ארגון המחשב ושפת סף

ברק גונן

מהדורת ניסוי

ארגון המחשב ושפת סף

גרסה 2.04

<u>כתיבה</u>:

ברק גונן

מבוסס על מצגות וחומרי לימוד מאת אנטולי פיימר

<u>ייעוץ פדגוגי</u>:

אנטולי פיימר

<u>עריכה</u>:

עומר רוזנבוים

<u>הגהה ופתרון תרגילים</u>:

אליזבת לנגרמן

עדן פרנקל

אריאל וסטפריד

ליאל מועלם

תודה מיוחדת: לעודד מרגלית וליהודה ויכסלפיש 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 הסייבר הצה"לי.

© כל הזכויות על החומרים ממקורות חיצוניים ששובצו בספר זה שמורות לבעליהן. פירוט בעלי הזכויות- בסוף הספר.

© כל הזכויות שמורות למטה הסייבר הצה"לי. מהדורה שניה תשע"ה 2015

http://www.cyber.org.il

תוכן עניינים

הקדמה 12
פרק 1 – מבוא ללימוד אסמבלי 15
פרק 2 – שיטות ספירה וייצוג מידע במחשב 18
מבוא – מהן שיטות ספירה? 18
משחק: Pearls Before Swine 3 19
השיטה העשרונית 20
חישוב ערך של מספר עשרוני בבסיס אחר 21
השיטה הבינארית 21
השיטה ההקסדצימלית 22
פעולות חשבון 25
חיבור 25
חיסור (הרחבה) 28
כפל (הרחבה) 30
חילוק (הרחבה) 32
ייצוג מספרים על־ידי כמות מוגדרת של ביטים 33
ייצוג מספרים שליליים 34
שיטת גודל וסימן 34
שיטת המשלים לאחת 35
שיטת המשלים לשתיים 36
אז איך מנצחים את חואן? 39
ייצוג מידע במחשב 41
סיבית – Bit 41
נגיסה – Nibble 41
בית – Byte 42
מילה – Word 43
מילה כפולה – Double Word 43
קוד ASCII 43
סיכום 44

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

Processor Status Register – FLAGS 87
דגל האפס – Zero Flag 88
דגל הגלישה – Overflow Flag 89
דגל הנשא – Carry Flag 90
דגל הסימן – Sign Flag 91
דגל הכיוון – Direction Flag 91
דגל הפסיקות – Interrupt Flag 92
דגל צעד יחיד – Trace Flag 92
דגל זוגיות – Parity Flag 92
דגל נשא עזר – Auxiliary Flag 92
סיכום 93
פרק 6 – הגדרת משתנים ופקודת mov 94
מבוא 94
הגדרת משתנים 94
הקצאת מקום בזיכרון 95
משתני Signed, Unsigned 97
קביעת ערכים התחלתיים למשתנים 99
הגדרת מערכים 100
פקודת MOV 102
העתקה מרגיסטר לרגיסטר 104
העתקה של קבוע לרגיסטר 105
העתקה של רגיסטר אל תא בזיכרון 105
העתקה של תא בזיכרון אל רגיסטר 106
העתקה של קבוע לזיכרון 108
רגיסטרים חוקיים לגישה לזיכרון 108
תרגום אופרנד לכתובת בזיכרון 109
Little Endian, Big Endian 111
העתקה ממערכים ואל מערכים 111

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

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

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

פסיקות חומרה – Interrupts ... 240
בקר האינטרפטים – PIC ... 243
אובדן אינטרפטים ... 244
זיכרון קלט / פלט – I/O Ports ... 245
המקלדת ... 247
הקדמה ... 247
יצירת Scan Codes ושליחתם למעבד ... 248
באפר המקלדת Type Ahead Buffer ... 249
שימוש בפורטים של המקלדת ... 250
שימוש בפסיקת BIOS ... 255
שימוש בפסיקת DOS ... 256
סיכום ... 259
פרק 13 – כלים לפרוייקטים ... 260
מבוא לפרוייקטי סיום ... 260
בחירת פרוייקט סיום ... 260
עבודה עם קבצים ... 262
פתיחת קובץ ... 262
קריאה מקובץ ... 263
כתיבה לקובץ ... 263
סגירת קובץ ... 264
פקודות נוספות של קבצים ... 265
תכנית לדוגמה – filewrt.txt ... 265
גרפיקה ... 268
גרפיקה ב־Text Mode ... 269
שימוש במחרוזות ASCII ... 269
גרפיקה ב־Graphic Mode ... 273
הדפסת פיקסל למסך ... 274
קריאת ערך הצבע של פיקסל מהמסך ... 276
יצירת קווים ומלבנים על המסך ... 277

קריאת תמונה בפורמט BMP ... 278
קטעי קוד וטיפים בנושא גרפיקה ... 285
השמעת צלילים ... 286
שעון ... 290
מדידת זמן ... 290
יצירת מספרים אקראיים – Random Numbers ... 293
ממשק משתמש ... 298
קליטת פקודות מהמקלדת ... 298
קליטת פקודות מהעכבר ... 298
שיטות ניפוי Debug ... 302
תיעוד ... 302
תכנון מוקדם – יצירת תרשים זרימה ... 302
חלוקה לפרוצדורות ... 304
מעקב אחרי מונים ... 305
העתקות זיכרון ... 305
הודעות שגיאה של האסמבלר ... 306
סיכום ... 307
נספח א' – רשימת פקודות חובה לבגרות בכתב באסמבלי ... 308
נספח ב' – מדריך לתלמידים: כיצד נכנסים לפורום האסמבלי הארצי ... 312
זכויות יוצרים – מקורות חיצוניים ... 317

הקדמה

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

הספר כולל את הבסיס שנדרש לכתיבת תוכניות באסמבלי, בהתאם לתכנית הלימודים של משרד החינוך עבור יחידת המעבדה. תלמידים המעוניינים להעמיק את הידע שלהם, ימצאו בספר את הרקע התיאורטי לתכנים שהם מעבר לתוכנית הלימודים. כמו כן, הספר כולל הדרכה וכלים לביצוע פרוייקטי סיום מורכבים, מעבר לדרישות משרד החינוך. הנושאים שהינם מעבר לתכנית הלימודים מסומנים בספר כ"הרחבה".

לכתיבה בשפת אסמבלי קיימות מספר סביבות פיתוח, שפותחו על-ידי חברות שונות. הקוד שבספר כתוב בסביבת העבודה TASM, והספר ממליץ על אוסף תוכנות לשימוש בתור סביבת עבודה - גם כדי לחסוך לקוראים את הצורך לחפש סביבת עבודה באופן עצמאי, וגם כדי שיהיה סטנדרט אחיד לעבודה ולתרגול בכיתות. מומלץ להתקין את סביבת העבודה ולהשתמש בה כבר בתחילת הלימוד, וכן לפתור את התרגילים המופיעים בפרקים 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 ללומדי מחשבים בכלל וסייבר בפרט.
- בפרק 2 נלמד איך לייצג מספרים בשיטות ספירה שמקלות על עבודה עם מחשב.
- בפרק 3 נלמד על מבנה המעבד במחשב. כיוון שיש לא מעט סוגי מעבדים, נבחר משפחת מעבדים נפוצה ונתמקד בה.
- בפרק 4 נלמד להתקין את סביבת העבודה, נכתוב ונריץ את התוכנית הראשונה שלנו באסמבלי.
- בפרק 5 נלמד על רכיבים במעבד שמאפשרים לו לדעת מה מצב התוכנית - רגיסטרים ייעודיים.
- בפרק 6 נלמד איך מגדירים משתנים בזיכרון, כולל מחרוזות ומערכים, ואיך מבצעים העתקה של ערכים מהזיכרון ואל הזיכרון.
- בפרק 7 נלמד פקודות אריתמטיות (חיבור, חיסור וכו'), פקודות לוגיות ופקודות הזזה.
- בפרק 8 נלמד איך להגדיר תנאים לוגיים ואיך לכתוב פקודות בקרה. בסוף הפרק תוכלו לכתוב אלגוריתמים שונים, לדוגמה תוכנית שמבצעת מיון של איברים במערך.
- בפרק 9 נלמד על אזור בזיכרון שנקרא מחסנית, נלמד לכתוב פרוצדורות, להעביר אליהן פרמטרים ולהגדיר משתנים מקומיים. בסיום הפרק נראה איך החומר שלמדנו מאפשר לנו להבין נושאי תוכנה מתקדמים.
- בפרק 10 נקבל רקע וכלים בסיסיים הדרושים להשתתפות בתחרות קודגורו אקסטרים, בין היתר נלמד את העקרון הבסיסי של Reverse Engineering על ידי פיצוח "זומבים", חידות תוכנה שמפרסם צוות התחרות.
- בפרק 11 נלמד על פסיקות. נלמד איך להשתמש במגוון פסיקות DOS כגון קריאה של תו מהמקלדת או הדפסה של מחרוזת למסך.

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

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

הערות לגרסה 2.0

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

- חומר בנושא תחרות קודגורו 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 כיחידת בחירה במחשבים (ולא כיחידת מעבדה במסגרת מגמת סייבר) ימצאו בספר התייחסות לפקודות אסמבלי שנדרשות על ידי משרד החינוך בבחינת הבגרות.

פורום ארצי לעזרה ושאלות

לרשות התלמידים עומדת "אוניברסיטת" גבהים. 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 גבהים, מורים ועוזרי הוראה. הוראות הרשמה והתחברות לאתר ניתן למצוא בנספח שבסוף ספר זה.

ספרי לימוד ותרגול

הספר Art of assembly מאת Randall Hyde הוא ככל הנראה המקור המקיף ביותר ללימוד השפה. אמנם הקוד שם כתוב בהתאם לחוקי כתיבה מעט שונים מאשר בספר זה (NASM לעומת TASM, למען הדיוק), אך הספר הינו מקור מעולה להבנת השפה ולהסברים אודות פקודות אסמבלי. ניתן להוריד אותו בחינם מהאינטרנט.

חוברת התרגילים "הכנה לבגרות 5 יח"ל במדעי המחשב" מאת רונית גל אור מרציאנו מכילה תרגילים רבים בנושאים שנדרשים על ידי משרד החינוך לפרק האסמבלי בבגרות במחשבים. התרגילים הם בהיקף ובסגנון השאלות בבחינות הבגרות.

לתלמידים

בספר זה תלמדו את הבסיס להמשך לימודי הסייבר. ספר הלימוד יעזור לכם להגיע להבנה עמוקה של מבנה המחשב ולרמת תכנות 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 במהלכו תלמדו לא מעט ותרכשו יכולת חדשה וחשובה להמשך. יחד עם לימוד האסמבלי כדאי שתפתחו את סט הכלים שיאפשר לכם ללמוד לפתור בעיות לבד: חיפוש באינטרנט, סינון חומר, התמקדות בעיקר, ניסוי וטעיה. בסופו של דבר, אלו הן המיומנויות שהכי יעזרו לכם להצליח, גם בסייבר וגם בחיים.

<u>אייקונים</u>

בספר, אנו משתמשים באייקונים הבאים בכדי להדגיש נושאים ולהקל על הקריאה:

הגדרה למונח

תרגיל לביצוע

הדגשת נקודה חשובה

חומר הרחבה

דוגמה. תרגיל מודרך שכולל הסבר

פרק 1 – מבוא ללימוד אסמבלי

ברוכים הבאים! אם אתם קוראים את השורות האלה, כנראה שהחלטתם ללמוד אסמבלי. או שאולי מישהו אחר החליט במקומכם שאתם צריכים ללמוד אסמבלי? כך או כך, זו החלטה טובה. מיד ננסה להבין למה חשוב שתדעו אסמבלי.

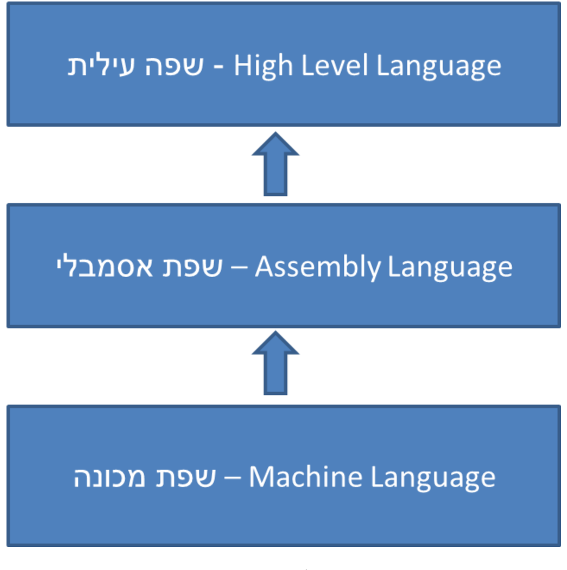
אסמבלי היא שפת התכנות הראשונה בעולם, שפותחה לפני עשרות שנים בתור תחליף פשוט יחסית לשפת מכונה, שבה כותבים באחדות ובאפסים. במילים אחרות, לפני שפותחה שפת אסמבלי, מי שרצה לתכנת היה צריך להזין למחשב רצף של אחדות ואפסים. ומה קורה אם מתבלבלים? אם מחליפים באיזשהו מקום 0 ב-1? חבל מאוד.

שפת אסמבלי הביאה לייעול ניכר בכך שהיא אפשרה לכתוב פקודות שניתנות להבנה בשפה אנושית. במקביל לפיתוח שפת אסמבלי פותח אסמבלר, שהוא כלי שמסוגל להמיר את פקודות האסמבלי לשפת מכונה.

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

אסמבלי נחשבת Low Level Language. הסיבה היא שמי שמתכנת באסמבלי כותב פקודות שעובדות ישירות מול החומרה של המחשב – אי אפשר לתכנת אסמבלי בלי להבין איך עובדים המעבד, הזיכרון ורכיבי החומרה שקשורים אליהם. כל פקודה באסמבלי מיתרגמת לפקודה אחת בשפת מכונה- לדוגמה העתקה, חיבור, כפל וכו'.

עם השנים פותחו שפות תוכנה עיליות, או High Level Languages. בשפות אלו שפת התוכנה מסתירה מהמתכנת את ה"קרביים" של המחשב ומאפשרת לתכנת בצורה יותר פשוטה וקלה. שפות תכנות אלו כוללות בין היתר את C++, Java ו-Python. פקודה בשפה עילית עשויה להיות מתורגמת למספר פקודות בשפת מכונה.



רמות של שפות תוכנה

אז למה בעצם ללמוד אסמבלי?

ישנן דווקא לא מעט סיבות למה **לא** ללמוד אסמבלי. הנה כמה מהן:

- מאז שאסמבלי פותחה העולם התקדם ויש שפות תוכנה מודרניות.
- אסמבלי היא שפה מורכבת יחסית ללימוד.
- מסובך לכתוב קוד באסמבלי.
- קשה לדבג (למצוא שגיאות ולנפות אותן) באסמבלי.
- קוד אסמבלי הוא תלוי מעבד. כלומר, קוד שנכתב למשפחת מעבדים מסויימת לא יתאים למשפחת מעבדים שונה.
- קוד המקור של תוכנית שכתובה באסמבלי כמעט תמיד יהיה ארוך יותר מאשר קוד של תוכנית שמבצעת את אותן הפעולות וכתובה בשפת תכנות אחרת.

ואמנם, מי שרגיל לתכנת בשפות עיליות בדרך כלל לא מתלהב לתכנת באסמבלי. מתכנתים מנוסים שלומדים אסמבלי, חשים לעיתים קרובות שבשפות אחרות אפשר לעשות הרבה יותר בקלות את הפעולות הנלמדות בשיעורי אסמבלי. זה באמת נכון, עד שמגיעים למגבלות של שפות תכנות אחרות, מה שמביא אותנו לסיבות למה **כן** ללמוד אסמבלי:

- שפת אסמבלי תורמת להבנה מעמיקה של המחשב על חלקיו השונים. העובדה שאתם נחשפים לחלקים הפנימיים ביותר של המחשב ושום דבר אינו מהווה "קופסה שחורה" בשבילכם, תאפשר לכם בעתיד להתמודד עם בעיות תכנות בלתי שגרתיות, שנדרשות בעולם הסייבר.
- שליטה בשפת אסמבלי היא כלי טכנולוגי חשוב בעולם הסייבר. לדוגמה, כדי לבצע מחקר קוד על ידי Reverse Engineering וכדי להבין בעיות אבטחה כמו Stack Overflow. הבנת בעיות האבטחה וכלי מחקר קוד מאפשרת, בין היתר, כתיבת קוד מוגן יותר בפני בעיות אבטחה.
- אסמבלי עובדת בצורה צמודה עם החומרה של המחשב. אם צריך לכתוב קוד שעובד מול חומרה, או להפעיל חומרה בצורה לא שגרתית, אז שימוש באסמבלי הוא עדיין בחירה נפוצה. מסיבה זו, חברת Apple לדוגמה, ממליצה לכותבי אפליקציות לשלוט באסמבלי.
- המקום שתוכנית אסמבלי תופסת בזיכרון הוא קטן למדי יחסית לתוכנות שכתובות בשפות עיליות.

יכול להיות שהשתכנעתם בחשיבות לימוד אסמבלי ויכול להיות שלא. בכל אופן – שימו לב לרשימת הדרישות שנוסחה על־ידי חברת אבטחת מידע ישראלית, ממי שמעוניין להגיש מועמדות למשרה בתחום הסייבר:

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 נציין, עוד מספר מטרות של לימוד האסמבלי במגמת גבהים.

ראשית, רכישת מיומנויות של סדר, ארגון וחשיבה מתודית. שפת אסמבלי היא מקום טוב במיוחד לרכוש בו יכולות תכנות מאורגן ומסודר, בגלל הדקדקנות והירידה לפרטים שנדרשת כדי לכתוב קוד תקין. יכולות אלו הן אבני הבניין לעבודה בתחום הסייבר.

שנית, הקניית יכולת לימוד עצמית והתמודדות עם אתגרים. במסגרת גבהים תידרשו לכתוב קוד בהיקף משמעותי ולדבג אותו. זהו אתגר שיכשיר אתכם לקראת הבאות.

שלישית, ידע מקדים באסמבלי נדרש לחומר הלימוד של גבהים במערכות הפעלה.

אז קדימה, אנחנו מוכנים להתחיל במסע נפלא בו נלמד רבות על דרך הפעולה של המחשב, על כתיבת קוד ועל איך ללמוד בעצמנו. נתחיל מהבסיס – שיטות ספירה וייצוג מידע במחשב.

פרק 2 – שיטות ספירה וייצוג מידע במחשב

מבוא – מהן שיטות ספירה?

לבני האדם יש עשר אצבעות ולכן ספירה בשיטה העשרונית (בסיס עשר, Decimal) נראית לנו טבעית מילדות, אך קיימות שיטות רבות לספור. לעומת ספירה בבסיס עשר, שיטות הספירה האחרות דורשות מאיתנו מאמץ – אדרבא כשמדובר בשיטה ההקסדצימלית (בסיס שש עשרה, Hexdecimal) שבה אותיות מייצגות חלק מהספרות. ייתכן שאילו היינו חייזרים בעלי שש עשרה אצבעות, היה לנו יותר נוח לספור בבסיס הקסדצימלי. בכל מקרה, תכנות בשפת אסמבלי מצריך הבנה של ייצוג מספרים בשיטה הבינארית (בסיס שתיים, Binary) ובשיטה ההקסדצימלית.

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

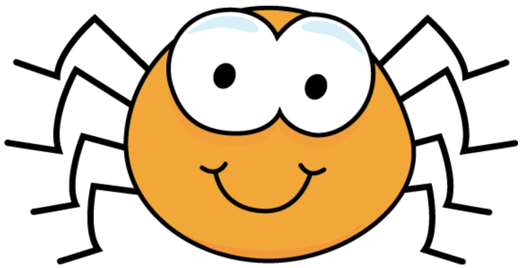
בפרק זה נלמד שיטות לייצג מספרים בבסיסי ספירה שונים תוך התמקדות בשיטה הבינארית ובשיטה ההקסדצימלית, נבצע את פעולות חשבון הבסיסיות בבסיסים שאינם בסיס עשר, נראה איך מייצגים מספרים שליליים בשיטות שונות ונתמקד בשיטה שנקראת "המשלים לשתיים". בואו נתחיל.

שיטות ספירה הן בסך הכל דרכים שונות לייצג כמות נתונה של פרטים שאפשר לספור אותם. נסתכל על הטבלה הבאה, שרשומים בה מספרים בשלושה בסיסים- בסיס 10, בסיס 8 ובסיס 3.

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

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

ניקח לדוגמה עכביש. כמה רגליים יש לו?



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

משחק: Pearls Before Swine 3

לפני שנתקדם, בואו נשחק משחק קטן. המשחק קשור לחומר הלימוד בפרק זה, אך בשלב זה לא נגלה יותר מזה.

היכנסו לקישור הבא: www.transience.com.au/pearl3.html

חוקי המשחק מוסברים על ידי "חואן", השדון בעל השיניים הצהורות. נסו לנצח אותו!

"חואן". נצחו אותו!

אל תהיו מתוסכלים אם לא הצלחתם 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 בבסיסים שאנחנו לא רגילים אליהם וכדי להקל, נתחיל דווקא מייצוג מספרים בבסיס עשר המוכר.בשיטה העשרונית קיימות עשר ספרות: 0,1,2,3,4,5,6,7,8,9. בעזרת הספרות האלו אנחנו מייצגים את כל המספרים. הערך של ספרה נקבע לפי המיקום שלה. כך, המספר 501 שונה מהמספר 105. במספר 501 הספרה 5 היא ספרת המאות, ואילו במספר 105 הספרה 5 היא ספרת האחדות.

