספר קבוע או הרגיסטר `cl`. הזרות הבאות תקיןות:

`shr register, const`

`shr register, cl`

`shr memory, const`

`shr memory, cl`

פעולת ה-`shr` משפיעה באופן הבא:

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

- על כל הזזה נכנס 0 אל הביט השמאלי ביותר (אם ההזזה היא של בית אחד, נכנס 0 אחד. אם ההזזה היא של חבייטים, נכנסים ח אפסים).

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

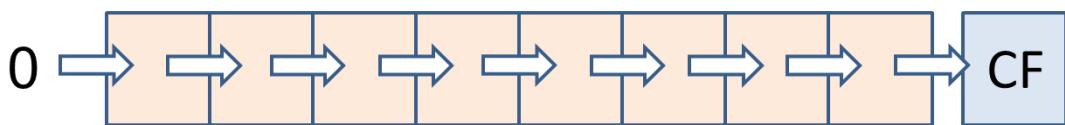
- דגל הגלישה OF יקבל את ערך הביט הcy השמאלי של האופרנד לפני ההזזה. כלל זה נכון רק עבור הזזה של בית יחיד.

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

- דגל הסימן יהיה שווה לערך של הביט הcy שמאל, 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 *shrl***שימושים של פקודות הזזה**

פקודות הזזה שיטות מיוחדות לביצוע פעולות:

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

כפל וחילוק: בסיסי עשרוני, כשורצים לכפול בעשר, כל מה שציריך לעשות זה להוסיף אפס בתור הספרה הциימנית (כלומר, להזין את הספרות ספרה אחת שמאללה). בסיס שתים, אם נוסיף אפס בתור הספרה הциימנית, נקבל כפל בשתיים. ככלומר כל הזזה שמאללה בבית אחד כופלת את הערך בשתיים. מיוון שערכיהם שמורים במשתנים עם גודל מוגבל, באיזשהו שלב נהרגו מהגודל של המשתנה ואז התמונה זו תאביד, אבל אם נעבוד נכון ונdagג לגדים נכונים של המשתנים, נוכל להשתמש בהזזה כדי לבצע כפל בשתיים, וכן הלאה – כפל בארבע באמצעות הזות שני ביטים, במשמעותו וכוכ. כמו שהזזה שמאללה היא כפל בשתיים או בחזוקות של שתים, הזזה ימינה היא חילוק בשתיים או בחזוקות של שתים. גם כאן צריך לשים לב שאם הביט הциימני שלנו הוא 1, ככלומר המספר אי זוגי, אז הזזה ימינה לא בדיקת חלק 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 לצד אחד שלו ומוציאים בית מצידו השני. בין הביטים במערך מבוצעות פעולות XOR שונות, ועם התוצאה עושים XOR לביט שהוא רוצה להצפן. הם 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 להתחiliar מאלגוריתם למפל זיו:

<http://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%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-`ai` את הערך 3. בעזרת פקודות הזזה, כיפלו את `ai` ב-4.
- ב. הכניסו ל-`ai` את הערך 120 (דצימלי). בעזרת פקודות הזזה, חלקו את `ai` ב-8.
- ג. הכניסו ל-`ai` את הערך 10 (דצימלי). בעזרת פקודות הזזה וחיבור, כיפלו את `ai` 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 שתים:

 - 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 למדוד כמה נושאים:

- איך קופצים למקום כלשהו בתוכנית (פקודה קופיצה 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, מודיע אל שורת הקוד הבאה בתחום CODESEG. תפקידן של הוראות הבקרה הוא לאפשר קופיצה קדימה או אחורה אל שורות קוד בתוכנית. קופיצה קדימה, למשל דילוג על שורות בתוכנית, הכרחית כדי לאפשר לנו להריץ קוד כלשהו רק אם מתקיים תנאי מתאים - "אם ערכו של המשתנה שווה ל-1, המשך ביצוע התוכנית. אחרת – דלג על המשך שורות קוד והמשך ממשם". קופיצה אחורה הכרחית כדי לאפשר לנו לחזור על הריצה 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 יעיל במצבים שונים. למען הפשטות, נתחיל בהוראות בקרה ללא תנאי קיום.

פקודת JMP

פקודה jmp שולחת את המעבד, ללא תנאי, לנקודת אחרת בתוכנית. פקודה jmp מקבלת כתובות בזיכרון, בתוך CODESEG. לאחר ביצוע פקודת jmp, תועתק אותה הכתובת המובאת לה בתור אופרנד לתוך רегистר ה-IP והריצה תמשיך מכתובת זו.

לדוגמה:

DATASEG

```
address      dw      000Ah
```

CODESEG

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

```
mov ax, 1
jmp [address]
```

כך נראהים הרגיסטרים לפני ביצוע פקודת 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 פקודת jmp, שימושו לבערך 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

וחתכוית המשיך את הריצה מהבית העשيري ב-CODESEG.

קפיות FAR ו-NEAR

בדוגמה שהשתמשנו בה כדי להסביר את פקודת jmp, מסרנו לאסמלר רק את האופטט שהוא צריך לкопיאז אליו – 000Ah – בעזרת שימושו במשתנה address. בזמן שהחכנית הגיעה לשורת ה-jmp שלנו, היא רצתה מתוך סגמנט הקוד ופקודת jmp שללה אותה לאחר באותו סגמנט. מצב זה, של קפיצה בתחום אותו סגמנט (במקרה זהה מסגמנט הקוד אל חלק אחר בסגמנט הקוד), נקרא קפיצה far. כדי לבצע קפיות מסווג near, מספיק שניתן לפקודת jmp את האופטט – הסגמנט הרוי לא משתנה.

קפיות 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 לאיזה סגמנט קוד אנחנו רוצים לкопיאז. 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 שנכתוב 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 (העתיקו אותה לתוך ה-`CODESEG` שבচচনিত `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, שככל 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 פקודה ה-`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 אינסוף פעמים, ובכל פעם יקדם את הערך הנמצא ברגיסטר `ax` באחד.

שימוש לב לאינדנטציה – הרוחחים בתחילת כל שורה. כל ה-`label`ים תמיד נמצאים בעמודה השמאלית ביותר של הקוד, ופקודות נמצאות טاب אחד ימינה.



אנחנו יכולים לקובע 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. האסמבולר כנובן לא יעיר עליהם, היות שהם תקניים מבחינה טכנית, אבל הבעייה תהיה שלנו (ושל מי שינסה להבין את הקוד שכתבנו). אבל יש סוג נוסף של שמות ל-label, שהאסמבולר יתריע עליהם כי ישנה שגיאה. אסור לקרוא ל-label בשם של רגייסטר (למשל: ax) או בשם של פקודה אסmbolic (למשל: MOV), ואין להשתמש בשמות מהשומות המופיעים להלן – אלו שמות שמורים.

נסים בטעות נפוצה של מתחילה: אין להת לשתי שורות קוד שונות את אותו שם label. האסמבולר פשוט לא יבין لأنם רוצים לкопין.



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	%MAC\$	MASM	MASM51
MODEL	MULTERRS	%NEWPAGE	%NOCOND\$	%NOCREF	%NOCTLS
NOEMUL	%NOINCL	NOJUMPS	%NOLIST	NOLOCALS	%NOMACS
NOMASMS51	NOMULTERRS	%NOSYMS	%NOTRUNC	NOWARN	P186
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 jmp, כך שפקודת jmp תהיה אל label. תנו ל-label שם משמעותי.

CMP – פקודה השוואתית

פקודת ההשוואה cmp (קיצור של המילה compare) משמשת להשוואה בין שני אופרנדים. המעבד לא "יודע" אם האופרנדים האלו שווים, או שהאחד גדול מהשני. הדרך בה המעבד מגלה את היחס בין האופרנדים, היא חיסור האופרנדים זה מזה. אם נדלק דגל האפס – האופרנדים שוים. הדלקה של דגלים אחרים (ג'ילשה, נשא, סימן) מאפשרת למעבד לבדוק אם אחד האופרנדים גדול יותר. במקרה דומה לפקודת cmp לפקודת sub, בהבדל אחד – היא לא משנה את ערכם של האופרנדים.

צורות כתיבה חוקיות של פקודת cmp:

תיאור	דוגמה	פקודה
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, שמורוצות אחת אחרי השנייה, 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 השנייה, אנו משווים את al ל-3. כפי שראינו, cmp מדמה חיסור של שני האופרנדים. תוצאה החיסור היא אפס (היות שהערך של al היה 3) ולכן נדלק דגל האפס. דגלי הנשא והסימן בהכרח יהיו 0 לאחר פקודה זו.

- בשורה השלישית אנו משווים את `a` ל-`2`. התוצאה של `a` פחות `2` היא חיובית ולכן דגל האפס כביה.
- בשורה הרביעית אנו משווים את `a` ל-`5`. תוצאה פולת החיסור `a` פחות `5` היא בעלת בית עליון שערכו '1' ולכן דגל הסימן, כמו כן כיון שהמחסר גדול מהמחסර יש צורך בנשא שלילי לחישוב התוצאה ולכן 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 פולת ה-`cmp` תהיה קביעת הדגלים לפי היחס בין האופרנדים, כפי שראינו קודם.
- פקודת הקפיצה בודקת אם דגל כלשהו, או כמה דגליים, מקיימים תנאי מוגדר. לדוגמה, האם דגל האפס שווה ל-`1`.
- אם התנאי מתקיים, הקוד קופץ לכתובת שהוגדרה על-ידי המשתמש. בדרך כלל כתובות זו תזוזין 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`, אינו 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 ה-`cmp`. הדבר הראשון שצורך לשים אליו לב, הוא ש אין דין השוואת של אופרנדים `signed` כדי השוואת של אופרנדים `unsigned`.

להלן שאלה למחשבה:

איזה מספר יותר גדול – `b` או `1b` או `10000001b`?



התשובה היא – כמו שבטח ניחתם בשלב זה – שתליי איך בודקים. אם מתיחסים אל שני המספרים בתור `unsigned`, הרי `sh-1b` (שווה `129` בבסיס עשרוני) גדול מ-`1b`. לעומת זאת, בהשואת מספרים `signed`, מיתרגם למינוס `127`, ולכן קטן מ-`1b`.

ברור, אם כך, ש כדי לבצע קפיצות מותנות בתוצאות של cmp, אנחנו חייבים להודיעו לאסמבולר איזה סוג השוואה ביצענו, או במילים אחרות – אילו דגלים לבדוק. לכן התפתחו שני סטים 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 כבר הוכרנו, יכולות להתווסף האותיות N בשביל Not ו-E בשביל Equal.

רכיב האפשרויות נמצא בטבלה הבאה. שימושם לב להגדרת 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 פקודה קפיצה מבוססת על בדיקה ותנאים לוגיים של דגל אחד או יותר. רק לטובת הבנה כללית, נסתכל לדוגמה על פקודה 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 שייבדקו הם אם CF שווה לאפס ואם ZF שווה לאפס. התוצאה הבדיקה תהיה חיובית, ולכן תבוצע קפיצה. פעולה JG, שמיועדת למספרים **signed**, צריכה להחזיר תשובה שלילית (מינוס 127 אינו גול מ-1). התנאים שייבדקו 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.

- א. כתבו תוכנית שבודקת אם ax גדול מ-ax (יש להתייחס לערך של ax ביצוג שלו כ-signed כ-unsigned כמוון), ואם כן מוריידה את ערכו באחד.
- ב. כתבו תוכנית שבודקת אם $ax=bx$, ואם לא – מעתקה את ax לתוכן bx.
- ג. כתבו תוכנית שבודקת אם המשתנה Var1 גדול מ-Var2 (שםו לב – יש להתייחס אל הערכים במשתנים כ-unsigned). אם כן, ax=1, אחרת ax=0.
- ד. 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 $var1-var2$.
- ה. קטע הקוד הבא מדפיס למסך את התו 'x':

```
mov dl, 'x'
mov ah, 2h
int 21h
```

כיתבו תכנית, שמוגדר בה משתנה בגודל בית בשם TimesToPrintX. תנו לו ערך התחלתי כלשהו (חיובי). הדפיסו למסך כמהות א'ים כערך של TimesToPrintX. הדרכה: החזיקו ברגיסטר כלשהו את כמהות הפעמים ש-'א' כבר הודפס למסך. צרו label שמדפיס א' למסך ומקדם את הרגיסטר ב-1. לאחר מכן בצעו השוואה בין הרגיסטר ל-TimesToPrintX, ואם לא מתקיים שוויון – קפצו 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 ש-'א' כבר הודפס למסך, להשוו את כמהות הפעמים זהו ל-TimesToPrintX, ואם לא מתקיים שוויון – לקפוץ אחריה למקום בו מודפס עוד 'א' למסך. פעולה זו – של ביצוע פעולה מוגדרת מספר פעמים – נקראת **לולאה (Loop)**. כיוון שפעולה זו נפוצה למדי, ישנה פקודה מיוחדת שעשויה בשילינו 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 זו:

```
loop SomeLabel
```

זהה הקוד הבא:

```
dec cx
cmp cx, 0
jne SomeLabel
```

דוגמה לשימוש בפקודת loop לביצוע התכנית שמדפסה '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

נקזה למחשבה: העתיקו את הקוד של מעלה והריצו אותו, עבר ערך היובי כלשהו של TimesToPrintX. האם

אתם מוזים מקרה כלשהו, שבו התכנית לא תבצע בדיקת מה שרצינו?



פתרון:

אתחלו את TimesToPrintX לאפס.

כמו אמרנו, הפקודה loop קודם כל מפתחה את ערכו של CX ואז 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 וرك 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 (הרתבה)



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. בΈך הכל, LoopB צריך להיות מבוצע 50 פעמים. מה לא תקין בקוד זהה?

הבעיה היא ששתי הוראות 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 LoopA לרוץ 65,536 פעמים. אך בכל ריצה כזו cx מתאפס... אי לך התכנית לעולם לא תיגמר.

כדי לפתור את הבעיה, אין ברירה אלא להשתמש ברגיסטר אחר לטובת שמירת כמות הפעמים שאחת הלולאות רצה (או לשמר את cx במחסנית- נושא שנלמד בהמשך). דוגמה לאותו הקוד, רק תיקון:

```
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 ושמירתם במערך 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 קולטות تو מהמשתמש:

```
mov    ah, 1h
int    21h          ; al stores now the ASCII code of the digit
```

- ג. צרו תוכנית שקולטת 5 תווים מהמשתמש ושומרת אותם לתוך מערך. בזדקו את התוכנית על-ידי הכנסת התווים O HELLO ובדיקה שהמערך מכיל תווים אלו.

- ד. צרו תוכנית שמחשבת את המכפלה $Var1 * Var2$, משתנים בגודל בית שיש להתייחס אליהם כ-`unsigned`. אך מבצעת זאת על-ידי פעולות חיבור. הדרכה: יש לבצע פעולה חיבור של `sum=Var1+sum` ולהזור עליה על-ידי `Var2 loop` פעמים.

- ה. 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, ממירות 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 המר'א':

```
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 כדוגמת jb, jg, ja וכו'

- פקודה loop

במקרים מסוימים, שימוש בפקודת קפיצה מותנית או בפקודת loop עלול לגרום לשגיאה הבאה בזמן ביצוע האסמלר: **Error- Relative jump out of range**.

האסמלר מתרגם כל פקודה ל-**opcode**. בעוד שלפקודת jmp האסמלר מקצה 16 ביט לייצוג טווח הקפיצה (כלומר טווח של מינוס 32768 עד פלוס 32767 בתים מהמקום הנוכחי), האסמלר מקטה 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 פקודת relative jump out of range לתוכית שנמצאת למרחק גדול מהטוויה שניתן נכתב ב-8 ביט, נקבל שגיאת

ישנן מספר דרכי להתחמזר עם בעיה זו-Colon סוגים של מעקים:

1. ההתחמזרות הכי פשוטה היא לא לייצר קפיצות מותניות שהורוגות מהטוויה וכך להימנע מהבעיה.

2. שיטה פשוטה אך לא אלגנטית- לקפוץ למקום קרוב וממנו מיד לקפוץ למיקום הבא, עד שmagicsים 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 לשגיאת:

cmp ax, bx

ja my_label

.... ; more than 127 bytes in code memory

my_label:

.... ; some code here

cut נבצע הمرة 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 מגננון שמחליף את הקפיצה המותנית בקפיצה בלתי מותנית. בואו נראה כיצד מתנהג הקוד החדש שכחבנו. בקוד המקורי התרחשה קפיצה ל-label `my_label` אם התקיים התנאי `ax>bx` (הפקודה `ja` משמעה "קפוץ אם גדול ממש", והיא הפוקודה ההפוכה ל-`jbe`, שמשמעותה "קפוץ אם קטן או שווה"). בקוד החדש, כאשר `bx>ax` התנאי `jbe` אינו מתקיים, כתוצאה לכך התוכנית מגיעה לשורת הקוד `jmp my_label` והקפיצה מתיקית.

סיכום

בפרק זה:

- למדנו על פקודת `jmp`, המשמשת לביצוע קפיצות. למדנו להיעזר ב-labels כדי לסמן מקומות בקוד שנרצה לקפוץ אליהם.
- למדנו על פקודת `cmp`, שמשמשת להשוואה בין שני ערכים ומשנה את מצב הדגלים בהתאם.
- למדנו סוגי שונים של קפיצות מותניות ("קפוץ אם ערך גדול מ...", "קפוץ אם ערך קטן או שווה...", "קפוץ אם ערך שווה אפס...").
- עמדנו על ההבדלים בין השוואת מספרים `signed` לבין השוואת מספרים `unsigned` וראינו שישנן פקודות שונות לאליה ולאלה.
- ראיינו איך מגדירים לולאות `loop` שימוש בפקודת `loop` וברגיסטר `cx` (או `cl`).

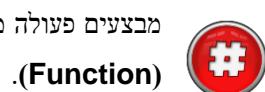
בסוף הפרק הגענו לרמה שמאפשרת לנו לתוכנת תוכניות שכוללות אלגוריתמים, תנאים לוגיים וחזרות על קטעי קוד. בשלב זה הקוד שאנו יוצרים מתאים לתוכניות קצירות, ועדין לא לביצוע תוכניות ארכוכות ומורכבות. חסירה לנו עדין היכולת לכתוב קטעי קוד שמקבלים פרמטרים ומבצעים פעולות שונות – פרוצדורות. על כך נלמד בפרק הבא.

פרק 9 – מחסנית ופראוצדורות

מבוא

המטרה של הפרק זהה היא ללמד איך לכתוב תוכניות מודולריות. ראשית, כדי שנבין – מה זאת אומרת לכתוב תוכנית מודולרית? ומדוע זה חשוב?

תוכנית מודולרית היא תוכנית שבנויות מודולרים – חלקים של קוד, שיש להם נקודת כניסה אחת ונקודת יציאה אחת, והם מבצעים פעולה מוגדרת. מודול כזה נקרא **פראוצדורה (Procedure)** (בשם העברי "תת תוכנית") או **פונקציה**



בשפה אסמלטי אין הבדל מעשי בין פראוצדורה לפונקציה ולכל נתיחה יכול בטור פראוצדורות. כאשרנו כתובים קוד מודולרי – קוד שבוני מפראוצדורות עם קוד שמקשר ביניהם – אנחנו מתכוונים מראש איך להלך את מה שהתוכנית שלנו צריכה לעשות לכמה פראוצדורות, כאשר לכל פראוצדורה יש תפקיד מוגדר. לעומת זאת בשיטה זו יש כמה יתרונות:

יתרון ראשון – הקוד שלנו קצר יותר. נניח שהתקנית 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 לשומר 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 הקריאה. כך חסכנו את כתיבת חוזרת של קוד קריית התווים.

רגע, למה שלא פשוט נגיד `label`, נקרא לו, לדוגמה – `ReadPassword`, ובכל פעם שנצטרך לקלוט סיסמה נעשה אליו `jmp`? אנחנו בהחלט יכולים לעשות את זה, וזה גם יעבוד, אבל لأن ממשיך אחרי קריית הסיסמה? אם לא ניתן לתוכנית הוראה אחרת, היא פשוט תמשיך אל שורת הקוד הבאה. אולי נפתר או זה באמצעות פקודת `jmp` בסוף קריית הסיסמה? זה יעבוד. 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 למקום אחר בתוכנית. נגיד שאנחנו צריכים לכתוב קוד שקולט סיסמה לגישה למחשב, ולאחר כך סיסמה לחשבון המיל. אחרי קריית הסיסמה, איך נדע לאן 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 ולבזוק אותו עליידי פקודת cmp – אבל כל הבדיקות האלה מתחילה להיות יותר ארכוכות מאשר לכתוב את הקוד של קליטת הסיסמה בכל המיקומות שאנו צריכים אותו... פְּרֹזְצָדָרוֹת מהוות פיתרון אלגנטי ויעיל לסוגיה זו.

יתרונו שני – אנחנו יכולים לבדוק חלקיים מהקוד שלנו ולדעת שהם עובדים בצורה טובה. נגיד שהגדכנו פְּרֹזְצָדָרוֹת 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 בפְּרֹזְצָדָרוֹת ReadPassword, מספיק שנתקן אותו פעם אחד – והוא יתוקן עבור כל הקריאות בתוכנית. העבודה שיש לנו נוספת של קטעי קוד בדוקים מקטינה משמעותית את המאמץ שלנו לגלות באגים.

יתרונו שלישי – כל יותר לקרוא את הקוד שלנו. במקום לכתוב הרבה קוד, אנחנו פשוט כותבים פקודה שנקראת `call` ואת שם הפְּרֹזְצָדָרוֹת. כך, לדוגמה:

`call ReadPassword`

יתרונו רביעי – יכולת לשף קוד גם בין תוכניות וגם בין אנשים. נניח שכתבנו קטע קוד שמבצע משהו שימושי ונפוץ, קליטת סיסמה מהמשתמש. אם נשים את הקוד בתוך פְּרֹזְצָדָרוֹת ואת הפְּרֹזְצָדָרוֹת נשמר בקובץ נפרד, יוכל להשתמש בה בכל תוכנית שנכתוב וגם לשף בה אחרים. העיקרונו 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 פְּרֹזְצָדָרוֹת.

חיסרונו ראשון – היתרונו של שיתוף קוד הוא פתח לביאות. אם יש באג בקוד שמייחסו כתוב והקוד שותף עם הרבה אנשים, שהשתמשו בקוד בתוכנות שלהם, הבאג נפוץ בכל התוכנות הללו וכתוצאה מכך קשה לתקן אותו. הקושי בתיקון הבאג הוא חריף במיוחד אם הוא נמצא בתוכנה שמותקנת אצל לקוחות גודלה של משתמשים, שנדרשים לעדכן את התוכנה שברשותם.

חיסרונו שני – כתיבת פְּרֹזְצָדָרוֹת דורשת השקעה מסוימת מצד המתכנת. יותר קל פשוט לכתוב בתוכנית הראשית את הקוד מאשר להגדיר אותו בתוך פְּרֹזְצָדָרוֹת, במיוחד בתוך מתכנתים מתחילה.

חיסרון שלישי – הקריאה לפרויקורות והזרה מהן דורשת **משאבים** של המחשב. לדוגמה, ביצוע פעולות קפיצה והזרה. לדוגמה, שימוש בזיכרון. מכאן שקריאה לקוד בתוך פרוצדורה היא "יראה" יותר עבר המחשב מאשר הרצה של הקוד שלא מתוך פרוצדורה.

הרף החסרונות שסקרנו, התוצאה שבכתיבת קוד מודולרי, שבסיס על פרוצדורות, עולה על החסרונות שלו. לכן נקבע לכתחום את הקוד שלנו בזורה זו.

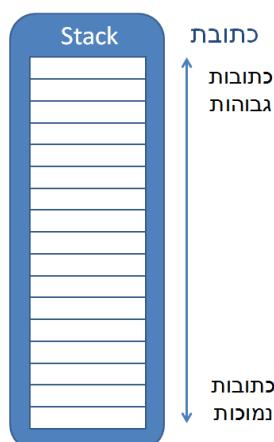
המחסנית STACK

המחסנית (**STACK**), היא סגמנט בזיכרון. המחסנית משמשת לאחסון בזמן קצר של משתנים, קבועים וכותבות

ולכן היא כלי חשוב בעבודה עם פרוצדורות ויצירת תוכניות מודולריות.



כדי שנוכל להבין באופן מלא איך לכתוב פרוצדורות, נצרך קודם כל להכיר היטב את המחסנית. לכן, נפתח בהסבר על המחסנית, הגדרה של מהסנית ופקודות הקשורות לשימוש במחסנית.



תיאור סכמטי של *Stack Segment*

הגדרת מהסנית

כמו כל סגמנט בזיכרון, המחסנית היא אזור בזיכרון שמהחיל בכתובת כלשהו וטופס 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 שמצוין גם בקובץ *base.asm*):

STACK 100h

لامחסנית יש שני רגיסטרים שקשורים אליה. עשינו אותם הিירות קטרה בעבר, כשסקרנו את כלל הרגיסטרים שיש למעבד:

- **stack segment – ss**

- מצביע stack pointer – **sp**

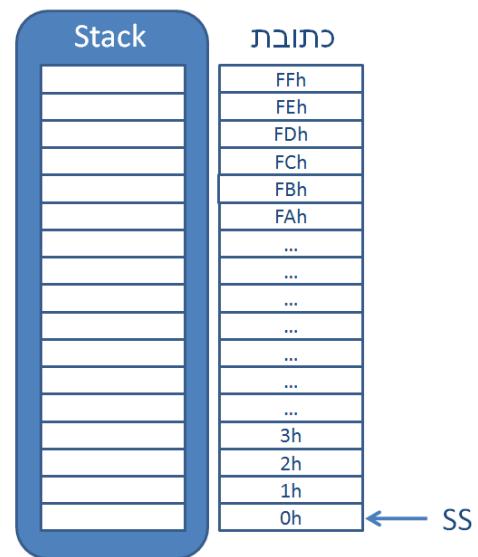
מצביע המהסנית, **sp**, הוא רגיסטר שמחזיק אופסט בזיכרון. בכך אין הוא שונה מהרגיסטר **ax**. כמו שאפשר להגיע לכל כתובות ב-SEG DATASEG בעזרת ה指挥符 **ax:bx**, כך אפשר להגיע לכל כתובות ב-STACK בעזרת הוספת האופסט הנתנו ב-**sp** לכתובת הסegment הנתונה ב-**ss**.

בתחילת התוכנית, הרגיסטר **sp** שווה לגודל המהסנית. בדוגמה שנשתמש בה, מהסנית בגודל **100h**, לפני שהכנסנו ערך **100h** ל**sp** מוחסנית מאותחל להיות שווה **100h**.

שימוש לב שבמהסנית בגודל **100h**, לעומת 256 בתים, מרחב הכתובות אינו מגיע עד **100h** אלא הוא בין 0 ל-**FFh**. לעומת, הערך ההתחלתי של **sp**, **100h**, מצביע על כתובות שהיא בדיקת אחד מעלה קצה המהסנית.



מהסנית בגודל **100h**. הרגיסטר **ss** מצביע על תחילת סegment המהסנית. בתחילת התוכנית הרגיסטר **sp** מאותחל לערך **100h**, לעומת לערך שנמצא מחוץ לmahsנית.