שימו לב – מעכשיו נכתוב את בסיס הספירה בקטן, בתחתית המספר, כך:

47_{10}

נעשה זאת כדי להבדיל מספרים שנראים אותו דבר אך מייצגים ערכים שונים בבסיסי ספירה שונים. לדוגמה:

$47_{10} \neq 47_8$

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

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

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

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

חישוב ערך של מספר עשרוני בבסיס אחר

נהפוך את המספר 199_{10} לייצוג שלו בבסיס 5:

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

נתחיל בחלוקה 199:5. התוצאה היא 39 והשארית היא 4.

נחלק את התוצאה 39:5. התוצאה היא 7, השארית 4.

חלוקה של 7:5 היא 1, שארית 2.

חלוקה של 1:5 היא 0, שארית 1.

כעת נרשום את כל השאריות, מלמטה למעלה - 1244.

ומכאן ש- $1244_5 = 199_{10}$.

נבדוק את החישוב שעשינו, על-ידי ביצוע הפעולה ההפוכה- 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

בספירה לפי בסיס שתיים, ספירה שנקראת גם השיטה הבינארית, קיימות שתי ספרות בלבד: 0,1. כלומר הספרה 2 לא קיימת וצריך לייצג אותה על-ידי שתי ספרות. השיטה הבינארית חשובה במיוחד בהקשר של מערכות מחשב, כיוון שכל המידע מיוצג בזיכרון המחשב באמצעות רצפים של אחדות ואפסים. המחשב לא מכיר את הספרה 2.

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

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

ניקח לדוגמה את המספר 10011_2. נתרגם את המספר הזה למספר עשרוני. לטובת קלות ההצגה נכניס 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**

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

המרה מבסיס עשרוני לבינארי מתבצעת בדיוק באותה שיטה שהדגמנו בסעיף "חישוב ערכו של מספר עשרוני בבסיס אחר", רק שהפעם הבסיס האחר הוא 2. ניקח את המספר 19:

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

וכשמעתיקים את השאריות מלמטה למעלה, מקבלים את 10011_2.

השיטה ההקסדצימלית

בספירה לפי בסיס 16, ספירה שנקראת 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 וכך הלאה. בטבלה הבאה מרוכזים 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

בשיטה ההקסדצימלית, מספר יכול להיות צירוף של אותיות וספרות. לדוגמה $F15_{16}$ $C1A_{16}$, $4C4_{16}$, $2B_{16}$, $1A_{16}$.
בשיטה זו אפשר להשתעשע עם מספרים בעלי ערכים נחמדים כמו $DEAD_{16}$, $C0FFEE_{16}$ או כמו $C0DE_{16}$.

שימו לב שאנחנו משתמשים בספרה 0, לאות האנגלית o אין משמעות בייצוג הקסדצימלי.

השיטות הבאות כולן שיטות תקפות לייצוג מספרים הקסדצימליים:

- רישום הבסיס 16 מתחת למספר – לדוגמה $C0DE_{16}$
- הוספת האות h בסוף המספר. בשיטת ייצוג זו, כאשר המספר מתחיל באות, מוסיפים '0' מצד שמאל. לדוגמה 0C0DEh (נדגיש כי ניתן להוסיף אפסים מצד שמאל בכל מקרה, אבל כאשר המספר מתחיל באות- חייבים להוסיף)
- הוספת הצירוף 0x בתחילת המספר – לדוגמה 0xC0DE

לאחר שהשתעשענו, נבצע תרגום של מספר, $4F_{16}$, מהקסדצימלי לעשרוני:

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

נבצע את הפעולה ההפוכה: תרגום מבסיס עשרוני להקסדצימלי, למספר 199_{10}:

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

נחלק 199:16. התוצאה 12, השארית 7.

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

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

כעת נבצע המרה של מספרים בינאריים להקסדצימליים. להמרה זו יש מאפיין מיוחד, מכיוון ש-16 הוא חזקה של 2. נראה מיד איך תכונה זו באה 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 הקסדצימלית ניתנת לייצוג על־ידי מספר בינארי בן ארבע ספרות. בעקבות זאת, ההמרה מהקסדצימלי לבינארי יכולה להתבצע פשוט על־ידי מעבר של ספרה אחר ספרה, והחלפת הספרה ההקסדצימלית בארבע ספרות בינאריות. ניקח לדוגמה את המספר $9B_{16}$:

$9_{16}=1001_2$

$B_{16}=1011_2$

$9B_{16} = 10011011_2$

התכונה הזו, המעבר הפשוט מבינארי להקסדצימלי, הוא מה שהופך את השיטה ההקסדצימלית לשימושית בעולם המחשבים. במקום לזכור רצפים של אחדות ואפסים שנמצאים בזיכרון המחשב, יותר קל להכניס ערכים הקסדצימליים והסיכוי לטעות נעשה נמוך יותר. חישבו על מספר כמו $9B2C_{16}$, איך יותר קל לכתוב אותו – כמספר הקסדצימלי או כמספר בינארי, שהוא 1001101100101100? איפה יש פחות סיכון לטעות?

תרגיל 2.1: מעבר בין בסיסי ספירה

השלימו את הטבלה הבאה – בטור השמאלי מספרים בבסיס 10, דצימלי. בטור האמצעי מספרים בבסיס 2, בינארי. בטור הימני מספרים בבסיס 16, הקסדצימלי.

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

פעולות חשבון

נלמד את ארבע פעולות החשבון הבסיסיות – חיבור, חיסור, כפל וחילוק – בבסיסים שאינם בסיס עשר.

חיבור

נתחיל מתרגול בבסיס עשר. נחבר את המספרים 133 ו־70.

```
   1
  133
+  70
  ---
  203
```

אנחנו מתחילים את החיבור מצד ימין ומחברים כל ספרה עם הספרה שמתחתיה. כאשר מקבלים תוצאה שהיא שווה או גדולה מ־10, אנחנו כותבים למטה רק את ספרת האחדות ואילו את ספרת העשרות אנו מוסיפים ("נושאים" או באנגלית carry – נשא) לזוג הספרות שמשמאל.

חיבור מספרים בינאריים אינו שונה ברמת העיקרון, פרט לכך ששארית מתקבלת עם כל תוצאת חיבור שהיא שווה או גדולה מ־2. נתבונן באפשרויות החיבור השונות, המרוכזות בטבלה למטה.

החיבור של 0 ועוד 0 שווה 0.

חיבור של 0 ועוד 1 שווה 1.

חיבור של 1 ועוד 0 שווה 1.

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

+	0	1
0	0	1
1	1	10

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

```
  1
 1010
+  11
 ----
 1101
```

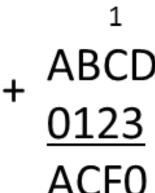
התחלנו מצד ימין. החיבור של 0 ועוד 1 שווה 1, ללא נשא.

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 שנגיע לתוצאה שווה או גדולה מ-16 נעביר נשא שמאלה. לדוגמה, חיבור של $ABCD_{16}$ עם 123_{16}:



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

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

B ועוד 1 שווה C, ללא נשא.

A ועוד 0 שווה ל-A.

תרגיל 2.2: חיבור

חיבור בבסיס בינארי:

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

חיבור בבסיס הקסדצימלי:

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

חיסור (הרחבה)

כאשר מחסרים שתי ספרות, ישנן שלוש אפשרויות:

1. הספרה הראשונה גדולה מהשניה. לדוגמה 9 פחות 6. במקרה זה פעולת החיסור פשוטה.
2. הספרות שוות, לכן תוצאת החיסור היא אפס.
3. הספרה השניה גדולה מהראשונה. לדוגמה 6 פחות 9. במקרה זה נשתמש ב"נשא שלילי". בשפה פשוטה, ניקח יחידה אחת מהספרה שמצד שמאל.

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

```
 -1 10
  619
-
   21
 ----
  598
```

החיסור של 9 פחות 1 שווה 8.

בשביל החיסור של 1 פחות 2 היינו 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 עכשיו בבסיס 2 את הפעולה 1010 פחות 1 (10 פחות 1).

```
    -110
  1010
-
     1
  ----
  1001
```

נתחיל מימין. חיסור של 0 פחות 1 מצריך נשא שלילי. אנחנו מורידים את הנשא השלילי מהספרה הבאה, ומוסיפים את הנשא השלילי (10 בבסיס 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 החיסור.

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

כעת נבצע תרגיל חיסור בבסיס 16:

```
    -1 16
  -1 16
- DEAD
  C0DE
  ----
  1DCF
```

הסבר:

חיסור של D פחות E מצריך נשא שלילי. נוריד יחידה מ־A ונפרוט אותה לשש עשרה יחידות שמתווספות ל־D. תוצאת החיסור היא לכן F (בהמרה לעשרוני – 13 ועוד 16 פחות 14, שווה 15).

בטור השני, הורדנו יחידה מ־A וצריך לחסר גם את D. שוב נצטרך להשתמש בנשא שלילי, ניקח יחידה מהטור השלישי. תוצאת החיסור היא לכן C (בהמרה לעשרוני – 10 פחות 1, ועוד 16 פחות 13, שווה 12).

בטור השלישי, הורדנו יחידה מ־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 הקסדצימלי:

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

כפל (הרחבה)

פעולת הכפל של מספרים בינאריים היא כנראה פשוטה אפילו יותר מאשר כפל מספרים עשרוניים. אחרי הכל, בזמן שלוח הכפל של מספרים עשרוניים הוא בגודל 10X10, לוח הכפל של מספרים בינאריים הוא בגודל 2X2 ואפשר לסכם אותו בטבלה הבאה:

X	0	1
0	0	0
1	0	1

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

```
 1010
x  11
 ----
 1010
1010-
-----
11110
```

הסבר:

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

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

לאחר מכן מחברים את התוצאות חיבור פשוט ומתקבלת התוצאה 11110 (או בבסיס עשר – 30=2+4+8+16).

תופעה מעניינת – כפל בחזקות של 2: כאשר לוקחים מספר בבסיס 10, וכופלים אותו בעשר, התוצאה היא המספר המקורי מוזז בספרה אחת שמאלה, כאשר במקום הכי ימני נכנסת הספרה 0. לדוגמה, 52x10=520.

דוגמה נוספת, עם חזקה יותר גבוהה של 10: 52x100=5200.

אותה התופעה מתרחשת בבסיס 2, כאשר כופלים במספר שהוא שתיים או חזקה של שתיים. על כל חזקה של שתיים, מזיזים ספרה אחת שמאלה ומוסיפים את הספרה 0.

לדוגמה, כפל בשתיים: 11x10=110.

כפל בארבע: 11x100=1100.

כפל בשמונה: 11x1000=11000.

וכן הלאה.

ביצוע פעולות כפל בבסיס הקסדצימלי: לוח הכפל בבסיס 16 כולל 16X16 איברים, כלומר 256 איברים. אך אל דאגה – אין צורך לזכור אותו בעל פה. הדרך הפשוטה יותר היא להמיר את המספרים לבסיס בינארי ולהמשיך משם. לדוגמה, כפל של C כפול 5, לפי טבלת ההמרה מתקבל בקלות 1100 כפול 101 והדרך משם ברורה.

A	B	AxB
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 בינארי 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 כשאין יותר ספרות להוסיף.

נבצע דוגמה – $10110_2/101_2$:

```
   100
 ______
 10110| 101
-101
 ___
   010
```

התוצאה היא 100, שארית 10.

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

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

זיכרון המחשב בנוי מתאים, שבתוכם שמורים מספרים בינאריים. כל ספרה בינארית (0 או 1) נקראת **סיבית (Bit)**. זהו קיצור של המילים ספרה בינארית (Binary digit). נניח עכשיו הנחת יסוד, שאומרת שיש לנו מגבלה על כמות הביטים שאנו יכולים להשתמש בה בשביל לייצג מספר. במחשבים זוהי מגבלה מאוד מעשית, כיוון שמחשבים שומרים מספרים בתאים בגודל קבוע. הגדלים המקובלים הם 8, 16, 32 או 64 ביטים. כלומר המגבלה על כמות הביטים לייצוג מספר היא חלק בסיסי באופן שבו מחשבים פועלים.

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

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

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

מה קורה אם אנחנו מנסים לייצג מספר שהוא יותר גדול מהמספר הכי גדול שאפשר לייצג על־ידי N ביטים? נניח, לבצע את פעולת 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 שמונת הביטים הראשונים מתאפסים, ומתקבל נשא. הדרך שבה המחשב מטפל במקרים כאלה היא כזו: הזיכרון שלנו, שלפני כן הכיל את הערך 11111111, מכיל עכשיו סדרה של שמונה אפסים. המחשב מדליק במקום אחר ביט, שאומר שהיה נשא בפעולה האחרונה (נלמד עליו בהמשך). זהו. מעכשיו בזיכרון שלנו יש 00000000. כלומר, מבחינת המחשב, כשמגבילים אותו לייצוג של 8 ביטים, אז המשוואה 255+1=0 היא נכונה. עלינו להבין כי כך המחשב עובד, ונצטרך להתאים את עצמנו כדי שלא יקרו לנו טעויות כאלו.

ייצוג מספרים שליליים

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

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

שיטת גודל וסימן

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

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

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

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

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

בשיטה זו, המספר הכי גבוה שאנחנו יכולים לייצג על־ידי 4 ביטים הוא 0111, כלומר 7, ואילו המספר הכי נמוך הוא 1111, כלומר מינוס 7.

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

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

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

החיסרון המרכזי של השיטה הוא שמסובך לעשות תרגילי חשבון. נסתכל לדוגמה על 3 ועוד (3-):

```
 0011
+
 1011
 ----
 1110
```

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

שיטת המשלים לאחת

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

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

ייצוג המספרים מ־0 עד 7 והנגדיים שלהם:

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

שימו לב לתופעה שהתרחשה מאליה - הביט השמאלי הוא ביט הסימן. כשהוא 0 המספר חיובי וכשהוא 1 המספר שלילי. כפי שאפשר לראות, חיבור של כל מספר עם הנגדי שלו נותן 1111, שבשיטה זו שקול לערך 0. כלומר התגברנו על מכשול אחד.

בשיטת המשלים לאחת, כדי שגם יתר פעולות החשבון ייתנו תוצאות נכונות, משתמשים בחוק הבא: אם לתוצאת הפעולה יש נשא, מחברים אותו לביט הימני ביותר.

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

```
    1  1
    0101
+
    1101
    ----
 (1)0010
    0011
```

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

היתרונות של שיטת המשלים לאחת, הם שפשוט מאוד למצוא את המספר הנגדי (רק צריך להפוך ביטים) ושפעולות החשבון יוצאות נכון. החסרונות של השיטה הם הסיבוך שבהוספת הנשא לביט הימני, אבל בעיקר – העובדה שיש שתי דרכים לייצג את הערך אפס. כדי ללמד את המחשב ש- 0000=1111 צריך להשקיע עוד זמן ומאמץ, ועדיף פשוט לעבור לשיטת המשלים לשתיים.

שיטת המשלים לשתיים

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

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

כדוגמה, נמצא את הייצוג של מינוס 6, בייצוג שלו על־ידי 8 ביטים:

הייצוג של 6 הוא 00000110.

לאחר הפיכת הביטים תוצאת הביניים היא 11111001.

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

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

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

וכפי שרואים, כל שמונת הביטים אכן התאפסו (זיכרו – ה־1 שמצד שמאל הוא נשא והוא אינו נכנס בשמונת הביטים).

נסתכל על הייצוג של כמה מספרים בינאריים בשיטת המשלים לשתיים. לטובת ההמשך, אנחנו נסתכל על הייצוג ב־8 ביטים, למרות שבשביל לייצג את המספרים שבטבלה אפשר היה להסתפק גם ב־4 ביטים.

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

כפי שאנחנו רואים, בשיטה זו יש רק ייצוג אחד לאפס – 0000 0000. זהו יתרון 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 $(2^{N-1}-1)$ לבין (-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 פעולת חשבון שהתוצאה שלה חורגת מתחום הערכים שניתן לייצוג, תגרום להחזרת תשובה שגויה.

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

- ראשית, אם המספר לא מכיל בדיוק 8, 16, 32, 64 וכו' ביטים, אז מוסיפים אפסים מצד שמאל של המספר עד שמגיעים לכמות זו של ביטים.
- אם המספר חיובי (הביט השמאלי ביותר שלו הוא 0) אז ההמרה לעשרוני היא בדיוק כמו שלמדנו בעבר- לכל ביט יש מיקום, וכל מיקום קובע חזקה אחרת של 2.
- אם המספר שלילי (הביט השמאלי ביותר הוא 1), אז ממירים את המספר למספר חיובי בשיטת המשלים ל־2, מחשבים את הערך שלו כמו שעושים למספרים חיוביים, ושמים מינוס לפני התוצאה.

לדוגמה, המספר 10111111. הביט 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 לשכוח שהמספר הוא שלילי – נוסיף סימן מינוס ונגיע לתוצאה: (-65)

תרגיל 2.6: המשלים לשתיים

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

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

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

אז איך מנצחים את חואן?

... ומה הקשר של המשחק Pearls3 לבסיסי ספירה?

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

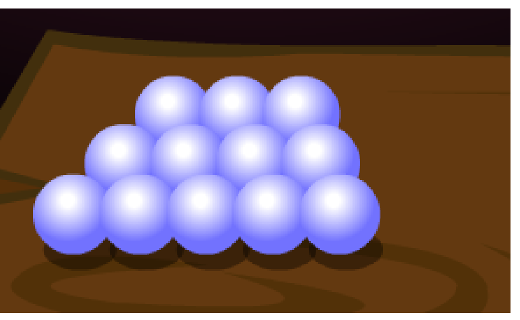
קיימת טכניקה שמבטיחה את הנצחון במשחק, ואנחנו נפרט עליה כאן. אם אתם סקרנים מדוע הטכניקה הזו עובדת, ישנו הסבר מתימטי, שהינו מחוץ להיקף של ספר זה, אך הוא מופיע בויקיפדיה ובמקומות אחרים באינטרנט.

כדי להבטיח ניצחון במשחק, יש לבצע את הצעדים הבאים:

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

דוגמה:

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



ייצוג בינארי (נשתמש בארבע ספרות בינאריות, המתאימות לייצוג מספרים עד 15, כך שסביר שהמספר יתחיל באפסים):

(3) 0011

(4) 0100

(5) 0101

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

0010

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

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

(1) 0001

(4) 0100

(5) 0101

הסכום:

0000

נמשיך בתהליך עד שנגיע למצב בו אנו יכולים להשאיר רק פנינה אחת ולנצח.

תרגיל 2.7: נצחו עשרה משחקים

נצחו את חואן בעשרת השלבים הראשונים! אתגר- נסו לא להפסיד באף משחק ולשמור על מאזן נצחונות מושלם.

ייצוג מידע במחשב

בתחילת הפרק, כשניסינו להסביר למה כדאי ללמוד בסיסי ספירה, נתנו הבטחה מעורפלת שבעזרת בסיס בינארי והקסדצימלי קל יותר לעבוד עם מחשבים. עכשיו הגיע הזמן לפרוע את השטר – נעבור לדבר על מה קורה בתוך המחשב, על הדרך בה מידע מיוצג במחשב. כך, נוכל לראות את השימושיות של החומר שלמדנו.

סיבית – Bit

היחידה הקטנה ביותר של מידע במחשב היא ביט בודד. ביט מקבל ערך אפס או אחד. באמצעות ביט בודד ניתן לייצג כל שני ערכים שונים זה מזה. לדוגמה, אפס או אחד, אמת או שקר, למעלה או למטה, ירוק או אדום, 340 או 519. כל שני ערכים שונים זה מזה ניתנים לייצוג על־ידי ביט בודד, אך ניתן לייצג באמצעותו רק שני ערכים שונים.

אם כך, כשמסתכלים על ביט, איך יודעים אילו שני ערכים הוא מייצג? התשובה היא, שאין דרך לדעת. כלומר, ניתן לפענח את המידע השמור על־ידי ביט במגוון דרכים, וכל דרך תיתן פרשנות אחרת. העיקר, בכתיבת קוד, הוא להיות עקבי לגבי הפרשנות שאנחנו נותנים לביטים השונים.

כדי להדגים את הנושא, ניזכר בחלק שבו דנו בייצוג מספרים בשיטה הבינארית. המספר 1100 בבסיס 2 יכול להיות מתורגם לבסיס עשרוני כ־12, אבל גם כמינוס 4 (היזכרו בשיטת המשלים לשתיים). הערך אותו רצף הביטים הזה מייצג תלוי בפרשנות שלנו – האם מדובר במספר signed או unsigned.

כל המידע שיש בזיכרון מחשב, תהיה משמעותו אשר תהיה, אגור תוך שימוש באחדות ואפסים בלבד. בזיכרון המחשב לא קיימת הספרה 2, לא קיים מינוס, לא קיים גדול או קטן. כל מה שקיים בזיכרון המחשב הוא רק רצף של אחדות ואפסים, שעל פי הפרשנות שהמחשב נותן להם מאפשרים ייצוג של כל דבר – החל מחישובים מתמטיים וכלה בסרטי וידאו.

כיוון שבאמצעות ביט בודד ניתן לייצג רק שני ערכים, השימוש בביטים בודדים הוא בעל שימושים מוגבלים. במקום זה, לרוב השימושים משתמשים באוספים של ביטים.

נגיסה – Nibble

Nibble הוא אוסף של ארבעה ביטים. Nibble יכול לייצג 2 בחזקת 4 ערכים שונים, כלומר 16 ערכים. במקרה של מספר הקסדצימלי, Nibble יכול לייצג את הערכים 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F ומכאן חשיבותו – כשאנחנו מסתכלים בזיכרון המחשב, לא נוח לקרוא אותו בבסיס 2, אחדות ואפסים בלבד. ייצוג בבסיס 16 ידחוס כל ארבעה ביטים לספרה אחת ויקל על הקריאה והזכירה. נראה דוגמה.

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

1101 1110 1010 1101 1100 0000 1101 1110

נתרגם את הרצף למספרים הקסדצימליים לפי טבלת ההמרה המוכרת:

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

ונקבל:

1101 1110 1010 1101 1100 0000 1101 1110

D E A D C 0 D E

...האם זכירת המידע קלה יותר עכשיו?

בית – Byte

בית (Byte) הוא אוסף של שמונה ביטים, והוא גם היחידה הקטנה ביותר שיש לה כתובת משלה – הזיכרון מחולק לכתובות, וכל כתובת היא של בית יחיד. משמעות הדבר היא שיחידת הזיכרון הקטנה ביותר שניתן לגשת אליה היא בית. כדי לגשת ליחידה קטנה יותר, נניח ביט, נדרש לקרוא את כל הבית שמכיל את הביט הרצוי, ואז על־ידי פעולה שנקראת מיסוך ביטים (נלמד עליה בהמשך, בחלק שמסביר פקודות לוגיות) אפשר לבודד את הביט.

הביטים בתוך כל בית ממוספרים לפי הסדר הבא:

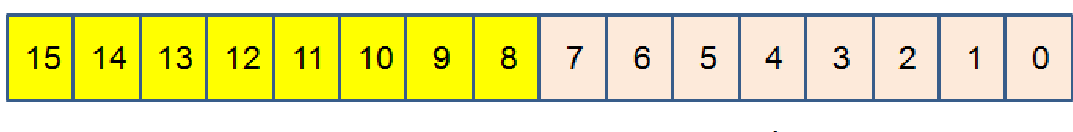
כאשר ניתן להתייחס אליהם גם כצירוף של שני Nibbles:

High Order Nibble Low Order Nibble

מילה – Word

מילה (Word) היא אוסף של 16 ביטים, או בדיוק שני בתים. באמצעות מילה ניתן לייצג שתיים בחזקת 16, או 65,536 ערכים שונים. באסמבלי 16 ביט, אליו מתייחס ספר זה, נשתמש במשתנים בגודל מילה בעיקר כדי לשמור ערכי מספרים או כתובות בזיכרון.

מספור של ביטים בתוך מילה וחלוקתם לבתים:



מילה כפולה – Double Word

מילה כפולה (Double Word או בקיצור DWORD) היא אוסף של 32 ביטים, או שתי מילים, או ארבעה בתים. מספר הערכים ש-**Word** Double יכול לקבל הוא 2 בחזקת 32, מה שמאפשר להחזיק מספרים Unsigned בתחום 0 עד 4,294,967,295, או מספרים Signed בתחום ‎-2,147,482,648 עד 2,147,482,647.

קוד ASCII

קוד American Standard Code for Information Interchange, או בקיצור ASCII, הוא הקוד הנפוץ לייצוג אותיות ותווים. הקוד משתמש ב-7 ביטים כדי לייצג 128 תווים, מכאן שלכל תו 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	♣ <enq>	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	↓ <em>	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 שאנחנו רגילים אליה. התמקדנו בשתי שיטות: השיטה הבינארית, שמשמשת לייצוג של ערכים בזיכרון המחשב, והשיטה ההקסדצימלית, שמקלה עלינו לקרוא ערכים בינאריים.

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

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

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

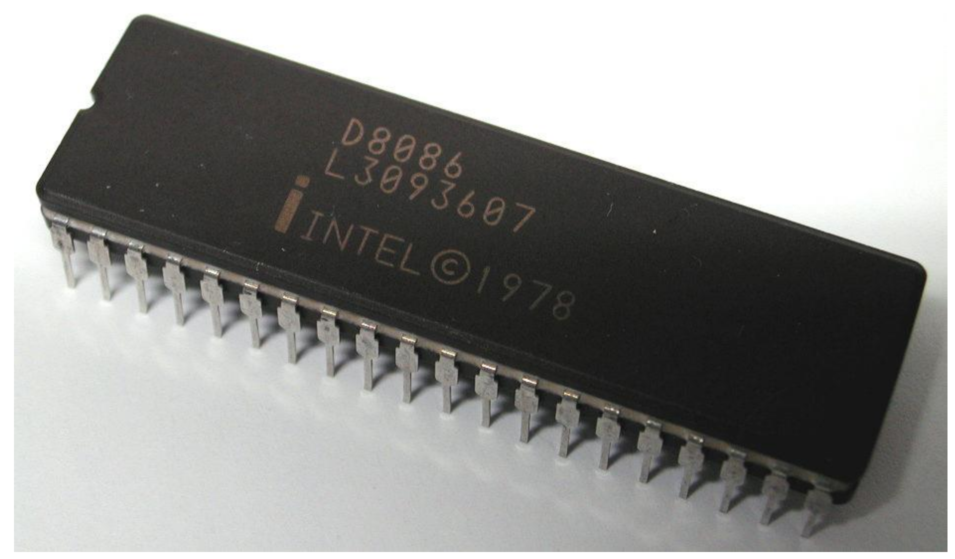
בסיום פרק זה, סקרנו איך מחשב שומר ערכים שונים בזיכרון וראינו את היתרונות של שימוש בבסיס בינארי והקסדצימלי. כעת אנחנו מבינים איך מספרים מסוגים שונים מיוצגים במחשב, ואנחנו בשלים לעבור לפרק הבא בו נרחיב ונתעמק במבנה המחשב.

פרק 3 – ארגון המחשב

מבוא

אסמבלי היא שפת התוכנה שעובדת בצורה הקרובה ביותר מול החומרה של המחשב. רק אסמבלר קטן מפריד בין הקוד שאתם כותבים באסמבלי לבין שפת מכונה, ובהמשך גם תתנסו בתרגום מאסמבלי לשפת מכונה. לכן כדי לכתוב תכנית, אפילו ***בסיסית***, בשפת אסמבלי, נדרשת הבנה של ארגון המחשב והחומרה. כדי לעסוק בסייבר באופן מקצועי צריך כישורים לביצוע פעולות מתקדמות כמו מציאת באגים בתוכנה, הקטנת גודל הזיכרון שקוד תופס או העלאת מהירות הריצה של תוכנה. לטובת פעולות מתקדמות כאלה נדרשת הבנה ***טובה*** של אופן פעולת המחשב והקשר בין החומרה לתוכנה.

אנחנו נלמד על ארגון המחשב באמצעות ניתוח הדרך שבה מאורגן מעבד ממשפחת 80x86 של אינטל. משפחת ה־80x86 כוללת מספר מעבדים, כאשר במקום ה־X ישנה ספרה שמציינת 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.



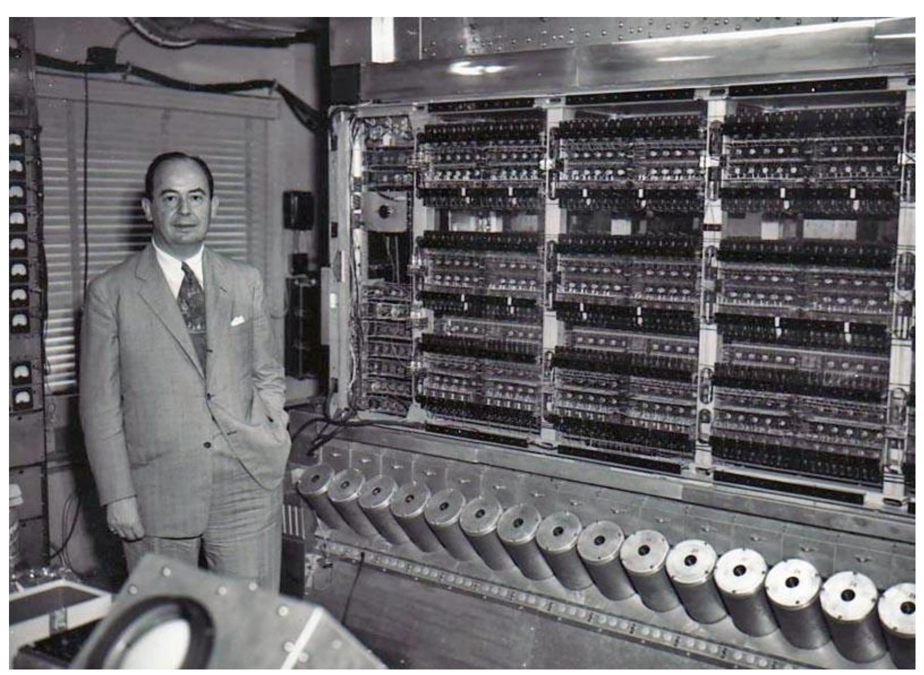
ראשית, ראוי שנשאל את עצמנו – איזו תועלת יכולה להיות ללימוד מעבד כל כך מיושן? הרי חומרת המחשבים משתנה כל מספר שנים בודדות. איך אפשר ללמוד משהו מעשי לעולם הסייבר באמצעות מעבד שמקומו בדפי ההיסטוריה?

ובכן, ישנן כמה תשובות לשאלה 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 שומרת על תאימות לאחור עם מעבדי ה־80x86. כלומר, אתם יכולים לכתוב פקודת אסמבלי שתרוץ על מעבד 8086, והמעבד החדש ביותר של אינטל יוכל להריץ אותה.
- בשביל ללמוד את ארגון המחשב צריך להתחיל ממקום כלשהו, ומעבד ה־80x86 הוא מקום טוב להתחיל ממנו.

מכונת פון נוימן – Von Neumann Machine

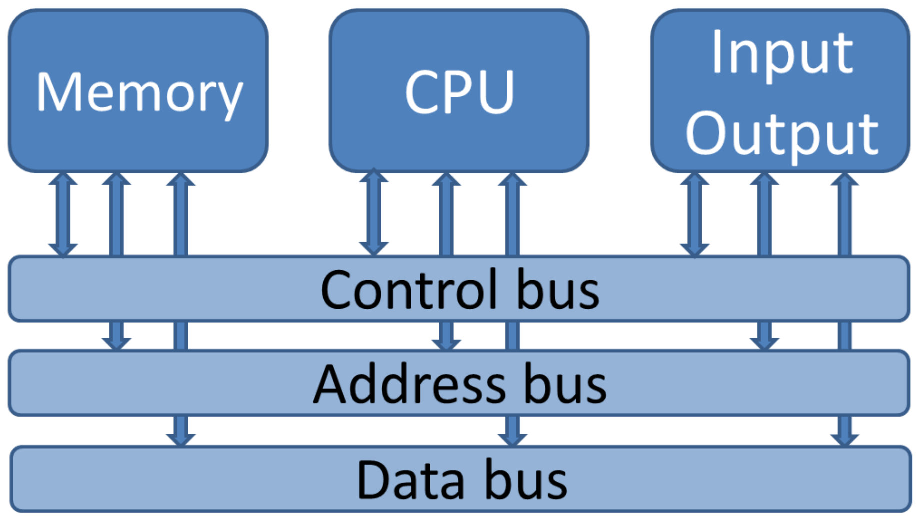
התכנון הבסיסי של מחשב נקרא ארכיטקטורה. ג'ון פון נוימן היה חלוץ בתכנון מחשבים, וניתן לו הקרדיט עבור הארכיטקטורה של רוב המחשבים שאנחנו משתמשים בהם היום. **ארכיטקטורת פון נוימן (או באנגלית Von Neumann Architecture – VNA)** כוללת שלוש אבני בניין מרכזיות: יחידת העיבוד המרכזית (Central Processing Unit – CPU),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



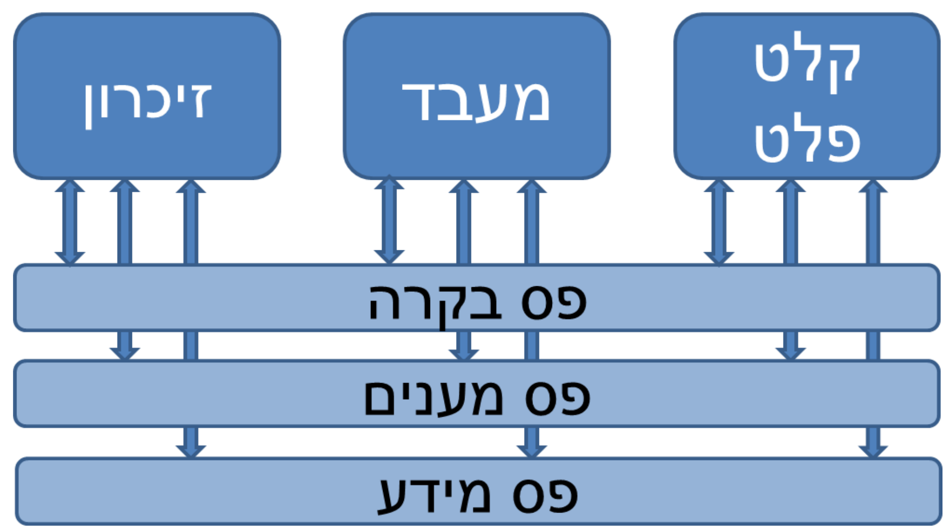
ג'ון פון נוימן John Von Neumann (1903–1957) והמחשב הדיגיטלי הראשון

במעבדים מבוססי ארכיטקטורת VNA, כמו משפחת ה־80x86, יחידת העיבוד המרכזית מבצעת את כל החישובים. מידע והוראות למעבד (מה שנקרא קוד התוכנית) נמצאים בזיכרון עד שהם נדרשים על־ידי המעבד. מבחינת המעבד, יחידות הקלט / פלט נראות כמו זיכרון, כיוון שהמעבד יכול לשלוח אליהן מידע ולקרוא מהן מידע. ההבדל העיקרי בין מקום בזיכרון לבין מקום קלט / פלט, הוא שיחידות הקלט / פלט בדרך כלל משויכות להתקנים שנמצאים בעולם החיצוני למעבד (כגון מקלדת).

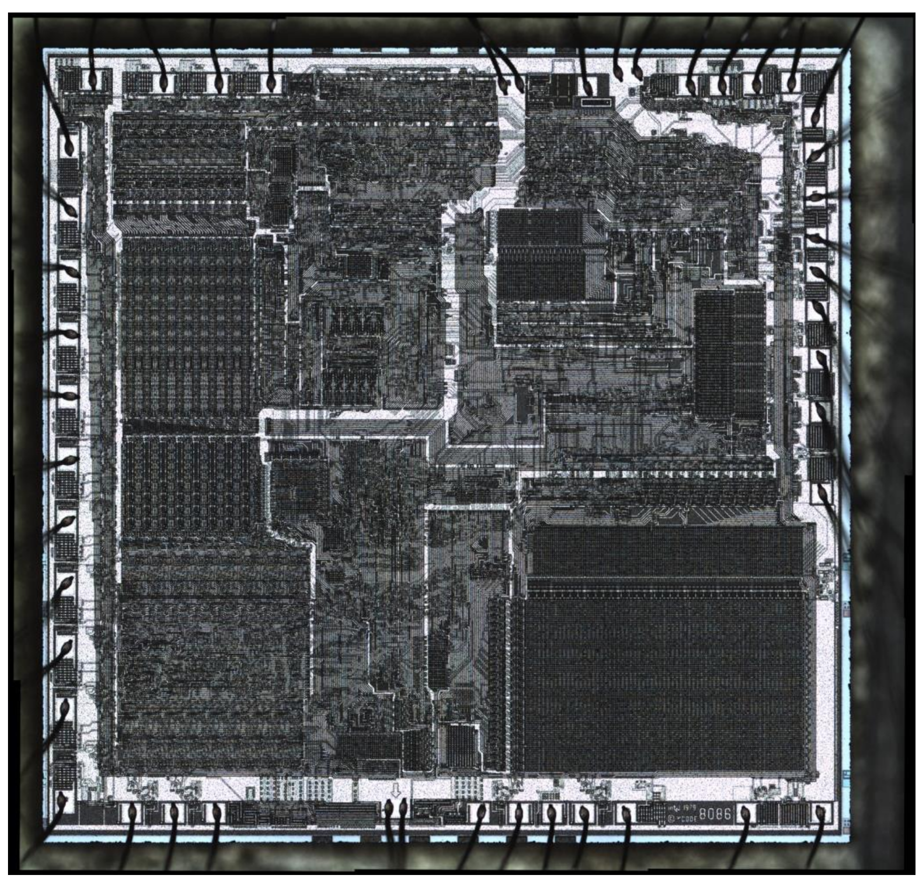
יחידת העיבוד המרכזית של המעבד מתקשרת עם הזיכרון ועם יחידות הקלט / פלט באמצעות קווי תקשורת שנקראים **Bus**, או בעברית **פסים**. יש סוגים שונים של קווי תקשורת, שלכל אחד יש תפקיד ייחודי.



שמות הרכיבים ב־VNA , במושגים המקובלים באנגלית



המושגים המקובלים בעברית



תמונת רנטגן של מעבד ה־8086 (התמונה בהגדלה, הגודל האמיתי הוא 33 מ"מ מרובעים). הקווים הבהירים הם הפסים השונים. המרובע מצד ימין למטה הוא זיכרון פנימי של המעבד. החוטים הכהים המחוברים להיקף הם קווי התקשורת לרכיבי ה-I/O שמחוץ למעבד.

פסי המערכת – SYSTEM BUSES

פסי המערכת (System Buses) מחברים את הרכיבים השונים של מכונות VNA. במשפחת ה-80x86 ישנם שלושה פסים עיקריים: פס הנתונים – DATA BUS, פס מְעָנִים – ADDRESS BUS ופס הבקרה – CONTROL BUS. כל פס הוא למעשה אוסף של קווי חשמל שמעבירים אותות בין הרכיבים השונים. האותות המועברים הם רמות מתח חשמלי שונות – יש מתח שמייצג 0 ויש מתח שמייצג 1. כיוון שרמות המתח שונות בין מעבד למעבד, נתייחס אליהן רק כ־0 או כ־1.

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

נניח שהמעבד שלח לזיכרון "1000h". האם מדובר בכתובת בזיכרון, או במידע שיש לשמור 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 שמחובר 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 שיכול לגשת ל־16, 32 או 64 ביטים בבת אחת, מסוגל לגשת גם לכמות קטנה יותר של מידע – לדוגמה, בית אחד. כלומר 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 לזיכרון. כיוון שכל פניה לזיכרון לוקחת זמן, מעבד שיש לו פס נתונים רחב יותר יוכל לכתוב ולקרוא מהזיכרון בקצב מהיר יותר ולכן יעבוד מהר יותר ממעבד זהה שיש לו פס נתונים עם פחות קווים.

פס מענים – ADDRESS BUS

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

כאשר התוכנה רוצה לפנות למקום כלשהו בזיכרון, או ביחידת I/O כלשהי, היא מכניסה את כתובת הזיכרון המבוקש לתוך פס המענים. רכיבים אלקטרוניים ששולטים על הזיכרון מזהים את הכתובת שבפס המענים ודואגים לשלוח למעבד את המידע בכתובת המבוקשת, או לכתוב לזיכרון בכתובת הנ"ל את המידע ששלח המעבד.

כמות הקווים בפס המענים קובעת את גודל מרחב הכתובות שהמעבד יכול לפנות אליו. לדוגמה, מעבד שיש לו שני קווים בפס המענים יוכל לפנות לארבע כתובות בלבד: 00,01,10,11. מעבד שיש לו n קווים בפס המענים, יוכל לפנות לשתיים בחזקת n כתובות שונות. למעבד ה־8086, לדוגמה, יש 20 קווים בפס המענים. כלומר מרחב הכתובות שלו הוא 1,048,576 (או שתיים בחזקת עשרים). כשתכננו את המעבד הזה, העריכו שמגבייט אחד של זיכרון הוא מעל ומעבר. מעבדים מתקדמים כוללים פס מענים של 32 ביט, כלומר הם מוגבלים ב־4,294,976,296 בתים – ארבעה ג'יגהבייט. בשלב מסויים, גם ארבעה גי'גה של מרחב כתובות זיכרון הפכו להיות מגבלה, וכיום מעבדים ומערכות הפעלה כגון Windows 7 תומכים במרחב כתובות בגודל 64 ביט.

פס בקרה – CONTROL BUS

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

פס הבקרה מכיל שני קווים, קו קריאה (read) וקו כתיבה (write), אשר קובעים את כיוון העברת המידע. כאשר קווי ה־read וה־write מכילים שניהם ערך 1, המעבד והזיכרון לא מתקשרים זה עם זה. אם קו ה־read מכיל אפס, המעבד קורא מהזיכרון. אם קו ה־write מכיל אפס, המעבד כותב לזיכרון.

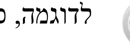
בסעיף הקודם הזכרנו שמעבד בעל פס מידע של 16, 32 או 64 ביטים, יכול לקרוא מידע של בית בודד. קווי בקרה שנקראים byte enable מאפשרים פעולה זו.