העובדת ש-**sp** מאותחל למצביע לא על תחילת המהסנית אלא על הקצה שלה נראה כרגע קצר מוזרה, אבל היא קשורה בדרך שבה המהסנית מנוהלת. המהסנית מנוהלת בשיטת LIFO – Last In First Out, לעומת הערך שנכנס אחרון הוא הראשון לצאת מהmhסנית. לפני הכנסה של נתונים לmahsנית, ערכו של **sp** יורד ולאחר הוצאה של נתונים מהmhסנית ערכו של **sp** עולה. נבין זאת כאשר נלמד על הפקודות המשמשות אותנו בעת הכנסה והויצאה של נתונים מהmhסנית, ונראה דוגמאות.

אילו פקודות מאפשרות הכנסה והויצאה של נתונים מהmhסנית?

PUSH

פקודת `push` גורמת להכנסה של ערך למחסנית. הפקודה נכתבת כך:

`push operand`

מה יבוצע?

- ערכו של `sp` יורד בשתיים: $sp=sp-2$.

- ערכו של האופרנד יועתק למחסנית, לכתחובה `ss:sp`.

שיםו לב: ערכו של `sp` תמיד יורד בשתיים עם פקודת `push`, כלומר הוא מצביע על כתובת שרחוקה שני בתים מהכתובת[האחרונה](#) עליה הצביע. המשמעות היא שאפשר לדוחף למחסנית רק משתנים רק מוגדים של שני בתים – `word`. כל ניסיון לבצע `push` לכמות אחרת של בתים – יוביל לשגיאה.

דוגמאות לשימוש בפקודת `push`:

`push ax`

`push 10`

`pash 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 מוגדר `word`.

אם הפקודה הבאה היא חוקית?



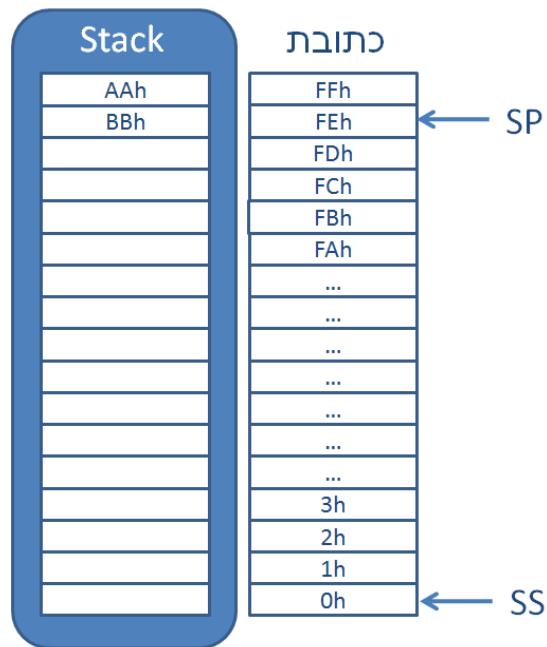
`push al`

תשובה: פקודת זו היא שגויה. `al` הוא לא בגודל מילה ופקודת `push` מקבלת רק רגיסטרים 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

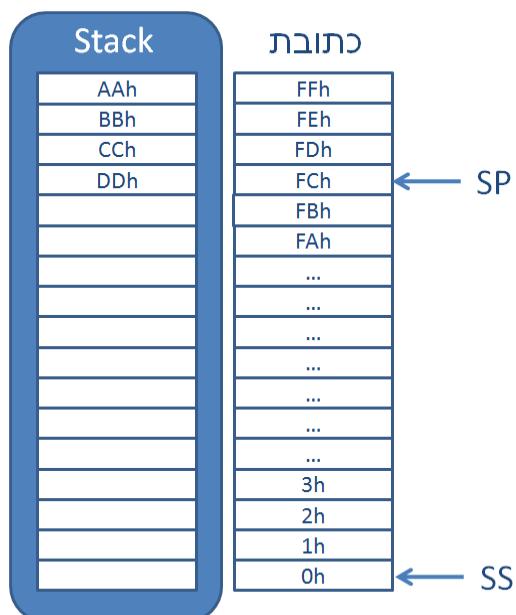
והוא מצביע על כתובת נמוכה יותר במחסנית

שימו לב לכך שהביטים הגבוהים של `ax`, אלו שהמורים ב-`ah`, 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]

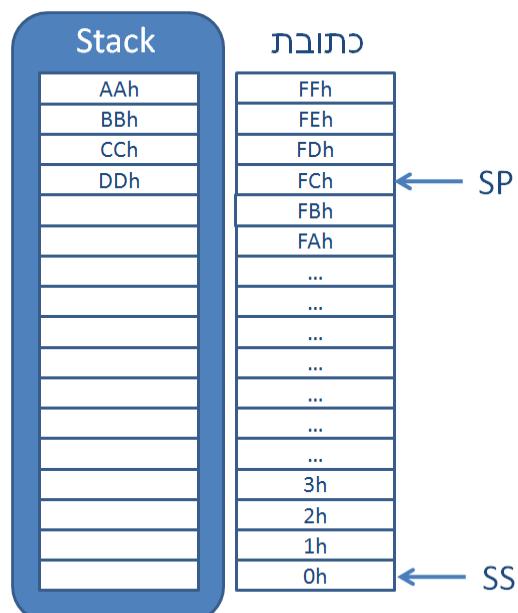
הפקודה הראשונה תעתק לתוכן **ax** את המילה שבראש המחסנית.

הפקודה השנייה תעתק לתוכן המשתנה **var** את המילה שבראש המחסנית (כמוון 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:





שאלה: מה יהיה ערכו של sp לאחר ביצוע הפקודה הבאה? ומה יהיה ערכו של bx?

pop bx

תשובה: ערכו של bx יהיה 0x0CCDDh, ערכו של sp יעלה בשתיים ויהיה שווה 0xFEh.



שאלה: לאחר ביצוע הפקודה הקודמת, בוצעה פקודת הדוק הבאה:

pop var

מה יהיו הערכים של sp ושל var לאחר ביצוע הפקודה הבאה?

תשובה: ערכו של var יהיה 0xAABBh, ערכו של sp יעלה בשתיים ויהיה שווה 100h.



מה קרה לערכים שבתווך המהסנית? האם הם נמחקו?

לא! הערכים שהכנסנו לmahsnet עדין קיימים בתוכה, אך כעת אין למעבד דרך לגשת אליהם כיוון ש-sp 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-disk



- א. הגדרו מהסנית בגדים שונים. 10h, 20h, 1234h. ראו כיצד משתנה ערכו של sp עם תחילת ההרצה.
- ב. הכניסו את הערך 1234h לתוך ax. בצעו push של ax. איך השתנה sp? הסתכלו על הזיכרון שבמהסנית ומיצאו את הערך שדוחף.
- ג. בצעו pop לmahsnet לתוך ax. איך השתנה sp? הסתכלו על הזיכרון שבמהסנית – האם הערך 1234h נמחק?
- ד. בצעו push לערך 5678h ?1234h. האם עכשו נמחק הערך ?1234h?
- ה. העתיקו את ax לתוך bx בעזרת mahsnet ללא שימוש בפקודת mov.

פראוצדורה

הגדרה של פראוצדורה

פראוצדורה היא קטע קוד שיש לו כניסה אחת, יציאה אחת (רצוי), והוא מבצע פעולה מוגדרת. 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 מנגנים להחזיר תוצאות העבודה. למשל, `num1+num2` לא רק ייחסב, אלא גם יועבר חוזה לתוכנית שזימה 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

```

#zeromem#zeroLoop: mov [bx], al
cs:0005 8807          mov     [bx],al
#zeromem#18: inc bx
cs:0007 43            inc     bx
#zeromem#19: loop ZeroLoop
cs:0008>E2FB         loop    #zeromem#zeroLoop (0005) ↑
#zeromem#20: ret
cs:000A C3            ret
#zeromem#start: mov ax, @data
cs:000B BB7B08         mov     ax,087B
#zeromem#25: mov ds, ax
cs:000E 8ED8           mov     ds,ax
#zeromem#26: mov bx, offset Digit
cs:0010 BB0000         mov     bx,0000
#zeromem#27: call ZeroMemory

```

ds:0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ss:0106 0000
ds:0008 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	ss:0104 0089
ds:0010 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	ss:0102 0403
ds:0018 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	ss:0100 52FB
ds:0020 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	ss:00FE>0016

F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

לאחר שהפְּרוֹצְׂדוֹרָה הספיקה לאפס 8 בתים ראשוניים (מודגש בצהוב)

תרגיל 9.2



העתיקו את קוד הדוגמה של ZeroMemory. הריצו אותו ב-IDTD שורה אחרי שורה וצפו בשינויים שתרחשים ב-**CODESEG** וב-**DATASEG** בשלבי התוכנית השונים

פקודות CALL, RET

בדוגמה ראיינו שפקודת **call** משמשת לזמן הפְּרוֹצְׂדוֹרָה. נראה לבדוק מה מבצעת הפקודה:

call ZeroMemory

לפני פקודת **call** לפְּרוֹצְׂדוֹרָה, הרגייסטר **IP=12**, **CODESEG**, כלומר מצביע על שורה **h 12**, שהוא השורה שאחרי פקודת **call** (בדוגמה שלנו: ...).

```

#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 7D 9D 00 EA FF FF = 3¥ 8
es:0008 AD DE 32 0B C3 05 6B 07 i [2d]ok.
es:0010 14 03 28 08 14 03 92 01 ¶( 0¶ff
es:0018 01 01 01 00 02 04 FF FF 888 8+
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 הוראת ה-call:

```

#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
es:0000 CD 20 7D 9D 00 EA FF FF = 3¥ 8
es:0008 AD DE 32 0B C3 05 6B 07 i [2d]ok.
es:0010 14 03 28 08 14 03 92 01 ¶( 0¶ff
es:0018 01 01 01 00 02 04 FF FF 888 8+
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

ערךן של ip השתנה ל-0000. זהו המיקום של הפְּרַזְצָדָרוֹה ZeroMemory בזיכרונו (זכרו, שמדובר ב-*offset* בתוך CODESEG, כלומר הפְּרַזְצָדָרוֹה נמצאת ממש בתחוםו של ה-G-SEG). כמובן, פקודת call משנה את ערכו של ip כך שהתוכנית קופצת אל תחילת הפְּרַזְצָדָרוֹה. אבל את הפעולה זו אפשר היה להציג גם באמצעות פקודה jmp. פקודה call עשויה משמשו נוספת. כפי שאולי שמתם לב, רגיסטר נוסף 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:0015h הווה 0000h. לפני פקודה ה-call ערכו היה 0Ah. 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 C0B9640008 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 0C 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 = 3F 9D 00 EA FF FF
es:0008 AD DE 32 0B C3 05 6B 07 i 2d ok.
es:0010 14 03 28 08 14 03 92 01 40 00 00 00
ss:0100 52FB
ss:00FE 0015

```

הגיע הזמן להזoor לתוכנית הראשית – לאחר ביצוע פקודת ret, הרегистר ip מצביע על שורת הקוד הבאה מיד אחרי שורה הקוד שקרה לפְּרוֹצְׂדָּוֶרֶת והתוכנית ממשיכה ממנה באופן טורי:

```

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 = 3F 9D 00 EA FF FF
es:0008 AD DE 32 0B C3 05 6B 07 i 2d ok.
es:0010 14 03 28 08 14 03 92 01 40 00 00 00
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.

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. מעטיקה לתוך הכתובת ss, את הכתובת בזיכרון אליה יש לחזור בסיום ריצת הפראוצדורה.
3. מעטיקה לתוך sp את הכתובת של הפקודה הראשונה של הפראוצדורה (פעולה זו שולחה לביצוע jump אל תחילת הפראוצדורה)

פקודת ret

1. קוראת מהכתובת ss, את הכתובת בזיכרון אליה יש לחזור.
2. מעלה את ערכו של sp בשתיים (כך במקרה זה; נראה מקרים אחרים בהמשך).
3. משנה את הדקן אל הכתובת שנקרה מ-sp, ובכך מזיזה את התוכנית לשורת הקוד שבה מיד אחרי הקראיה לפראוצדורה.

כמו שראינו, זו לא מקרים שבסיום ריצת הפראוצדורה, הערך של sp חזר אל השורה הבאה בתוכנית. פקודות ה-call וה-ret דאגו לשמר את ערכו של sp במחסנית ולהחזיר אותו בסיום ריצת הפראוצדורה!

פראוצדורות NEAR, FAR

התוכנית שלנו רצתה 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 שאנו מגדירים מודל זיכרון **small**, 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 מקומיות, שאין נמצאות ב-**CODESEG**.

אם הפראוצדורה נמצאת בתחום ה-**CODESEG**, היא נקראת פראוצדורות **near**. הפראוצדורה **ZeroMemory** היא דוגמה לפראוצדורה מסוג **near**. במקרה כזה, פקודת **call** מכניסה לתוך המחסנית רק את האופסט של כתובות החזרה – אחרי הכל, הסגמנט כבר ידוע – **CODESEG**. כיוון שהואופסט הוא בגודל שני בתים, הריגיטר sp משתנה ב-2.

לעומת זאת, אם הפראוצדורה נמצאת מחוץ ל-**CODESEG**, היא נקראת פראוצדורות **far**. במקרה כזה, פקודת **call** מכניסה למחסנית גם את האופסט וגם את הסגמנט של כתובות החזרה, כדי שהמעבד יוכל להحسب בדיקות לאן לחזור. האופסט והסגמנט תופסים ביחד ארבעה בתים, ולכן במקרה זה הריגיטר 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 כותבים 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, הגירה החדשה זהה לגרסת הקודמת. להלן צילום מסך של הכניסה לפראוצדורה. שימו לב לדברים הבאים:

- במקום `cs:0012` בתוכנית, האסמבילר הוסיף שורת קוד: `"push cs"`. שורת קוד זו גורמת להעתקה של רגיסטר `CS` לקוד `CS` לתוך המחסנית. כתוצאה לכך נדחף הערך `0879h`, שהוא ערכו של רגיסטר ה-`CS` (מודגם בכחול).

- לאחר מכן נדחף למחסנית הרגיסטר `sp`, ערכו `15h` (ולא `16h`) כמו בגרסת הקודמת, משום שנוסף שורת הקוד `("push cs")`.

- עם הכניסה לפראוצדורה, ערכו של `sp` הוא `00FCh` (ולא `00FEh`, כמו בגרסת 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 משמיטים את הכניסה של `CS` למחסנית. למעט הבדל זה, אין הבדל בין שימושם בפראוצדורות `near` ו-`.far`.

שימוש במחסנית לשמירה מצב התוכנית

נתרנו 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: בתחלת התוכנית **CX** מאותחל ל-3. בתוך הפְּרוֹצְדוּרָה ערכו משתנה ל-4. ביציאה מהפְּרוֹצְדוּרָה ערכו הוא 0, ואז הפקודה **loop Row** מפהיתה את ערכו באחד והופכת את ערכו ל-455,535 (כזכור, זהו הייצוג ה-**unsigned** של מינוס אחד). כיוון שתנאי העצירה של **loop Row** לא מתקיים (**CX** אינו שווה לאפס), היא ממשיכה לרווץ ולקראוב שוב לפְּרוֹצְדוּרָה, ששוב מוחזרה את **CX** עם ערך 0 וכן הללו...

לכן, אנו זוקקים 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 שייתבצע בדיקת מה שאנחנו רוצים. הטכניקה היא פשוטה: בכניסה לפְּרוֹצְדוּרָה צריך לשמור את הרגיסטרים במחסנית, ביציאה מהפְּרוֹצְדוּרָה לשלווף את הרכים מהמחסנית 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 למחסנית בכניסה 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 שפרוצדורה 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 לכמות פרוצדורה חדשה? ואם אחר כך נרצה לאפס שמונה בתים בזיכרון, שוב נצטרך לכמות פרוצדורה חדשה? לא הגיוני לעבוד בצורה זו. למה שלא נבנה פרוצדורה ש יודעת לקבל את מספר הבטים שעיליה לאפס בתום נתון, וכדי לאפס מספר שונה של בתים פשוט נציג

להודיע לפְּרוֹצְדוּרָה כמה בתים עליה לאפס בזיכרונו? לנตอน זהה, שהפְּרוֹצְדוּרָה מקבלת מוקד שקורא לה (למשל התוכנית הראשית), קוראים פרמטר.

פרמטר יכול להיות כל דבר שאנו רוצים להעביר לפְּרוֹצְדוּרָה. לדוגמה, שני מספרים שפְּרוֹצְדוּרָה צריכה לחבר ביניהם, יכולים לעبور כפרמטרים. מה לגבי תוצאת החיבור? איך מחזירים אותה לתוכנית הראשית? גם את תוצאת החיבור, ובאופן כללי כל תוצאה שהפְּרוֹצְדוּרָה צריכה להחזיר, אנחנו יכולים להעביר בעזרתו פרמטר.

יש יותר מושיטה אחת להעביר פרמטרים לפְּרוֹצְדוּרָה, השיטות הן:

- שימוש ברגיסטרים כלליים
- שימוש במשתנים שמוגדרים ב-DATASEG
- העברת הפרמטרים על גבי המחסנית

המושיטה הראשונה, שימוש ברגיסטרים כלליים, היא פשוטה ביותר. הפְּרוֹצְדוּרָה ZeroMemory, לדוגמה, מקבלת בתוך bx כתובת בזיכרון שמננה עליה להתחיל את איפוס הזיכרון. באותה שיטה הינו 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 שאנו צריכים לעשות בתוכנית הראשית הוא לזראג ש-ax יחויק את כמהת הבטים שאנו רוצים לאפס.

הערה: אפשר כמובן לבצע את אותה פעולה על-ידי ה הכנסת הפרמטר ישירות לתוך CX, אך כדי להקל על ההסביר נעשה בדוגמא זו שימוש ב-ax.

למרות פשוטות השימוש ברגיסטרים כלליים להעברת פרמטרים לפְּרוֹצְדוּרָה, 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 הפְּרוֹצְדוּרָה ולדעת באילו רגיסטרים היא משתמשת בתור פרמטרים.

השיטה השנייה, שימוש במשתנים שモוגדרים 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. שלפני כן היה פרמטר שקבע את כמות הבתים של התוכנית לאפס. המשתנה MemoryStart מחליף את הרגיסטר bx, שלפני כן היה פרמטר שקבע את המיקום בזיכרון שאנו רוצים לאפס. הצלחנו לפנות שני רגיסטרים לטובת שימושות אחרות (נכון, אנחנו עדיין משתמשים ב-al וב-bx בפְּרוֹצְדּוֹרָה, אבל כרגיסטרים כלליים לחישובי עזר ולא כפרמטרים, וזה אומר שהתוכנית הראשית כבר לא צריכה להיות עסוקה בקביעת הערכים הנכונים לרגיסטרים לפני הקריאה לפְּרוֹצְדּוֹרָה).

העברה פרמטרים על המהסנית

גם לשיטה הקודמת ישנו חסרונו – כותב התוכנית הראשית צריך להכיר את הוראות הפרוצדורה, לדעת שצורך להגדיר ב-SEG שני משתנים, ולא סתם שני משתנים אלא עם שמות קבועים שאי אפשר לשנות בלי לשנות את הפרוצדורה. התוכנית הופכת קשה לשימוש, יותר מכך – מה אם נרצה להשתמש בשתי פרוצדורות שלקחנו מקורות שונים, ושתייהן עשוות שימוש במשתנים בעלי שמות זהים? ומה אם נרצה לכתוב פרוצדורה שקוראת לעצמה (רקורסיה)?

נלמד קצת איך להעביר פרמטרים לפרוצדורה על ידי המהסנית. זהה השיטה המקובלת להעביר פרמטרים לפרוצדורות, משום שהיא פותרת את הבעיה שהציגו בשיטות הקודמות: אין מגבלה מעשית של מקום בmahsinit (אפשר להגדיר מהסנית יותר גודלה, ולא סביר שנctrdr יותר מ-64K, מגבלת גודל סגמנט), אין הגבלה על מספר הפרמטרים שאחננו שלוhim לprocדורה, אנחנו לא מוגבלים על-ידי כמות הרегистרים, ואחרון חביב – מי שקורא(procדורה) שלנו לא צריך לדעת איך אנחנו קוראים למשתנים בתחום(procדורה).

ישנן שתי שיטות להעביר פרמטרים אל procדורה:

1. העברה לפי ערך – Pass by Value

2. העברה לפי ייחוס – Pass by Reference

Pass by Value

בשיטת זו מועבר(procדורה) ערך הפרמטר. כמובן, על המהסנית נוצר העתק של הפרמטר.(procדורה) אין גישה לכתובת בזיכרון שמכיל את המשתנה המקורי, ולכן אין היא יכולה לשנות את ערכו – רק את ערכו של העתק. ביציאה מהprocדורה, ערכו של הפרמטר שנשלח(procדורה) יישאר ללא שינוי.

נמחיש זאת באמצעות דוגמה. נדמיין procדורה בשם SimpleAdd, שמקבלת פרמטר אחד ומוסיפה לערכו 2. נניח שהprocדורה זו היא חלק מספירה של פרוצדורות שימושו כתוב, קימפל ומסר לנו. הנתון זהה חשוב, כדי להציג שלאסמלר שקיים את SimpleAdd לא יהיה מושג מה הכתובות של המשתנים שאחננו מגדירים ב-SEG של התוכנית שלנו. יכול להיות שהprocדורה SimpleAdd נכתבה בכלל לפני שאחננו כתבנו את התוכנית שלנו.

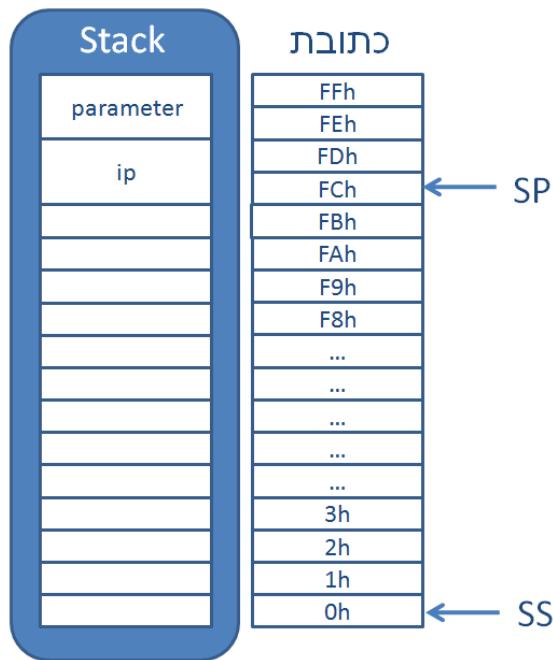
בתוכנית שלנו מוגדר משתנה, נקרא לו parameter. כתעת אנחנו רוצים SimpleAdd תקבל את הערך שנמצא בתחום parameter.

כדי להעביר את ערכו של הפקטר לפְּרוֹצְדוּרָה, התוכנית הראשית צריכה לדוחוף 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. מבחינהה, היא מכירה רק את ההעתק שלו שנמצא בכתובת כלשהו על המחשנית. לאחר שהפְּרוֹצְדוּרָה תוסיף 2 לערכו של הערך, 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 פְּרַמְטְּרִים: **i**, **j** ו-**k** ומחשבת בתוך ax את **k-j+i**. קוד אסמלטי מתאים לקרוא לפְּרוֹצְדוּרָה (בשיטת Pass by Value) יכול להיות:

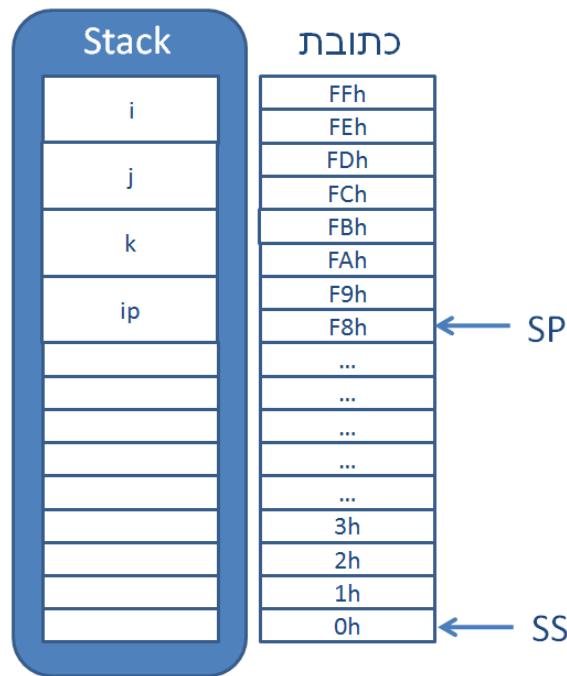
```
push [i]
```

```
push [j]
```

```
push [k]
```

```
call SimpleProc
```

עם הכניסה לפְּרוֹצְׂדָּוֶרֶת, המהסנית תראה כך (בහנחה שלא הכנסנו למחסנית מידע נוסף):



מצב המהסנית לאחר דחיפה i, j, k

בתוך הפְּרוֹצְׂדָּוֶרֶת SimpleProc אנחנו יכולים להוציא את הפרמטרים שהועברו אליה בעזרת פקודה pop. יש רק עניין אחד – ראש המהסנית מצביע על כתובות החזרה מהפְּרוֹצְׂדָּוֶרֶת, זו, שנדחף למחסנית עם פקודה ה-call. ככלمر ה-code הראשון שנעשה יוציא את כתובות החזרה מהמחסנית.