במשפחת ה־80x86 יש הפרדה בין מרחב הכתובות שמיועד לזיכרון לבין מרחב הכתובות שמיועד ל-I/O. בעוד גודל מרחב הכתובות של הזיכרון משתנה בין דורות שונים במשפחת ה־80x86, גודל מרחב הכתובות של ה־I/O הוא בעל ערך קבוע של 16 ביט. דבר זה מאפשר פניה ל־65,536 כתובות שונות של התקנים חיצוניים. כאשר מוזנת כתובת כלשהי לפס המענים, פס הבקרה קובע האם היא מיועדת לאזור בזיכרון או לאזור ב־I/O.

הזיכרון

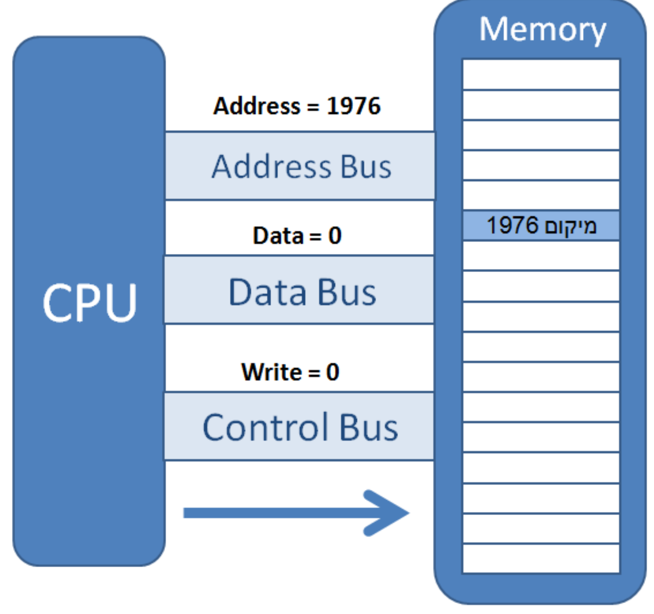
בסעיפים הקודמים הזכרנו, שכמות המקומות בזיכרון היא 2^n, כאשר n היא כמות הביטים בפס המענים. בפרק זה נדון בהרחבה בנושא הגישה לכתובות שונות בזיכרון.

אפשר לחשוב 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 במקום ה־1976 במערך את הערך "0", מתבצעות הפעולות הבאות:

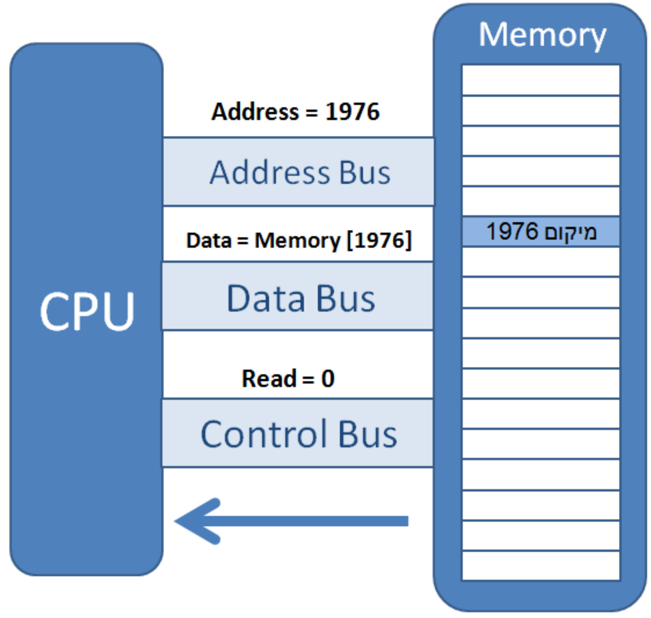
1. המעבד שם את הערך "0" בפס המידע.
2. המעבד שם את הכתובת 1976 בפס המיעון.
3. המעבד משנה את קו ה־write בפס הבקרה וקובע את ערכו ל-0 (הקו "פעיל").



פעולת כתיבה לזיכרון

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

1. המעבד שם את הכתובת 1976 בפס המיעון.
4. המעבד משנה את קו ה־read בפס הבקרה וקובע את ערכו ל-0 (הקו "פעיל").
2. המעבד קורא את הערך שבפס המידע.



פעולת קריאה מהזיכרון

כעת נבחן איך ערכים בגדלים שונים שמורים בזיכרון.

נניח שהגדרנו בזיכרון שלושה ערכים:

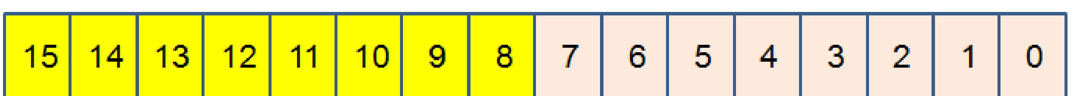
1. במיקום 1970 בזיכרון שמנו ערך בגודל 8 ביטים, או Byte, שערכו הוא 0ABh.
2. במיקום 1974 בזיכרון שמנו ערך בגודל 16 ביט, או word, שערכו הוא 0EEFFh.
3. במיקום 1976 בזיכרון שמנו ערך בגודל 32 ביט, או double word, שערכו הוא 12345678h.

שימו לב לאופן הכתיבה של ערך הקסדצימלי – אם הערך מתחיל באות, נקדים אותו עם הספרה אפס. בצד ימין נשים h כדי לציין שמדובר בערך בבסיס הקסדצימלי. הסיבה לכך היא צורך להפריד בין צירופים כגון עשר בכתיב הקסדצימלי, שניתן לרשום כ־ah, אך כפי שנלמד בהמשך יש ל־ah גם משמעות אחרת.

לאחר שמירת הערכים, הזיכרון שלנו ייראה כך:

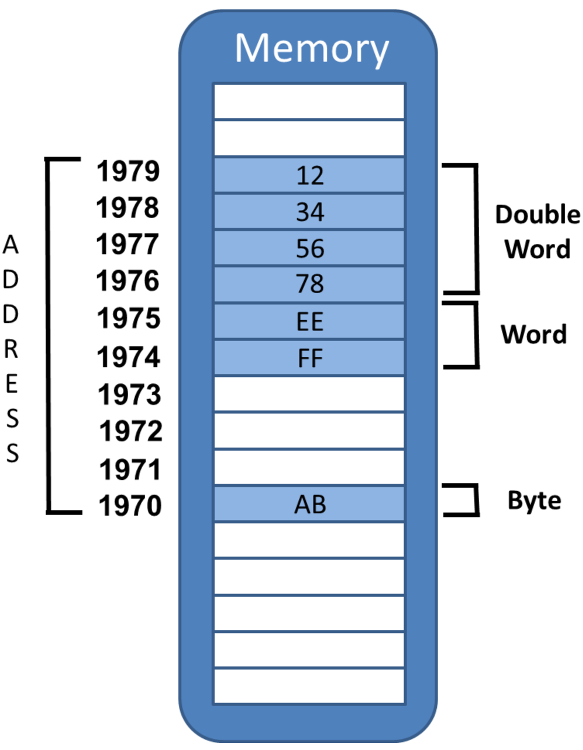
1. במיקום 1970 בזיכרון שמור הערך 0ABh.
2. במיקום 1974 בזיכרון שמור הערך 0FFh, במיקום 1975 בזיכרון שמור הערך 0EEh.
3. במיקום 1976 בזיכרון שמור הערך 78h, במיקום 1977 שמור 56h, במיקום 1978 שמור 34h ובמיקום 1979 שמור 12h.

שימו לב – הבית ה־L.O שמור בכתובת נמוכה יותר בזיכרון, הבית ה־H.O שמור בכתובת גבוהה יותר בזיכרון.



High Order Byte Low Order Byte

תזכורת – L.O. Byte, H.O. Byte



אופן השמירה של גדלים 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 ששמרנו 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 עושה שימוש ביחידות בגודל 16 ביטים, שנקראות **רגיסטרים (registers)**. כל רגיסטר של 16 ביטים מסוגל להחזיק מרחב כתובות מ־0 עד 65,535 (או 0FFFFh). כלומר, צריך למצוא שיטה לגשר על הפער שבין מרחב הזיכרון הקיים לרשות המעבד לבין מרחב הזיכרון, המצומצם יותר, שרגיסטר מסוגל לפנות אליו. השיטה שנבחרה היא לחלק את הזיכרון למקטעים קטנים יותר – **סגמנטים (segments)**. כל כתובת בזיכרון ניתנת לביטוי על-ידי מספר ה-segment שלה, וההיסט (offset) מתחילת הסגמנט. הצורה המקובלת לרישום של כתובת בזיכרון היא:

Segment:offset

במשפחת ה־80x86, פניה לזיכרון מתבצעת על־ידי שילוב של שני רגיסטרים בני 16 ביטים: הרגיסטר הראשון מחזיק את מיקום תחילת הסגמנט בזיכרון. הרגיסטר השני מחזיק את האופסט של הזיכרון מתחילת הסגמנט.

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

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

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

הכתובת בזיכרון תחושב כך:

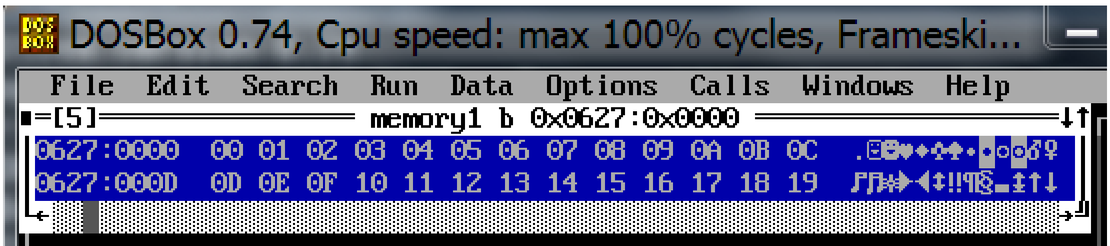
3DD60h + 0012h = 3DD72h

שימו לב לכך שתוספת 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 הבאות לקוחות מתוך תוכנת codeview שרצה בסביבת dosbox. בהמשך הספר, נעבוד בתוכנה אחרת. נשתמש ב-codeview רק בסעיף הזה כיוון שהיא ממחישה היטב את נושא הסגמנט והאופסט.

מצד שמאל, רשום מרחב הכתובות בזיכרון שאנחנו מסתכלים עליו. בדוגמה שלנו, השורה הראשונה מתחילה במיקום בזיכרון שכתובתו היא 0627:0000. בשורה זו יש 13 בתים, ולכן הכתובת האחרונה בשורה הראשונה היא 0627:000C. השורה השניה מתחילה בכתובת של הבית הבא אחריו: 0627:000D.

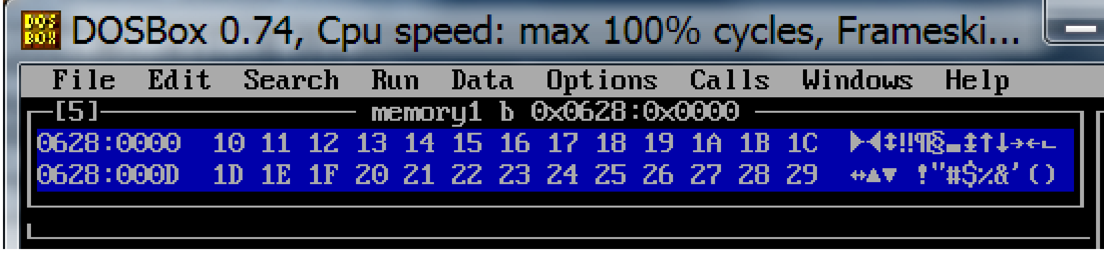
במרכז רשומים הערכים השמורים בזיכרון. בדוגמה זו, יש בזיכרון ערכים עולים – 0,1,2 וכו'. באותה מידה היינו יכולים לשמור בזיכרון אוסף אחר של ערכים בגודל בית אחד. בצד ימין מוצג התרגום של הערכים בזיכרון לתווים טקסטואליים לפי קוד ASCII. אתם מוזמנים לחזור לסעיף שמסביר על קוד ASCII ולהיווכח שהתווים הרשומים הם אכן התרגום של הערכים שבזיכרון.



כעת נדגים את מושג האופסט. נעשה זאת על־ידי הוספת אופסט של בית אחד למרחב הכתובות בזיכרון. כעת השורה הראשונה מתחילה בכתובת 0627:0001.

שימו לב שגם הערכים בזיכרון וגם ערכי ה-ASCII מוסטים כעת בבית בודד.

להמחשת מושג הסגמנט, נאפס את האופסט ובמקומו נקדם את הסגמנט ביחידה אחת. כעת השורה הראשונה מתחילה בכתובת 0628:0000. שימו לב לכך שפעולה זו גורמת להסטה של 16 בתים בזיכרון:



לסיכום, בדוגמה אנחנו רואים בצורה מוחשית שסגמנט תמיד מתחיל 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 לעשות את הפעולה שלו – עיבוד של ביטים. נתחיל 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 ביחידת הבקרה (Control Unit), שאחראית על פענוח פקודות המכונה, ניהול רצף התכנית, הוצאה לפועל של פקודות ואחסון התוצאות המתקבלות.

אוגרים – Registers

רגיסטרים, או במינוח העברי אוגרים, הם סוג מאוד מיוחד של זיכרון. הם אינם חלק מהזיכרון המרכזי, אלא נמצאים ממש בתוך המעבד. כיוון שהם מטופלים ישירות על־ידי המעבד הם גם מהירים ביותר – במקום לפנות לזיכרון ולחכות לקבלת הנתונים, המעבד אינו מחכה כלל. מצב זה נקרא zero wait. יש כמות מצומצמת של רגיסטרים ולחלקם יש תפקידים מיוחדים, שמגבילים את השימוש בהם, אבל באופן כללי רגיסטרים הם מקום מצוין לשמור בו מידע באופן זמני.

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

קיימים דורות שונים של מעבדים במשפחת ה־80x86. כמות הרגיסטרים וגם הגודל שלהם בביטים עשוי להיות שונה בין דור אחד למשנהו. אנחנו נדון בדור הבסיסי הכולל רגיסטרים של 16 ביט.

רגיסטרים כלליים – General Purpose Registers

קיימים שמונה רגיסטרים כלליים, אשר מרוכזים בטבלה הבאה:

הרגיסטר	שם לועזי	שם עברי	תיאור ושימוש עיקרי
ax	Accumulator register	צובר	משמש לרוב הפעולות האריתמטיות והלוגיות. למרות שניתן לבצע פעולות חישוב גם בעזרת רגיסטרים אחרים, השימוש ב־ax הוא בדרך כלל יעיל יותר.
bx	Base address register	בסיס	בעל חשיבות מיוחדת בגישה לזיכרון. בדרך כלל משמש לשמירת כתובות בזיכרון.
cx	Count register	מונה	מונה דברים. בדרך כלל נשתמש בו לספירת כמות הפעמים שהרצנו לולאה, לכמות התווים בקובץ או במחרוזת.
dx	Data register	מידע	משמש לשתי פעולות מיוחדות: ראשית, ישנן פעולות אריתמטיות שדורשות מיקום נוסף לשמירת התוצאה. שנית, כשפונים להתקני I/O, רגיסטר dx שומר את הכתובת אליה צריך לפנות.
si	Source Index	מצביע מקור	ניתן להשתמש בהם בתור מצביעים כדי לפנות לזיכרון (כמו שהראינו שניתן לעשות עם bx). כמו כן הם שימושיים בטיפול במחרוזות.
di	Destination Index	מצביע יעד	
bp	Base Pointer	מצביע בסיס	משמש לגישה לזיכרון בסגמנט שקרוי "מחסנית" Stack.
sp	Stack Pointer	מצביע מחסנית	sp מצביע על כתובת בזיכרון בה נמצא ראש המחסנית. באופן נורמלי, לעולם לא ניגע ב־sp ולא נעשה בו שימוש לטובת ביצוע פעולות אריתמטיות. התפקוד התקין של התכנית שלנו תלוי בכך שערכו של sp תמיד יצביע למיקום הנכון בתוך המחסנית.

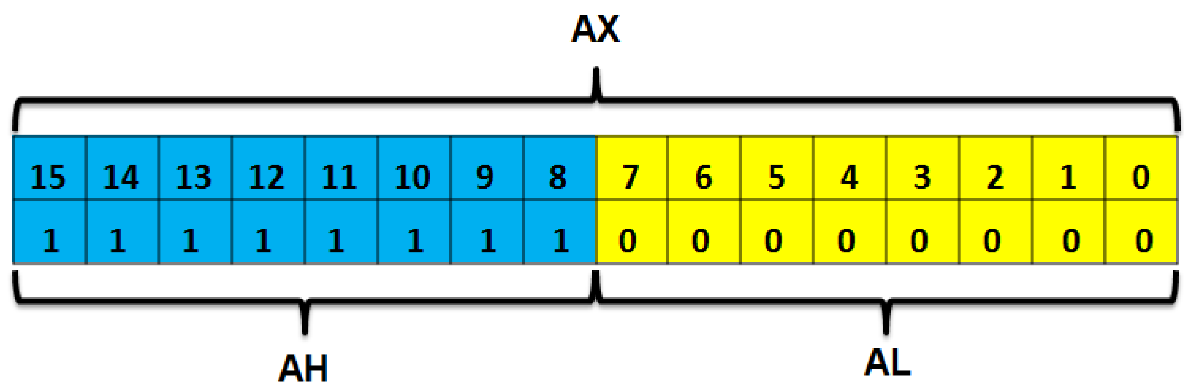
כאמור כל אחד מהרגיסטרים הנ"ל הוא בעל 16 ביטים, כלומר שני בתים. לעיתים אנחנו צריכים לעסוק רק ב־8 ביטים. לדוגמה, כשאנחנו רוצים להעתיק מידע לבית כלשהו זיכרון. אם נעתיק לזיכרון את ax כולו, הזיכרון ישתנה בשני בתים – שינינו את הבית שרצינו לשנות בזיכרון, אבל כתוצאת לוואי לא רצויה שינינו גם את ערכו של הזיכרון 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 ול־al. H מסמן high, כלומר 8 הביטים העליונים של ax, ואילו L מסמן low, כלומר 8 הביטים התחתונים של ax.