אנחנו נראה שיטה פשוטה לטיפול בכתובות החזרה. זו **אינה שיטה שימושית 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 ונדוחוף אותן למחסנית לאחר ביצוע ה-pop לכל הפרמטרים שהעברנו לפְּרוֹצְׂדָּוֶרֶת.

```
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 משטנה כלשהו שהגדכנו ב-DATASEG. אם נרצה, אפשר להחליף אותו ברגיסטר. בשיטה זו הצלחנו לקרוא את הparmsים שהועברו מהשנית אל הפרוצדורה ועדיין לחזור אל המוקם הנוכחי בתוכנית הראשית. כאמור-בקרוב נחליף את השיטה זה בשיטה נכונה יותר.

תרגיל 9.4 Pass by Value



א. צרו פרוצדורה שמקבלת פרמטר אחד בשיטת pass by value ומדפיסה למסך מס' 'X' אם הנתון בפרמטר. דגשים: יש לבדוק קודם הערך הפרמטר היובי! בסיום הריצה יש לשוחזר את ערכי הרגיסטרים שנעשה בהם שימוש. עזרה – הדפסתתו למסך:

```
mov dl, 'X'
mov ah, 2h
int 21h
```

ב. צרו פרוצדורה שמקבלת שני_Parms בשיטת pass by value ומדפיסה למסך 'A' אם הפרמטר הראשון גדול מהשני, 'B' אם הפרמטר השני גדול מהראשון ו-'C' אם הם שווים. בסיום הפרוצדורה יש לשוחזר את הערכים המקוריים של הרגיסטרים.

ג. צרו תוכנית שמוגדרים בה ארבעה מספרים קבועים בתחלת התוכנית, ומשתנים בשם max ו-chin. צרו פרוצדורה שמקבלת כ_Parms 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 לפרוצדורה הכתובת של הפרמטר בזיכרון. כמובן, על המשנית לא נוצר העתק של הפרמטר אלא רק הכתובת שלו בזיכרון מועתקת למשנית. הפרוצדורה לא יודעת מה ערך של הפרמטר שהועבר אליה אבל יש לה גישה לכתובת בזיכרון שמכילה את המשתנה, וכך היא יכולה לשנות את ערכו – היא פשוט צריכה לגשת למיקום בזיכרון. במקרה מהפרוצדורה, ערכו של הפרמטר שנשלח לפרוצדורה עשוי להשתנות.

נמחיש בឧרת דוגמה: שימו לב לפקודה חדשה שמו פיעעה בה. אנחנו הולכים לקרוא לפרוצדורה SimpleAdd, אבל הפעם בשיטת Pass by Reference:

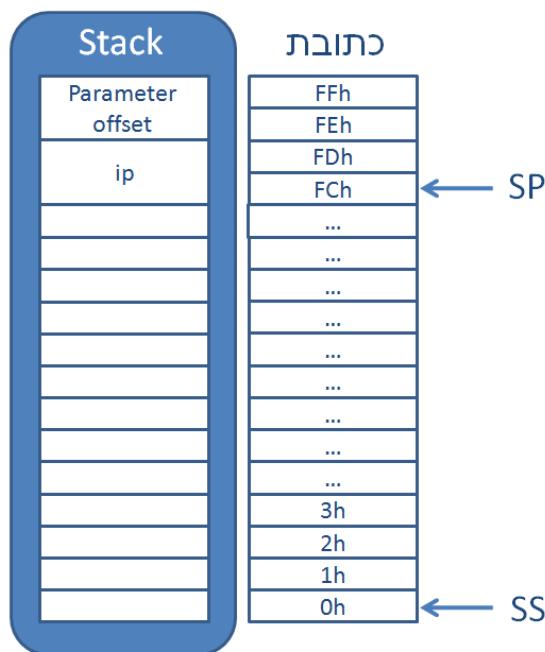
```
push offset parameter ; Copy the OFFSET of "parameter" into the stack
call SimpleAdd
```

אם היה לנו יותר סגמנט נתונים אחד – מה שכמובן אינו המצב – היינו צריכים להכנס למשנית גם את כתובתו של סגמנט הנתונים ש-parameter מוגדר בתוכו. כיוון שיש לנו רק סגמנט נתונים אחד, אין צורך בפקודה הבאה והיא לידע כליל בלבד:



```
push seg parameter ; Copy the SEGMENT of "parameter" into the stack
```

בזהירה למחסנית שלנו. מצב המחשנית 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 (לדוגמה – לפניהם).
`.(var1=5, var2=4. var1=4, var2=5` הпроцедура

שימוש ברגיסטר BP

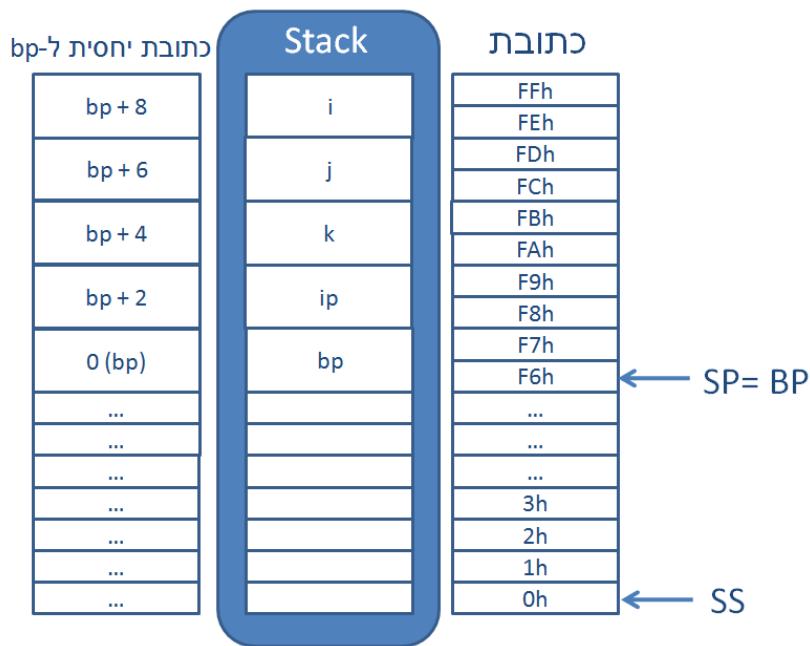
כשהצגנו את השימוש ב-[ReturnAddress] pop ציינו שזו שיטה שאינה מקובלת ולא נעשה בה שימוש בפועל. הסיבה היא, שככל פעם צריך לדאוג להוציא, לשמר ולהחזיר למחסנית את הרגיסטר ip.

הרגיסטר bp, קיצור של Base Pointer, מסיע 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 החדשות בזו אחר זו.

שתי הפקודות הראשונות מבצעות שמירה של bp למחסנית והעתקת sp לתוך bp. בשביל מה זה טוב?
 יצרנו פה בעצם מגנון, ששמור את ערכו ההתחלתי של sp. מעכשיו, גם אם sp ישנה כתוצאה מדחיפה או הוצאה של ערכאים מהמחסנית, bp נשאר קבוע ותמיד מצביע לאותו מקום. מגנון זה פותח לנו אפשרות לקרוא לכל ערך במחסנית לפי הכתובת היחסית שלו ליד bp.



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 שהמראתקים מ-bp נשארים קבועים למשך כל חי הפְּרֹצֶדָׁרָה. נראה איך משתמשים בمسקנה האחרונה. נדגים זאת שוב על הפְּרֹצֶדָׁרָה SimpleProc, שבוצעת את הפעולה $k = j + i = ax$. 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, שתשדרג 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
```

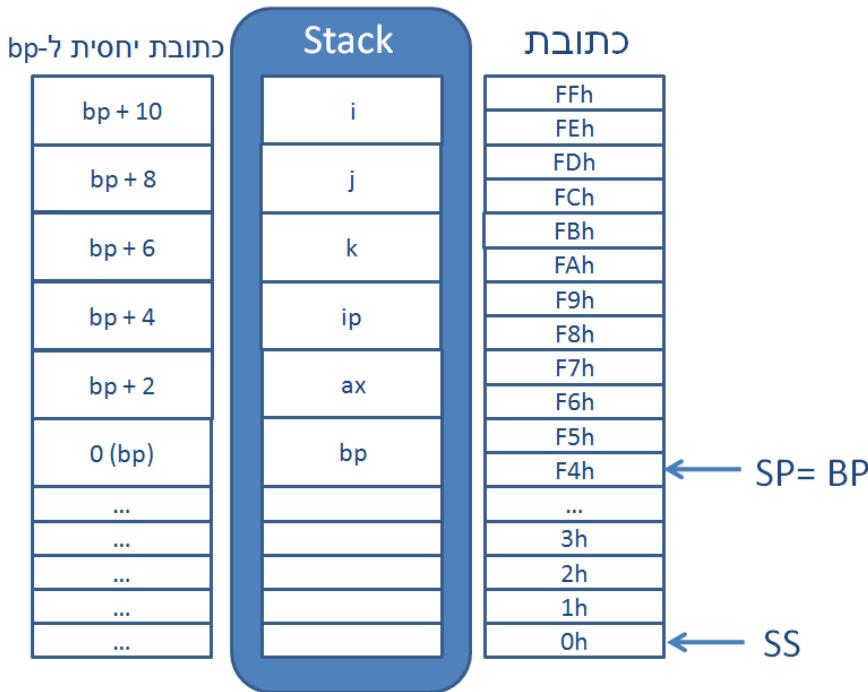
טיפ: השימוש ב-`equ` נוח במיוחד כאשר עובדים בשיטת `pass by value`. כך, לכל משתנה שדחפנו למחסנית יש שם מוגדר בפְּרוֹצְׂדָּוֶרֶת. לו היינו עובדים בשיטת `pass by reference`, היינו מרויחים פחות שימוש ב-`equ`, שהוא שਬשיטת `pass by reference` לא פונם יישירות אל הערך שנשמר במחסנית, אלא מעתקים אותו לתוך רגיסטר שמשמש לפניה לזכרון (לדוגמא - מעבירים מערך ומעתיקים את כתובות תחילת המערך ל-`ax`. במקרה זה אין תועלת להעת שם מיוחד לתחילת המערך משום שאנו תמיד ניגשים אל המערך דרך `ax`). 

שימוש לב: כדי שהמבנה שיצרנו יယב, העתקה של `bp` למחסנית ופקודת `mov bp, sp` חייבת להיות הפוקודת הראשונית בפְּרוֹצְׂדָּוֶרֶת. 

הסיבה היא, שאם בטעות נבצע פקודה `push` או `pop` לא מאוזנת (דחיפת והוצאה של כמהות לא שווה של בתים) לפני פקודות אלו, כל המבנה שיצרנו, של יצרת משתנים במרקדים מ-`bp`, יהיה משתבש. הנה דוגמה לשימוש לא נכון: `bi bp`:

```
proc WrongBP
    push ax
    push bp
    mov bp, sp
    ...

```



המהשנית של ret

בעקבות הוכנה של ax לפני bp, מבנה המרחקים מ-bp שיצרנו אינו נכון. עכשו bp+8 מצביע על j במקום על ax וכולו.

נסים בהסביר על פקודת `ret -6` שבסוף הפראצידורה.

ראינו שפקודת `ret` מבצעת את התהליך הפוך מאשר פקודת `call`: פקודת `ret` מוציאה את כתובות החזורה מהמהשנית, מעלה את ערכו של sp ב-2, ו קופצת אל כתובות החזורה. 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 `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 `ret` נועד לשחרר מקום במחשנית שתאפשר ב Amendments פקודות `push` לפני הכניסה לפראצידורה. בדוגמה שלנו עשינו `push` לשולחה משתנים בגודל 2 בתים כל אחד, כלומר הוכנו בסך הכל 6 בתים למחשנית. הפקודה `ret 6` מוחירה את sp למצווי המקורי טרם הכניסה לפראצידורה ובכך "משחררת" את הזיכרון שבחשנית. באותה מידה הינו

יכולים לבצע `ret` רגיל בלי ספרה לידי, ולאחר כך שלוש פעמים `pop`, אך השימוש ב-`6 ret` יותר אלגנטי, כיוון שהוא לוקח פחות מקום ומבצע את הפעולה המבוקשת באמצעות פקודה אחת בלבד.

סיכום ביןימים של יתרונות השימוש ב-`bp`:

1. בתחילת הפרוצדורה לא צריך לעשות `pop` לכנתובת החזרה ולשמור אותה.
 2. לא צריך לעשות `pop` לכל הערכים שדחיפנו למחסנית. פשוט ניגשים ישר אל הכתובות שלהם במחסנית בעזרת `bp`.
 3. אפשר ליצג כל תא בזיכרון המחסנית עליידי שם קרייא ובעל משמעות.
- יתרונו נוספת של `bp`, שנראית מיד, הוא `sh-bp` עוזר לנו לגשת למשתנים מקומיים.

תרגיל 9.6: פרוצדורה ושימוש ב-`bp`



- A. כתבו פרוצדורה שמקבלת שני משתנים, `pass by reference`, בשיטת `var1, var2, max`, ומחליפה ביניהם (לדוגמה – לפני הפעוץודורה `var1=5, var2=4`. אחרי הפעוץודורה `var1=4, var2=5`). בתוך הפעוץודורה 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 מהфункциיה, `max` יוכל את הערך הגבוה `var1, var2` מבין.

שימוש במחסנית להגדרת משתנים מקומיים בפראוצדורה (הרחה)



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 מקצתה עברו המשתנה זיכרון על המחסנית וגם דואגת לשחרר את הזיכרון לפני החזרה לתוכנית שקרה לה.

איך זה מתבצע בפועל? כמו שראינו, כל הקצתה 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, ההקצתה תבוצע באמצעות הפקודה:

```
sub    sp, 6
```

כמובן שלפנוי היציאה מהפראוצדורה צריכה להתבצע הפעולה ההפוכה, כדי לשחרר את הזיכרון (וכדי להביא את sp לערך הנכון – שיזבייע 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 והפרש 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

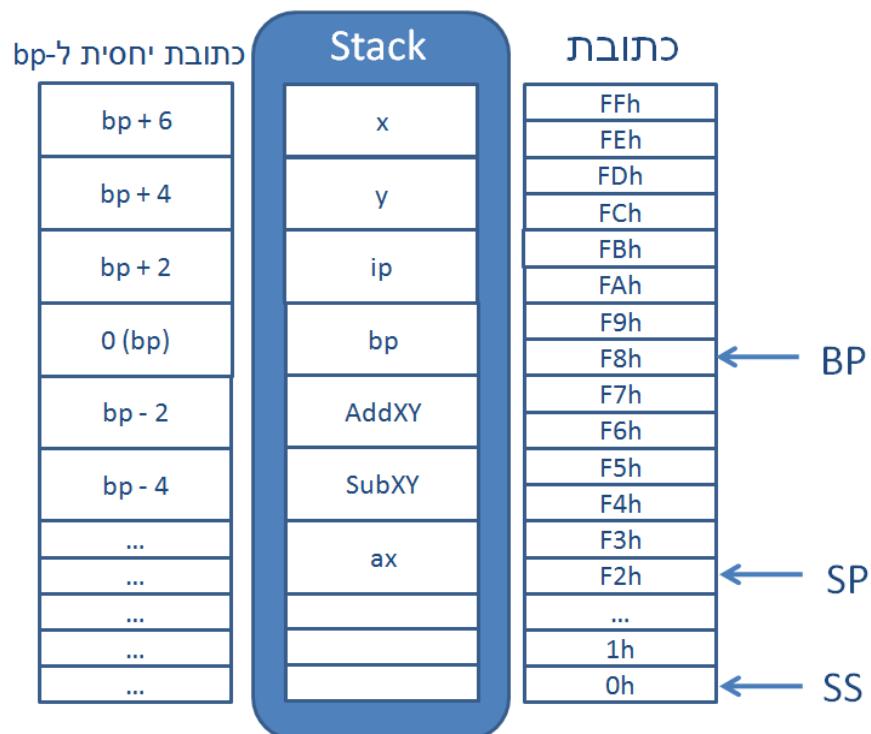
```

```

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` :



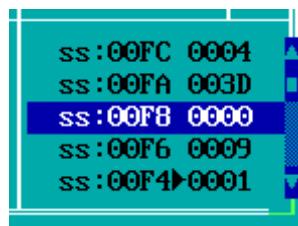
אם לדוגמה אנחנו קוראים ל- XYZ בתוכנית הראשית באמצעות שורות הקוד הבאות:

```
push 5
```

```
push 4
```

```
call XY
```

כך נראה המהסנית, לאחר ביצוע החישובים ושמירת הערכים בתוך AddXY ו- SubXY . השורה המודגשת בכהול היא השורה עליה מצביע bp:



דחפנו 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 הן את הסכום (0009) והן את הפרש (0001), במקומות זייכרונו בו הגדרנו את המשתנים המקומיים AddXY ו- SubXY , בהתאם.

שיםו לב לדמיון שבין צילום הזיכרון במחשב לבין האירור שמתאר את מצב המהסנית: במקום ss:00F6, שהוא [bp-2], נמצא הסכום של שני הפרמטרים. במקום ss:00F4, שהוא [bp] נמצא החיסור של שני הפרמטרים.

תרגיל 9.7: ייצרת משתנים מקומיים במחסנית



כתבו תוכנית ראשית, שדוחפת למחסנית שלושה ערכים כלשהם וקוראת לפראצדרה בשם XYZ . 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`. הסונותה השיטה זו:

- מייגעת, אם המערך ארוך.
 - توפסת הרבה מקום במחסנית.
 - הפראוצדורה לא יכולה לשנות את הערכים של המערך המקורי.
- בשיטה `pass by reference`, לעומת זאת, מעבירים לפראוצדורה רק את:
- הכתובת של האיבר הראשון בזיכרון
 - מספר האיברים בזיכרון

כעת נראה דוגמה. נגדיר מערך בתוך DATASEG:

DATASEG

```
num_elements      equ   15
Array            db    num_elements      dup (?)
```

בתוך CODESEG, לפני הקראיה לפראוצדורה, נדחוף למחסנית את כתובות האיבר הראשון בזיכרון ומספר האיברים בזיכרון:

```
push  num_elements
push  offset Array
call  SomeProcedure
```

בתוך הפראוצדורה אפשר להתייחס לכל אלמנט בזיכרון על-ידי כתובות הבסיס שלו, שהוועתקה אל המחסנית, בתוספת היחס של האלמנט מתחילה בזיכרון.

תרגיל 9.8: העברת מערכים לפראוצדורות



שימו לב - בתרגילים הבאים, אין להשתמש בתחום 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 בזיכרון. לדוגמה, עבור המערך `2,2,3,4,5` התוצאה תהיה `sum=16`.

ב. כתבו פרכוזדרה SortArray שמקבלת מצביע לערך ומספר איברים בערך, וממיינת את המערך מהאיבר הקטן לגודול. לדוגמה עבור המערך 3,6,5,2,1 הפרכוזדרה תגרום למערך להכיל את הערכים: 1,2,3,5,6.
הדרך לכתיבת הפרכוזדרה:

- כתבו פרכוזדרה עוז FindMin שמקבלת מצביע לערך, מוצאת את האיבר הקטן ביותר ומהזירה את האינדקס שלו.

- כתבו פרכוזדרה עוז Swap שמקבלת שני פרמטרים באמצעות שיטת pass by reference ומחליפה את הערכים שלהם.

- הפרכוזדרה SortArray תורץ בולולה על המערך ותקרא ל-`FindMin`. לאחר מכן תקרה ל-`Swap` עם שני פרמטרים: האינדקס שהזירה `FindMin` והאינדקס הראשון בערך.

- לאחר שהפרכוזדרה העבירה את האיבר הקטן ביותר לאינדקס הראשון, היא תקרה ל-`FindMin` כאשר המצביע לערך הוא על האינדקס השני בערך. לאחר מכן SortArray תקרה ל-`Swap` עם שני פרמטרים: האינדקס שהזירה `FindMin` והאינדקס השני בערך.

- הפרכוזדרה תמשיך בשיטה זו עד לסיום מיון הערך.

ג. כתבו פרכוזדרה שנקראה Sort2Arrays, שמקבלת מצביעים לשני מערכים ומספר איברים בכל מערך, ומצביע נוסף לערך יעד sorted אליו יש להכניס את תוכנתה המיוון של שני המערכים, כאשר ערכים כפולים מסוננים החוצה. לדוגמה:

`Array1 = 4,9,5,3,2`

`Array2 = 3,6,4,1`

לאחר הרצת הפרכוזדרה:

`Sorted = 1,2,3,4,5,6,9`

הדרך לכתיבת הפרכוזדרה:

- כתבו פרכוזדרה בשם Merge שמקבלת מצביעים לשני מערכים ומעתיקה אותם לערך אחד, ללא מיון או סינון.

- קיראו ל-`Merge` עם המערך שיצרה SortArray.

- כתבו פרכוזדרה בשם Filter שעוברת על המערך הממיין ומאפסת את כל הערכים שמופיעים יותר מפעם אחת.

גִּלְשַׁת מַהשְׁニָה - Stack Overflow (הרחבה)



Stack Overflow הינו מונח חשוב מתחום אבטחת המידע. באמצעות ביצוע פעולות שונות ניתן לשנות את אופן פעולתה של תוכנית ולגרום לה להריץ קוד שונה مما שתוכנן (ולכן בהיבט של אבטחת מידע מדובר בבעיה קשה). איןנו מודדים כלל שינוי קוד מקור של תוכנות והדבר אף אינו חוקי כמובן. מאידך מוכא לפניכם הרקע התיאורטי לנושא, מהסיבות הבאות:

1. מודעות לביעות אבטחה עשויה תעוזד אתכם לתוכנת קוד ברמה גבוהה יותר, שאיןו חשוף לביעות אבטחה קלאסיות.
2. הבנה של הנושא נדרש מכמ' חזקה והתעמקות בחומר הלימוד בפרק זה, שהינו אחד הפרקים החשובים ביותר להבנת אופן הפעולה של תוכנות מחשב.

ראשית נגידיר את המונח **Buffer Overflow**. מונח זה מתיחס למצב בו לתוכן מערך בגודל מסוים, מה שנקרה "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 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 גליישת זיכרון מוחז לבאפר. אם הבאפר שלוש הוגדר על המהשנית, הgliisha נקראת **Stack Overflow**.



הבה נראה איך עושים 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 עליו להקשין `.enter` או התוכנית מדפסה למסך הודעה `Program finished`.

```
Please enter your name, press enter to finish
Jon Snow

Program finished
```

את קליטת שם המשתמש מבצעת פְּרוֹצְדוּרָה, שומרת את הקלט על המחסנית, בבאפר בגודל 10 בתים.

שימוש לב שלמרות שלמהשנית אפשר לעשות `push` רק בנסיבות של שני בתים, בגישה ישירה לזיכרון אפשר להעתיק אל המחסנית גם ערכים בגודל בית יחיד.



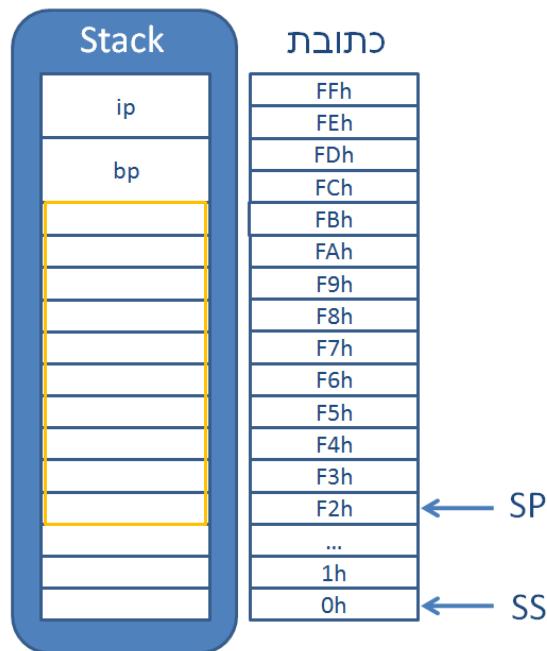
באופן לא שגרתי, בסיום קטע הקוד שמסים את ריצת התוכנית, יש קטע קוד נוספת נוסף, שמדפיס הודעה שונה למסך. זה קטע קוד מוזר, מכיוון שהוא לא אמור להיות מורץ. אין בתוכנית שום פעולה שמקפיצה את כך כך שיגיע אל קטע הקוד הזה.

אם כך, איך אפשר להסביר את ההרצחה הבאה?



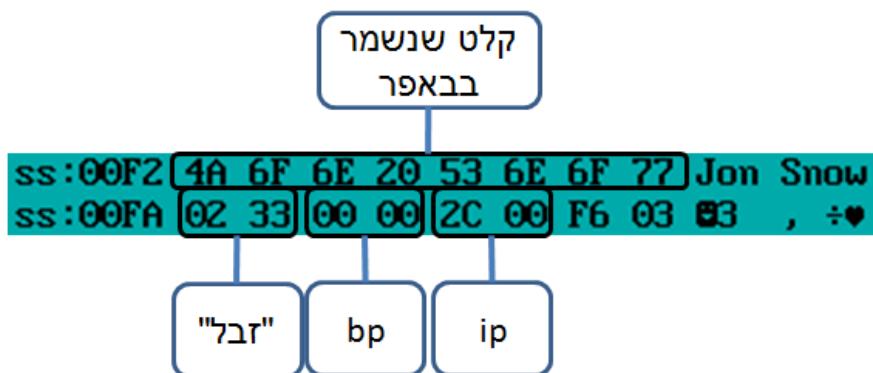
```
Please enter your name, press enter to finish
Jon Snow !  
Here be dragons
```

לאחר שנוכחנו שיש כאן התנוגות מוזרה, נסביר מה התרחש כאן שלב אחריו שלב. הפְּרֹזְצָדָרָה `GetName` באפר בגודל 10 בתים על המהסנית. בתוך הפְּרֹזְצָדָרָה, המהסנית נראה כך:



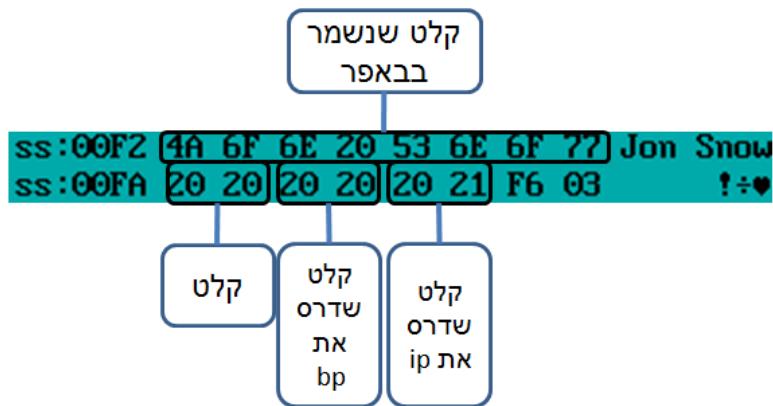
הmahסנית בתחום `GetName`. בצהוב- הבאפר בגודל 10 בתים.

בהרצתה הראשונה משתמש הוזן את שמו, `Snow`, סך הכל 8 תווים (כולל הרווח בין המילים). ערכי ה-ASCII של התווים נכנסו לmahסנית בזה אחר זה והmahסנית נראה如此:



נבחן את מצב mahסנית. הקלט הראשון – האות J (4Ah : ASCII (4Ah) – נכנסה לתחילה הבאפר במקום ss:00F2. יתר האותיות שנקלטו הועתקו לזכרון 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 קו, כתובות החזרה.

כעת נניח שהמשתמש שלנו ממשיך להזין תווים. לאחר שהוא כתב את המחרוזת 'Jon Snow', המשתמש מזין 5 תווים נוספים ואז סימן קרייה. כתוצאה מהעתקה של 14 בתים לתוך באפר שגודלו 10 בתים, נוצר מצב של Stack Overflow. בשלב זה mahסנית נראה如此:



ארבעת הבטים האחרונים שניכנו למחסנית, דרכו את `kp` ואת כתובת החורה, הערכיהם המקוריים הוחלפו בערכי ה-ASCII של התווים שהוקלו: ערך ה-ASCII של תוו הרוחה הוא `20h` ואילו של סימן קריאה-`21h`. פעללה זו לא גרמה למעבד לשגיאה, כיוון שלא התבכעה פעולה לא חוקית, ההתקאה לזכרון בוצעה באופן תקין מבחינה שפה אSEMBL. עם זאת, בפעולה זו שוכתבה כתובת החורה של הפְּרוֹצְדוּרָה. חישבו – מה היו ההשלכות של מצב זה?

התו הבא שהמשמש מזון הוא `enter` ובשלב זה הפְּרוֹצְדוּרָה מסתiyaת ומגיעה לקוד היציאה. בשלב הראשון נמחק הבאפר שהוקצתה בזיכרון. בשלב השני מבוצע `pop kp` ל-`bp`. כמובן, שהערך שנכנס לתוך `kp` אינו הערך המקורי שנשמר במחסנית אלא הערך החדש-`2020h`. לבסוף מבוצעת פקודת `h-ret`. הערך `2120h` מועתק לתוך `ko` ובוצעת קפיצה לכתובת `2120h` (מדוע לא `?2021h`? הזכירו – הזכירו פועל בשיטת `little endian`).