נלמד כעת פקודת אסמבלי ראשונה – הפקודה mov (קיצור של move) מבצעת העתקה של הערך שנמצא בצד 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
```

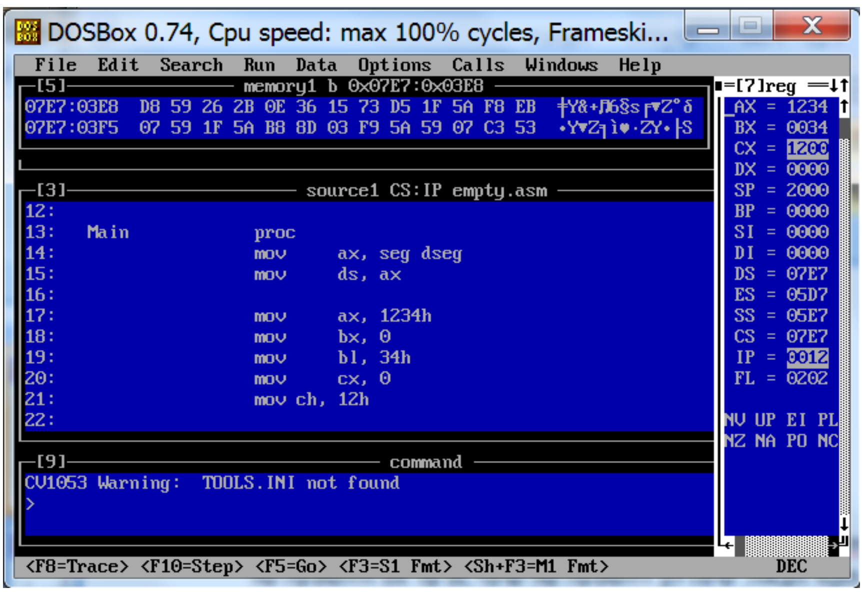
שימו לב שרגיסטרים של 8 ביטים אינם רגיסטרים עצמאיים. שינוי של al ישנה גם את ax, ולהיפך.

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

להלן טבלה המסכמת 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.

שימו לב:

- להשפעה של שינוי bl על bx: שינוי של bl משנה רק את 8 הביטים התחתונים של bx.
- להשפעה של שינוי ch על cx: שינוי של ch משנה רק את 8 הביטים העליונים של cx.

יתר הרגיסטרים הכלליים, sp, bp, si ו־di, הם בעלי 16 ביטים ואין כל דרך לפנות באופן ישיר אל הבית העליון או התחתון שלהם כמו שעשינו עם ax, bx, cx ו־dx.

לתרגול נושא הרגיסטרים הכלליים, להלן טבלה הכוללת פקודות העתקה ורגיסטרים. כל פקודת העתקה היא של ערך או רגיסטר לתוך רגיסטר כלשהו. יש להשלים את ערכי הרגיסטרים לאחר כל פקודת העתקה. הניחו שהמצב ההתחלתי הוא שכל הרגיסטרים מאופסים.

הפעולה		AX		BX		CX		DX	
העתק את	לרגיסטר	AL	AH	BL	BH	CL	CH	DL	DH
הערך 1234h	AX								
הערך 0ABh	AL								
הערך 0ABCDh	BX								
הערך 0EEh	CH								
הערך 0BBh	DL								
הרגיסטר CH	DH								
הרגיסטר DL	AH								
הרגיסטר BX	CX								

הפתרון בעמוד הבא.

פתרון:

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

רגיסטרי סגמנט – Segment Registers

במעבד ה-8086 יש ארבעה רגיסטרים מיוחדים (בדגמים מאוחרים יותר במשפחת ה-80x86 נוסף רגיסטר חמישי- FS):

CS - Code Segment

DS - Data Segment

SS - Stack Segment

ES - Extra Segment

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

הרגיסטר cs שומר את מיקום תחילת סגמנט הקוד. סגמנט זה מכיל את פקודות המכונה שמתבצעות כרגע על־ידי המעבד. כזכור, הגודל המקסימלי של סגמנט הוא 64K. האם זה אומר שהגודל המקסימלי של תכנית הוא לא יותר מאשר 64K? לא, משום שאפשר להגדיר כמה סגמנטים של קוד, ולהעתיק ל- cs ערכים שונים, כך שכל פעם יצביע על הסגמנט הנכון (כאשר מודל הזיכרון מאפשר זאת. בתוכנית base.asm אנחנו מגדירים בתחילת התוכנית model small ולכן ישנו רק סגמנט קוד אחד). באופן מעשי לא סביר שתכתבו תכנית אסמבלי שתדרוש יותר מסגמנט קוד אחד.

לגבי הרגיסטר ds, בדרך כלל בתחילת התוכנית מתבצעות פעולות העתקה כך ש-ds מצביע על המיקום של סגמנט ה־DATA, בו נשמרים המשתנים הגלובליים בתכנית. גם כאן, במודל זיכרון small בו אנחנו משתמשים אנחנו מוגבלים ל־64K של בתים, אבל במודלים אחרים של זיכרון ניתן להגדיר מספר סגמנטים מסוג DATA ולשנות את ds כדי לפנות אליהם.

הרגיסטר ss מצביע על מיקום בזיכרון שמכיל את המחסנית (STACK) של התוכנית. נושא המחסנית יוסבר בפירוט בהמשך. אותה האזהרה שנתנו לגבי שינוי הרגיסטר sp תקפה גם כאן – אין לשנות את ss אם אתם לא יודעים בדיוק מה אתם עושים.

הרגיסטר es, Extra Segment, נותן לנו גמישות נוספת – אם אנחנו רוצים לפנות לסגמנט נוסף, זה בדיוק מה שהוא מאפשר לנו. רגיסטר זה שימושי, לדוגמה, בהעתקות של רצפי נתונים אל הזיכרון.

רגיסטרים ייעודיים – Special Purpose Registers

ישנם שני רגיסטרים ייעודיים:

IP - Instruction Pointer- מצביע ההוראה

FLAGS - דגלים

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

רגיסטר הדגלים 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 ולכן הוא שימושי לפעולות בקרה על התוכנה. כמו כן כולל רגיסטר הדגלים אותות שמשנים 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, הערכים של רגיסטר ה-IP והדגלים ישתנו על־ידי המעבד בזמן ריצת התכנית. נציין, כי קיימות פקודות שמאפשרות למתכנת לשנות את רגיסטר הדגלים, אולם הן מחוץ להיקף של ספר לימוד זה.

אנחנו נקדיש 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) היא המקום במעבד שמתרחשת בו רוב הפעילות. ה־ALU טוען את המידע שהוא צריך מהרגיסטרים, לאחר מכן יחידת הבקרה קובעת ל־ALU איזו פעולה צריך לעשות עם המידע, ולבסוף ה־ALU מבצע את הפעולה ושומר את התוצאה ברגיסטר שנקבע כרגיסטר יעד.

לדוגמה, נניח שאנחנו רוצים להוסיף את הערך 3 לרגיסטר 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 תוצאת החישוב מה-ALU לתוך הרגיסטר ax.

יחידת הבקרה – Control Unit

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

למחשבים הראשונים היה פאנל עם שורות של מעגלים חשמליים, שניתן היה לחווט אותם. באמצעות חיווט החוטים החשמליים אפשר היה לקבוע איזו פקודה המעבד יריץ. בשיטה זו, כמות הפקודות שניתן היה להריץ בתוכנית אחת הייתה שווה למספר השורות בפאנל. אחת ההתפתחויות המשמעותיות במדעי המחשב הייתה המצאת הרעיון לשמור את התוכנה בזיכרון המחשב ולהביא אותה ליחידת העיבוד פקודה אחרי פקודה. בשיטה זו כל פקודה מתורגמת לרצף של אחדות ואפסים ונשמרת בזיכרון.

יחידת הבקרה, Control Unit, מביאה מהזיכרון את פקודות המכונה, הידועות גם בשם **Operational Codes** או **OpCodes**, ומעתיקה אותן אחת אחת אל רגיסטר פענוח.

כדי לייעל את הבאת ה־OpCodes למעבד, גודל של OpCode הוא בדרך כלל כפולה של 8 ביטים. יחידת הבקרה מכילה רגיסטר הוראות, Instruction register, או IP, אותו הזכרנו בסעיפים הקודמים. ה־IP מחזיק את הכתובת בזיכרון של הפקודה הבאה שיש להעתיק. לאחר שיחידת הבקרה מעתיקה את ה-OpCode לתוך רגיסטר הפענוח, היא מקדמת את ה־IP אל כתובת הפקודה הבאה וכך הלאה.

שעון המערכת (הרחבה)

מכונות פון נוימן מעבדות פקודה אחרי פקודה, באופן טורי.

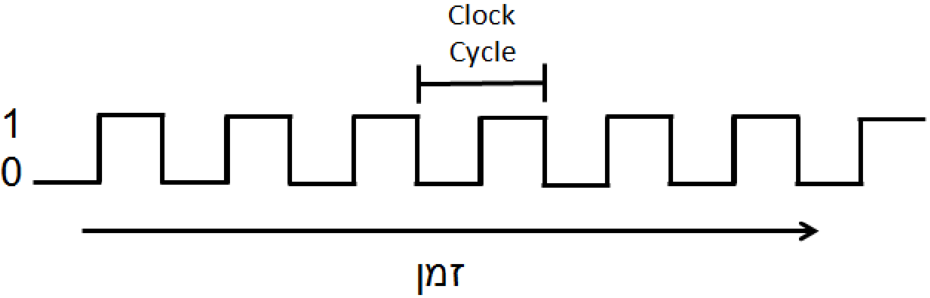
נתבונן בפקודות הבאות, הטוענות לתוך ax את bx+5:

```
mov    ax, bx
add    ax, 5
```

ברור שפקודת החיבור צריכה להתרחש אחרי פקודת הטעינה של bx לתוך ax, אחרת נקבל תוצאה לא נכונה. אם שתי הפקודות תתבצענה בו זמנית, ולא טורית, לאחר הרצת הפקודות ax יכיל את bx או יכיל 5, אבל לא bx+5.

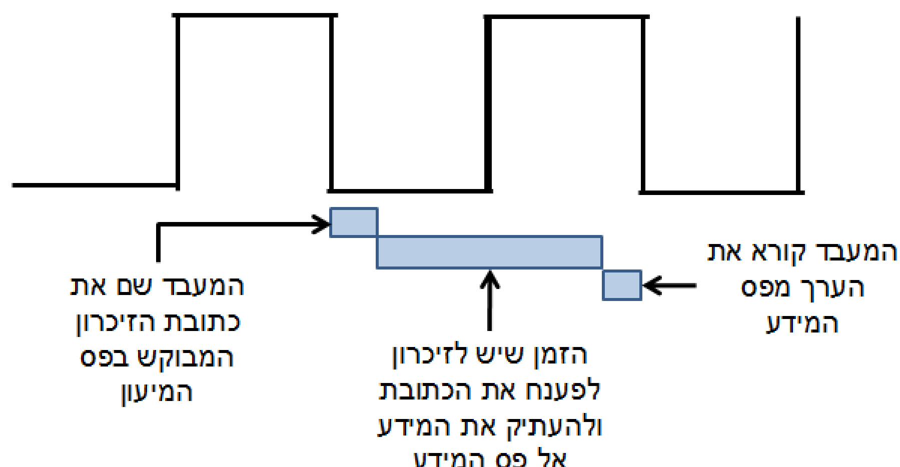
כל פעולה דורשת זמן לביצוע. זמן זה כולל פענוח ה־OpCode, פניה לזיכרון (בדרך כלל), טעינת ערך כלשהו לרגיסטר, ביצוע פעולה אריתמטית ועוד. כדי שפעולות יתבצעו לפי סדר, המעבד צריך מנגנון שיתזמן את הפעולות – שעון המערכת.

שעון המערכת הוא אות חשמלי על פס הבקרה Control Bus, אשר משנה את ערכו כל פרק זמן קצוב בין 0 ל־1. התדירות שבה האות החשמלי משתנה בין 0 ל־1 היא תדירות שעון המערכת. הזמן שלוקח לאות להשתנות מ־0 ל־1 וחזרה ל־0 נקרא **clock cycle**. הזמן של clock cycle הוא אחד חלקי **תדירות השעון**. לדוגמה, מעבד בעל תדר שעון של 1MHz הוא בעל clock cycle של 1 מיקרו שניה (1/1,000,000 של שניה).

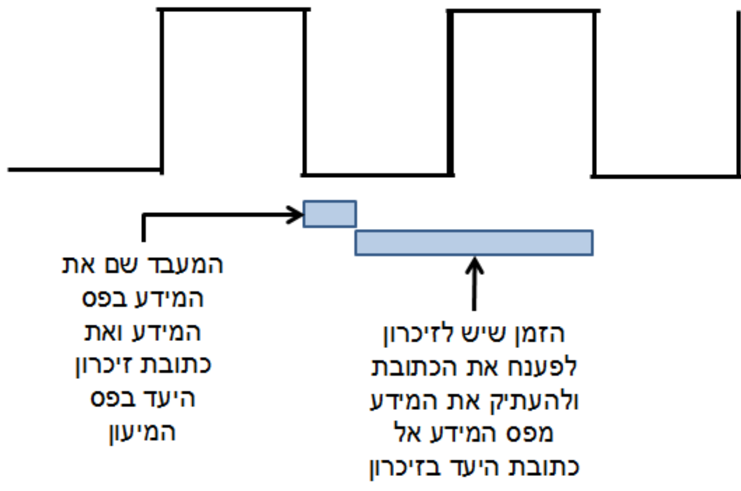


כדי להבטיח סנכרון, המעבד יתחיל לבצע פעולה או בנקודת הזמן בה מתרחשת עליית אות השעון (הזמן שהאות משתנה מ־0 ל־1) או עם ירידת אות השעון (הזמן בו האות משתנה מ־1 ל־0). כיוון שכל המשימות של המעבד מסונכרנות על ידי שעון המערכת, משימה לא יכולה להימשך פחות מ־clock cycle יחיד, כלומר תדירות שעון המעבד מגבילה למעשה את כמות הפעולות המקסימלית שהמעבד יכול לבצע בפרק זמן נתון. עם זאת, אין משמעות הדבר שהמעבד תמיד יבצע את כמות הפעולות המקסימלית, משום שפעולות רבות אורכות יותר מ־clock cycle יחיד.

אחת הפעולות הנפוצות שהמעבד מבצע היא גישה לזיכרון. גישה לזיכרון חייבת להתרחש באופן מסונכרן עם שעון המערכת. להלן סכימה של הפעולות המתבצעות על־ידי המעבד בזמן גישה לזיכרון. חשוב לציין שבדוגמה זו הגישה אורכת clock cycle יחיד, כמו במעבדים מתקדמים, אך תיתכן גישה לזיכרון שאורכת מספר clock cycles:



סכימה של פעולת קריאה מהזיכרון על ציר הזמן של שעון המערכת



סכימה של פעולת כתיבה לזיכרון על ציר הזמן של שעון המערכת

סיכום

בפרק זה למדנו אודות ארגון המחשב, בדגש על מעבדי משפחת ה־80x86: התחלנו מפירוט הסיבות שבגללן רלבנטי ללמוד על משפחת מעבדי ה־80x86 למרות שקיימים מעבדים חדשים יותר, בין היתר מכיוון שכל המעבדים חולקים ארכיטקטורה משותפת וכולם מכונות VNA. הסברנו מהי ארכיטקטורת VNA ונכנסנו להסבר מפורט אודות מרכיביה השונים. ראינו איך פסי המידע, הנתונים והבקרה משמשים את המעבד לתקשורת עם רכיבים שונים, תוך דגש על קריאה וכתיבה לזיכרון.

בהמשך, סקרנו את מבנה הזיכרון של משפחת ה־80x86 והכרנו את החלוקה לסגמנטים ואופסטים, שמשמשת להגעה לכל כתובת בזיכרון. לאחר מכן, עברנו לדון ביחידת העיבוד המרכזית על חלקיה השונים. התחלנו מהרגיסטרים השונים ועבור מה משמש בדרך כלל כל רגיסטר. משם עברנו ליחידה האריתמטית, האחראית על ביצוע הפעולות, וליחידת הבקרה, שאחראית להביא את פקודות המכונה לפענוח. סיימנו בהסבר על שעון המערכת – הבנו שכדי לבצע פקודות באופן טורי יש צורך בשעון המערכת, והראינו את תהליך הפניה לזיכרון על ציר הזמן.

פרק זה היה תיאורטי בעיקרו, והקנה את הכלים הבסיסיים להבנת הפרקים הבאים. חשובה בעיקר ההבנה של נושא הכתובות בזיכרון ומהם הרגיסטרים השונים, אז אם אינכם בטוחים שאתם שולטים בנושאים אלו – חיזרו וודאו שאתם מבינים את התיאוריה. בפרק הבא נתקין את סביבת העבודה שתאפשר לנו להתחיל לתכנת.

פרק 4 – סביבת עבודה לתכנות באסמבלי

מבוא

בפרק זה נלמד על כלי העבודה בהם נשתמש בשביל לתכנת בשפת אסמבלי. ראשית, נציין כי יש מגוון של כלים לעבודה באסמבלי, ופרק זה אינו מתיימר להציג את כולם. קיים אוסף של אסמבלרים (תוכנות הממירות אסמבלי לשפת מכונה) – ההבדלים ביניהם קטנים, אבל קוד אסמבלי 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 האסמבלי, בחרנו להציג אפשרות אחת בלבד לכל תוכנה. היעד אליו אנו שואפים להגיע בסיום פרק זה הוא ליצור קובץ ראשון בשפת אסמבלי, קובץ שנקרא לו base.asm, להפוך אותו לקובץ בשפת מכונה ולהיות מסוגלים להריץ את התוצאה באמצעות כלי שנקרא **דיבאגר (Debugger)**. בסיום הפרק יש תרגיל מחקר קטן, לתלמידים המעוניינים להרחיב את הידע שלהם. בפרק הקודם למדנו באופן תיאורטי שהאסמבלר מתרגם פקודות בשפת אסמבלי לשפת מכונה – Opcodes. תרגיל המחקר משלב את הכלים שנלמדים בפרק זה על מנת להבין יותר לעומק את הדרך בה מתורגמות פקודות האסמבלי ל־Opcodes.

Editor – Notepad++

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

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

התקנה: מחפשים Notepad++ בגוגל, ממשיכים לאתר

http://notepad-plus-plus.org/download/v6.5.5.html ובוחרים באפשרות Notepad++ installer.

טיפים:

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

קובץ Base.asm

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

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

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

הסבר על base.asm

ניתן כעת שני הסברים לקובץ base.asm. הסבר אחד בסיסי, שכולל את מה שצריך לדעת כדי להתחיל לתכנת בשפת אסמבלי, והסבר שני מתקדם. בשלב זה עדיין אין לנו את הכלים להבין את ההסבר המתקדם, אבל אין סיבה להיות מתוסכלים – עיברו הלאה וחיזרו אליו בסיום הפרק על שפת אסמבלי – הידע על מבנה השפה והזיכרון יאפשר לכם להבין את ההסבר המתקדם.

הסבר בסיסי:

נתמקד רק באזורים שאותם נרצה לערוך בקובץ: האזור הראשון נמצא תחת הכותרת DATASEG. את המושג סגמנט אנו מכירים מהפרק על ארגון המחשב. הסגמנט שקוראים לו DATASEG הוא הסגמנט שבו מגדירים שמות למשתנים בזיכרון המחשב. נזכיר כי משתנים הם שמות שאנחנו נותנים לכתובות בזיכרון של המחשב. בתוך DATASEG לא נהוג לשים קוד.

אנו יכולים לכתוב ב־DATASEG, לדוגמה:

```
var1    db      5
```

ובכך הגדרנו משתנה בשם var1 שמקבל את הערך 5. האסמבלר (נגיע אליו בהמשך) כבר ידאג להקצות אזור בזיכרון, לקרוא לאזור הזה var1 ולשים בזיכרון את הערך 5.

האזור השני שמעניין אותנו הוא מה שנמצא תחת הכותרת CODESEG. זהו הסגמנט שבו כותבים את הפקודות שאנחנו רוצים שהמעבד יריץ. בתחילת הסגמנט יש כמה פקודות עזר, ואז – מיד אחרי ההערה – יש אזור שבו אפשר להקליד את הפקודות שלנו.

אנו יכולים לכתוב ב־CODESEG, לדוגמה:

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

ופקודה זו תגרום למעבד להעתיק את הערך שהכנסנו ב־var1, 5 , אל תוך הרגיסטר al.

זה כל מה שצריך לדעת בשביל להתחיל לערוך שינויים בקובץ base.asm.

הסבר מפורט:

נעבור שורה שורה על הקובץ:

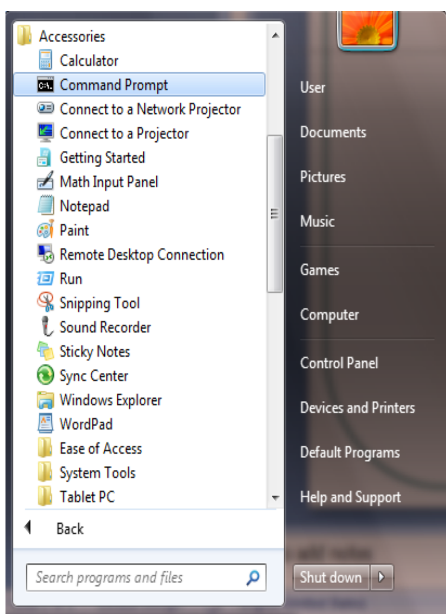
- IDEAL – לתוכנת Turbo Assembler שאנחנו עובדים איתה, יש כמה צורות כתיבה. IDEAL היא צורת כתיבה פשוטה שמתאימה למתכנתים מתחילים.
- MODEL small – מודל זיכרון small, קובע לאסמבלר שהתכנית מכילה שלושה סגמנטים – Data, Code, Stack ושגודל סגמנטי הקוד והנתונים הוא 64K כל אחד.
- STACK 100h – גודל המחסנית. הסבר מפורט תמצאו בחלק על המחסנית.
- DATASEG – סגמנט הנתונים.
- CODESEG – סגמנט הקוד.
- start – תווית שמסמנת למעבד מאיפה להתחיל את ריצת התוכנית. כל שם אחר יכול לבוא במקומה- לדוגמה "main"- העיקר להיות עקביים עם הוראת ה־end שבשורה האחרונה.
- mov ax, @data – מטרת הוראה זו וההוראה הבאה, לקבוע ש־ds יצביע על מקטע הנתונים. ההוראה @data מחזירה את הכתובת של סגמנט ה־data. בסיום ההוראה הבאה mov ds, ax מועתק לתוך ds כתובת סגמנט ה-data.

- exit – יציאה מסודרת מהתכנית. שתי השורות שמתחת למילה exit הן קוד שגורם ליציאה מסודרת של התכנית ושחרור הזיכרון שהיא תפסה, כך שמערכת ההפעלה תוכל להשתמש בו לצרכים אחרים. הסבר מפורט תמצאו בחלק על פסיקות DOS.
- end start – המילה end אומרת לאסמבלר להפסיק את הקומפילציה, כלומר פקודות שיופיעו אחרי מילה זו לא יתורגמו לשפת מכונה. לאחר end מופיעה תווית שאומרת לאסמבלר מאיפה המעבד צריך להתחיל את הריצה על התוכנית. אם אין תווית, הריצה תתחיל מתחילת קובץ ה־exe. במקרה שלנו, אנחנו רוצים שהריצה תתחיל מהתווית start.

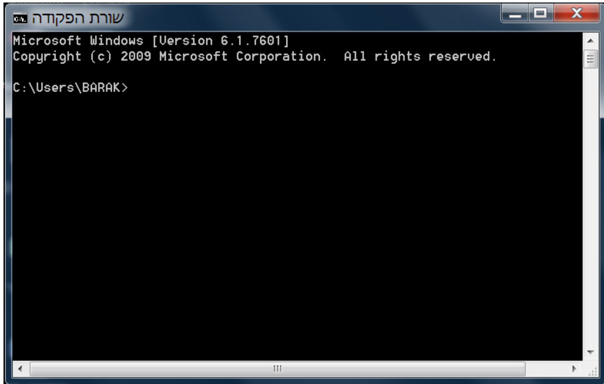
Command Line

בעבר, טרם מערכות ההפעלה הגרפיות, ניהול הקבצים והספריות התבצע על־ידי Command Line. ב־ Command Line אין אפשרות להשתמש בעכבר, לכן כדי להריץ קובץ, לדוגמה, במקום לעשות עליו דאבל קליק עם העכבר צריך לכתוב את שם הקובץ.

אנחנו נלמד פקודות בסיסיות של Command Line משתי סיבות. הראשונה היא שהממשק לאסמבלר שלנו הוא טקסטואלי. כלומר, כדי להורות לאסמבלר איזה קובץ asm לתרגם לשפת מכונה, הדרך היחידה היא באמצעות הקשת הפקודות ב־Command Line. הסיבה השניה, היא שמערכות הפעלה מודרניות רבות כבר לא מסוגלות להריץ את התוכנות הנדרשות לעבודה בסביבת אסמבלי. מערכות אלו עובדות ב־64 ביט (כלומר, מייצגות כתובת בזיכרון על־ידי 64 ביטים) ואין להן תאימות לאחור לייצוג על-ידי 20 ביט שאנחנו נזדקק לו כדי לדמות עבודה עם מעבד 80x86. הפיתרון המקובל, כפי שגם אנחנו נעשה, הוא לעבוד עם תוכנה שנקראת אמולטור – מדמה מערכת הפעלה ומעבד ישנים יותר. כדי לעבוד עם אמולטור, נזדקק לתוכנות שעובדות בממשק טקסטואלי בדומה ל־Command Line. ההגעה ל- Command Line שונה במערכות הפעלה שונות. בחלונות 7, תמצאו אותה בתוך רשימת התוכניות ----> עזרים ----> Command Prompt. בדרך כלל ניתן יהיה להגיע אליה על ידי כתיבת cmd בחלון חיפוש התוכניות ב-Start menu.



לאחר מכן ייפתח חלון כדוגמת החלון הבא:



קיימים אתרים שונים שמסבירים את הפקודות השונות שניתן להריץ ב-Command Line. אתר מומלץ הוא http://www.computerhope.com/msdos.htm ובו רשימת כל הפקודות.

אנחנו נשתמש בעיקר בפקודות הבאות:

❖ **CD – קיצור של Change Directory**

כדי לעבור לספריה כלשהי יש לרשום:

CD DirectoryName

כאשר שם הספריה בא במקום "DirectoryName". לדוגמה:

CD Games

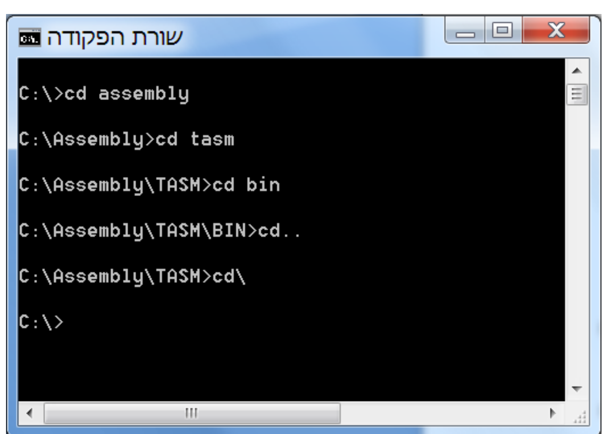
כדי לחזור ספריה אחת אחורה בעץ הספריות יש לרשום:

CD ..

וכדי לחזור לתחילת העץ יש לרשום:

CD \

לדוגמה:



❖ DIR

פירוט כל הקבצים ותתי הספריות הקיימים בספריה כלשהי. לעיתים יש קבצים וספריות רבים מדי בשביל הצגה על מסך אחד, ולכן רצוי לכתוב dir /p. הסימון p הוא קיצור ל־page והתוכן יוצג על המסך עמוד אחרי עמוד:

```
שורת הפקודה - dir  /p

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>          TASM
09/30/2013  10:43 PM         1,181,760 WinAsm.rar
Press any key to continue . . .
```

❖ EXIT

כדי לצאת מה־Command Line ולחזור למערכת ההפעלה, מקישים exit ואחר כך enter.

❖ DOSBOX

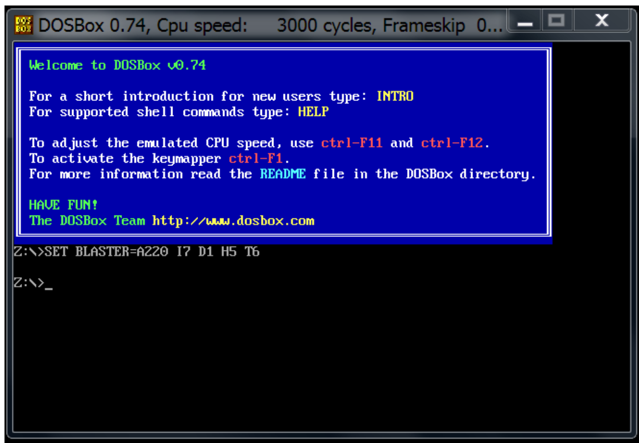
כיוון שלמערכות הפעלה מודרניות אין תאימות לאחור עם מעבדי ה־80x86 ומרחב הכתובות שלהם. נדרשת תוכנה שנקראת בשם כללי **אמולטור (Emulator)**. אמולטור היא תוכנה שמדמה מחשב או מערכת הפעלה כלשהי. לדוגמה, אנחנו יכולים להוריד מהאינטרנט תוכנת אמולטור שתגרום למחשב שלנו, ואפילו לסמארטפון שלנו, להתנהג כמו מחשב מיושן.

אמולטור של מחשב Commodore 64, מחשב קלאסי שיצא בשנת 1982, מאפשר לאייפון להריץ משחקים שנכתבו ל־Commodore 64

כדי להריץ את סביבת העבודה של אסמבלי, נצטרך מערכת הפעלה מוקדמת של מיקרוסופט. מערכת הפעלה זו קרויה DOS, קיצור של Disk Operating System, והגרסה האחרונה שלה יצאה בשנת 1994. האמולטור של DOS נקרא Dosbox ונכון לזמן כתיבת שורות אלו הגרסה העדכנית היא 0.74. חפשו בגוגל "Dosbox download" והתקינו את התוכנה.

ניתן להוריד את גרסה 0.74 גם מהלינק: www.cyber.org.il/assembly/dosbox

הקלקה על דוסבוקס תפתח מסך כדוגמת המסך הבא:



הפעולה הראשונה שאנחנו 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, נכתוב:

Mount c: c:\

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

C:

ועברנו לכונן הקבצים C.

כל הפקודות של ה-Command Line תקפות גם כאן.

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

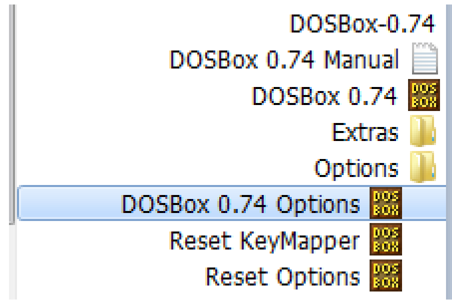
כיוון שאנחנו מבצעים אמולציה של מעבד ישן, מהירות הריצה של הדוסבוקס הופחתה בהתאם ובררית המחדל היא ריצה בקצב של 3000 cycles לשניה בלבד (שימו לב לכתוב בכותרת למעלה). ניתן לשנות את הקצב הזה באמצעות כתיבת הפקודה הבאה במסך ה-DosBox:

Cycles = max

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

טיפ לעבודה קלה עם Dosbox: אפשר לקבוע לדוסבוקס אוסף של פקודות שירוצו באופן אוטומטי עם עליית התוכנה. קביעת פקודות אלו יכולה לחסוך לנו הקלדה רבה בעתיד- לדוגמה, נוכל לקנפג את דוסבוקס כך שבכל פעם שהוא עולה, הוא יגיע מיד לספריה הנכונה.

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



לאחר מכן הקלקה על לחצן שמאלי של העכבר תפתח לנו את הקובץ. נגלול 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

אסמבלר Assembler היא כל תוכנה שמסוגלת להפוך קוד בשפה כלשהי לשפת מכונה. קיימים מגוון אסמבלרים לשפת אסמבלי, אנחנו בחרנו לעבוד עם TASM, קיצור של Turbo Assembler. הגרסה האחרונה של TASM היא 5.0 והיא יצאה בשנת 1996.

לינקר Linker היא תוכנה אשר מבצעת המרה משפת מכונה לקובץ הרצה. הלינקר יכולה לקבל קובץ אחד או מספר קבצים בשפת מכונה, ולהמיר אותם לקובץ הרצה יחיד. הלינקר שימושי במקרים שבהם התוכנה מחולקת בין מספר קבצים. לדוגמה, קובץ אחד כולל את התוכנית הראשית, שמבצעת קריאה לקטעי קוד שנמצאים בקבצים אחרים. הלינקר יודע לקשר בין הקריאה לקטעי קוד לבין קטעי הקוד שנמצאים בקבצים האחרים.

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

ניתן להוריד את הקובץ tasm.rar בלינק: http://cyber.org.il/assembly/TASM.rar

לאחר ההורדה:

- צרו ספריה בשם c:\tasm\bin.

- פיתחו את קובץ ה־rar והעתיקו לספריה את הקבצים (אם אין לכם תוכנה לפתיחת rar, חפשו "rar download" והורידו תוכנה חינמית).

- ודאו שהקובץ base.asm נמצא בספריה bin.

- כעת היכנסו ל־Dosbox והגיעו אל ספריית ה־bin (על ידי פקודות cd, דוגמה בצילום המסך שלמטה).

- הפקודה הבאה ממירה את base.asm לשפת מכונה:

```
tasm /zi base.asm
```

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

- הפקודה הבאה – שמבוצעת על ידי תוכנת הלינקר tilnk - ממירה את הקובץ בשפת המכונה 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, כיתבו base בשורת הפקודה והקישו enter, או כיתבו td base כדי להריץ אותו מהדיבאגר.

```
DOSBox 0.74, Cpu ...
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
```

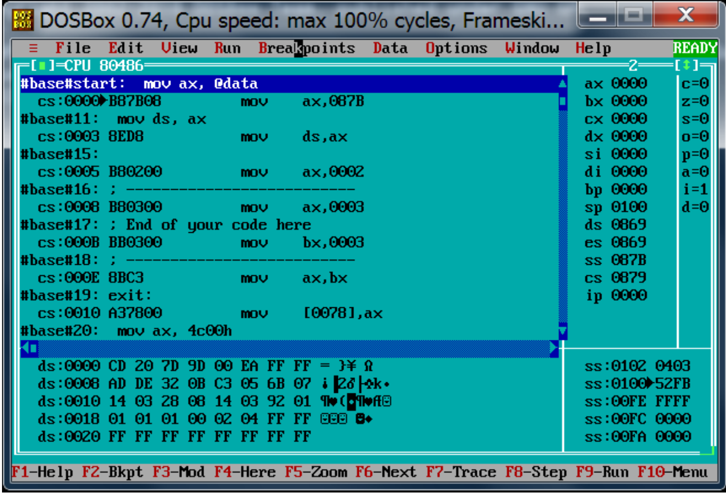
כעת, נריץ את הקובץ base.exe באמצעות תוכנת דיבוג בשם Turbo Debugger.

Turbo Debugger – TD

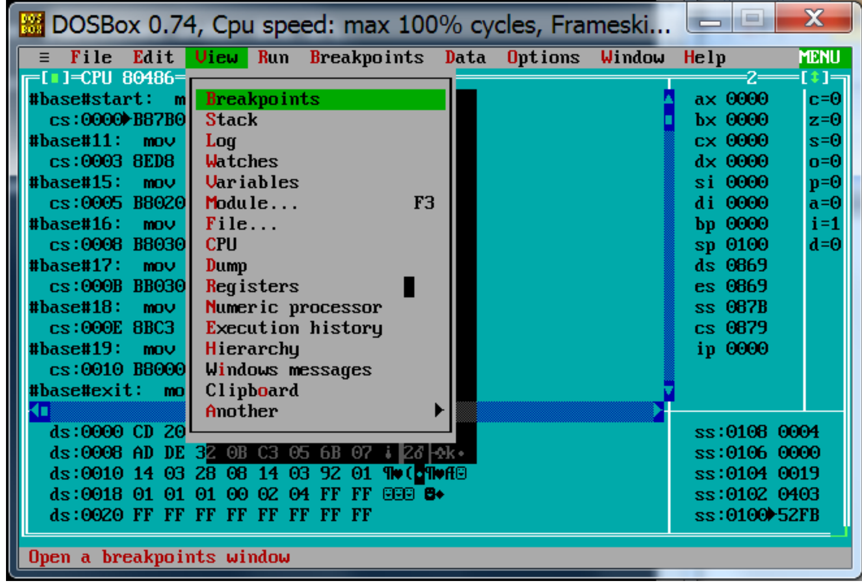
כדי להפעיל את תוכנת הדיבוג שלנו, נכתוב בדוסבוקס:

td base

יופיע המסך הבא:



אם במקום המסך הזה מופיעים דברים אחרים, היכנסו לתפריטים למעלה (לחיצה על מקש F10), בחרו view ובתפריט שנפתח, בחרו את האפשרות CPU:



השימוש ב־Turbo Debugger

הסברים על כל הפונקציות של Turbo Debugger, או בקיצור TD, ניתן למצוא בתפריט ה־Help. אנחנו נתייחס לפונקציות העיקריות של TD:

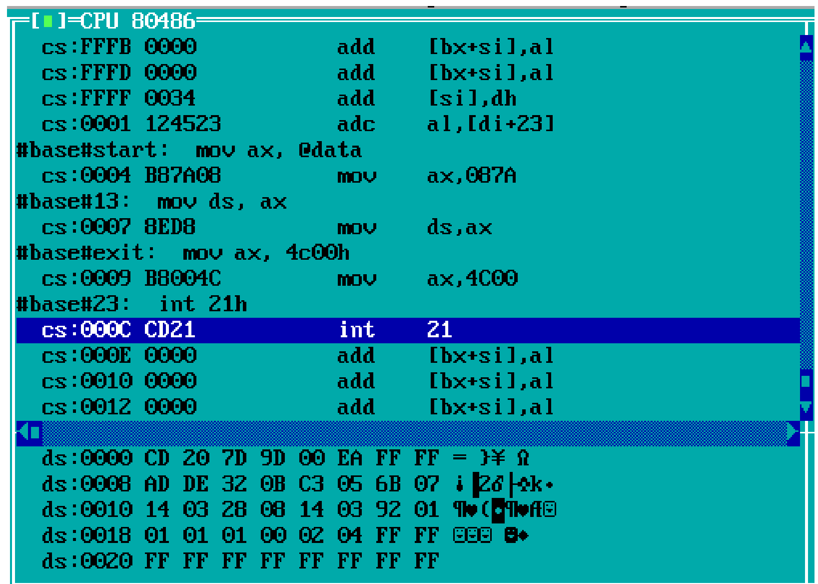
הרצת הקוד שורה אחרי שורה – ניתן לעשות באחד משני מצבים. מצב Step, על־ידי לחיצה על F8, עובר לשורת הקוד הבאה. אם שורת הקוד הנוכחית היא פרוצדורה, בהרצה של שורה יחידה נעבור את כל הפרוצדורה ונצא ממנה. כדי לדבג פרוצדורה, נשתמש במצב Trace, על־ידי לחיצה על F7. במצב זה, כשאנחנו מגיעים לתחילתה של פרוצדורה, הדיבאגר יתקדם כל פעם שורה יחידה בתוך הפרוצדורה.

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

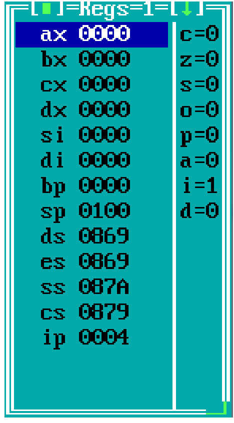
אפשרויות הרצה שימושיות נוספות הן F4, שמשמעותו Go to cursor – התוכנה תרוץ עד למקום שאנחנו עומדים עליו. הפקודה F9 מריצה את התכנית עד עצירה.

אפשרויות View – באמצעות האפשרויות השונות אפשר לחקור כל אלמנט בתכנית. ה־Views השימושיים ביותר הם:

- ה־CPU, שמציג לנו את שורות הקוד ותרגומן לשפת מכונה, וכמו כן את סגמנט ה ־DATA:

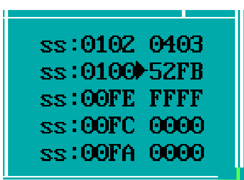


- הרגיסטרים והדגלים:



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

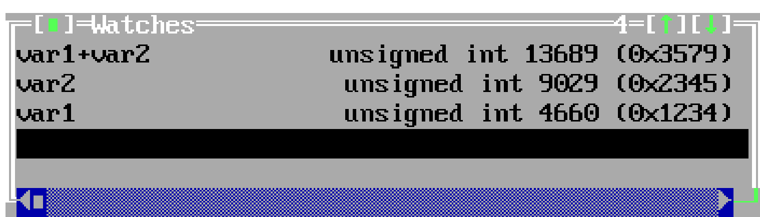
- ה־STACK, שמציג לנו את סגמנט ה־STACK – כרגע אנחנו לא יודעים עליו הרבה, אבל נשתמש בו בהמשך:



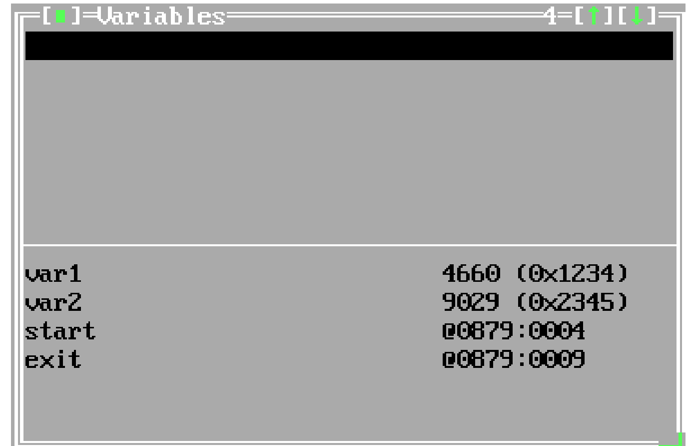
- Watches – אנחנו יכולים לקחת משתנה או ביטוי כלשהו ולהכניס אותו בתוך watch. ערך הביטוי יעודכן באופן דינמי. לדוגמה, יצרנו שני משתנים ב־DATASEG:

```
Var1        dw      1234h
Var2        dw      2345h
```

הכנסנו לתוך Watch את הביטוי var1+var2 וקיבלנו:

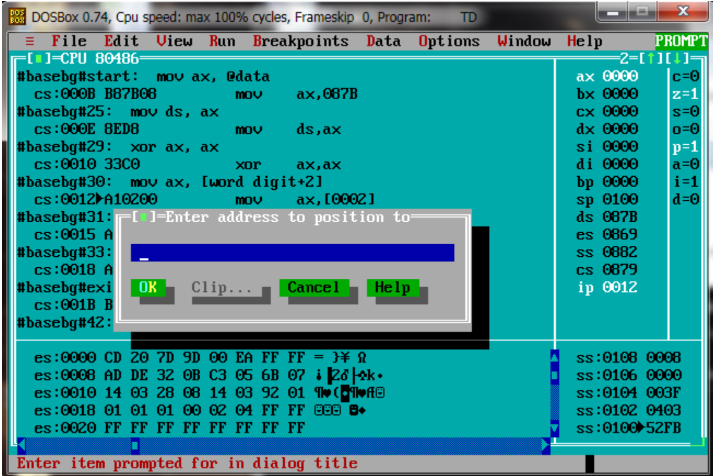


- Variables – מציג לנו את רשימת כל המשתנים והתוויות בתוכנית. לדוגמה:

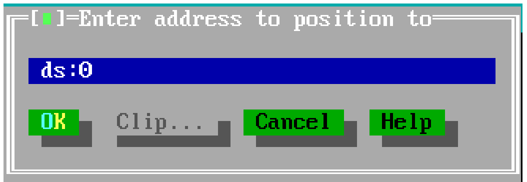


מעבר למקום מבוקש בזיכרון

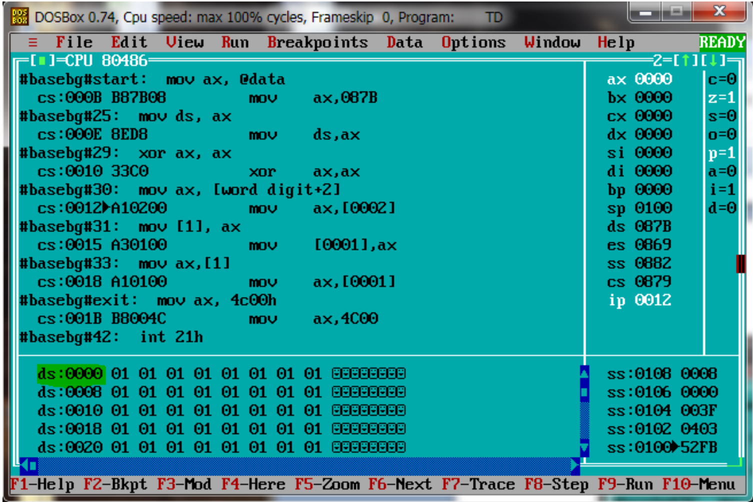
עמדו על החלק התחתון של המסך, היכן שכתובים הערכים שנמצאים בזיכרון. לחיצה על CTRL+G תפתח את המסך הבא:



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



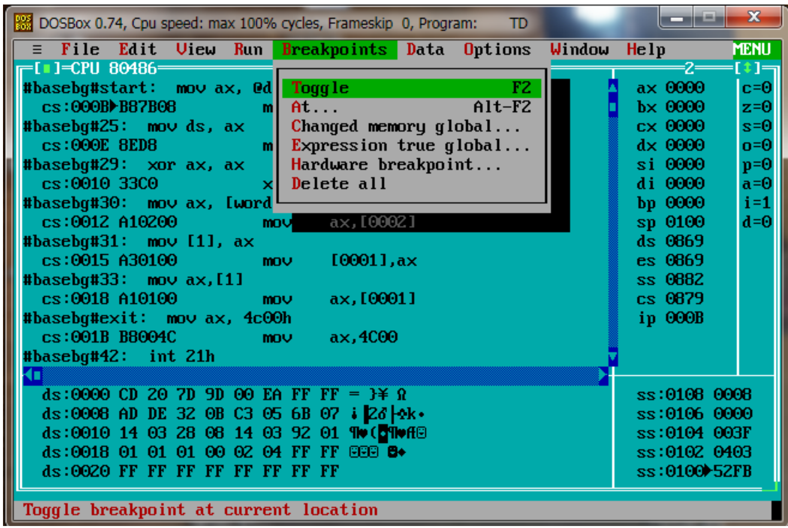
ופעולה זו "תקפיץ" אותנו לאזור המבוקש:



שימו לב לכך שבתחילת התוכנית ds עדיין אינו מוגדר להצביע על המקום של DATASEG, רק אחרי שתי שורות הקוד הראשונות ds מקבל את הערך הנכון. לכן, עלינו ללחוץ פעמיים על F8 לפני שנבצע את הפעולות האחרונות.

שימוש ב-breakpoints

אנחנו יכולים להגדיר נקודת עצירה, או breakpoint בכל שורה שנרצה בקוד. בכדי לעשות זאת, עלינו לסמן את השורה בה אנחנו שהתכנית תעצור ובתפריט Breakpoints לבחור ב-Toggle. השורה הופכת לצבועה באדום. חזרה על הפעולה תבטל את נקודת העצירה.



לאחר שקבענו breakpoint, אם נריץ את התוכנית בעזרת פקודת Run (F9) היא תעצור בשורת הקוד שיש עליה breakpoint. זוהי דרך יעילה להגיע מהר לנקודות בקוד שאנחנו חושדים שיש בהן באגים.

הריצו את base.exe בעזרת הדיבאגר. קיבעו breakpoint בשורה mov ax, 4C00 והגיעו אל שורת הקוד בזמן ריצה תוך שימוש ב-F9. בידקו את הערכים של הרגיסטרים השונים והשוו את הערכים שלהם למה שציפיתם שהם יהיו.

טיפים לעבודה עם TD

- יציאה מהתוכנה אל מסך ה-Dosbox: לחיצה על ALT+X
- הרצה מחדש של התוכנית: CTRL+F2
- דפדוף בתפריטים בעזרת המקלדת (למקרה שהעכבר נתקע): לחיצה על F10 ושימוש במקשי החיצים. לחיצה על Tab למעבר בין החלונות השונים של הדיבאגר.
- מעבר למצב מסך מלא (ויציאה ממנו): לחיצה על ALT+Enter
- יציאה מחלון הדיבאגר לתוכנות אחרות, ללא שימוש בעכבר: מקש "חלונות" (המקש שבין Alt ל- Ctrl)

תרגיל מחקר (הרחבה) – קידוד פקודות אסמבלי ל-Opcodes

נחקור כיצד פקודת mov מיתרגמת לפקודות מכונה.

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

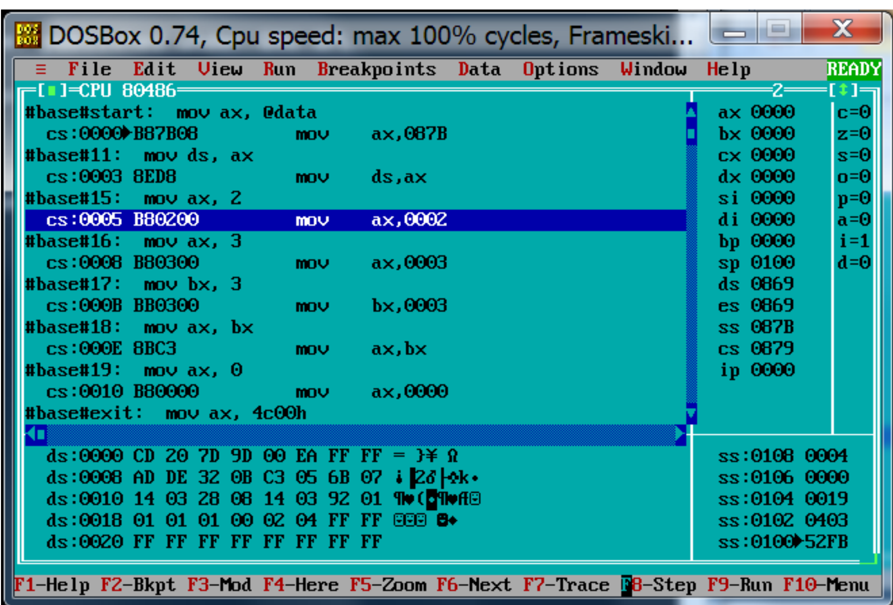
הדרכה וכלים לביצוע המשימה:

1. הקובץ base.asm מכיל שלד של תכנית. הכניסו את הקוד שאתם רוצים להריץ לאחר ההערה "Your code here". שמופיעה לאחר שורת הקוד

mov ds, ax

2. לאחר ההמרה לקוד מכונה ויצירת קובץ הרצה, הריצו את תוכנת TurboDebugger.

3. התבוננו איך כל פקודת אסמבלי מיתרגמת לקוד מכונה. לדוגמה, הפקודה mov ax, 2 היתרגמה לקוד המכונה B80200.



נשאל את עצמנו כמה שאלות מנחות:

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, 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 לתוך רגיסטר אחר:

```
mov bx, ax
mov ax, cx
```

חזרה למשימת המחקר: נתונה הפקודה mov ax, dx. יש למצוא את התרגום שלה לשפת מכונה. יש למצוא את הפתרון בלי לתכנת את הפקודה ולבדוק איך האסמבלר מתרגם אותה לשפת מכונה, אלא רק על־ידי מחקר איך האסמבלר מתרגם לשפת מכונה פקודות דומות.

סיכום

בפרק זה הכרנו את כלי העבודה שלנו ואת סביבת התכנות באסמבלי, שנעבוד איתה מעכשיו ואילך. קיימות סביבות עבודה שונות ומגוונות, אך למען האחידות התמקדנו בסביבת עבודה אחת. הכלים שקיבלנו בפרק זה הם:

- Editor Notepad++
- הרצה דרך Command line, כולל פקודות DOS
- אמולטור DOSBOX
- אסמבלר Turbo Assembly
- לינקר Tlink
- דיבאגר Turbo Debugger

שליטה טובה בכלים אלו היא הכרחית להמשך. זה הזמן לוודא שכל הכלים מותקנים אצלכם במחשב ועובדים כמו שצריך. בפרק הבא אנחנו מתחילים לתכנת.

פרק 5 – IP, FLAGS

מבוא

חלקים מפרק זה – רגיסטר IP ודגל האפס – נדרשים עבור יחידת המעבדה שנלמדת בתכנית גבהים. יתר הנושאים מסומנים כנושאי הרחבה (תמצאו לידם את הסימן), הם אינם נדרשים ליחידת המעבדה של תכנית גבהים אך הם נדרשים על ידי משרד החינוך בבחינת הבגרות במחשבים.

כשערכנו היכרות עם המעבד ומרכיביו, הזכרנו בקצרה שני רגיסטרים ייעודיים, Special Purpose Registers, רגיסטר IP ורגיסטר FLAGS.

IP - Instruction Pointer- מצביע ההוראה

FLAGS – דגלים

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

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

IP – Instruction Pointer

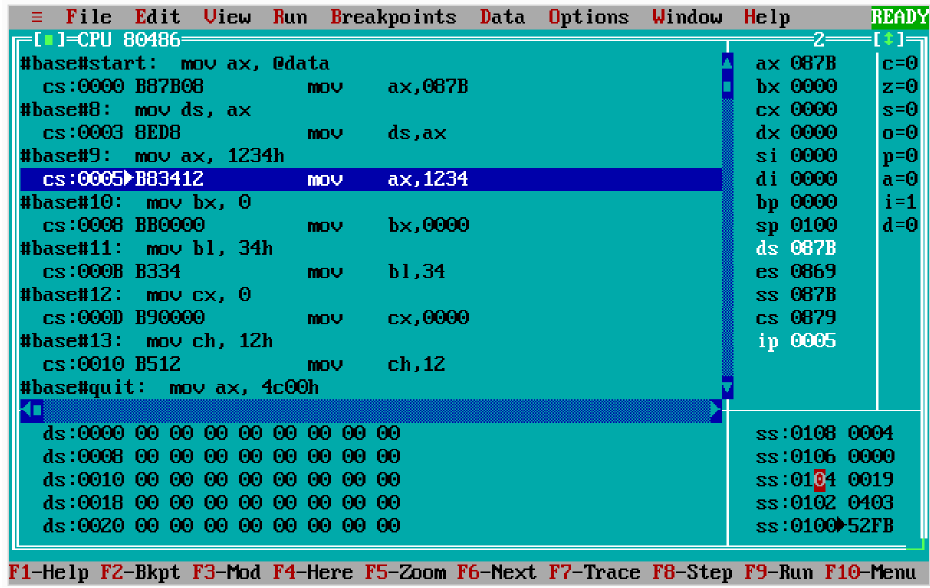
רגיסטר ה-IP מחזיק את הכתובת של הפקודה הבאה לביצוע. זהו רגיסטר של 16 ביט שלמעשה משמש כמצביע (pointer) לתוך ה-code segment.

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

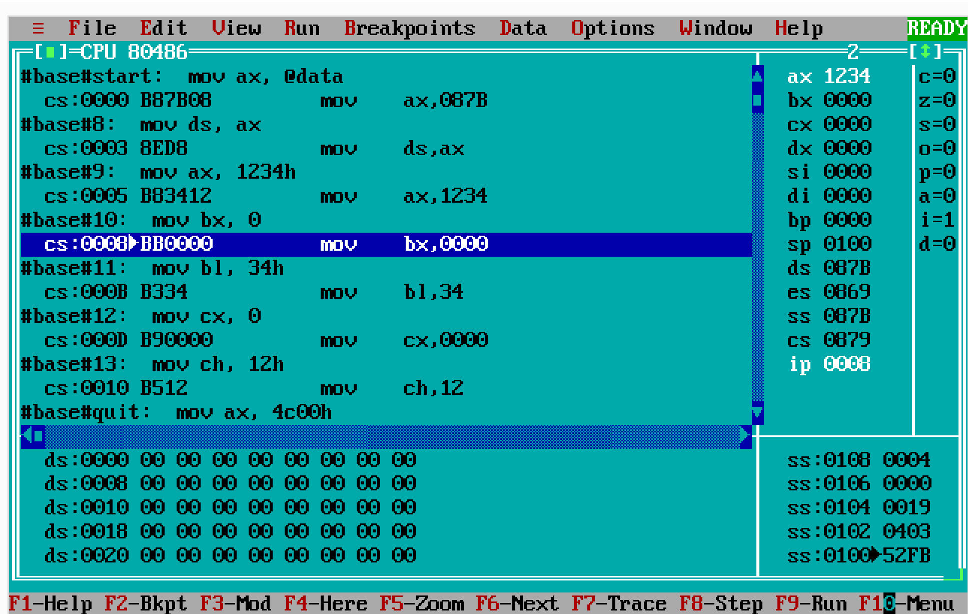
```
IDEAL
MODEL       small
STACK       100h
DATASEG
CODESEG
start:
        mov     ax, @data
        mov     ds, ax
        mov     ax, 1234h
        mov     bx, 0
        mov     bl, 34h
        mov     cx, 0
        mov     ch, 12h
exit:
```

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

בואו נראה מה קורה ל-IP עם ריצת התוכנית: התכנית נמצאת כרגע בשורה mov ax, 1234h, IP=0005, כלומר IP מצביע על הבית השישי בסגמנט הקוד.



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



השינויים שהתרחשו הם:

1. הפס הכחול במסך, שמציג את השורה הבאה שהמעבד יריץ, התקדם בשורה אחת (לשורה mov bx, 0).

2. הפקודה mov ax, 1234h בוצעה, וערכו של ax השתנה בהתאם.

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

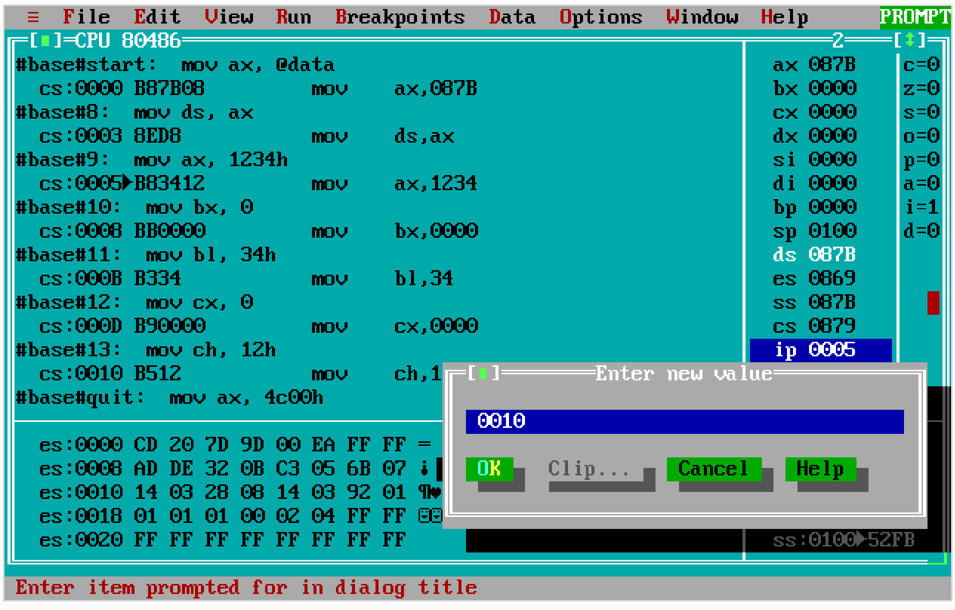
נתעכב רגע על מנת להסביר את השינוי שחל ב־IP.

הסיבה שההתקדמות הייתה בשלושה בתים, היא משום שהפקודה mov ax, 1234h תופסת שלושה בתים בזיכרון התכנית. 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 ועיקבו אחרי השינוי של IP בין פקודה לפקודה. האם אתם מזהים מצבים שבהם IP אינו משתנה בשלושה בתים?

ב. הריצו את התוכנית מההתחלה, ועיצרו כאשר IP=0005h. כדי לשנות את ערכו של IP באופן ידני, הקליקו עליו עם העכבר (או עימדו עליו ולחצו על מקש enter) והכניסו ערך כלשהו. ייפתח לכם חלון כדוגמת החלון הבא:

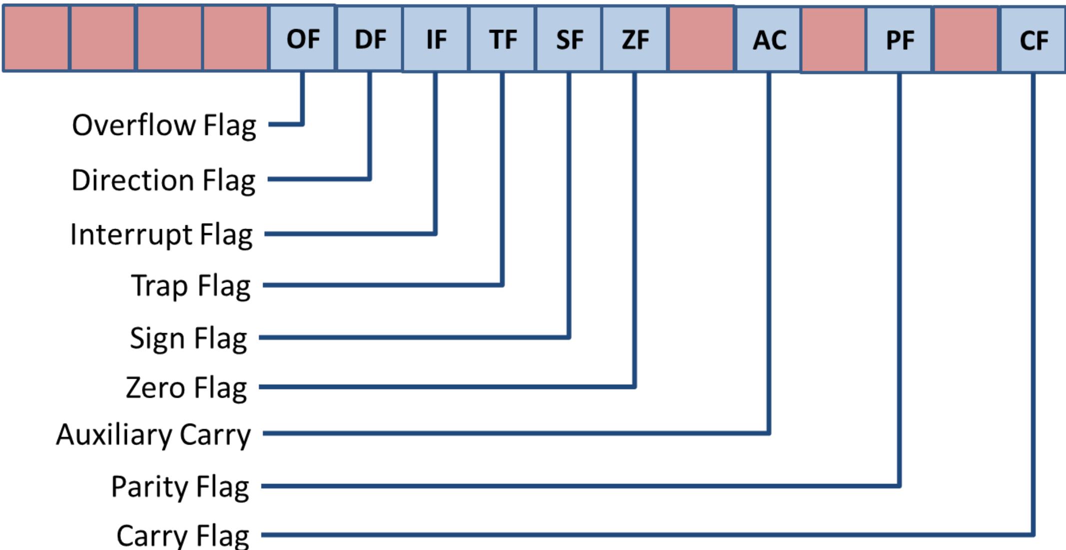


ג. שנו את ערכו של IP ל־0010h, ליחצו על enter וודאו ש־IP מקבל את הערך שהזנתם. עיברו לפקודה הבאה על־ידי לחיצה על F8. לאיזו פקודה הגעתם? האם הפקודות שבין 0005 לבין 0010 במקטע הקוד אכן התבצעו (שימו לב לערכם של הרגיסטרים ax, bx, cx)?

Processor Status Register – FLAGS

הרגיסטר Processor Status Register, מוכר יותר בכינויי 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 ביט אחד, שעוזרים לקבוע את מצב המעבד. למרות שיש ב-FLAGS 16 ביטים, רק לחלקם יש שימוש.

להלן שמות הדגלים והמיקום שלהם בתוך רגיסטר ה-FLAGS:



מבין הדגלים השונים, קיימים ארבעה דגלי תנאים – Condition Codes:

- Zero Flag
- Overflow Flag
- Carry Flag
- Sign Flag

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

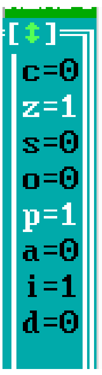
דגל האפס – Zero Flag

דגל האפס יהיה 1 אם הרצת הפקודה האחרונה גרמה לאיפוס של אופרנד היעד. בכל מקרה אחר, ערכו יהיה 0. (הערה: לכלל זה קיימים שני חריגים: פעולות כפל וחילוק ופקודת העתקה mov. לאחר פעולות כפל וחילוק לא ניתן להסתמך על ערכו של דגל האפס לקביעת התוצאה, ואילו פקודת mov אינה משפיעה על מצב הדגלים).

אופרנד היעד- Destination Operand - המקום אליו מועתקת התוצאה. מקום זה יכול להיות רגיסטר או כתובת בזיכרון.

דוגמה: נטען את הערך 4Bh גם לתוך al וגם לתוך ah (חישבו: איך אפשר לעשות את זה באמצעות פקודה יחידה?). כעת נבצע פעולת חיסור (**sub**tract) בין הרגיסטרים. שימו לב לתוצאה של הרצת הפקודות על הדגלים:

```
mov     al, 4Bh         ; 75 decimal
mov     ah, 4Bh         ; 75 decimal
sub     al, ah          ; subtract al minus ah, result is 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 לתשומת לב מיוחדת. תוצאת תרגיל זה היתה אמורה להיות 100h, אך כיוון שתוצאת החיבור נשמרת 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 פעולות השוואה בין זוג ערכים. כל פעולת השוואה גורמת למעבד להריץ "מאחורי הקלעים" פעולת חיסור. אם תוצאת 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 ההיכרות עימו.

תרגיל 5.2

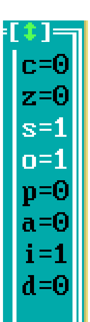
כיתבו קוד שמדליק את דגל האפס בעזרת פעולת **חיסור**, השתמשו ברגיסטרים של 16 ביט. כיתבו קוד שונה, שמדליק את דגל האפס בעזרת פעולת **חיבור**, השתמשו ברגיסטרים של 16 ביט.

דגל הגלישה – Overflow Flag

פקודות שהתוצאה שלהן "גולשת" מתחום ערכים מוגדר, גורמות לדגל הגלישה לקבל את הערך 1, כלומר "לדלוק". תחום הערכים תלוי במספר הביטים שמשמשים לייצוג מספרים בפעולה, כאשר מתייחסים אל הייצוג של המספרים בתור signed. לדוגמה, אם אנחנו מבצעים פעולות בעזרת משתנים או רגיסטרים בגודל של 8 ביטים, שכזכור יכולים לשמור מספרים signed בתחום מ (-128) עד +127, דגל הגלישה יידלק אם התוצאה הנשמרת באופרנד היעד חורגת מהתחום (-128) עד +127. אם תוצאת הפעולה המתמטית אינה יוצרת גלישה, המעבד ינקה את הדגל ויציב בו את הערך 0.