הקפיצה לכתחבת `2120h` מביאה את `ko` היישר אל קטע הקוד שמדפיס את הודעת הסיום 'Here be dragons'. המשפטbekohot mifpatot utikot vemesuot hiruhot shelo hiya "azor la-moker, shein le-utat ma nafza bo. zfu l-skenot...".



מפת *Psalter* מ-1265. בתחום המפה, מופיע טרייטוריה המוכרת לאדם, דרקונים.

לסייעם בעי' זה, ראיינו שבוזרת שימוש ב-Stack Overflow ניתן להקפיין את התוכנית להוראות שהמתקנה לא תכנן שיתבצעו. אולם במקרה הזה הפקודות שאלייהן קופצים נמצאות בתוכנית, אולם באמצעות טכניקות שונות (שאין בהיקף של ספר זה) ניתן לשתול פקודות חדשות בטור 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.عليיכם למנוע לא רק את האפשרות שתודפס המחרוזת 'Here be dragons' אלא גם כל אפשרות ל- Stack 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 שבשפה עילית לבין הקוד שבשפת המcona מתבצע על ידי הLINKER. אותו הLINKER בפרק אודות סביבת העבודה – אשר מקשר בין מספר קבצים ליצירת קובץ הרצה היחיד בשפת המcona. 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** (פרק ב-**big endian** והגדלת 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 יכול לגרום לתופוקד.

קונבנצייה – Convention – מוסכמה. התנוגות רוחות.

כפי שלמדנו על קונבנציות הקשורות לזיכרון, ישנו גם קונבנציות של הפעלת פראוצדורות, ואלו נקראות **Calling Conventions**. תחילת נמחיש מדוע יש בכלל צורך ב-**Calling Conventions**:
נניח הגדרה של פונקציה בשפת C:

```
int MyProc (int a, int b);
```

מה אנחנו יכולים לדעת על הפראוצדורה **MyProc**? אפשר לקבוע שהיא מקבלת שני פרמטרים מטיבוס **integer** ומהזירה ערך מטיבוס **integer**. מה בדיקות עשויה הפראוצדורה אין מונה כרגע.

קריאה לפראוצדורה יכולה להיות לדוגמה:

```
int c = MyProc(1,2);
```

قطع הקוד שקורא לפראוצדורה (ה"אבא") נקרא **Caller** ואילו קטע הקוד שנקרא, ככלומר הפראוצדורה עצמה (ה"בן"), נקרא **Callee**.

כעת הנה נדמיין שהפראוצדורה **MyProc** נכתבה בשפת אסמבלי ואילו הקריאה לפראוצדורה 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 לפראוצדורה ביותר מדרך אחת. אפשרות אי':

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 ה-callee קורא את הפרמטרים מהמחסנית, ערכיהם של הפרמטרים יהלפו. המסקנה היא, ש כדי למנוע תקלות, ה-caller וה-callee חייבים לעبور לפי אותה הקונבנצייה.

2. החזרת ערך מהפראוצדורה:

הדרך הנוחה ביותר להחזיר את הערך שהשיטה MyProc אל הקוד שקרה לה, היא להשתמש ברגיסטר. כלומר, הפראוצדורה MyProc תעתק את תוכנת הפעולה שלה אל רגיסטר כללי כלשהו, והערך יועתק על ידי ה-callee מהרגיסטר אל המשתנה ששומר את ערך החזרה, במקרה זה – המשתנה C.

כמובן שה-caller וה-callee חייבים להיות מתואימים לגבי הרגיסטר שמשמש להחזרת הערך. אין זה משנה באיזה רגיסטר כללי יוחזר הערך, רק שה-caller וה-callee יתיחסו אותו הרגיסטר.

3. ניקוי המחסנית:

כזכור, כאשר מעבירים פרמטרים לפראוצדורה על גבי המחסנית, יש צורך לנוקוט את המחסנית בסיום ריצת הפראוצדורה. פעולה זו מתבצעת על ידי העלאת ערכו של sp בהתאם למספר הבטים שהועתקו למחסנית. למדנו שישנן שתי דרכי לעשות זאת. האחת, היא להוסיף לפקודה ret קבוע. לדוגמה, ניקוי 4 בתים מהחסנית:

ret 4

הדרך השנייה היא לקדם את ערכו sp:

add sp, 4

ה-caller וה-callee צריכים להיות מתואימים לגבי מי מנקה את המחסנית. אם ה-callee מבצע את הניקוי, בתוך הפראוצדורה תהיה פעולה ret עם קבוע. אם ה-callee מבצע את הניקוי, לאחר החזרה מהפראוצדורה 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) ובמגוון אתרים אינטרנט, אך לצורך המחתה הנושא נסתפק בפירוט של CDECL ו-STDCALL.

לפי קונבנציית CDECL:

- ערבים מועברים מהסנית Right to Left.
- רגיסטר החזירה הוא ax (או רגיסטר ax מורחב, אם מדובר על מעבד של למעלה מ-16 ביט)
- ה-caller הוא שאחראי על ניקוי המהסנית.

לפי קונבנציית STDCALL:

- ערבים מועברים מהסנית Right to Left.
- רגיסטר החזירה הוא ax (או רגיסטר ax מורחב, אם מדובר על מעבד של למעלה מ-16 ביט)
- ה-callee הוא שאחראי על ניקוי המהסנית.

היתרון של STDCALL על פני CDECL, הוא מאפשר לשולח לפראצ'זרה כמהות לא קבועה של פרמטרים. מדוע? משום שה-caller אחראי על ניקוי המהסנית, כלומר מי שביצע את פעולה עדכון sp הוא ה-caller. כיוון שה-caller יודע כמה פרמטרים הוא עבר לפראצ'זרה והוא גם יכול לבצע את הניקוי. ה-callee, לעומת זאת, אינו יכול לבצע את הניקוי כיוון שהוא לא אחראי על ניקוי המהסנית.

מתי נרצה להעביר לפראצ'זרה כמהות לא קבועה של פרמטרים? לדוגמה, כאשר אנחנו קוראים לפראצ'זרת print בשפה עילית, אנחנו רוצים גמישות לקרוא לה עם כמה פרמטרים שאחנו צריכים להדפיס. כל פרמטר יועבר לפראצ'זרה print, וכך ניתן לכתוב את print כך שהיא תנקה את המהסנית בסוף הריצחה.

מהו היתרון של STDCALL על פני CDECL? כאשר כמהות הפרמטרים היא קבועה וידועה מראש, הפראצ'זרה יכולה לנוקוט את המהסנית על ידי פקודת ret עם קבוע. זאת לעומת המצב שבו הפראצ'זרה מבצעת ret ואז ה-caller מבצע פקודה שמעלה את sp. ככלומר חסכנו פקודה. אמנם חסכו של פקודה אחת נראה כמו משהו צנוע, אך כשמכפילים את החיסכון בכמות הקריאה לפראצ'זרות 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 בנויה תוכניות מודולריות. כשתעבדו בעtid עם שפות תוכנה עיליות, חלק מהמנגנוןם שלמדנו בפרק זה יהיו נסתורים מעיניכם. לדוגמה, כדי להגדיר משתנים מקומיים לא נדרש להזכיר מקום על גבי המחסנית – אולם הרקע שנותן בפרק זה יאפשר לכם להבין לעומק את פעולות המעבד, והבנה 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 בהסביר 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 להגדיר 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. כתה 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 אסמבלי.

בפרק הבא נלמד על פסיקות. בפרק הלימוד רأינו כמה דוגמאות קוד שלא הבנו – למשל, השימוש בקוד של הדפסת תווים או קריאת תווים מהמשתמש, אך לא הבנו איך הקוד פועל. לאחר שנלמד פסיקות נבין את קטיעי הקוד הללו ונדע לכתוב בעצמינו 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 אסמבלי ונקראות "שורדים". התחרות מתקיימת אחת לשנה והשתתפות בה מתבצעת בקבוצות של 2-5 תלמידים.



לונן תחרות CodeGuru Extreme

אתר המתחרות: <http://www.codeguru.co.il/xtreme>

השתתפות בתחרות היא בעליית יתרונות רבים. ראשית, זהה הזדמנויות מעוללה לחודד את כישורי התוכנות שלכם בשפת אסמבלי. שנייה, זהה הזדמנויות להתנסות במחקר תוכנה על ידי Reverse Engineering. שלישיית, השתתפות בתחרות היוקרתית היא פרט שנייתן להוסיף אותו לקורות החיים ולצין אותו בראיניות עבודה ובמיונים לתפקידים שונים. אחרון- התחרות היא חוות כיפית ומהנה, גם מי שלא זוכים בה זוכרים את החוויה.

הסביר קצר על חוקי התחרות:

כל קבוצה מגישה לתחרות קוד אסמבלי, שנקרא שורד. השורדים שהגיבו כל הקבוצות נטען אל זירה וירטואלית, קטע זיכרון שגודלו 64 קילו בתים בסך הכל. כל שורד מוגREL למיקום כלשהו בזירה. בכל תור, כל שורד מקבל אפשרות להריז פקודת אסמבלי אחת. באמצעות פקודת האסמבלי 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 חידה תכנותית או מתמטית, שפתרון שלה מאפשר השתלטות על הזרם ושימוש בו נגד שורדים מתחרים.

נסתפק בהסביר קצר זה, כיוןSCP הפתרים והדרכה על התחרות מצויים באתר קודגورو אקסטרים ובמיוחד בחוברת הדרכה ובמצגת, שבקישוריהם הבאים:

חוברת הדרכה: <http://data.cyber.org.il/assembly/codeguru-guide.pdf>

מצגת: <http://data.cyber.org.il/assembly/codeguru-slides.pdf>

כמו כן מומלץ להעזר בפורום שבאתר קודגورو אקסטרים, שם גם מפורסמים עדכוניים שותפים לגבי התחרות:

<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 הדרכה 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 חולשה שלהם.

פקודות אסמללי שימושיות

פקודות אלו אינן הכרחיות לטובת ייחודה המעבדה באסמללי, כיון שניתן לכותב 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 וכן רצוי להכיר אותן כדי לכתוב קוד יעיל לשורה.

1. XCHG: פקודה XCHG מקבלת שני רגיסטרים, או רגיסטר וטא בזיכרון, ומחליפה בין הערכים שלהם

לדוגמה:

xchg ax, bx

מחליפה בין הערכים של ax ו-bx. פעולה זו הוסכת 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 המרה מסט אחד של ערכים לסט אחר של ערכים. 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 פרוצדורה 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 שככל אינדקס ב-LUT מצביע על הערך אותו יש להחרור. לדוגמה, קוד ASCII של a הוא 97 ואנחנו רוצים להחליף אותה עם האות m, שקוד ASCII שלה הוא 109. את האות b, שקוד ASCII שלה הוא 98, אנחנו רוצים להחליף אותה עם האות f, שהקוד שלה הוא 102. נדאג שבאינדקס 97 יישמר הערך 109 ובאינדקס 98 יישמר 102. הנה כך:

Cipher db 97 dup (0), 'mf'

כמובן שאנחנו יכולים להוסיף אחרי mf גם את הצופן של יתר האותיות. בעת בתוך סגמנט הקוד נכתב:

mov bx, offset Cipher

mov al, 'a'

xlat

ואם יכיל את ערך ASCII של m, כפי שרצינו.

.3. **NOP**: זה קיצור של No Operation, כולם פקודה שאומרת למבצע "אל תבצע כלום". פקודה זו יכולה להיות שימושית כאשר צריך למלא אזור זיכרון בפקודה חוקית, אך בלי שהיא תשפיע על מצב התוכנית.

.4. **STD / CLD**: הפוקודה STD מדliquה את דגל הכוון ואילו CLD מכבה אותו. לשם מה זה נחוץ? בשבייל פקודות .MOVSW

.5. **MOVSW**: פקודה זו מסיימת לנו להעתיק בצורה יעילה מידע מאזור זיכרון אחד לאחר. לדוגמה, יש לנו מהרוות של 200 איברים, ואני מעתיק אותה למקום אחר בזכרון. דרך אפשרית היא ליצור לו לאה שתרוץ על המהרוות הראשונה תוך כדי שהיא מקדמת בכל פעם אינדקס שימוש לקריאה מהזיכרון, ובעזרת פקודה mov תעתק את המחוורת למקומות החדש, תוך קידום אינדקס שימוש לכתיבה לזכרון. זה יעבד, אולם פקודה MOVSW עוזרת לפשט את התהליך.

פקודה זו מעתקה מילה מכתובת di אל הכתובת si, es:di, תוק עדכון ערכי הרגיסטרים si, di. ככלומר, נחERICA מאיתנו המתכנתים פועלות 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 שיכיל מספר הפעמים שהוראת העתקה צריכה להתבצע ואז כותבים rep movsw

קוד זה שקול לפועלות הבאות:

my_label:

```
movsw
dec    cx
jnz    my_label
```

פקודת אסמבלי נוספת ניתן למצוא בנספח א'.

Reverse Engineering

נושא ה-RE, Reverse Engineering, או בקיצור RE, הוא עמוק ורחב מכדי לסכם אותו בפרק אחד. לכן לצערנו אין ביכולתנו ללמד לביצוע RE במסגרת לימודי האסמבלי. מאידך, כן נלמד איך מבצעים RE לשורדים ולזומבים של קודגورو אקסטרים. זהה משימה בהיקף מצומצם, שמאידך אחד ניתן לכנות בפרק ייחד ומצד שני מספקת טיעמה מילולית מעניינת זו.

הזומנים שבפרק זה נמצאים בקישור:

data.cyber.org.il/assembly/zombies.zip

שיםו 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 לפרסוץ סיסמאות או לשנות דברים בתוכנה, הוא אסור לפי החוק. היזהרו והישארו בכך הטוב של החוק.



duck.com

נתחיל בניתוח זומבי פשוט. הורידו את הזומבי duck.com. כתת ענו על השאלה - מה מבצע הזומבי? נסו לגלות.



אפשרות אחת היא להפעיל את קובץ ההרצה. מתוך חלון ה-cmd הקישו `duck` ואו `enter`. התוכנית רצתה אך אינה מגיבה... הבעה 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 שונות 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 `cs: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, נעבר לזומבי הבא, שמו coffee.com. הפעם גם נבצע RE כדי להבין מה הזומבי מבצע וגם נלמד איך אפשר לגרום לו להרין קוד שאנו חתנו.



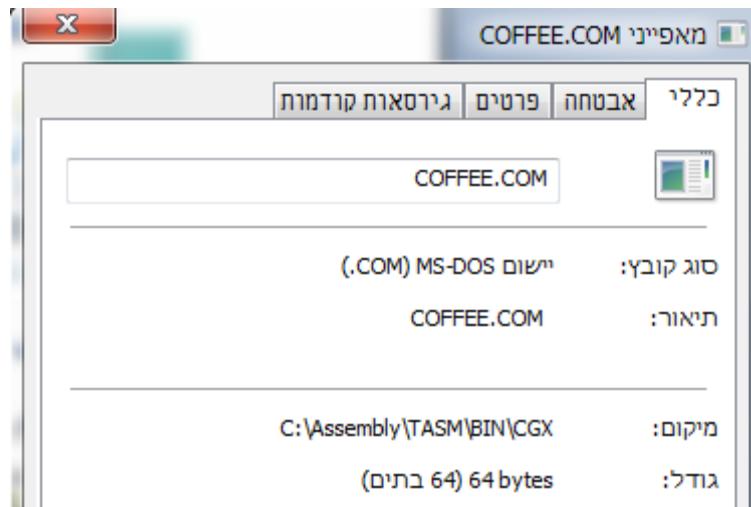
```

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 לציין 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- $h-0140:cs$ כבר אינה חלק מהקוד של הקובץ. כך וידאונו באופן סופי את ההשערה שלנו.

כעת – מה עושה הזרומי `?coffee`

שלוש הפקודות הראשונות צריכות להיות מוכרכות לכל מי שקרא את המדריך – פקודת `int 87` היא הפצתה חכמה. הזרומי נפטר מהפצתה החכמה שלו.

ב-4-`cs:0104` מועתק בית מקום `0000h` בזיכרון לתוך `dl`. בפקודה הבאה הערך של `dl` מושווה ל-`43h` ואם אין שוויון, מתחיצה 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 הבא – האם הבית הבא בזיכרון מכיל את הערך `43h` (קוד ה-`43h` ASCII של הספרה 0). המשיכו מכאן – אילו תנאים צריכים להתקיים כדי שהתוכנית הגיע אל פקודת `ret?`

נניח שאנו רוצים להשתלט על זומבי זה, לומר לגורם לו להריץ את הקוד שלנו. איך אפשר לבצע זאת? נctrkr לשילוח אותו ל-`cs:013Fh` שבו נמצא הקוד שלנו. פקודת `ret` שבכתובת `cs:013Fh` יכולה לסייע לנו ביצוע התהליך. אנחנו צריכים לדאוג לשני תנאים. הראשון, שהזרומי הגיע לירץ את שורת `ret`. ראיינו שעיל ידי הזנת ערכים מתאימים למיקומים מוגדרים בזיכרון, אנחנו יכולים "לשחרר" אותו ולגרום לו להגיע לפקודת `ret`. התנאי השני, הוא שעם ההגעה לפקודת `ret` בראש המחסנית יהיה ה-`cs` של הקוד שלנו. לעוזרתנו נמצאות הפקודות שמעתיקות את מקום `0006h` אל `ax` ואז דוחפות את `ax` בראש המחסנית. אם נdag לך שזמן שהזרומי מריץ את שורות אלה בכתובת `0006h` נמצא כתובות של פקודה בקוד שלנו, הזרומי יסיים את הריצה ויקפוץ אליה. שם הוא ימשיך ויריץ את הקוד שלנו.

סיימנו את הטיפול ב-`coffee.com`. זה זומבי חביב ולא מורכב שכל מטרתו למד את הרעיון הכללי של RE לתוכנית קצרה ושליחת המעבד אל קוד במיקום אחר.

כעת נטפל בזומבי אמיתי מתחרות 2015. CodeGuru Extreme. כנהוג בתחום, הזרומי 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 הזרומי 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 ההתחלמי של הזרומי ואילו `bx` מאותחל לקבוע `1Dh`.
- ד. לאחר מכן נטען לתוך `ax` המילה שמתחליה במיקום `h 2015` בזיכרון.
- ה. לאחר ביצוע `or` בין `ax` ו-`cx` מ被执行 פעלות `xlat`. נזכיר במה מבצעת `xlat`: היא מעתיקה לתוך `ax` את הערך שנמצא במיקום `al+bx:ds`. נציין כי בנויגוד לתוכניות שאיתן עבדנו בפרקם הקודמים, בקודגورو אקסטרים מודול הזיכרון מוגדר כך `sh-ds` ו-`cs` מצביעים על אותו מקום בזיכרון. אם כך, על איזה זיכרון מצביע `?ds:bx`? 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:011`.

נחקור את המיקום בזיכרון `011Dh:cs`. זה המיקום הראשון שנמצא אחרי פקודת `ret`. נמצאת שם פקודת `add`.

משונה. גם יתר הפקודות הן צורות משונות של `add`. מדובר מטבח `xlat` למיקומים אלו בזיכרון? נניח `0=al`.

הפקודה `xlat` תחזיר את הערך שבמיקום `011Dh:cs`. ערך זה הוא `0`. נניח `1=al`, הפקודה `xlat` תחזיר את הערך

שבמיקום `h:cs:011E` 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

עזרה- הרשו בהקדם לאולימפיאדה למתמטיקה לנעור.

ח. המסקנה עד כאן היא שהזומבי שלנו לוקח את המיקום ההתחלתי שלו בזיכרון ואת הערך שנמצא בתא `2015` בזיכרון, עושה להם `orx` ומשתמש בתוצאה על מנת לgesht ל-`LUT`, שהינה סדרה מתמטית בעלת חוקיות מעניינת.

ט. פעולה `xlat` חוזרת על עצמה פעמיים, פעם עבר `ah` ופעם עבר `ah`.

י. לאחר מכן מחושב הסכום שלהם- אם הוא איינו עולה על `6`, מטבח `ret` ייצאה על ידי `ret`. איזה ערך יוכל `di`?

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 מהולאה בה הוא נמצא ולהגיע לפקודת `ret`, יש צורך להכניס ערך מסוים ל-`word` במיקום `2015h`. אותו הערך גם ה-`di` שהזומבי יקוף אליו בסומו. הבנת הערך ה"נכון" תלואה בהבנת חוקיות הסדרה המתמטית שאיתרנו.

תרגיל: Make it – Break it – Fix it

בתרגיל זה נכתוב תוכנית שמקשת סיסמה מהמשתמש ובודקת אם הסיסמה "נכונה". אם כן, תודפס למסך הודעה מתאימה: "Access granted". בשלב הראשון, נסה לנכתב תוכנה מסוובכת ככל הניתן לפענוה. בשלב השני נמסור את התוכנה לחברינו לכיתה, שניגסו לגרום להדפסת ההודעה "Access granted" על ידי פענוה הסיסמה. בשלב השלישי נפיק לקחים ממה שהברינו עשו ונשפר את התוכנה כך שמציאות הסיסמה תהיה קשה יותר.

שלב א' - Make it

בשלב זה אנחנו מעוניינים לנכתב תוכנית שקולטת סיסמה מהמשתמש, 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 בתוך, 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 הזיכרון...). התווים 13,10 מורים לרדת שורה בסוף ההדפסה. לדוגמה:

```
message db 'Hello World',13,10,'$'
```

לאחר מכן טוענים לתוך ad את האופטט של המחרוזת:

```
mov dx, offset message
```

נותר לנו רק לקבוע את ah=9h ולהפעיל את הפקודה int 21h:

```
mov ah, 9h
int 21h
```

הסברים מפורטים על פקודות אלו נמצאים בפרק אודוט פסיקות, תחת הסעיף פסיקות DOS.

להלן תוכנית פשוטה, שמקשת סיסמה מהמשתמש ועל סמך בדיקת הסיסמה מדפיסה הודעה הצלחה או כישלון. שקולטה נתונים, 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 שמציאות 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 לנסיונות מציאת הסיסמה?

הדרך:

- .1. בחרנו את השורות בהן מתבצעת הבדיקה של הסיסמה. האם תוכלו לפתח מגנון 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 הלאה בכוחות עצמכם. גלו יצירתיות!

שלב ב' - Break it

קחו את התוכנית שכ כתבו חבריכם (כמוון, רק את קובץ ההרצאה - 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 חברייכם פיצזו את סיסמת הכניסה לתוכנית שלכם, מצאו פיתרון שיקשה על חברייכם למצוא את הסיסמה באמצעות方法ה השיטה.

סיכום

בפרק זה סקרנו בקצרה את תחרות קודגورو אקסטרים וקיבלנו מספר כלים חשובים להצלחה בתחרות. הכליל הראשון הינו ידיעת פקודות אסמלוי מיוחדות. הכליל השני הוא יכולת מחקר של קוד שנכתב על ידי מי שהו אחר באסמלוי, הכליל השלישי הוא הבנת טכניקת ההשתלטות על זומבים. יתר הידע הנדרש מרוכז בחומרה הדרוכה - המשיכו מכאן בכוחות עצמכם.

איחולי הצלחה בתחרות והנאה מההשתתפות ומהתרגול לקרהת 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).



סביר למה הכוונה. משתמשים בפסיקה כשותפים 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 תעבור בדיקו באוטו אופן כל הזמן – לדוגמה, תמצא את האיבר האגול ביותר במערך – אלא תגיב לאיורים שונים. דמיינו משחק מחשב, שבו השחקן משתמש במקלדת כדי להפעיל דמות שזהה על המסך. התוכנה צריכה להגיב לפקודות השחקן – כל לחיזה על המקלדת צריכה לגרום לתוכנה להרים קוד אחר. גם התזמון של היזותה המקלדת לא ידוע מראש – התוכנה לא יכולה להניח שברגע מסוים השחקן יקיש על המקלדת. הצורך ביכולת לשנות את אופן ריצת התוכנה, גורם לכך שנוצר מנגנון שיודע לשנות את סדר ביצוע הפקודות במעבד באופן דינامي.

במשפחת ה-80x86 יש שלושה סוגים של איורים 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 לאיור 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 איור חיצוני שדורש טיפול. המעבד עוצר את ביצוע הקוד שרצן, משרת את רכיב החומרה וחזור לתוכנית למקום שעצר בה.

במשך נדון בהרחבה בכל אחת מסוגי הפסיקות.

קריאה לפסיקה מתבצעת באמצעות הפקודה int. לכל פסיקה יש מספר, שמייחד אותה מיתר הפסיקות. לאחר ה-int יבוא אופrnd, שהוא מספר הפסיקה.

int operand

לדוגמה, הפעלת פסיקה מס' 1:

int 1h

בפרקים הקודמים נתכלנו בפקודה זו. כאשר רצינו לבצע פעולות של קריאתתו מהמקלדת או הדפסתו למסך, למשל, השתמשנו בפקודה:

int 21h

במשך השנים חברות שונות פיתחו קוד שככלו אוסף של פסיקות תוכנה וחומרה שימושיות. לדוגמה, חברת מיקרוסופט, פיתחה מערכת הפעלה בשם DOS – קיצור 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, ונטען 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 בתוכני חומרה, לדוגמה קליטת תווים מהמקלדת, נושא שנגע אליו בהמשך.

כמובן מכך שיש לנו פסיקות זמניות לשימוש מקורות שונים, ישן לעיתים דרכים שונות לבצע 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 הטיימר.

- פסיקה מס' 1Ch היא פסיקה BIOS, שגם עובדת מול הטיימר. פסיקה זו למעשה "עוטפת" את פסיקה 8h עם קוד נוסף.

- פסיקה מס' 21h של DOS, עובדת גם היא מול הטיימר. 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 ולעתים גם ביצועים.

במסגרת לימוד הפסיקות רק יהיה מודעים לכך שיש דרכים שונות לבצע אותה פעולה, אבל לא נלמד את כל הדרכים. בדרך כלל נבצע את הפעולות באמצעות פסיקות DOS, וכך נקייש לפסיקות אלו הסבר מפורט.

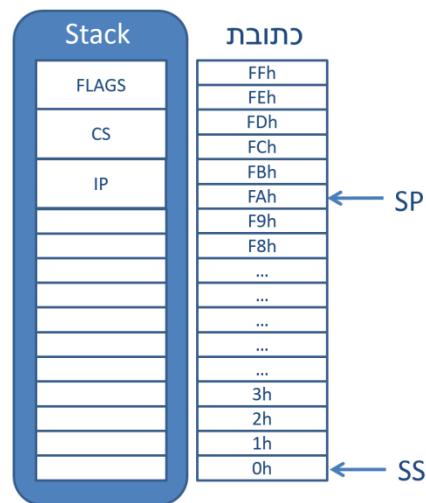
כל פסיקה, יהיה אשר יהיה המקור שלה, מפעילה קוד מיוחד לטיפול בפסיקה. הקוד הוא כמובן שונה בין פסיקה לפסיקה. קוד זה נקרא בשם כללי Interrupt Service Routine או בקיצור ISR. 

כשמעבד מבצע פסיקה, הוא מפסיק את ביצוע התוכנית, פונה אל ISR שנמצא בזיכרון המחשב ולאחר מכן חזר להמשך התוכנית שהופסקה. נסקרו כעת את המנגנון שמאפשר למעבד לבצע את הפעולה זו.

שלבי ביצוע פסיקה

תהליך ביצוע פסיקה מורכב משלבים הבאים:

1. המעבד מסיים את ביצוע הוראה הנוכחית. למשל – אם לדוגמה היונו באמצעות הוראת `mov ax,5` בזמן שהמשתמש לחץ על מקש במקלדת, המעבד יעתיק לתוך `ax` את ערך 5 ואוזיתפה לשרת את הפסיקה מהמקלדת.
2. המעבד שומר במחסנית את תוכן רגיסטר הדגלים ואת כתובת הוראה הבאה לביצוע בתום ביצוע תוכנית הפסיקה (ה-ISR). כזכור, כתובת הוראה 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).
4. המעבד מחשב את כתובת ISR ומעתיק אותו לרגיסטרים `cs` ו-`ip`. תהליך חישוב כתובת ISR ראוי להסביר בפני עצמו ונתיחה אליה בהמשך.
5. המעבד מבצע את ISR.
6. בתום ביצוע ISR, המעבד מוציא מהמחסנית את הערכים שנדרשו אליה (ראו סעיף ב') ומשחזר 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 המלכודות 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 להרין 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 תוקן כדי ביצוע הפסיקה הנוכחית. המונח המקצועי של פעולה זו הוא disable interrupts. פעולה זו חשובה כדי שלא תהיה "תחרות" בין פסיקות, שבה פסיקה חדשה מגיעה וגורמת למעבד לטפל בה לפני שטסים לטפל בפסיקה הקודמת.

לאחר היציאה מה-ISR, המעבד משוחרר מהמחסנית את רегистר הדגלים, ובין היתר את ערכו של הדגל if, וכך שפסיקות החומרה שוב מאפשרות. המונח המקצועי של פעולה זו הוא enable interrupts.



המבנה הכללי של ISR הוא כזה:

```
proc ISRname far
    ...
    iret
endp ISRname
```

פקודת ה-iret היא כמו פקודת ret – היא דואגת לשזרור כתובת החזורה אל המיקום שבו הייתה התוכנית לפני הקראיה ל-ISR. ההבדל בין iret ל-ret הוא שiret עושה pop נוספת, כדי לשזרור 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 לד-255 דצימלי, אך נהוג לקרוא לפסיקה לפי הערך הheximal שלו. לדוגמה, כשכתבנו את הפקודה:

```
int 21h
```

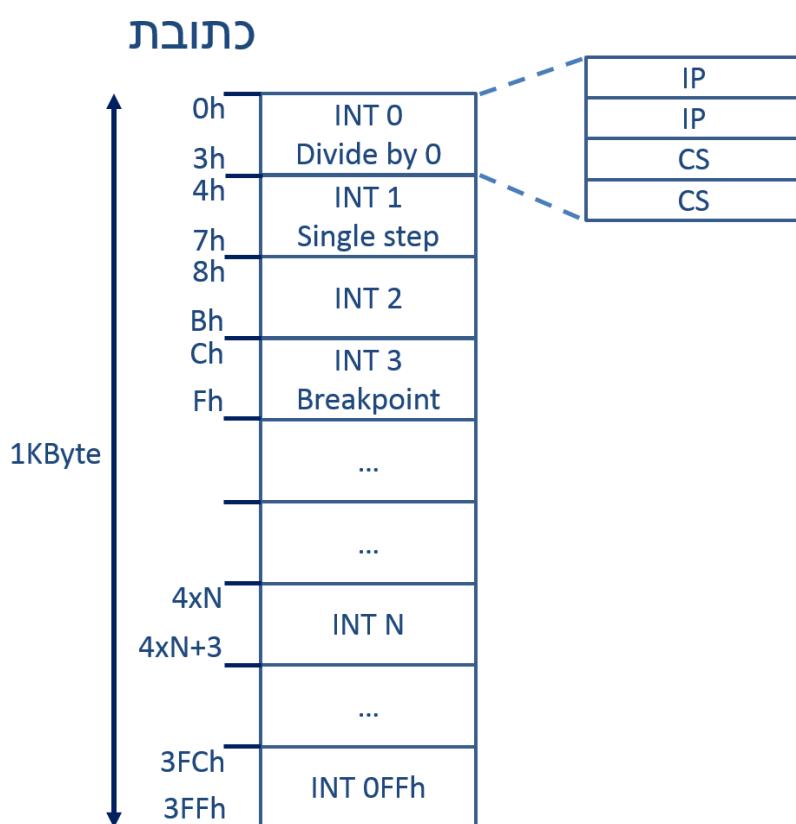
הודענו למעבד שהוא צריך לשרת את פסיקה מספר h.21.

כעת המעבד ניגש למלאכת תרגום מספר הפסיקה לכטובת בזיכרון. לטובה פועלת התרגומים, המעבד פונה לטבלת תרגום, ששומרה אצל 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 (בטים בסך הכל).

איך ה-IVT עוזר למעבד לתרגם את מספר הפסיקה לכטובת 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 הסגןט של ה-ISR. כך למשל, ארבעת הבטים הראשונים ב-IVT הם הכתובת של ISR של פסיקה מס' 0. ארבעת הבטים הבאים אחוריים 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 לkapo'ז אליה. דוגמה: פסיקה 21h מיתרגרמת למקום 84h בזיכרון. המעבד ניגש למקום .21h, שנמצא כموבן ב-IVT, וקורא ארבעה בתים, שמוררים לו לאיזו כתובה לkapo'ז לביצוע ה-ISR של פסיקה 21h.



מבנה ה-IVT בזיכרון

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, אז הוחלפה על ידי מערכת הפעלה Windows.

מערכת הפעלה מספקת לנו שירותים שונים. בין היתר היא מקשרת בין רכיבי החומרה במחשב לבין אפליקציות – תוכנות ומשחקים שרוצים על המחשב שלנו. כך היא חוסכת עבודה למי שמתכונת אפליקציות, שבמוקם לגשת שירותים לרכיבי החומרה, מקבל שירותים מערכות הפעלה. יכולת זו גם מונעת התנגשות בין אפליקציות לצורכי השתמש בהם רכיבים באותו זמן. לכן, אחד מהדברים שמערכות הפעלה מודרנית כוללת הוא אוסף של ISR'ים, שיעודם לעבוד מול רכיבי חומרה שונים.

לISR'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 הפסיקות שמוקצחות לדOS בתוכן ה-IVT

אחד מהפסיקות היא 21h, שמודדרת כפסקת שירות של מערכת הפעלה. כתע, כל פעם שנרצה שירות של מערכת הפעלה נוכל לבצע זאת על ידי השורה `int 21h`. איך אפשר בעזרת פסיקת 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.

ספרט, הרегистר ah מחזק את סוג השירות שאחנו רוצים.

באתר <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.

קליטתתו ממקלדת – AH=1h

כדי לקבל בעזרה int 21h שירות של קריאתתו ממקלדת, נשים בתוך ah את הקוד "1". כך:

```
mov ah, 1
int 21h
```

התו שיופיע ייטען בתוך al.

יש לציין ש-al יכול את ערך ה-ASCII של התו שהוקלד. לדוגמה, אם המשתמש הקליד "2", ah לא יכול, אלא 32h (כפי אפשר לראות 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
001 ☐ <soh>	017 ▲ <dc1>	033 !	049 1	065 ☑	081 Q
002 ☑ <stx>	018 ♫ <dc2>	034 "	050 2	066 ☒	082 R
003 ♥ <etx>	019 !! <dc3>	035 #	051 3	067 ☑	083 S
004 ♦ <eot>	020 ¶ <dc4>	036 \$	052 4	068 ☒	084 T
005 ☈ <eng>	021 § <nak>	037 %	053 5	069 ☑	085 U
006 ☉ <ack>	022 – <syn>	038 &	054 6	070 ☒	086 V
007 ☊ <bel>	023 ¶ <eth>	039 ,	055 ?	071 ☑	087 W
008 ☋ <bs>	024 ↑ <can>	040 <	056 8	072 ☒	088 X
009 <tab>	025 ↓ 	041 >	057 9	073 ☑	089 Y
010 <lf>	026 <eof>	042 *	058 :	074 ☒	090 Z
011 ☃ <vt>	027 ← <esc>	043 +	059 ;	075 ☑	091 [
012 ☄ <np>	028 ↴ <fs>	044 ,	060 <	076 ☒	092 \
013 <cr>	029 ↵ <gs>	045 –	061 =	077 ☑	093]
014 ★ <so>	030 ☁ <rs>	046 .	062 >	078 ☒	094 ^
015 ☆ <si>	031 ☂ <us>	047 /	063 ?	079 ☑	095 _
					111 o 127 ☈

כדי ש-al יוכל ממש, 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 שקולעתתו מהמשתמש ושומרת את ערך ה-ASCII שלו בתוך al:

IDEAL

MODEL small

STACK 100h

DATASEG

CODESEG

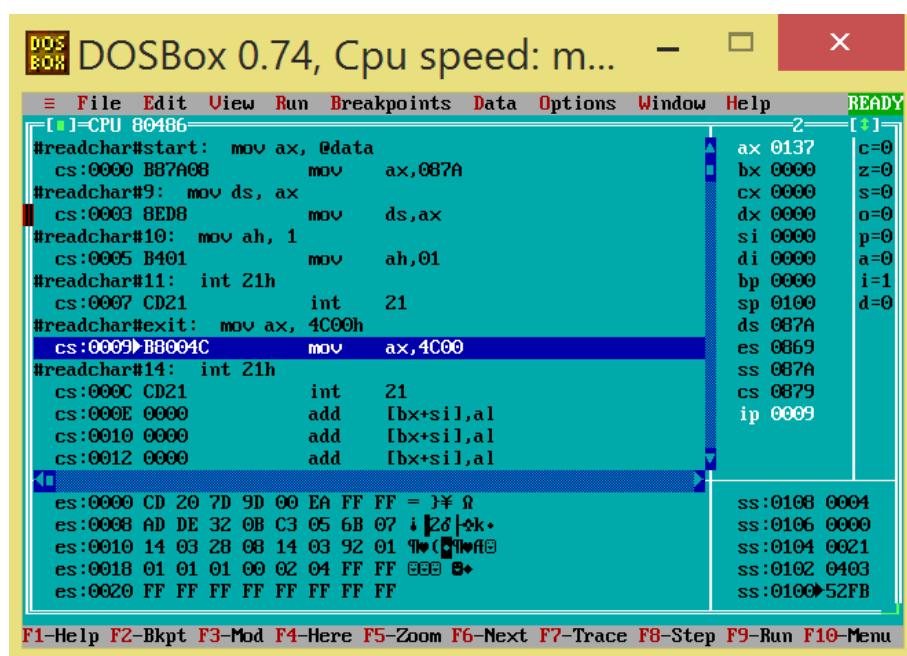
start:

```
    mov ax, @data
    mov ds, ax
    mov ah, 1
    int 21h
```

exit:

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

END start



המשתמש הזין את התו 7 (קוד ASCII 37h)

הקוד הועתק לתוך al

תרגיל 11.1: קליטת تو ממקלדת



- א. כתבו תוכנית שקולטת שני תוים ובבודקת לאיזה تو יש ערך ASCII גדול יותר.
- ב. כתבו תוכנית שקולטת تو ממקלדת ובבודקת אם מדובר בספרה (כלומר ערך ה-ASCII נמצא בטוחה עברית ה-ASCII שבין 0 ל-9).
- ג. כתבו תוכנית שקולטת עשרה תוים ממקלדת, ואם כולם ספרות התוכנית מחשבת את סכום הספרות לתוכן הריגיסטר 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 לערך התו האחרון שנשלח להדפסה.

קיימים שני קודי ASCII שימושיים, שאינם מדפסים תוים למסך אלא נתונים הוראות בקרה:

1. קוד 10, או 0Ah – Line Feed – מורה להתחליל שורה חדשה, הסמן נמצא באותה עמודה בה הייתה בשורה היישנה.

2. קוד 13, או 0Dh – Carriage Return – 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

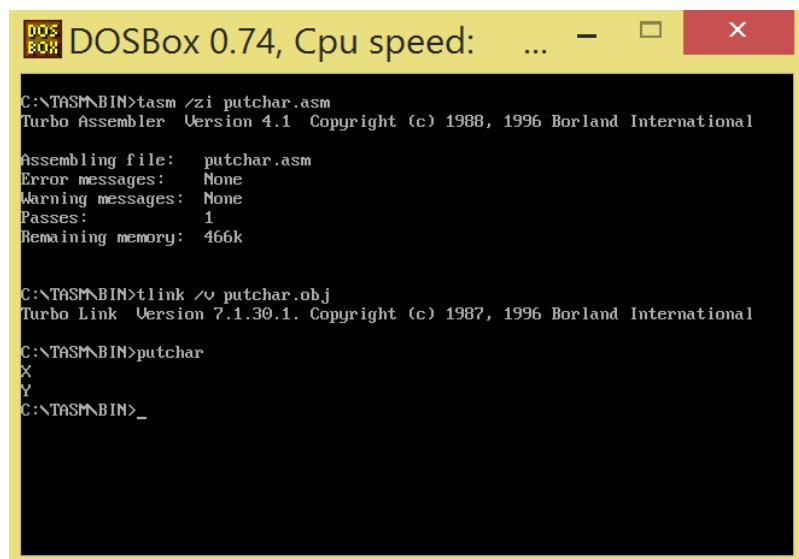
 int 21h

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 אחת, ואת המילה 'WORLD', تو אחרתו, בשורה מתחילה.
- 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 כרך את ההפרש בין המספר השני לראשון.

הדפסת מחרוזת למסך – AH=9h

כדי להדפיס מחרוזות למסך אנחנו צריכים קודם כל ליצור מחרוזת, שמשמעותה בתו '\$' (זה הסימן ל-SR) להפסיק את הדפסה למסך – אחריה יודפס כל הזכרונות...). לדוגמה:

```
message      db      'Hello World$'
```

לאחר מכן טענים לתוך dx את האופט של המחרוזת:

```
mov      dx, offset message
```

נותר לנו רק לחת את הקוד הנוכחי ולהפעיל את הפסקה:

```
mov      ah, 9h
```

```
int      21h
```

ירידת שורה בסוף המחרוזת:

שיטה נוחה לכתוב מחרוזת שבסוף הדפסה שלה מתחילה שורה חדשה, היא להגדיר את תוכו 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
```

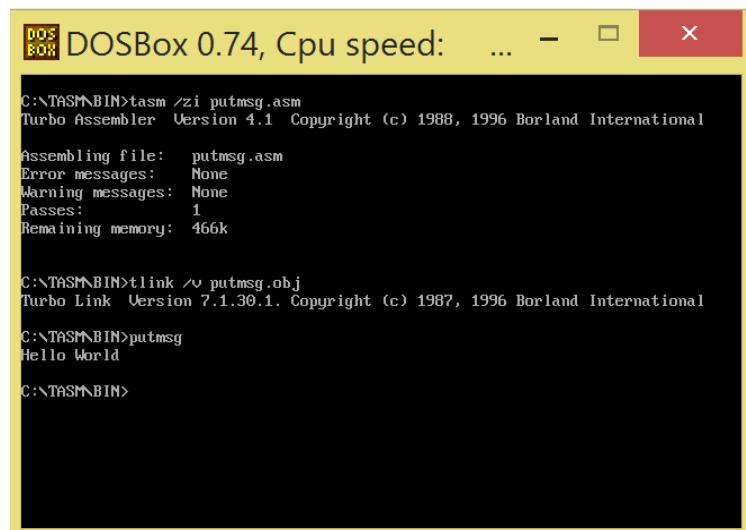
```

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 הודעה: 'Enter a digit' ואז קולעת 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, ובשורת הבאה כותבת את שמהם. לדוגמה:

I like to write assembly code

Barak

קליטת מהרוזת תווים – AH=0Ah –

כדי לקבל מהרוזת תווים מהמקלדת, צריך קודם כל להזכיר בזיכרון מקום לקלטת התווים- מקום זה נקרא "באפר" (Buffer). בתוך הבית הראשון של הבאפר צריך להכניס את מספר התווים המקסימלי שאנו מאפשרים לקלוט. לאחר מכן מכנים סימן dx את האופט של הבאפר (יחסית ל-ds) ואז מפעילים את Int 21h כאשר ah שווה לקוד 0Ah. כשההשתמש סימן

להקליד תווים ולחץ על מקש ה-Enter, הבית השני בבאפר יכול את מספר התווים שהוכנסו, ואילו התווים עצם יהיו מהמקום השלישי והלאה.



דוגמה לתוכנית שקולטת עד 20 חווים מהמשתמש:

(שים לב - אנחנו מזמנים 23 בתים, כיון שני בתיים בתחילת הבאפר תפוסים על-ידי מספר התווים המקסימלי 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 הtekסט שהוכנס על ידי המשתמש:

```
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.oj
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		

Memory Address	Value	Memory Address	Value
ds:0000	14 13 48 65 6C 6C 6F 20	ss:0108	0005
ds:0008	77 6F 72 6C 64 21 20 2D	ss:0106	0000
ds:0010	world! -	ss:0104	0025
ds:0018	BarakJ	ss:0102	0403
ds:0020		ss:0100	52FB

תרגיל 11.4: קליטת מחורזת תווים



כיתבו 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 שמהם הפרטי.

11.6: כתבו תכנית שקולטתתו בודד מהקלדת ומציג אותה 10 פעמים בשורה חדשה.

11.7: כתבו תכנית המציגת על המסר את הוצאות הבאות (כל צורה – תכנית חדשה!)

א. *****	ב. *****	ג. *****	ד. ***	ה. ***	ו. *	ז. *	ח. *	ט. ****	י. ****
* ***	* ***	* * *	* * *	* * *	* * *	* * *	* * *	* * *	* * *
**	**	* * *	* * *	* * *	* * *	* * *	* * *	* * *	* * *
*	*	* * * *	* * * *	* * * *	* * * *	* * * *	* * * *	* * * *	* * * *

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 מהקלדתו ועובדו כל תו מציג הודעה מתאימה:

Small letter, Capital letter, Number, Other

11.11: כתבו תוכנית הקולטת רצף של 5 תווים מהקלדת "קוד סודי", במקומות כל תוו שנקלט מוצגתו "*". על התוכנית לבדוק האם הקוד הוא: "12345" ואם כן - להציג הודעה מתאימה. אחרת, להודיע על הטעות ולאפשר עוד 2 נסיונות.

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` במקרה במידה והתכוון הסתימה בהצלחה, ולכנן יציאה סטנדרטית מתוכנית נראית כמו שורות הקוד המוכנות לכם מ-`base.asm`:

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

קריאה השעה / שינוי השעה – AH=2Ch ,AH=2Dh (הרחבה)



לצד המעבד קיימ רכיב חומרה שתפקידו לתחזקון למעבד – הטימר. הטימר שולח למעבד אותן חמומי כל 55 מילישניות (0.055 שניה), או בערך 18.2 פעמים בשניה. לכן 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` מאות השניה יועתקו לתוך `al`.



שימוש לב שערק מאות 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 בריגיסטרים השונים:

- `ch` יועתק לתוך הדקות.
- `dl` יועתק לתוך מאות השניה.
- `ah` יועתק לתוך השעות.
- `dh` יועתק לתוך השניות.

תוכנית לדוגמה – קריאת הטימר והדפסת השעה למסך:

התוכנית הבאה משתמשת ב-int 21h כדי לקרוא את ערכו של הטימר. לאחר מכן נעשה שימוש בפראצדורה שומרה 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 שייתכן ששעון השניות ישתנה 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 מסווג exception מתרחשת אוטומטית כתוצאה מאירוע תוכני שהתוכנה לא יוזם אותו – בדרך כלל אירוע חריג, וכך הן 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 בזאג'ר debugger ומריצים את הקוד שלנו שורה אחרי שורה. צריך להיות מנגנון כלשהו שעוצר את ריצת התוכנית אחרי כל פקודה. מנגנון זה הוא פסיקת חריגה, שמשנה את ערכו של אחד הדגלים בריגיסטר הדגלים. המעבד יודע האם דגל זה מופעל, עליו לעזור לריצה עד לקבלת פקודה "המשך".

פסיקה זו, של הריצת צעד בודד, היא .int 1h.

דוגמה נוספת היא השימוש בdebugger breakpoints בתוכן ה-IVT. כל breakpoint גורם למעבד להריץ את הפקודות בלי עצירה עד להגעה לשורת קוד שיש בה breakpoint וואז מתבצע עצירה עד לקבלת פקודה "המשך".

פסיקה זו, של ביצוע breakpoint, היא .int 3h.

פסיקות תוכנה – Traps

בניגוד ל-exceptions, שתרחשות אוטומטית, מי שיוזם פסיקות תוכנה הוא מי שכוחב 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 – שימירת המיקום והדגלים במחסנית, חישוב כתובת ה-ISR בעזרת ה-IVT וקבעה 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 ויזורו – אפשר פשוט להשתמש בפראוצ'ורה.

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 ההפעלה, היא לא יודעת את הכתובות של הפ逻צדרות של מערכת ההפעלה. תיאורטיות היינו יכולים לפתור את הבעיה זו אם היינו מengl'ים את התוכנה שלנו יחד עם הקוד של מערכת ההפעלה, אבל אופציית זו לא מעשית. לכן בפועל מערכת ההפעלה מגירה 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.

נתהיל מ-ISR ריק:

```
proc SimpleISR far
    ...
    iret
endp SimpleISR
```

מוסיף ל-ISR שלנו את ההודעה אותה אנחנו רוצים להדפס.

שים לב, שההודעה צריכה לבוא אחרי פקודת ה-`iret`, אחרת המעבד עלול להגיע למצב שהוא מנסה לתרגם את ההודעה ל-`opcodes` ולהריץ אותם, מה שעலול להיגמר בקריסה.

```
proc SimpleISR far
    ...
    iret
    message db 'Hello World$'
endp SimpleISR
```

מוסיף קריאה ל-`int 21h`. הקוד שמדפיס מחרוזת 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 שגוראת לו עלולה לא לתקן כמו שצורך אחרי שהשליטה תוחזר אליה.

```

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 שבחרנו דוקא במקום זה, היא שאנחנו רוצים לשתול את ה-ISR שלנו במקום שאיןו 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`, שירותת שזהה הזמן המתאים לסקור אותו. קוד `AH=25h` מכניס כתובת לתוך ה-IVT. מספר הפסיקה צריך להיות בתוך `al`, וככתובת ה-ISR בתוכה `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 ah 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 הכנה לפרויקט הסיום - פיצוח צופן הזזה

נבער תרגיל שהוא חוזה על החומר העיקרי שלמדנו עד כה (פרוצדורות ושימוש בפסיקות DOS לקלט ופלט) ותוך כדי נלמד מעט על צפנים ופיצוח צפנים.



נתחיל בהסביר קצר על צופן הזזה. צופן הזזה הוא צופן עתיק בו כל אות מוחלפת באות אחרת, שנמצאת למרחוק קבוע מהאות המקורית (למרחוק הזזה). לדוגמה צופן הזזה 1- האות a מוחלפת באות b, האות b מוחלפת באות c וכו' עד שmagim לאות z, אשר מוחלפת באות a. לדוגמה, בצופן הזזה במרחק 3, המילה cat מוחלפת במילה fdf.

1. כתבו פרוצדורה שמקבלת מהירות ומרחוק הזזה, ומיצפינה את המחרוזת לפי מרחוק הזזה. תוויה רוחה יש להשאיר כמו שם.

2. לפניכם העמוד הראשון מתוך הספר הנפלא Anna Karenina מאת טולסטי. כדי להקל על תהליך ההצפנה הכתוב מובא רק עם אותיות קטנות ולא סימני פיסוק. הצפינו אותו בעורצת צופן הזזה. שימו לב לכך שבסוף העמוד ישנו סימן '\$'. סימן זה נועד גם להקל עליכם למצוא את סוף המחרוזות 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 ומחזרה כמה פעמים מופיעתו בתוך המחרוזת.

4. באמצעות הפרוצדורה שכתחבם, סיפו כמה פעמים מופיע כל אות באلف בית הלועזי בטקסט המוצפן. פעולה זו נקראת ניתוח תדייריות. הדפיסו למסך את התוצאה עבור כל אות.

5. צרו פרוצדורה שמקבלת את ניתוח התדייריות שעשיהם, ובאמצעות טבלה של ניתוח תדייריות שבשפה האנגלית מהליטה מה הייתה אותן המקורית שהוצנה. מצורפת טבלה של ניתוח תדייריות אותן 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 לאחוזים 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'ים.

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 ו איך שותלים אותו ב-IVT.

עסקנו בפעולות 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

כשענסנו בנושא הפסיקות ראיינו שקיימות פסיקות DOS שבמציאות פעולות שונות מול רכיבי חומרה, לדוגמה קריית תוו מהמקלדת. כמו כן למדנו להשתמש בכמה פסיקות DOS שימושיות. עם זאת, לא ענסנו בשאלת איך המידע מהתקני החומרה מגיע אל המעבד? איך, לדוגמה, הקשה על המקלדת גורמת להופעתתו 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 צורך פסיקות חומרה ובאיזה דרך מגיעות פסיקות החומרה אל המעבד.
- פורטים I/O – תאים מיוחדים בזיכרון ששמשים לעובודה מול התקני קלט / פלט.

לאחר מכן, נעבור לדוגמה מעשית על עובודה מול התקני חומרה ונפרט אודות פעולות 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 הייצוניים 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, לחתת את התו שהוקלד במקלדת ולהדפיס אותו למסך, להזין 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 זמן יקרה אירוע חומרה ולהתכוון אליו. אם כך, איך המידע שמאגייח מהתקנים הייצוניים מועבר לתוכנה בזמן ריצה?

קיימות 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 אחרי התקן, בצורה מעגלית, ובודק אם יש



התיקן חומרה שצורך שירות. המקלדת 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 שהוא גישה פשוטה יחסית למימוש, ולא צריך רכיבי חומרה נוספים שיעזרו לנו בתהליכי מפסיקות חומרה, שנגיעה אליהן מיד). החיסרונו של Polling הוא שהתקן 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 עזתי, מקרים שבהם הפעולה התקינה של תוכנה שלנו תלואה בזמן שלוקח לה לשרת אירועים מהתקן חומרה כלשהו. לדוגמה, במערכות שבמציאות מסחר אלקטרוני בגיןheiten, 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 גובה של טיפול בפנויות מהטיימר. דוגמה אחרת – נניח שיש מערכת שמשגרת טיל ברגע שימושו לחץ על כפתור במקלדת, במקרה זה מובן של מהירות הטיפול באירוע הלחיצה על המקלדת יש חשיבות רבה. פתרון פשוט הוא להעלות את קצב התשאול של התקני החומרה. נניח שבמקרה לשאול את המקלדת 10 פעמים בשניה אם יש מыш לוחז, התוכנה תישאל את המקלדת 1000 פעמים בשניה אם יש מыш לחוז. שנית זה יוריד את הזמן שהמקלדת ממתינה לשירות מעשרית 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. כמובן התוכנה שלנו תגרום לכך שהמעבד יקצת חלק ממשמעותי מהזמן לביקורת התקני חומרה ולא לפועלות אחרות שאנחנו רוצים שיבצעו.



גישה *Polling* – אחת לזמן מוגדר, המורה שואלת כל תלמיד אם יש לו שאלה. כל עוד המורה לא פונה אליהם, התלמידים מהלכים.

הגישה השנייה היא פסיקת חומרה, שנקראת **איןטרupt** (Interrupt). מעכשיו כשנכתוב "איןטרפט" נכוןן לפסיקת חומרה. כשייש להתקן החומרה צורך בשירות של המעבד, התקן החומרה שולח אותה חשמלי שמנגיע אל המעבד וגורם לו לעצור לאחר סיום הפעולה הנוכחית שלו ולבצע קוד, ISR, שנכתב מראש לטיפול באירוע החומרה. בסיום ISR התכנית המשיך ברכיצה. היתרון של Interrupt הינו שה-ISR גם שומר על שיחוי נמור (בקשות של התקני חומרה מטופלות עם שיחוי נמור יחסית) וגם חסכו במשאבי מעבד – כל עוד לא מגיע איןטרupt, המעבד פשוט מרין את הקוד שנקבע לו להריצן בלי לדאוג לגבי התקני החומרה.

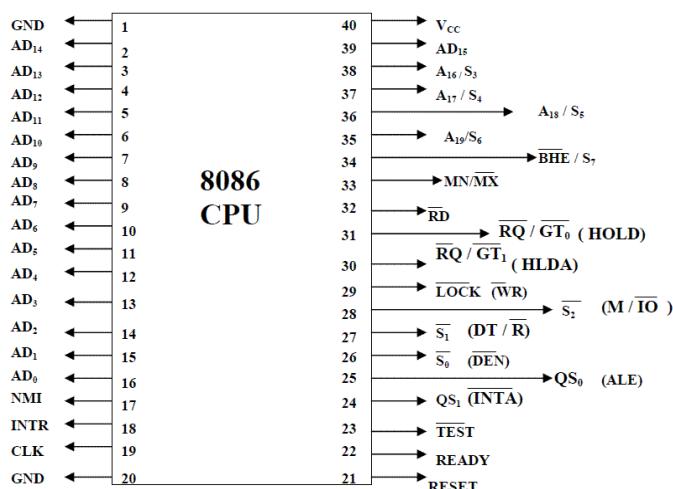




גישת תלמיד שיש לו שאלת מרים יד. המורה עוצרת את השיעור, עונגה לתלמיד וממשיכה את השיעור מהמקום שהפסיקה אותו.

כמו שתלמיד שיש לו שאלת מרים יד, רכיב החומרה שצרכיך טיפול שולח אותה המשמעי. לمعالג יש "רגליים", או "פינים" – חוטי מתכת דקים שמחברים אותו אל העולם החיצוני. שניוי המתה החשמלי על אחת הרגליים גורם למעגל להניע תהליך של אינטראקט.

Pin Diagram of 8086



דיagramת פינים של מעגל ה-8086

핀 מס' 18, שמוסמן "INTR", מקבל אינטראקטים מרכיבי חומרה חיצוניים

בקר האינטראפטים – PIC

כפי שראינו באIOR של מעבד ה-8086, יש למעבד רgel אחת שמיועדת לאינטראפטים. בעזרתו ניתן להזמין מעבד מכבלי אינטראפטים מכל ה התקנים שקשורים אליו. איך בעזרתו יכול רgel אחד המעבד יכול להיות קשור למספר רב של התקני חומרה?

Programmable Interrupt הוא בקר האינטראפטים, שימוש באינטראפטים מציריך רכיב חומרה נוספת. רכיב זה הוא **PIC Controller**, או בקיצור **PIC**.

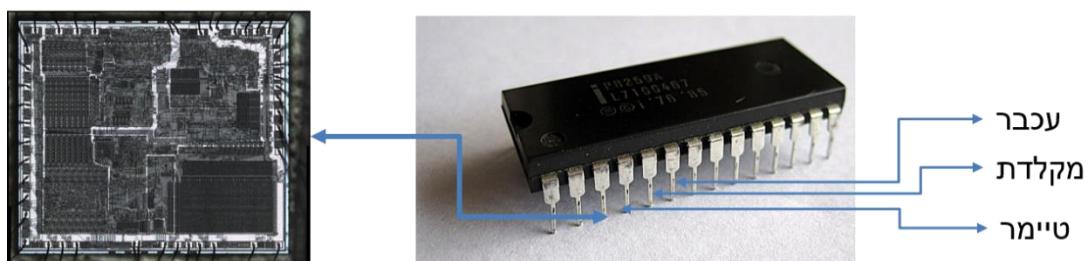


תמונה של PIC מדגם 8259A משנת 1976, מהסוג ששימש ליבורנה עם מעבד ה-8086.

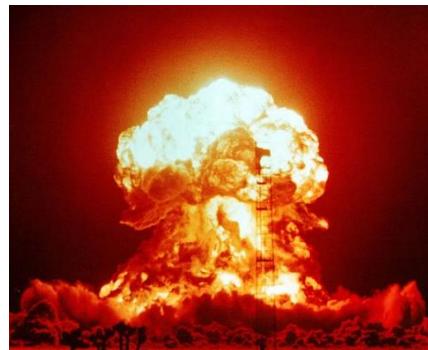
כפי שניתן לראות בתמונה, ל-PIC יש אוסף של רגילים. הרגילים שמסומנות IR0 עד IR7, בסך הכל 8 רגילים, כאשר כל אחת מהם יכולה להיות מקושרת לרכיב חומרה בודד. כלומר PIC מדגם 8259A מסוגל לעבוד עם עד 8 רכיבי חומרה במקביל. לדוגמה – רgel IR0 מחוברת לטימר, רgel IR1 מחוברת למקלדת, רgel IR6 מחוברת למוניטור דיסקטים, ורגל IR2 מחוברת לעכבר (לא באופן ישיר, אבל לצורך הפשטות נקבע להגדיר 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 במסלול שעושה אינטראפט. ניקח בתוור דוגמה אינטראפט של המקלדת. האות החשמלי עובר מהמקלדת אל רgel IR1 של PIC. כתגובה, ה-PIC מוציא אות חשמלי מרgel INT, שמגיעת 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 חומרה. דמיינו מערכת שמקושרת לשני התקני חומרה: מקלדת, וכור גרעיני. המשתמש הקיש על המקלדת והכור גרעיני מדוחה על בעיה... במאי המעבד צריך לטפל קודם?



טיפול לא מ[hash]ב בשיטת כל הקודם זוכה. ה-PIC מחזיק תור של אינטראפטים שמחכים לטיפול. אם הגיע אינטראפט חדש, הוא לא יזחף לסוף התור אלא יכנס 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 **charmli** שמסמן שיש אינטראפט.
- שולח מידע שקורא לאינטראפט לאזור מיוחד בזיכרון המעבד, אזור שנקרא **Port O/I** או בקיצור – פорт.
- מפסיק לשולוח אינטראפטים למעבד.
- ממחכה לאות **charmli end of interrupt** מהמעבד, ובינתיים שומר בתור אינטראפטים חדשים שמגיעים אליו מרכיבי חומרה.
- חוזר לשולוח אינטראפטים למעבד.

אובדן אינטראפטים

זכור, 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 שהמתכנת קבוע אותו בקוד), פסיקות חומרה מגיעה בזמן לא צפויים מראש. מה קורה אם בזמן שהמעבד מטפל באינטראפט מגיע אינטראפט נוסף? לדוגמה – לחצנו על המקלדת והעכבר בהפרש זמן קטן ויצרנו אינטראפט אחד מהמקלדת ואינטראפט אחד מהעכבר. האינטראפט מהעכבר מתרחש בזמן שבו המעבד כבר מטפל באינטראפט של המקלדת וחומרם נוספים. האם האינטראפט של העכבר "ילך לאיבוד"?

ובכן, כפי שציינו קודם, ה-PIC מרכז את האינטראפטים של כל רכיבי חומרה ושומר אותם אצלו בזוגמה שלנו, עם סיום האינטראפט של המקלדת, ה-PIC יעביר למעבד את האינטראפט של העכבר. התור של ה-PIC עוזר במצב שבו מספר

איןטרפטים הגיעו בפרש זמינים קצר. לעומת זאת, אם המעבד לא מצליח לעמוד בקצב הפסיקות, התור של ה-IC יילך ויתארך עד לנקודה מסוימת בה לא יהיה ל-IC מקום לשמר אינטרפטים חדשים והם ילכו לאיבוד.

דוגמה תיאורטית – שעון המערכת, הטימר, שולח עדכון שעה כל 55 מילישניות. נדמיין מעבד איטי, שלוקח לו יותר מ-55 מילישניות לבצע את אינטרפט הטימר. לאחר זמן מה אינטרפטים מהטימר ילכו לאיבוד ופעולתו התקינה של המעבד תשתבש.

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 לכתובתו.

למעשה, למעבד יש שני פסי מענים. נוסף על פס המענים המשמש לגישה לזכרון, משפחת ה-86x80 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 מאוד 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 פקודת mov, משתמשים בפקודות in ו-out.
- פס הבדיקה דואג שפקודות **in** ו-**out** יעבירו כתובות על פס המענים של ה-O/I (לעומת פקודות mov, שמתייחסות לפס המענים של הזיכרון).

פקודות out, in: פקודת **in** משמשת לקריאה מפורט, ופקודת **out** משמשת כתיבת הפורט. פקודת **in** מעתקה זיכרון בגודל מילה או בית מפורט מסוים אל ax או al. פקודת **out** מעתקה זיכרון בגודל מילה או בית מ-ax או al אל פорт.

- 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 פорт בתחום שמעל 255, במקום פорт משתמשים ברגיסטר **dx**.

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, 00000011b      ; change some bits
out   61h, al           ; copy the value in al back to the port
```

משמעות הביטים ששינו איננה חשובה כרגע, (גיאע אליה בהמשך, כשהנוסף בפרק יקיי סיום). החשיבות היא בתהיליך – אנחנו קוראים באמצעות פקודה **in** את הסטוס של פורט כלשהו, משנים בו כמה ביטים וכותבים חורה את הסטוס אל הפורט עליידי פקודה **out**.

דוגמה לעבודה מול פורט שמספרו גדול מ-255:

```
mov  dx, 300h
```

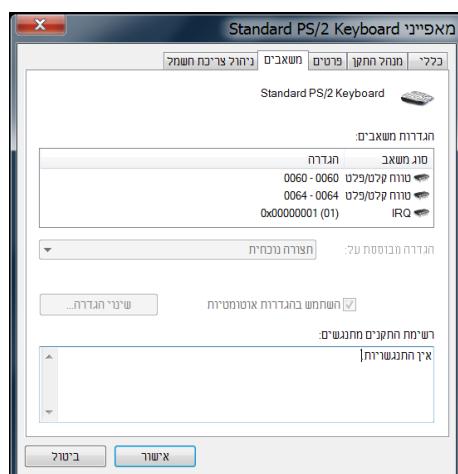
```
in   al, dx
```

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 ההתקנים (חיפוש תכנית-<Device manager>).



בתוך מנהל ההתקנים, נסמן את המקלדת ונ קיש על עכבר ימני – נבחר "מאפיינים" (Properties). בתוך מאפיינים, נבחר "משאים". 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 זמין לשימוש בזיכרון המעבד. לפני שנצלול אל השלבים 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-I/O של המעבד והמידע שמגיע ממנו מועתק אל פורט קבוע בזיכרון, פורט **60h**.
3. פסיקת חומרה, אינטראפט, אוספת את הד-codes **60h** ומעתיקה אותו אל באפר מיוחד שmagדר לשימורת המידע מהמקלדת. מספרו של האינטראפט הוא **9h** והשם של הבאפר של המקלדת הוא **Type Ahead Buffer**.



כמו שאמרנו במאמר לנווט הפסיקות, למי שמתכוна באסmbly יש יותר דרך אחת לעבוד מול חומרה. ספציפית לגבי המקלדת יש שלוש דרכים לעבוד מולה:

1. לעבוד יישירות מול הפורט של המקלדת בשביל לקלל את המידע על המקשים שהוקשו.
2. להשתמש בפסיקה של BIOS (תזכורת – BIOS היא חבילת תוכנה של אינטל, שנתקלה בקצתה בראשית הפרק הקודם. אחד מתפקידיה הוא קישור אל רכבי חומרה). ל-BIOS יש פסיקה מס' **16h** שנותנת לנו לבצע פעולות שונות מול המקלדת – כמו קליטתתו – בפשטות יחסית.
3. להשתמש בפסיקה של DOS, פסיקה **21h**, שכבר הכרנו חלק מהשירותים שהוא נותן בנושא העבודה מול המקלדת.

מקוצר היריעה, ניתן הסבר תמציתי בלבד לנושא 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 **scan codes**. 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 **h80**, כלומר בית מס' 7 של ה-**code** מקבל 0 בלחיצה ו-1 בשחרור, יתר הביטים נשארים קבועים בין לחיצה ושחרור של אותו תוו (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 ה-**ESC** יגרום לשילוחת קוד **81h**. באופן כללי, כל פעולה של לחיצה או שחרור של מקש גורמת לשילוחת **scan code** יחד עם אינטראפט. להלן התהליך שמתתרחש עם הלחיצה על המקלדת:

1. רכיב חומרה שנמצא במקלדת מעביר לפורט **60h** את ה-**code** של הלחיצה / שחרור של המKeySpec.
2. ה-**PIC** מקבל דרך **IR1** אינטראפט מהמקלדת.
3. ה-**PIC** שולח למעבד אינטראפט, שאומר למעבד שיש מידע בפורטים של המקלדת. האינטראפט שמודיע הוא **int .9h**.
4. כתגובה לאינטראפט, המעבד מרים **ISR** שמתפלב באינטראפט **9**.

.5. ה-ISR מטפל בהעתיקת 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.

באור 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 באור המקלדת, *Type Ahead Buffer*. הבאור מוגדר בצורה הבאה:

- מיקום A 0040:001A – מצביע על ראש הבאור

- מיקום C 0040:001C – מצביע על זנב הבאור

- מיקום E 0040:001E – 16 מילים (words)

כמו שאפשר לראות מוגדל הבאור, הוא יכול להחזיק עד 16 הקלדות של המקלדת (משום שככל הקלדה מתורגמת לשני בתים – *scan code*, ASCII *code*). מה קורה אחרי 16 הקלדות? הבאור הוא באור מעגלי – יש מצביע לראש ומצביע לזנב שלו. בכל פעם שרץ אינטרפט מקלדת, הוא מגדיל את הערך של זנב הבאור ב-2. אם הערך יוצא מתחום של הבאור, הוא מוחזר לתחילת הבאור. ככה עובד באור מעגלי.

כמו שיש מנגנון שדוגג להכנס נתונים לבא/or, יש מנגנון שדוגג להוציא נתונים מהבא/or. זה פסיקה BIOS, שمعתיקה את המידע מהבא/or במיקום ראש הבא/or מצביע עליו, ומעלה את ערכו של מצביע ראש הבא/or ב-2.

כך הראש של הבא/or "רודף" אחרי הזנב של הבא/or במסלול מעגלי שגודלו 16 מילים. יש פסיקת מקלדת שדוגגת לקדם את הערך של הזנב ויש פסיקת BIOS שדוגגת לקדם את הערך של הראש. מה קורה אם הראש "משיג" את הזנב? המשמעות של המקרה זה היא שהבא/or 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 שמיילאתם את בא/or המקלדת, הקשתם 15 פעמים בדיק על המקלדת (בקלדה הבהה הראש יעקו את הזנב) ובגלל שהמחשב תקוע, הפסיקה של ה- *BIOS* לא מրוקנת את הבא/or.

הסיבה שנתנו את כל הסקירה זו, היא כדי להגיע למסקנה שניקיי הבא/or הוא פועל הכרחי וצריך לדאוג לכך שהבא/or לא יהיה מלא – לאחרת אנחנו מאבדים את יכולת לקבל תווים חדשים מהמקלדת. זה מה שצורך לזכור, ומיד נשימוש במידע זהה.

עכשו, אחרי שסקרנו את התיאוריה של שרשרת הפעולות שגורמות להקלדת 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

בקצרה, המעבד עובד מול שני רכיבי חומרה, microcontrollers. אחד נמצא בתחום המקלדת, שני נמצא על לוח האם של המעבד. ישנו שלושה פורטים שימושיים בעובדה עם המקלדת:

1. פорт 60h משמש לתקשורת בין המעבד לבין ה-microcontroller של המקלדת. עוברים בו נתונים מסוימים שונים, הנתון חשוב לנו הוא ה-scan codes שמגיעים מהמקלדת.

2. פорт 64h משמש לתקשורת בין המעבד לבין ה-microcontroller של לוח האם. הם מעבירים בו נתונים חזותיים בקרה, שדרכו אפשר לדעת אם יש scan code בפורט 60h.

3. פорт h 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 ולכון נגיעה אליו בהמשך (כשנרצה לעבד מול כרטיס הקול).

קוד שבודק בפורט 64h אם הגיעו אליו נתונים חדשים (הבית השני מחזק את הסטטוס):

```
in    al, 64h      ; Read keyboard status port
```

```
cmp   al, 10b     ; Data in buffer ?
```

את הקוד הזה ניתן להכניס ללולאה, שרצחה עד שתנאי העצירה – יש מידע חדש מהמקלדת – מתקיים:

WaitForData:

```
in    al, 64h
cmp   al, 10b
je    WaitForData
```

לאחר שבדקנו שיש מידע חדש מהמкладה, נבצע קריאה מהמיוקם שבו המידע נמצא – פורט: 60h

in al, 60h

לABIי השלב האחרון, ריקון הבAPER, ניתן לבצע אותו עליידי קידום מצבייע ראש המкладה ב-2.

תוכנית דוגמה – קליטת הקלדות ובדיקה האם מקש ה-ESC נלחץ. שימושו ליב שבודוגמה זו אנחנו לא מנקים את באפר המкладה אחרי כל הקלדה. אחרי 15 תווים הבAPER כבר מלא – ברגע 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 עדין 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 שהקלדנו יותר מ-15 תווים, רק 15 תווים נשמרו בbaarך המקלדת

והודפסו למסך עם הייציאה מהתוכנית.

לעתים אנו צריכים לזהות לא רק איזה מקש הופעל במקלדת, אלא גם אם הפעולה היא לחיצה על המKeySpec 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 בפסקת BIOS

פסקת BIOS מס' 16h נותנת לנו כלים נוחים לבדיקת מצב המקלדת, קריאת התו שהוקלד (אם הוקלד) ו"נקיי" באפר המקלדת (כלומר שינוי מצביע הראש ומצביע הזנב של באפר המקלדת כך ש מצב הבאפר יהיה ריק, אין נתונים).

כדי לקרוא את התו הבא מתוכך באפר המקלדת, מפעילים את ah=0h עם קוד 16h. הפסקה מחזירה בתוך al את קוד ASCII של התו שנמצא בראש הבאפר ובתוך ah את הדסטוט ah scan code שלו. בנוסף, הפסקה "מנקה" את התו מהבאפר עליידי קידום ערכו של ראש הבאפר ב-2.

הבעיה היחידה עם הפסקה זו, היא שאם איןתו שמתינו באפר – הפסקה תחכה לו. כתוצאה לכך, אם אנחנו רוצחים לתכנת משחק שלא עוצר בהמתנה לפועלה של השחקן, הפסקה זו לבקשת לא מתאימה.

הפתרון הוא לשלב את הפסקה 16h עם קוד ah=1. במקרה זה, הפסקה מחזירה את הסטוטוס של המקלדת – 0 אם ישתו מוכן לקרוא, 1 אם איןתו מוכן. אם ישתו מוכן, ah ו-ah יקבלו את ערכי ASCII והדסטוט ah scan code של התו.

שילוב הפסקות אפשרי לנו:

- לדעת מתי יש מידע מהמקלדת (בלי לעזר את הריצזה ולהחות למשתמש).
- לקלות 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
```

התכנית הבאה מבצעת גם היא קליטת הקלדות מקלדת ויציאה 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 21h מנקה את הבאפר של המקלדת, ואז מבצעת טרייק נחמד – היא לוקחת הערך ששמננו באל כפרמטר, וMRIיצה int 21h עם הקוד הזה. אנחנו נראה דוגמה בה al=7h, כלומר – לאחר ניקוי באפר המקלדת, הפסיקה תעבור לקוד 7h, שהוא קוד של קליטתתו מהמקלדת ללא הדפסת התו על המסך. באופן זה אנחנו מבצעים שתי פעולות:

- 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 בROUTRN מבחן פשוטות תכונות. עם זאת ישנו שני חסרוןות:

החיסרון הראשון, היותר ברור מאליו, הוא שהתוכנה עוצרת בזמן שהוא מנסה לקלט מהמשתמש. זה יכול להיות בעייתי 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 אמוד 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 ה-`scan code` אל ה-`Type Ahead Buffer` שנמצא במיקום `0040:001Eh`.

כתבו תוכנית שקוראתתו ממקלדת (השתמשו ב-`int 21h` עם הקוד המתאים), הריצו את התוכנית ב-`TD` במצב `step by step`, וצפו בשינוי זיכרון במיקום של ה-`type ahead buffer`. בטור קלט, הכניסו את התו 'a' ומצאו ב-`scan codes` את `type ahead buffer` שלהם.

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 הkowski 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 הkowski 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 תשתחמם בפסיקה **h16** 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
שמשמייע 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 PIC בקר האינטראפטים, מבצע את עבודת הקישור בין התקני החומרה לבין המעבד.

למדנו על פורטים, 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 שבסיומו של דבר התו שנלחץ מופיע בזיכרון המחשב, ב-`Type Ahead Buffer`. ראיינו דוגמאות לעובדה מול המקלדת במגוון שיטות:

- עובדה ישירה מול הפורטים של המקלדת, `60h` ו-`64h`
- שימוש בפסיקות `int 16h`, BIOS
- שימוש בפסיקות DOS, `21h`, לניקוי באפר המקלדת

בפרק הבא נעסק בנושאים שימושיים לכנתיבת פרויקטי הסיום.

פרק 13 – כלים לפרויקטים

מבוא לפרויקטי סיום

זהו, כיסינו בפרק הקודמים את כל החומר התיאורטי שנכלל בתוכנית הלימודים. כמו כן ישנו עוד חומר תיאורטי רב הקשור לשפט אסמללי, אבל מדובר בעיקר על הרחבת נושאים שדנו בהם בקצרה. עכשו אנחנו מתקדמיים אל העבודה המעשית, אל הפרויקטים. פרויקט הסיום הוא ההזמנות שלכם לכתוב תוכנה "אמיתית" שמשלבת בין הדברים שלמדתם לדברים שימושיים אתכם, להציג שאתם מבינים את החומר היטב ולהוכיח את זה. הפרק הזה מיועד לкриאה וללימוד עצמי. הוא אינו מתיימר לכוסות באופן מלא את הנושאים אלא רק לתת הסבר ראשוני ומקורות ללמידה עצמי. קיימות לכך שתי סיבות עיקריות. הראונה, סיבה מעשית: במסגרת פרק אחד אין אפשרות לכוסות לעומק נושאים, שלא הרבה מהם פרק משל עצמו או אפילו ספר. הסיבה השנייה היא הרצון להכין לצורת העבודה בה תנתנו בעתיד – המידע שאתם צריכיםקיים אי שם, עליו לחשוף אותו, להגיאו 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 ותשקיעו פחות זמן בעיסוק בנושאים אחרים. כמובן, שם יש לכם עניין ללמוד 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 מוצאים את מלאו היכולות שלכם. אם הפרויקט הוא הרבה מעל ל-2000 שורות קוד, ייתכן שאתם לוקחים על עצמכם משימה שתיקח זמן רב מדי. אפשר לסוג את הפרויקטים לשלוש קטגוריות עיקריות:

- **משחקים – סנייק, פונג או כל דבר שעולה על דעתכם.** היתרונות בבחירה פרויקט שהוא משחק, הוא שהעבודה ש策ירית לעשوت בפרויקט די מוגדרת ובדרך כלל הפעלת התוכנה די אינטואיטיבית, דבר שמקל על תכנון הפרויקט.

- **אפליקציה קטנה – לדוגמה כלי נגינה (הקשה על המקלדת או העכבר גורמים להפעלת הכללי, המשמע צלילים ושינויי גרפיקה), מכונת מזקה (בחירה שירים שהוקלטו מראש), צייר (הוזת העכבר תוך כדי להיזה גורמת לצירור על המסך) או כל דבר שעולה על דעתכם.** היתרונות בפיתוח אפליקציה הוא שיש מקום רב ליצירתיות ואפשר לעשות מה שימושיים אתכם.

- **בעיות אלגוריתמיות – לדוגמה מימוש 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 הדעת התיאורטי תצטרכו ללמידה בכוחות עצמכם. אי לכך, כדאי שתיכנסו לנושאים כאלה רק אם אתם נלהבים ללמידה את**

כל הידע בעצמכם מה האינטרנט. 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

- גרפיקה:

- ייצירת גרפיקה בעזרת תוכנת ASCII
- ייצירת גרפיקה בעזרת הדפסת פיקסלים למסך
- יבוא של תמונה בפורמט BMP

- צילומים:

- המשמעת צילום בתדרים שונים

- שעון:

- שימוש בשעון למדידת זמן (השניה)
- שימוש בשעון לייצרת מספרים אקראיים

- ממתק משתמש:

- טיפים לעובדה עם מהמקלדת

- קליטת פקודות מהעכבר

- שיטות דיבוג

עבודה עם קבצים

פתיחה קובץ

הדבר הראשון שאנו צריכים לעשות כדי לקרוא מקובץ, הוא לפתח אותו לקריאה (כמו ספר – אי אפשר לקרוא אותו כשהוא סגור, קודם צריך לפתוח את הכריכה...). הדרך פשוטה לבצע פתיחה 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 שהמחזרות של שם הקובץ צריכה להסתיים ב-0. לדוגמה:



Filename db 'file.txt',0

בסיום הריצה, ax יוכל את ה-ah=3Dh שהוקצת לו על ידי מערכת הפעלה DOS. אם הייתה שגיאה, יודלק דגל CF ו-ax יוכל 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. הפרמטרים שצורך לשולח לפסיקה 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 יחזק את כמות הבתים שנקרו מהקובץ, או קוד שגיאה 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. פרמטרים:
 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 שמאפשר כתיבה (**cx=1**, **cx=2**) או יוחזר קוד שגיאת **5**  **access denied** שימושותו

```
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 ונוגרים כל הקבצים שהתוכנית פתחה. אם כך, מדוע בכלל לסגור קבצים שאנחנו משתמשים בהם? אחת הסיבות היא שלא תמיד יוכל לסמוק על היציאה מהתוכנית, שתסגור את הקבצים שלנו. לדוגמה, אם התוכנית קרסה בזמן ריצה, היא לא תסגור את הקבצים שפתחנו. זהו גם הריגל טוב לעתיד: לסגור משאבי חיצוניים שאנחנו משתמשים בהם (קבצים, זיכרון, קישורים למחשבים אחרים, תוכנות עוזר ועוד). בדומה זו, תוכנות אחירות יכולות לנצל את המשאבי שפינינו והקוד שלנו הופך ליעיל יותר.

סגירת קובץ מתבצעת על-ידי קוד **ah=3Eh**. הפקטור היחידי שציריך 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 והעובדת שמקורות שונים מתמקדים בהסבירם 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=...!' כאשר שלושת הנקודות מוחלפות בקוד הרלבנטי:

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     openerror
    mov    [filehandle], ax
    ret

```

openerror:

```

    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
```

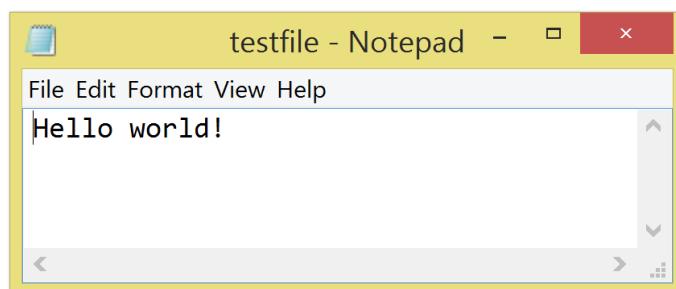
start:

```
    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 בעבודה 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. ה-**video memory** מזמין את המקום בזיכרון שבין A000:0000 ל-B000:FFFF.

כרטיסים המסך יכולים לעבוד בשני מצבים, שנקראים **modes**:

- במצב טקסטואלי, או **text mode**, אנחנו קובעים לCards המסך שלוו לקרוא את המידע מה-**video memory** במקומות שמתחליל ב-B800:0000 וגודלו 4K (4,096 בתים). במצב זה, כרטיס המסך מדפיס 25 שורות כפול 80 עמודות של תווים ASCII.

- במצב גרפי, או **graphic mode**, אנחנו קובעים לCards המסך שלוו לקרוא את המידע מה-**video memory** במקומות שמתחליל ב-A000:0000, וגודלו 64K (65,536 בתים). במצב זה, כרטיס המסך מדפיס 200 שורות כפול 320 עמודות של פיקסלים.

אפשר לפרק את תהליך הדפסה למסך לשולש שלבים:

שלב ראשון – קביעת מצב העבודה של כרטיס המסך (כ**শক্তিশালী ক্ষমতা** DOSBOX), מצב ברירת המחדל של הגרפיקה הוא מצב טקסטואלי).

שלב שני – תרגום הגרפיקה שלנו לביטים שכרטיס המסך יוכל לפענה (מה שנקרא פורטט של תמונה).

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 שחייב לחשב בעצמו את הכתובת שצריך לפנות 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. עדין, עם קצת יצירתיות ומשחק עם תווי ASCII אפשר להציג לדברים נחמדים מאוד.

שלב ראשון – קביעת מצב העבודה **text mode**. לשמהנו, 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 דרכי – 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 כתובות. לחילוףין, 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@@@@@\n
db "~-` .-/v]Z@@@@d|-
db "~-` .-/v KN8@@@@ (.i!vGG_
db "~-` .-/v)8K@@@@Kb@b@d~@~T4(
db "~-` .-/v\148@K@@@@8@0d*@@bVi
db "~-` .-/v,\Kb@@d@.~t` !*~!`.
db "~-` .-/v, \8M@@@@ -~,gvz`
db "~-` .-/vi @8@K@@D
db "~-` .-/ve8d5@@@@@
db "~-` .-/v8dZ8@M@@@@-
db "~-` .-/vb@AK@@b@[
db "~-` .-/v@b@M@@@@OP-
db "~-` .-/v@@@@MA@@@`- ._)g2i
db "~-` .-/v@@@@bK@@K@)i 'c, Kb@@K
db "~-` .-/v@Kb@@@@AGAA/i- ~M@@@@Mc
db "~-` .-/v@@@@Mb@@@@(`\` PPK(,i)v|-\`v)8XNaDMK@@@@b@EMK@@@@([" ,13,10
db "~-` .-@8@8@C@MK@d@A@L!--c)s, (ZsLBb@`~- .N)/KM@@@@d@e@A@@@@C@@" ,13,10
db "~-` .@8@8d@Kb@@@@Kb@@@@- .`!`~@ff@N@F-`-, `))KK@@@@C@MK@d@e@M8d@b@C@[" ,13,10
db "~-` .@8@8d@b@C@K@A@@@@C@2-- ,,_JJ|i)/- |v)NK@8d@e@C@A@@@@C@M@C@[" ,13,10
db "~-` .@8dY@8d@K@C@b@C@C@d!, 'VVV)\`)\`7(-)4Jb@8@A@C@Ked@C@C@C@C@C@C@[\$"

המחרוזת – אתם כבר בטח מבינים מדוע הוא שם.

את ההדפסה למסך נבצע בשיטה המוכרת, באמצעות הפקודה של הדפסת מהירותו למסך int 21h ah=9h.

דוגמה לתוכנית קצירה שמדפיסה למסך את הקובץ `monalisa.asm` (שםו לב לנכ' שיש להוריד את הוראה `IDEAL` בראש התוכנית, על מנת שה-`include` יעבדו):

```
; Display ASCII graphics
; Author: Barak Gonen 2014
;
model small
stack 100h
DATASEG
include monalisa.asm
CODESEG
start:
```

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

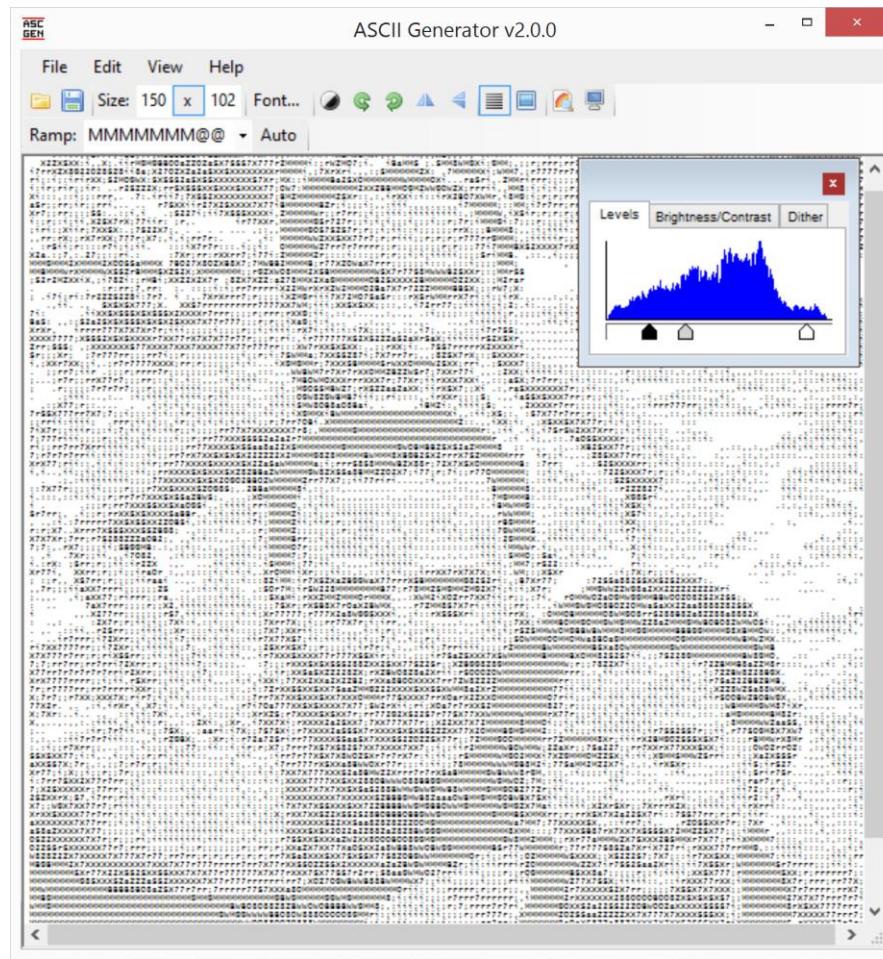
; Print string
mov    ah, 9h
mov    dx,offset monalisa
int    21h

; Wait for key press
mov    ah, 0h
int    16h

exit:
mov    ax, 4C00h
int    21h

end start
```

אפשר להפוך כל תמונה לתווית ASCII Generator שנותנת להורדה מ-[לדוגמה: http://sourceforge.net/projects/ascgen2](http://sourceforge.net/projects/ascgen2)



אתם יכולים להשתמש במגוון המקורות הבאים, או לחפש 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 נעבור לדבר פיקסלים. פיקסל הוא היחידה הקטנה ביותר שנitin לשנות את הערך שלו בمسך. פיקסל הוא כמו אטום, אבל של גרפיקה. ממות הפיקסלים במסך נקראת רזולוציה. אנחנו נדבר על פורמט VGA, קיצור של Video Graphics Array. סקירה של פורמט זה ניתנת למצוא במקומות רבים, מומלץ כרגע לתת 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 לעبور לפורט גרפי זה השתמש בפסקית BIOS, int 10h שנעשה חביבה עליינו בפרק זה:

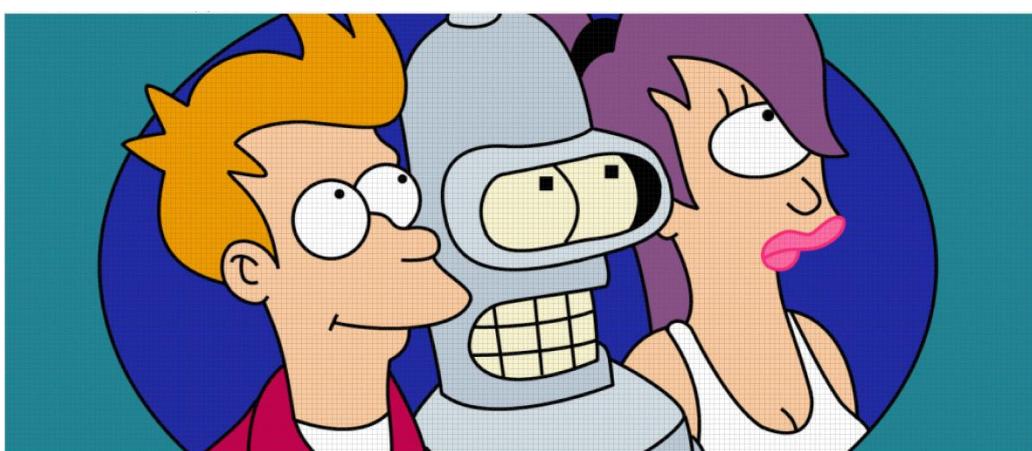
```
mov ax, 13h
```

```
int 10h
```

פסקה זו קובעת את מצב המסך שלנו, למטריצה של 200x320 פיקסלים.

0,0		...		0,319
...				...
199,0		...		199, 319

בתוך ה-video memory, באזור שמתחליל ב-A000:0000, שומר 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 הבית 0:0000:A000:0001 קשור לפיקסל 0:0, הבית 1:0:0:0 קשור לפיקסל 0:1 וכך הלאה. כדי להגיע לפיקסל בשורה Y, כופלים את מספר השורה ב-320, מוסיפים את מספר העמודה X ופונם לזכרון במיקום שוחשב.

נעמיק 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 המילה מכילה רק 256 אפשרויות (כדי שנitin יהיה לייצג צבע באמצעות בית בווד). נניח ששמננו ב-RGB הם 0,0,0 – מספר צבע 0. כרטיס המスク יפנה לטבלת המרה 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-**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 בירית המהדר, באמצעות טענה של קובץ בפורמט bmp שמכיל palette מותאם לקובי.

הדפסת פיקסל למסך

לאחר שסקרנו את המנגנון שעומד מאחורי הדפסת פיקסלים למסך, נותר לנו רק לתאר את הפקודות שגורמות לכך בפועל. האפשרות הראשונה היא פשט לגשת ל-video memory ערכיהם לתוךוUrchin בגודל בית באמצעות פקודה **out**. לפני כן צריך רק לחשב את מיקום התא בזיכרון, כדי שנקלע לקוואורדיינטות **yx**, הנכונות.

האפשרות השנייה היא לבצע אותה פעולה, אבל בעזרת פסיקת BIOS, קוד **ah=0Ch**. הפסיקה צריכה לקבל את הפרמטרים הבאים:

al – צבע

ax – עמוד (צריך להיות 0)

cx – קוואורדיינטת X

dx – קוואורדיינטת 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
```

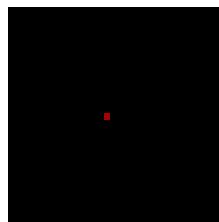
```

int      16h
; Return to text mode
mov     ah, 0
mov     al, 2
int      10h

exit:
mov     ax, 4c00h
int      21h

END start

```



הדףת פיקסל למסך

קריאה ערך הצבע של פיקסל מהמסך

פסיקת BIOS מאפשרת לנו גם לקרוא את ערך הצבע של פיקסל מסך, 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 רוצים לבדוק האם הנחש שלנו התנגש בקיר. אנחנו יכולים פשוט לחת לקיר צבע ייחודי ולבזוק 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
```

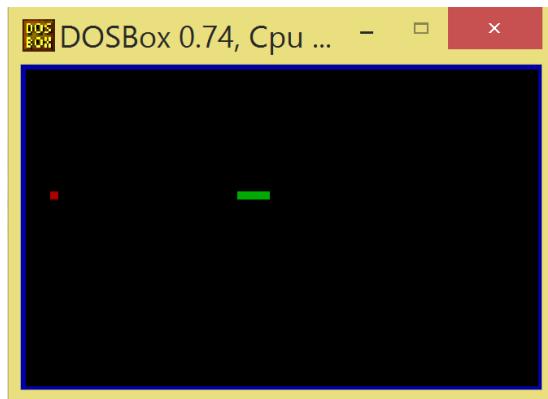
```
; Read dot
```

```
mov     bh,0h
```

```
mov     cx,[x]
```

```
mov     dx,[y]
```

```
mov ah,0Dh
int 10h ; return al the pixel value read
```



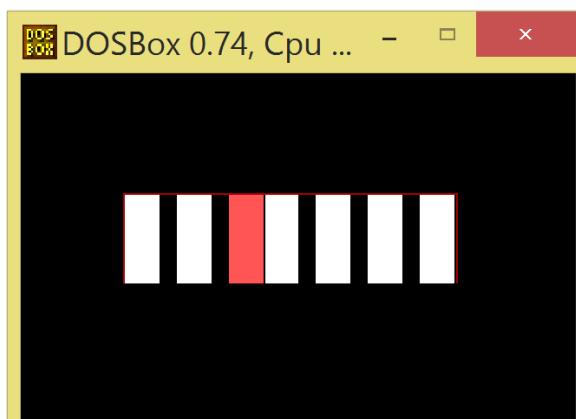
(משחק סנייק שתוכנת באסמבלי (เครดיט: נעם אולאי)

יצירת קוים ומלבנים על המסך

ברגע שאנו יודעים ליצור פיקסל על המסך, אנחנו יכולים ליצור גם קוים ומלבנים (למעשה אנחנו יכולים ליצור כל שורה, אבל נתמקד בקוים ובמלבנים 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 מתבצעת באמצעות אוסף קריאות לפרוזדורה שיוצרת קו אופקי, כאשר בכל פעם נקודת ההתחלה בציר ה- Y גדלה ב-1. כדי ליציג כפטור 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, במקומות לצורך צייר תמונה בכוחות עצמוו, נרצה לשלב בתוכנה שלנו תמונה נאה שמצאנו. קיימים פורמטים (הגדירות מקובלות לייצוג מידע בקבצים) רבים של תמונות, אנחנו נתמקד בפורטט BMP, קיצור של Bit Map, משתי סיבות. הראשונה היא שהוא פורטט נפוץ, והשנייה היא שהוא פשוט. את הנתונים שיש בקובץ BMP אפשר לטען בלי הרבה שינויים ומשחקים לתוך הדעת memory video, זאת בניגוד לפורטטים אחרים, כמו JPG הנפוץ, שמצריך אלגוריתמיקה מורכבת לפתחה.

נתאר בקיצור רב את פורטט BMP. לפני כן, נקדים ונאמר שאת הלימוד של פורטט ה-BMP מומלץ وكل לעשות עליידי היפוש באינטרנט. הסברים מפורטים ניתן למצוא באמצעות החיפוש *in assembly* "read bmp file". קישורים לדוגמה:

www.brackeen.com/vga/bitmaps/html

www.ragestorm.net/tutorial?id=7

פורטט BMP מרכיב מהשדות הבאים:

Header .1 – פtile בין 54 בתים. בתחילת header נמצאים התווים 'BM' שמצוין שהקובץ הוא בפורטט .BMP

.2 – Palette 256 צבעים, כל צבע תופס ארבעה בתים (בסך הכל 1,024 בתים). שימוש לב, שבעוד שכרטיס המסך מקבל את הצבעים בפורטט RGB (כלומר, אדום–ירוק–כחול), הצבעים שבpalette של ה-BMP שמורים בפורטט BGR (כחול–ירוק–אדום), ולכן נדרש להפוך את סדר הצבעים לפני שnettן אתpalette לכרטיס המסך.

.3 – המידע על כל פיקסל ופיקסל שמור בבית יחיד. הבית הזה מכיל מספר בין 0–255, שהוא מספר הצבע ב-palette. שימוש לב, שהשורות שמורות מלמטה למעלה. ככלומר, אם ניקח את data ונווטיק אותו כמו שהוא לתוך הדעת memory video, 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 למקום. נניח שיש לנו תמונה בגודל 200x320 פיקסלים. ניקח את הפיקסל הראשון בשורה התחתונה (שמוצג במאט מצד שמאל). כיוון ש-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, הוא שהצבע של הפיקסל השמאלי בשורה התחתונה הוא מה שנמצא באינדקס 0 ב-palette. אנחנו נגישים ל-data, 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 ליצר 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 הגדים 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 לקרוא.
- לאחר מכן נקרא ה-`header`.
- לאחר מכן נקרא ה-`palette`.
- ה-`palette` נתען לרטיס המסך, בפורטים `3C9h`, `3C8h`, `3C8h`. 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 של 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 וטיפים בנושא גרפיקה

פורום הפיאה מכיל מידע איקוטי בנושא שימוש בגרפיקה, כולל קטעי קוד וטיפים. בחלון החיפוש אתם יכולים לחפש כל ביטוי. לדוגמה bmp או graphics. הדיוון הבא מומלץ במיוחד:

<https://piazza.com/class/i98qbkdp1mg15m?cid=20>

דיון זה נקרא "Graphics- advanced" והוא מרכז מספר נושאים. להלן הודעה שפותחת את הדיון:

"שלום לכלום

אני רוצה שדיון זה ירכז פרקטיקות טובות לטיפול בגרפיקה, שלא ספק זהו נושא שמעסיק תלמידים רבים. לדיוון זה מוזמנים מאד לענות תלמידים שעשו פרוייקטים עם גרפיקה איקוטית, והידע שלהם גם יסייע לאחרים וגם ישמר לשנתונים הבאים.

נא להעלות הסברים וקטעי קוד לנושאים הבאים:

1. איך מעלים תמונה BMP למקום מוגדר במסך? לדוגמה, ציר של מטרה שגודלה 10x10 פיקסלים במקומם 80,100.

2. איך מזיזים תמונה על המסך בצורה חלקה בלבד 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 (:

בברכה

ברק"

เครดיטים על תשובה:

סעיף 1 - **יואב שטרנברג**, אהל שם רמת נ

סעיפים 2,3 - קורן ברנד, אהל שם רמת גן

סעיף 4 - שני שורץ, אורט בנימינה

השمعת צלילים

כפתיה לנושא השמעת צלילים, מומלץ לקרוא קודם כל תיאוריה על צלילים, גלי קול ותדריות. אתם יכולים למצוא מגוון עצום של מקורות באינטרנט. מומלץ במיוחד להוריד את גרסה ה-pdf החינמית של הספר *Art of Assembly* ולקראא אודות הפיזיקה של גלי קול (*The Physics of Sound*).

נתמקד בחלקים הטכניים של השמעת צלילים מהמחשב.

למבחן יש רכיב חומרה שצמוד אליו – טיימר, שעון, שלמדנו עליו בפרקם הקודמים. הטיימר מחובר לכרטיס הקול של המחשב ומעבר לו תקתוכי שעון, או "טיקום". כל "טיק" של הטיימרגורם לסגירת מעגל השםלי בכרטיס הקול, שמהרגם אותו לקול שאנחנו מסוגלים לשמעו. כמות ה"טיקום" שSEGMENTS כל שנייה מהטיימר אל כרטיס הקול הם התדר. ככל שמתגברים יותר "טיקום" – 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. בתוך אוקטבה, הצלילים רוחקים זה מזה במכפלות של שורש 2 (בערך 1.06), כלומר אם התדריות של TWO מסויים היא 110 הרץ, אז התדריות של הAU הבא אחריו היא $1.06 \times 110 = 122$. כלומר, כפוף לערך ה-1.06, הצליל השני יהיה כפוף לערך ה-1.06², והצליל השלישי יהיה כפוף לערך ה-1.06³.

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 חזרה 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 לשלוח את תדר הAU 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 לדחף port 42h "מחלק" (divisor) בגודל 16 ביט, עבור התו שאנו רוצים להשמי. 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, ובסיס הקסדצימלי 0A98h. לכן השילוחה תבוצע 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 שאותם מבינים את התיאוריה לפני שאתם מתקדמים הלאה. כזכור, אפשר לקרוא את השעה באמצעות פסיקה $h\ 21\ int\ 0\ ah=2Ch$. במקרה זה אנחנו עוסקים בכלים לפרויקטם ולכון נתמך בשני כלים מעשיים שונים לנו השעון.

הכלי הראשון הוא יכולת למדוד פרקי זמן קבועים שקבענו מראש – נניח שאנו רוצים שפעולה כלשהי תבוצע כל זמן קבוע. לדוגמה:

- במשחק סניף, קבוע כל כמה זמן הנחש יקדם צעד אחד.
- כשהמנגנים צליל כלשהו, קבוע למשך כמה זמן יונגן הצליל.
- במשחק שחמט, קבוע פרק זמן מקסימלי 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 מגנון שנותן לנו ערכים משתנים בכל פעם שאנו מרים את המשחק, ואת הערכים האלה אנחנו יכולים אחר כך לתרגם לדברים אחרים שאנו צריכים. לדוגמה:

- במשחק סניף, הצבה של אוכל במקום אקרי במסעדה.
- במשחקי קובייה, כמו סולמות וחלבים, קבוע את הערך של הקובייה.
- במשחקי פעולה, תזוזה אקראית של דברים על המסלך.

מדידת זמן

דרך אפשרית אחת למדוד זמן שעובר היא על-ידי הפסיקה שקוראת את השעון – מבצעים לולה של קריאה חוזרת לפסיקה, ובכל פעם משווים את השעה עם השעה הקודמת (מספיק להשוות את ערך מאות השניה, שנשמר לתוך `if` – אם הוא לא השתנה אזשאר הערכים בהכרח לא השתנו). הדרך זו נראה מרווחת מילישניות ולכון לא עמוקה. 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 מהמדידה השנייה והלאה אנחנו יכולים להניח שהפרש הזמנים הוא מדויק.

הדרך השנייה לדעת ש-55 מילישניות עברו היא להשתמש בעובדה שכל פסיקה של הטימר גורמת לעדכן של מונה שנמצא בכתובת `0040:006Ch`. אנחנו לא נדע מה השעה, אבל כל שינוי בערך שנמצא בכתובת הזו בזיכרון יקבע על כך שעברו 55 מילישניות. היתרון של השיטה הזו הוא מהירותה. קריאה ישירה תמיד תהיה מהירה יותר מפסיקה (עם כל הפעולות

הנוספות שהיא גוררת, כגון שמירת ערכיהם במחסנית וביצוע קפיצות). באופן כללי, אם יש קוד שאנו מರיצים לעיתים קרובות או עדיף לשים לצד את 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 מחכים 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 מחשב היא בעיה מעניינת מאוד עם השלכות פילוסופיות. בערךון, הטענה היא שאי אפשר ליצור באמצעות מחשב – 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 מעניות רבות, לדוגמה בעולם ההימורים אונליין.

לכן אנחנו נזכיר מושג שנקרו מספרים **פסאודו-אקראיים** (**Pseudo-random**). אלו מספרים שניתנים לשחזר אותם באמצעות ידיעת האלגוריתם ותנאי התחלה, 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 פסאודו אקראיים, תלמידים המתעניינים בנושא מוזמנים לפנות ללמידה עצמי). מעכשו, בכל פעם שנכתב "אקראי" נתכוון בעצם "פסאודו אקראי".

קיימות שיטות שונות ליצירת מספרים אקראיים. לא נוכל להתייחס במסגרת ספר זה לכלן ואנו לא למקצתן. בהתאם 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 וכן 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: 0,00, 0,01, 0,10, 0,11. כלומר, מספר בין 0–3. נבצע זאת באמצעות שינוי קטן בשורה האחורונה:

```
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. לכוארה אפשר לקבל את התחום בין 0–9 על ידי הייבור של שלוש פעולות ליצירת מספרים אקראיים:

- ייצור מספר אקראי בתחום 0–1.

- ייצרת מספר אקראי נוסף בתחום 0–1.

- ייצרת מספר אקראי 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 שMOVIL לתוכה 7 (0,0,7 או 1,0,6 או 1,1,5 או 1,1,1 או 1,5,1 או 6,1,0 ועוד...).

פתרון פשוט הוא ליצור מספר אקראי בתחום גדול 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, ניצור 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 שככובת הביטים 0040:006Ch, שכאמור מתעדכן כל 55 מילישניות. כמובן, אם נרצה ליצור כמה מספרים אקראיים בפרק זמן קצר, לא נקבל גיון בתוצאות, או שנקבל מספרים שיש ביניהם קשר ברור. כמובן, השיטה זו 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 הפעם מהטימר במספר קריאות נפרדות, אכן נפיק מהם ביטים אחרים בכל פעם. אנחנו יכולים, לדוגמה, לעשות לביטים שקוראים מהטימר XOR עם ביטים במקום כלשהו בזיכרון – בכל פעם עם מקום אחר. למשל, ניתן לנקח 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 ו XOR עם ביטים שנמצאים ב-GCODESEG. נסכם את הסעיף על ייצור מספרים אקראיים בכך שקיימות שיטות רבות וטובות מזו לייצור 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
```

ממתק משתמש

בסעיף זה עוסוק בשני כלים שכלי תוכנית שעובדת עם ממתק משתמש צריכה – מקלדת ו/או עכבר.

קליטת פקודות מהמקלדת

בפרק על הפטיקות כיסינו את התיאוריה והפרקטיקה של פעולה המקלדת ואת הגורמים הקשורים לפעולה של המקלדות:

- Scan Codes -

- פורטימ 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 (פורטימ / BIOS / DOS). כמו שראינו, פסיקות BIOS מאפשרות לנו לקבל קלט מבלי לעזרה

את ריצת התוכנה, ופסיקות DOS מאפשרות לנו להוכיח למשתמש. כמובן, המימוש המומלץ תלוי בצהרה

בה אנו רוצים שהתוכנה תפעל.

.2. שימוש בפסיקת הד-ASCII int 21h מוחיר את קוד הד-ASCII של המקש שהוקלד. ישנו מקשיים –

ודווקא ככל שצורך להשתמש בהם במשחקים (מקשי חיצים לדוגמה) – שיש להם קוד הד-ASCII מורחב.

בקוד הד-ASCII מורחב, לטור באפר המחסנית מועתק קוד הד-ASCII במקומו של הד-code scan, ובמקום

של קוד הד-ASCII מועתק אפס. מבלבל? נכון. מסיבה זו, עדיף תמיד לבצע את פעולות הבדיקה

וההשוואה עם הד-code scan. 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, כפי שלמדנו לעשות מוקדם יותר בפרק זה:

mov ax,13h

```
int    10h
```

העכבר נשלט בידי פרוצדורות שונות שפעילה פסיקה 33h. הקודים נשלחים על גבי ax (שימו לב, ax ולא ah כמו שאנו רגילים עם int 21h). כדי להפעיל את העכבר, ראשית יש לאותל אותו ואת החומרה שלו. הקוד ax=0h מבצע את פעולה האותל:

```
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 מזירה את הערכים הבאים:

- ax – מצב הלחיצה על כפתורי העכבר

○ xxxx xxx1 – הביט במקום 0 שווה '1' – כפטור שמאלி לחוץ.

○ xx xxxx – הביט במקום 1 שווה '1' – כפטור ימני לחוץ.

כלומר, כל ערך של ax שהbiteים הימניים שלו הם '00' – משמעתו שאין כפטור לחוץ.

- ax – מיקום העכבר, שורה בין 0–639 (שימו לב, כשאנו עובדים במוד גרפי כמוות השורות שיש לנו היא 320 בלבד, לכן צריך לבצע התאמת ולחולק את ax 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 במקומות 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 יותר. לעומת זאת, בפרויקט סיום יש למעלה אלפי שורות קוד. אפילו אם אתם מתכוונים לעוברים ויש לכם באג רק אחת למאה שורות קוד, זה עדין אומר שתיקלו בעשרות באגים לפני שהתוכנית שתכתבם תעבור כמו שתכננתם. אנחנו עוד לא מדברים על באגים שנובעים מבעיה בתוכנון, כזו שתאלץ אתכם להבהיר חלקו קוד מקום למקום, או באג באлогריתם, שעלול לגרום לכם לבנות ערבות מול מסך המחשב בעודכם כוסים ציפורניים, נעים בין ייאוש לזעם.

אין הכוונה לייאש אתכם, רק לציין את המובן אליו – כתיבת תוכנית גדולה דורשת מילונות תכנות ושימוש בטכניקות דיבוג, שאינן נחוצות בשביל כתיבת תוכנית קטנה כמו התוכניות שנתקלנו בהן עד עכשו.

אנחנו עוסקים עכשו בטכניקות שונות שיכללו, אולי, לעזור לכם לצלוה יותר בבעיות את אותן ערבים בהם התוכנה לא עובדת ואין לכם מושג למה.

тиיעוד

כתיבת 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 שתכתבם. תתעדו. רמה נכונה של תיעוד היא זו:

- בתחילת כל פרוצדורה, הסבר מי נגד מי: ציון פרמטרים ותוצריים.
- אחת לפחות שורות קוד, לרשום הערכה מה עשו קטע הקוד הבא.
- אם יש שורה מסובכת, כזו שלא מובן لماذا 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
```

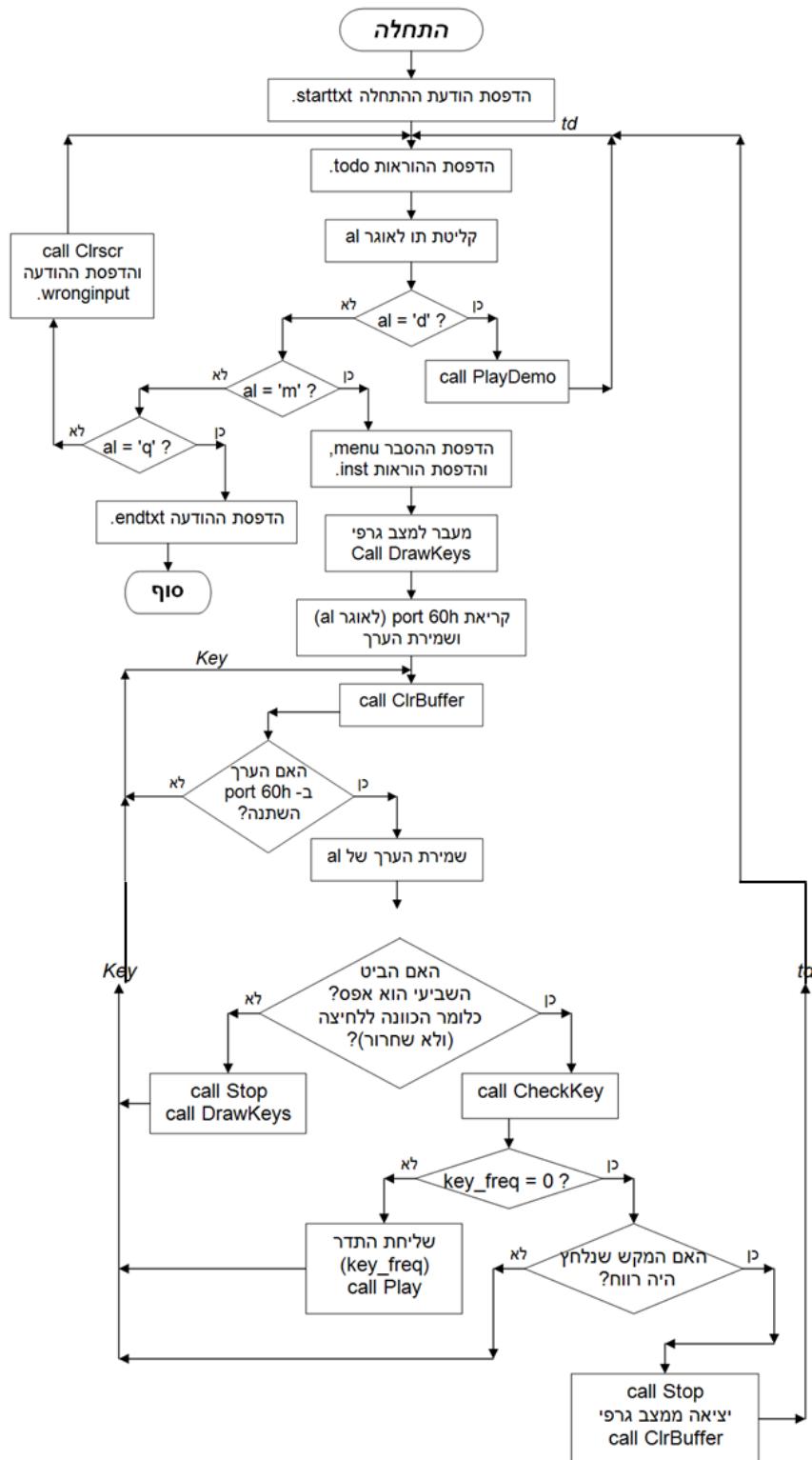
עם זאת, חשוב לא להציגם. מתקנים מתחילה נוטים לעתים להוסיף תיעוד רב, שרבו מיותר, או שאיןו מסביר למה שעשינו את מה שעשינו. דוגמה לתיעוד שלא משרת שום מטרה טובה:

```
mov ax, 5      ; copy '5' into ax
```

תיכון מוקדם – יצירה תרשימים זרימה

תרשימים זרימה לכל התוכנית יכול להקל עליכם לעשות סדר בתוכנית המורכבת שלכם ולמנוע טעויות תיכון שיגולו מכם זמן יקר בהמשך. אל תותרו על תיאור מסודר של התוכנית שלכם לפני שאתם מתישבים לתוכנה. בכל אופן, גם אם תותרו לעצמכם, משרד החינוך דורש שלכל תוכנה يتלווה תרשימים זרימה, כך שבעל מקרה עלייכם להזכיר אחד... עדיף שתתחלו מתרשימים זרימה טוב שיעשה לכם סדר بما שאתה מתקננים לתוכנה.

لتרשימי זרימה יש שפה מיוחדת. כדי להקל על הקריאה וההבנה יש טקסט ששמות במלבן, טקסט ששמות בתוך מעוין, טקסט ששמות בעיגול וכו'. תוכלו ללמידה בקהלות את שפת תרשימי זרימה באינטרנט (חפשו "איך לכתוב תרשימים זרימה").



חלוקת לפרויקטים

קרוב לוודאי שאחרי שיצרתם תרשימים זרימה, זההיתם מספר קטעי קוד שצרכיהם לרוץ מספר רב של פעמים. אלו הן הפרויקטים שאתם מיעודים לכתוב. אם אתם רוצים שהיה לכם סיכוי סביר למצוא באגים, אתם צריכים לחשוף היבט על החלוקה לפרויקטים. היתרונות העיקרי של פרויקטים הוא בשלב הדיבוג – אם כתבתם פרויקט טובה ובודקתם אותה, אתם לא צריכים להיות מודאגים בקשר אליה. אנחנו נציג כרגע כמה דרכי לדבג תוכנית שכוללת פרויקטים:

- **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 תמונה 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 להערה, ו"נשחרר" אותן מההערה 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
```

נרים את התוכנית, שכרגע כל מה שהיא עשוה זה רק לפתוח קובץ. נבדוק שהקובץ נפתח בצורה תקינה (לא מוחזר קוד שגיאת). הכל נכון? המשכנו להלאה.

- **לקראן זיכרון תוך כדי ריצה.** ה- TD מספק לנו תמייה טובה בבדיקה מצב הזיכרון תוך כדי ריצה. נניח שהוצאנו את הקראנה ל- **ReadHeader** מהערה, אנחנו רוצים לבדוק אם הפרויקט עופדת בצורה תקינה. בתוך **header**.header נוכל לראות איך header נראה בזיכרון מיד בסיום DATASEG הגדרנו מערך ששומר את header.

הקריאה. אנחנו יודעים שני הบทים הראשונים שלו צריכים להיות 'BM' – אם קיבלנו ערך אחר, יש לנו בעיה בקריאה.

להחליפ פרוצדורות 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 גרפּי 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 יועתק ערך קבוע. אם הבעיה ממשיכה, נראה שיש בעיה בפרוצדורה שモזיאת את הגרפּיקה על המסק.

שים לב, שכדי שתוכלו לדבג בצורה ייעלה, החלוקה לפרוצדורות צריכה להיות של קטעי קוד שמבצעים משימות קטנות ומוגדרות היטב. חישבו תמיד אם אפשר לפצל את המשימה שהפרוצדורה שלכם מבצעת ליותר מפרוצדורה אחת, אם יש קטעי קוד שהווירים 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 מפרוצדורה אחת, הרצתם לולאה ואוזרתם לפרוצדורה שמשנה את CX).

אם אתם לא שמים לב בעיה, מומלץ להכנס לתוכנית שלכם קטעי קוד שמדפיסים את הערכים של המונחים למסך או אפילו לקובץ. כך תוכלו לוודא שהמונחים שלכם מקבלים רק ערכים בתחום הצפוי וההגיוני. יעזר לכם אם כתבו פרוצדורה או מקרו, שמקבלים כפרמטר רегистר ומדפיסים את הערך שלו למסך או לקובץ, במקומות למש את הלוגיקה הזו בכל פעם מחדש.

העתיקות זיכרון

עוד באג נפוץ מאוד הוא העתקה למקום לא נכון בזיכרון. אין הכוונה להעתקה למקום לא נכון בזיכרון, בעיה נפוצה בפני עצמה, אלא להעתקה של מידע מחוץ 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 שימושים אנחנו צריכים לשנות את הערכים של רегистרי הסגמנט. לאחר השינוי צריך להחזיר אותם למצב המקורי, 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 החיפוש, קרוב לוודאי שתגיעו לפורום מתכנתים (לדוגמה StackOverflow) ותמצאו שמשיחו כבר נתקל בבעיה שלכם וקיבל עזרה.

סיכום

בפרק זה עסקנו במגוון 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 עבודה עם קבצים, עבודה עם עכבר, ייצרת מספרים אקראיים. סיימנו בטיפים לכתיבת פרויקט ובאופן כללי כיצד יש לגשת לפרויקט כתיבת תוכנה בהיקף ממשמעותי.

סיימו את לימוד שפת האסמבולי ומבנה המחשב, אנו מוכנים לأتגרים הבאים שלנו.

لتלמידים: זיכרו, הودות לאינטרנט, ידע אינטובי נמצא במרחק כמה הקשות מקלדת. זהה מתנה שלא הייתה קיימת בדורות קודמים. כשתלמדו למצוא לבד מקורות לימוד, תתרמו את כה הידע של אחרים לטובתכם. מכאן והלאה, דבר לא יכול לעזור את הסקרנות שלכם. בהצלחה!

נספח א' – רשימת פקודות חובה לבגרות בכתב באסמליה'

לפניכם רשימת פקודות החובה לבגרות בכתב באסמליה'. שימו לב – אלו אינן פקודות חובה עבור תלמידים שלומדים אסמליה' במסגרת יחידת המעבדה (כמו ב"גבהים") אלא רק עבור תלמידים שניגשים לבגרות במחשבים בפרק בחירה באסמליה'.

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:

CLC - הורדת דגל הנשא

STC - הדלקת דגל הנשא

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 שׂוֹהָ	JGE (JNL)	JAE (JNB)	
קפוֹץ אֶם קָטָן או שׂוֹהָ	JLE (JNG)	JBE (JNA)	
קפוֹץ אֶם שׂוֹהָ	JE	JE	
קְרוּץ אֶם שׂוֹנוֹה	JNE	JNE	

cx=0 - קפוץ אֶם JCXZ

פקודות קפיצה שבודקות ישירות את מצב הדגלים:

תיאור	הוראה
קְרוּץ אֶם דָגֵל הַנְשָׁא דָלוֹק	JC
קְפוֹץ אֶם דָגֵל הַנְשָׁא כָבּוּי	JNC

קפוץ אם דגל האפס דולק	JZ
קפוץ אם דגל האפס כבוי	JNZ
קפוץ אם דגל הסימן דולק	JS
קפוץ אם דגל הסימן כבוי	JNS
קפוץ אם דגל הגלישה דולק	JO
קפוץ אם דגל הגלישה כבוי	JNO
קפוץ אם דגל הזוגיות דולק	JP / JPO
קפוץ אם דגל הזוגיות כבוי	JNP / JPE

פקודות רегистר דגלים:

-LAHF – מעתיק את 8 הביטים הנמוכים של רегистר הדגלים לא-ah

-PUSHF – מעתיק את רегистר הדגלים למחסנית

-POPF – מעתיק את תוכן ראש המחסנית אל רегистר הדגלים

פקודות לולאות נוספות:

-LOOP – שאנחנו מכירים, ישנן פקודות 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 **SHL, SHR** שאנו מכירים, ישנן הפקודות הבאות-

ROL- כמו **SHL**, כל הביטים זרים שמאליה והבית השמאלי ביותר מועתק לדגל הנשא, בעוד מועתק 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 זרים ימינה והבית הימני ביותר מועתק לדגל הנשא, בעוד מועתק אחד- הבית הכוי ימני מועתק גם אל הבית הכוי שמאליה.

RCL- כמו **SHL**, אך דגל הנשא מועתק אל הבית הכוי ימני.

RCR- כמו **SHR**, אך דגל הנשא מועתק אל הבית הכוי שמאליה.

SHL-SAL – זהה ל-

SHR-SAR – זהה ל-

פקודת חסור מיוחדת:

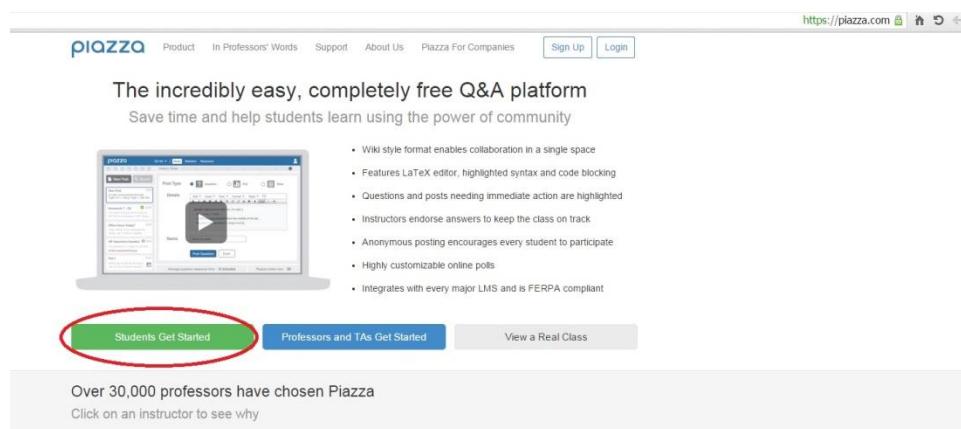
SBB-SUB – כמו **SUB**, אבל מוסיפה לתוכאה את ערכו של דגל הנשא.

נספח ב' – מדריך לתלמידים: כיצד נכנסים לפורום האסמלבי הארץ'

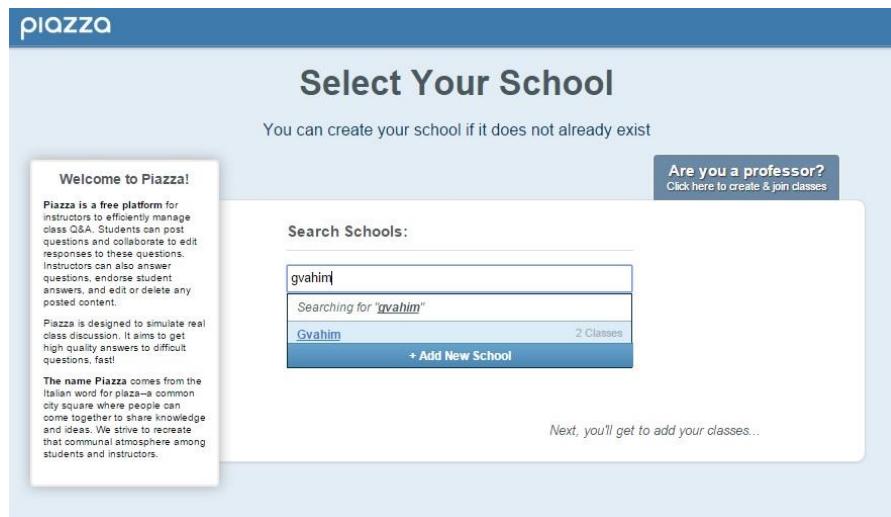
Piazza היא פורום שאלות ותשובות שמיועד לתלמידים בכל הארץ. ניתן לפתח פורום בכל נושא, לכתוב שאלת ותשובות. כיוון שתלמידים בכיתות גבהים בכל הארץ נתקלים בעיות דומות, עומדת לרשותכם "אוניברסיטת" וירטואלית, שם תוכלו לסייע אחד לשני. בנוסף, התשובות שלכם יסייעו לתלמידים בשנים הבאות!

לכניסה ל-Piazza:

1. היכנסו לאתר www.piazza.com ובחרו "students get started"



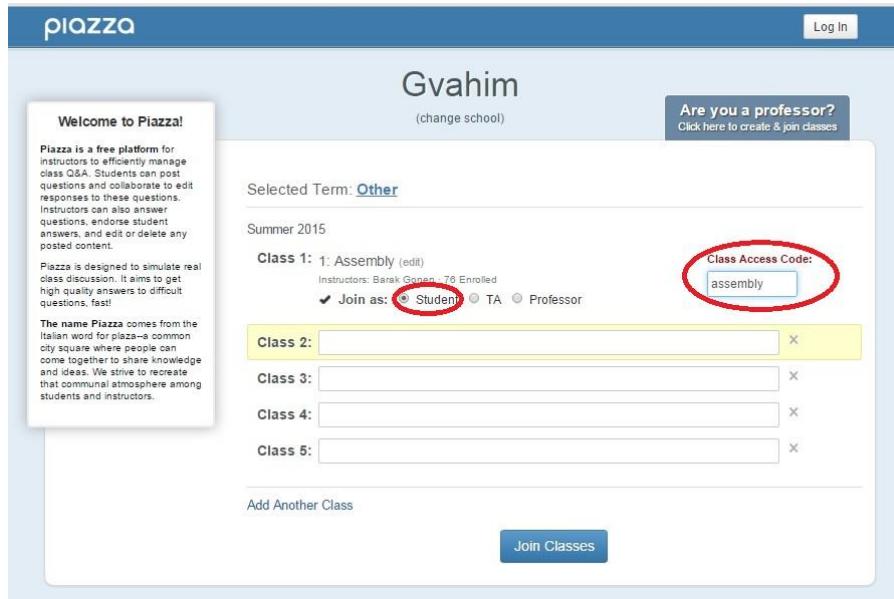
2. בחרו באוניברסיטה "gvahim"



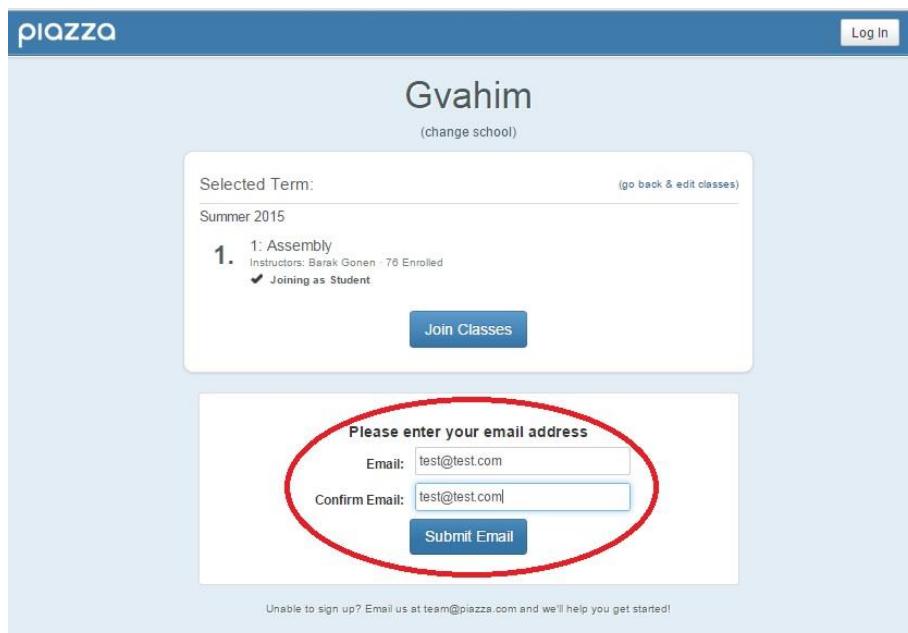
.3. בחרו סטטוס "other".

.4. בחרו קורס assembly.

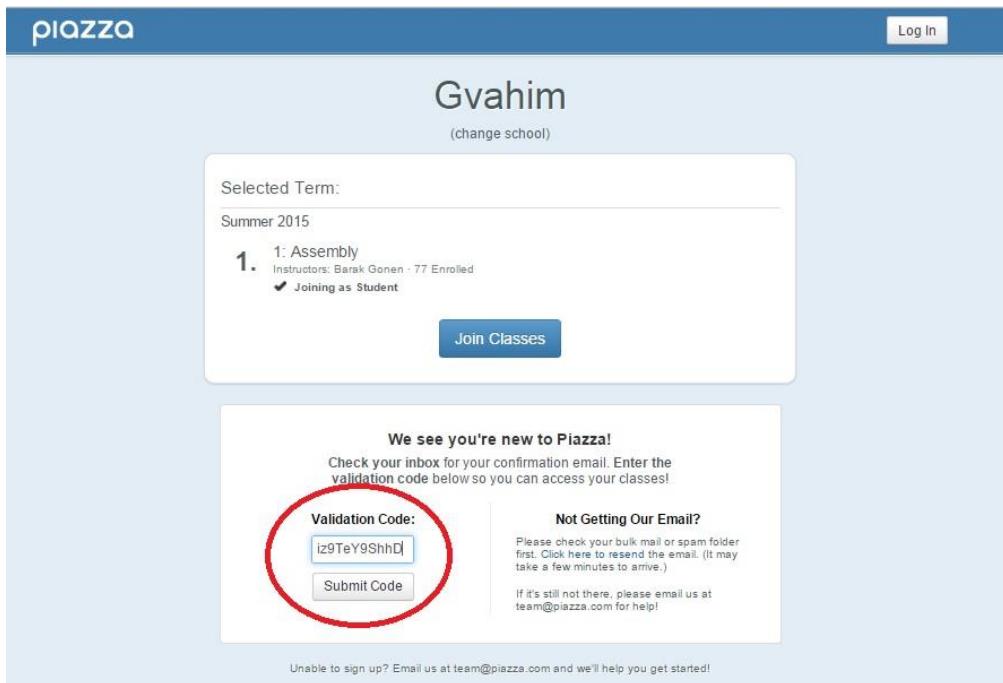
.5. בחרו להצטרף כ-student והכניסו סיסמה "assembly".



.6. הכניסו כתובת מייל, אליה יישלח קוד הפעלה של האתר, ולחצו על submit email.



. 7. הכניסו את הקוד שקיבלתם ולוחזו על submit .



. 8. בחרו שם משתמש וסיסמה. למטה בחרו באופציה I am not pursuing a degree ואשרו שקראותם את תנאי השימוש. לאחר מכן לחזו על Continue - Continue .

הכנסת שאלה חדשה

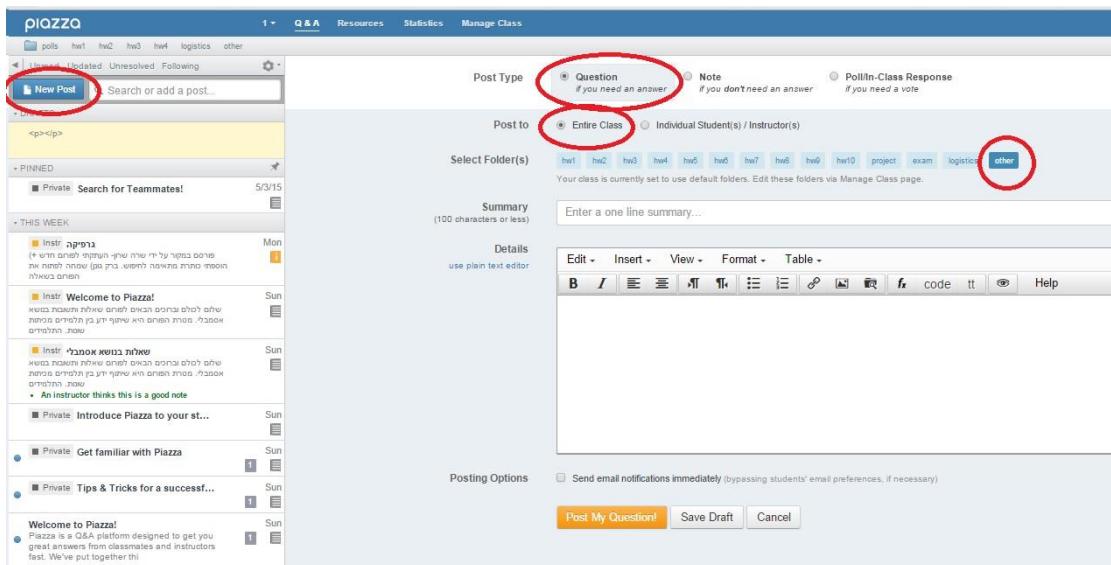
על מנת לשאול שאלה, הקישו על **new post**. בחלון שייפתח בחרו

- Post type = Question

- Post to = Entire class

- Folder = Other

Post my question אל תשחחו تحت כותרת לשאלת, 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/>