ניקח 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
```



דגל הגלישה מסומן בתוכנה באות 'o'

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

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

תרגיל 5.3

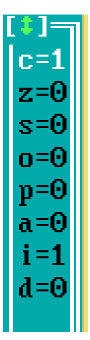
העתיקו את שורות הקוד שבסעיף זה לתוך base.asm, הריצו ועיקבו אחרי השינוי בדגל הגלישה.

דגל הנשא – Carry Flag

כמו דגל הגלישה, גם דגל הנשא, Carry Flag, נדלק עקב חריגה מתחום ערכים מוגדר, אך תחום הערכים שמוגדר לו הוא שונה. תחום הערכים תלוי במספר הביטים שמשמשים לייצוג מספרים בפעולה, כאשר מתייחסים אל הייצוג של המספרים בתור unsigned. לדוגמה, אם אנחנו מבצעים פעולות בעזרת משתנים או רגיסטרים בגודל של 8 ביטים, שכזכור יכולים לשמור מספרים unsigned בתחום מ-0 עד 255+, דגל הנשא יידלק אם התוצאה הנשמרת באופרנד היעד חורגת מהתחום 0 עד 255+. עבור אופרנד יעד בגודל 16 ביטים התחום 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
```



דגל הנשא מסומן בתוכנה באות 'c'

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

נסתכל על דוגמה אחרת, שאינה מדליקה את דגל הנשא, ונבין את הסיבה לכך:

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

לכאורה תוצאת החישוב האחרון צריכה להיות 88-, מחוץ לתחום המוגדר בתור unsigned, ולהדליק את דגל הנשא. אולם, בפעולה זו לא היינו עקביים- התייחסנו 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. זהו גם הייצוג של 128+, כאשר מפרשים 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

ערכו של דגל הסימן יהיה 1 אם הביט העליון (השמאלי ביותר) של התוצאה הנשמרת באופרנד היעד הוא 1. אחרת ערכו של דגל הסימן יהיה 0. דרכים פשוטות לראות אם מספר הוא שלילי, כאשר מתייחסים אליו כמספר מסומן Signed:

- בייצוג בינארי – הביט השמאלי הוא בעל ערך 1.
- בייצוג הקסדצימלי – ה־nibble השמאלי נמצא בתחום שבין 8 ל־F. לדוגמה: פעולה שהתוצאה שלה היא 0F1h, 0A3h, 088h (משתנים או רגיסטרים בגודל בית), או פעולה שהתוצאה שלה היא 0F100h, 0A300h, 08800h (משתנים או רגיסטרים בגודל מילה).

תרגיל 5.5

שנו את base.asm כך שבזמן ההרצה ישתנה ערכו של דגל הסימן. הריצו ועיקבו אחרי השינוי.

דגל הכיוון – Direction Flag

דגל הכיוון משמש בעבודה עם מחרוזות. כשדגל הכיוון שווה 0, המעבר עובר על אלמנטים במחרוזת מהכתובות הנמוכות אל עבר הכתובות הגבוהות. כשדגל הכיוון שווה 1, המעבר הוא מהכתובות הגבוהות אל הנמוכות.

דגל הפסיקות – Interrupt Flag

דגל זה שולט על היכולת של המעבד להיענות למאורעות חיצוניים שנקראים **"פסיקות" (Interrupts)**. תוכניות מסויימות מכילות קוד שחייב לרוץ ברצף וללא הפרעה של מאורעות חיצוניים. לפני הרצת קטעי קוד אלו, משנים את דגל הפסיקות ל־0, וכך מבטיחים שהקוד ירוץ ללא הפרעה. בסיום ריצת הקוד מאפשרים חזרה את הפסיקות, על־ידי קביעת ערך דגל הפסיקות ל־1.

דגל צעד יחיד – Trace Flag

דגל צעד יחיד מכניס את המעבד למצב Trace. במצב זה, המעבד מפסיק את פעולת העיבוד לאחר כל שורת קוד ומעביר את השליטה לתוכנה חיצונית. פעולה זו מאפשרת את פעולתם של תוכנות debugger כגון turbo debugger. אם ערכו של הדגל שווה ל־0, המעבד מריץ את התוכנה ללא עצירה.

דגל זוגיות – Parity Flag

זהו אינו דגל שנשתמש בו. ערכו של דגל הזוגיות נקבע לפי כמות האחדות בשמונת הביטים התחתונים של כל פעולת חישוב. אם בסופה של פעולת החישוב, בשמונת הביטים התחתונים יש מספר זוגי של '1' (0,2,4,8), דגל הזוגיות יקבל ערך 1. אחרת יתאפס.

דגל נשא עזר – Auxiliary Flag

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

תרגיל 5.6

האם 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 שעל ידי פעולה אחת של חיבור או חיסור, מדליק בו זמנית את דגל הגלישה, את דגל הנשא ואת דגל הסימן.

תרגיל 5.9 (אתגר)

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

סיכום

בפרק זה למדנו אודות הרגיסטרים הייעודיים (Special Purpose Registers), רגיסטר ה־IP ורגיסטר הדגלים. ראינו איך ערכו של IP משתנה בזמן ריצת התוכנית וגילינו מה קורה כשמשנים אותו.

חקרנו את רגיסטר הדגלים, והעמקנו במיוחד בדגלים שישמשו אותנו בהמשך:

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

ראינו איזה סוג של פעולות גורמות לכך שערכם של הדגלים הללו ישתנה.

בהמשך, נשתמש בשילוב של רגיסטר ה־IP עם רגיסטר הדגלים על מנת לבצע פעולות בדיקה וקפיצה, שהינן הבסיס לכתיבת אלגוריתמים בתוכנה.

פרק 6 – הגדרת משתנים ופקודת mov

מבוא

את הפקודות הבסיסיות בשפת אסמבלי אנחנו נלמד בשלושה חלקים:

- בחלק הראשון, בו נעסוק בפרק זה, נלמד איך מגדירים משתנים, איך קובעים בהם ערכים ואיך משתמשים בפקודת mov כדי להעתיק ערכים אל משתנים בזיכרון או אל רגיסטרים.
- בחלק השני נלמד פקודות אריתמטיות (פעולות חשבון), פקודות לוגיות ופקודות הזזה.
- בחלק 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 צנועות. לדוגמה, מציאה של האיבר הגדול ביותר מתוך רשימת איברים, מיון של איברים מהגדול לקטן, ספירת מספר איברים ברשימה וכדומה.

אז קדימה, בואו נראה איך מגדירים משתנים באסמבלי.

הגדרת משתנים

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

```
mov     al, [ds: 1h]
```

תטען לתוך al את הערך שבסגמנט ds ובאופסט 1h (כלומר בית אחד לאחר תחילת הסגמנט DS). הסוגריים המרובעים אומרים לאסמבלר שצריך להתייחס לערך שנמצא בכתובת שבתוך הסוגריים. כלומר אם בכתובת ds:1h נמצא הערך 5, אז הערך 5 יועתק לתוך al. הבעיה היא שאם התוכנית שלנו מלאה בהצבות זיכרון בשיטה הזו, יהיה לנו קשה למדי לדבג אותה. נצטרך לזכור בראש מה אנחנו שומרים במקום 1h. כמו כן, אם נחליט לשנות מעט את הסדר שבו אנחנו שומרים את הערכים, לדוגמה לשים משתנה אחר במקום 1h, אז נצטרך לשנות את הקוד של התכנית.

אחד מתפקידיו החשובים של האסמבלר הוא לאפשר למתכנת להשתמש בשמות משמעותיים בשביל מקומות בזיכרון. לדוגמה, אם הכתובת ds:1h תיקרא age, אז כל פעם שנפנה ל-age נגיע לכתובת ds:1h. כעת ניתן יהיה לכתוב את הקוד שלנו כך:

```
mov     al, [age]
```

האסמבלר לא רק נותן תווית שם למקומות בזיכרון, הוא גם דואג בשבילנו להקצאת המקום ואפילו – אם נרצה בכך – לקביעת ערך התחלתי למיקום שהקצנו בזיכרון. התוויות שאנחנו נותנים לכתובות בזיכרון קרויות **משתנים** **(Variables)**.

משתנה הוא שם שניתן לאזור מוגדר בזיכרון, שיכול לקבל טווח מוגדר של ערכים.

אפשר לחשוב על קובייה בתור 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 ה־DATA. היזכרו בקובץ base.asm – מיד לאחר התווית "start" יש פעולה של הצבת ערך לתוך ds:

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

פקודות אלו גורמות ל־ds להחזיק את ערך סגמנט ה־DATA. בכל פעם שאנחנו פונים למשתנה, אם לא הגדרנו אחרת, האסמבלר יחפש אותו בסגמנט שכתובתו שמורה ב־ds. כדי למנוע בעיות מיותרות, אנחנו פותחים את התוכנית בהצבה של הערך הנכון בתוך ds. אחרת, פניה למשתנה age תגרום לטעינת ערך עם אופסט נכון, אך מסגמנט אקראי כלשהו, ולכן לתוך age יועתקו ערכי "זבל".

הקצאת מקום בזיכרון

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

כדי להגדיר משתנה בגודל בית אחד בתוך סגמנט ה־DATA, נשתמש בהגדרה כגון:

```
ByteVarName db      ?
```

במקום ByteVarName יבוא שם המשתנה. ההגדרה db מסמנת לקומפיילר שהמשתנה תופס מקום בגודל בית אחד (DB – קיצור של Define Byte). סימן השאלה מציין שאנחנו רק מקצים מקום בזיכרון למשתנה ByteVarName, אך לא קובעים ערך למשתנה. במילים אחרות איננו משנים את הערך האקראי שישנו בזיכרון. חשוב לזכור שבזיכרון תמיד יש ערך כלשהו, ואם נכתוב:

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

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

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

```
DATASEG
ByteVarName db      ?
ByteVar2    db      ?
ByteVar3    db      ?
```

ארגון המשתנים בזיכרון יהיה כך:

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

אם נרצה לשלב משתנים מגדלים שונים, כל מה שנצטרך 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. הגדירו ב־DATASEG משתנה בשם var (קיצור של variable, משתנה) בגודל byte.
העתיקו את הקוד הבא לתוך התוכנית שלכם בסגמנט CODESEG. הקוד מעתיק את הערך '5' לתוך var.
כעת קמפלו את התוכנית, היכנסו ל־(Turbo Debugger (TD והריצו 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 הסברנו על חשיבות הטעינה של כתובת סגמנט הנתונים לתוך DATASEG. הכניסו את השורה mov ds, ax להערה בעזרת הוספת נקודה פסיק בתחילת השורה. כעת קמפלו שנית, היכנסו ל־ Turbo Debugger (TD) והריצו את התוכנית. האם קיבלתם את הערך שציפיתם לו?

משתני Signed, Unsigned

שימו לב שהגדרות אלו אינן קובעות אם המשתנה שהגדרנו הוא signed או unsigned – כלומר מה טווח הערכים שניתן להכניס במשתנים. כל מה שהגדרות אלו קובעות, הוא כמה בתים Bytes בזיכרון יוקצו לטובת המשתנים.

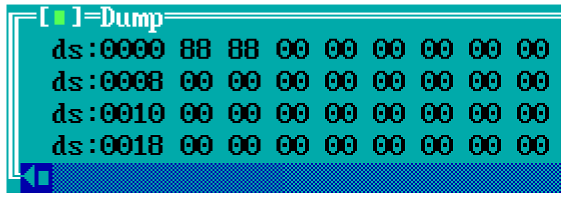
נראה את הטענה האחרונה שלנו בעזרת דוגמה. נגדיר משתנים בגודל Byte אחד:

```
DATASEG
Var1        db      ?
Var2        db      ?
```

בתוך CODESEG נכניס למשתנים אלו ערכים:

```
mov         [Var1], -120
mov         [Var2], 136
```

נסתכל על הזיכרון ב־DATASEG לאחר הצבות אלו. אפשר לראות ששני המשתנים הם בעלי אותו ערך! שני המשתנים מכילים את הערך 88h.


[▪]=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 שמינוס 120 שווה לפלוס 136? יכול להיות שמצאנו באג במעבד?

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

האם העובדה שלשני מספרים שונים יש את אותו הייצוג אינה גורמת לשגיאות חישוב במעבד? בואו נבדוק את העניין היטב. אנחנו נראה שאפשר לקחת מספר, +120, לחבר אותו פעם אחת למינוס 120 ופעם אחרת לפלוס 136 ולקבל בשני המקרים את אותה תוצאה:

- בפעולת חיבור של פלוס 120 עם מינוס 120, התוצאה היא כמובן אפס.
- בפעולת חיבור של פלוס 120 עם פלוס 136, התוצאה היא 256, או 100h. כיוון שמדובר במשתנה בגודל בית, לא ניתן לשמור בתוכו את התוצאה במלואה, ונשמרים רק שמונת הביטים הימניים – 00h, כלומר אפס.

בעזרת דוגמה זו ראינו, שאת הערכים ששמורים בזיכרון המחשב אפשר לפרש בתור מספר signed או unsigned - האחריות לפרשנות היא של המשתמש. במעבד אין באג.

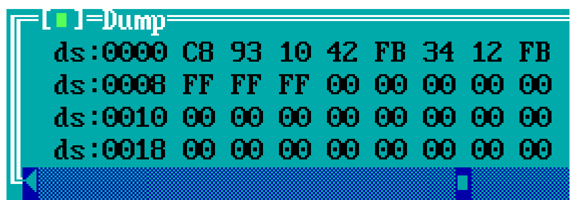
קביעת ערכים התחלתיים למשתנים

לא חייבים לחכות ל-CODESEG בשביל לטעון ערכים התחלתיים למשתנים. כבר בזמן ההגדרה, אנחנו יכולים לקבוע למשתנים ערכים התחלתיים.

שימו לב לכך שניתן להגדיר כל ערך שניתן לשמור בכמות הביטים שמוגדרת למשתנה, ושהאסמבלר מתייחס לכל המספרים כאילו שהם בבסיס 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 (0FFFFFFFBh)
```

כך ייראה הזיכרון לאחר פעולת ההקצאה:



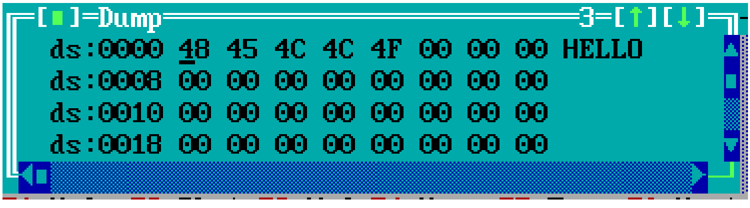
שימו לב במיוחד לכך שהגדרנו שני משתנים שקיבלו את הערך מינוס 5. הראשון בגודל בית, השני בגודל מילה כפולה (ארבעה בתים). למרות שנראה לנו שלשניהם אותו ערך – כל אחד מהם מיוצג בזיכרון בצורה אחרת.

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

```
DATASEG
ByteVarName db     'HELLO'
```

לכאורה יש כאן בעיה, מכיוון שהגדרנו משתנה בגודל בית, שמכיל חמישה תווי ASCII, כל תו בפני עצמו הוא בגודל בית. למעשה, האסמבלר יודע להתייחס להגדרה זו כאילו הגדרנו חמישה תווים שונים ושמרנו אותם בזיכרון בזה אחר זה:

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



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

תרגיל 6.2: הקצאת זיכרון

באמצעות התוכנית base.asm, הגדירו ב-DATASEG משתנים בגדלים שונים – משתנה בגודל בית, משתנה בגודל מילה, משתנה בגודל מילה כפולה, תנו לחלקם ערכים התחלתיים, הוסיפו משתנה ששומר אוסף של תווי ASCII. בסיום, עיברו על הזיכרון ב-DATASEG ומיצאו כל אחד מהמשתנים שהגדרתם.

הגדרת מערכים

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

כתובת הבסיס של המערך היא הכתובת ממנה המערך מתחיל, ששווה בדיוק לכתובת של האלמנט הראשון במערך. אפשר לדעת מה הכתובת של כל אלמנט במערך, בעזרת כתובת הבסיס של המערך, אינדקס האלמנט וגודל האלמנט, באמצעות חישוב פשוט:

ElementAddress = ArrayBaseAddress + Index * ElementSize

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

הגדרת מערך בסגמנט ה-DATA מתבצעת בצורה הבאה:

ArrayName   SizeOfElement   N dup (?)

ArrayName הוא שם המערך, שניתן לקבוע לפי רצוננו.

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

N הוא כמובן כמות האיברים במערך. N חייב להיות מספר שלם וחיובי.

Dup הוא קיצור של duplicate, שכפול.

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

לדוגמה:

ArrayOfTenFives   db   10 dup (5)

ייצור 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
 ds:0018 00 00 00 00 00 00 00 00
```

האסמבלר ישכפל לתוך המערך את **כל** מה שכתבנו בסוגריים, גם אם זה יותר מאיבר אחד.

אם נגדיר מערך כזה:

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

התוצאה תהיה:

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

כפי שאנחנו רואים הוגדר בזיכרון מערך בגודל 32 בתים – שמונה פעמים הרצף 1,2,3,4, בגודל ארבעה בתים.

תרגיל 6.3: הקצאת זיכרון למערך

בכל התרגילים הבאים, הריצו את התוכנית שיצרתם ב־TD, ומיצאו את המקום בזיכרון בו נשמרים המשתנים שהגדרתם.

א. הגדירו ב־DATASEG מערכים בגדלים שונים: 3, 5 ו־7 בתים (Bytes).

ב. הגדירו מערך של 10 בתים שמאותחלים לערך '5'. הגדירו מערך של 10 מילים שמאותחלות לערך '5'. השוו את תמונת הזיכרון בשני המקרים!

ג. הגדירו מערך ששומר 20 פעמים את הרצף 4,5,6, כל משתנה בגודל בית.

פקודת MOV

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

בפרקים הקודמים הזכרנו בקצרה את פקודת mov, קיצור של "move". הסברנו עליה ברפרוף, רק כדי לאפשר דיון במספר נושאים חשובים. כעת הגיע הזמן להסבר מפורט אודות הפקודה.

פקודת ה־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)**. במילים אחרות, פקודת mov מעתיקה את המידע שבאופרנד המקור לתוך אופרנד היעד.

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

```
mov     ax, 22
```

אנחנו יכולים להכניס את הערך 22 באמצעות שימוש בהצגה ההקסדצימלית או הבינארית שלו:

```
mov     ax, 16h
mov     ax, 00010110b
```

באופן דומה אנחנו יכולים להכניס ערכים ליתר הרגיסטרים:

```
mov    bx, 199
mov    cx, 2321
mov    dx, 10
```

אפשר גם להעתיק לרגיסטר את הערך שנמצא ברגיסטר אחר:

```
mov    ax, bx
```

פקודה זו תעתיק ל־ax את הערך שנמצא ב־bx. שימו לב לכך ש-bx לא ישתנה בעקבות פעולת ההעתקה. הערך שבתוכו פשוט ישוכפל ל־ax. באופן דומה ניתן גם לכתוב:

```
mov    ax, cx
mov    ax, dx
mov    ax, ax
```

הפקודה האחרונה היא חוקית, למרות שלא תשנה כלום. היא פשוט תעתיק את ערכו של ax חזרה לתוך ax.

בתתי־הסעיפים הבאים נסקור את הצורות המותרות לשימוש בפקודת mov:

```
mov    register, register
mov    register, constant
mov    register, memory
mov    memory, register
mov    memory, constant
```

כפי שאתם רואים, לא ניתן לבצע העתקה memory אל memory. במילים אחרות, כל העתקה מהזיכרון אל הזיכרון צריכה לעבור דרך רגיסטר. לאחר שקראתם את הפרק על מבנה המחשב והזיכרון אתם ודאי מבינים את ההיגיון בדבר: פסי הבקרה והנתונים, שנחוצים 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
```

הוראה זו מעתיקה רגיסטר בן 8 ביט, 16 ביט (או יותר, במעבדים מתקדמים יותר) לתוך רגיסטר אחר, שחייב 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 קומפילציה.

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

תרגיל 6.4

א. העתיקו את ax לתוך bx.

ב. העתיקו את bx לתוך ax.

ג. העתיקו את ah לתוך ch.

ד. העתיקו את al לתוך dl.

העתקה של קבוע לרגיסטר

```
mov     register, constant
```

הוראה זו מעתיקה קבוע לתוך רגיסטר.

שימו לב לכך שגודל הרגיסטר צריך להיות בתואם לגודל הקבוע – ניסיון להעתיק ערך שניתן לשמור 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 ובייצוג הבינארי (הערה: ייצוגי בינארי מסתיים באות b, לדוגמה 00001111b). הריצו את התוכנית ב־TD ובידקו ש־al מקבל את הערכים הרצויים.

העתקה של רגיסטר אל תא בזיכרון

```
mov     memory, register
```

פקודה זו מעתיקה את ערכו של הרגיסטר לתוך הכתובת בזיכרון שמוחזקת על־ידי אופרנד היעד. אופרנד היעד יכול להיות ערך קבוע, ששווה לכתובת אליה אנחנו רוצים לפנות (שיטת Direct addressing), רגיסטר המחזיק את הכתובת (שיטת Indirect addressing) או שילוב של רגיסטר עם קבוע (שיטתIndexed 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 השניה, ערכו של ax יועתק לתוך הזיכרון לכתובת אליה מצביע המשתנה Var.

בדוגמה השלישית, ערכו של ax יועתק לתוך הזיכרון לכתובת השמורה ב־bx. הפקודה הזו:

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

שקולה לפקודות הבאות:

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

בדוגמה הרביעית, ערכו של ax יועתק לתוך הזיכרון לכתובת השמורה ב־bx ועוד 1, כלומר בית אחד אחרי הכתובת השמורה ב־bx.

תרגיל 6.6

א. הגדירו בתוך DATASEG את המשתנה var בגודל בית, קבעו את ערכו ההתחלתי בתור 0. העתיקו לתוך al את הערך 100 (דצימלי) ולתוך 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־TD ועיקבו אחרי השינויים ב־DATASEG בזמן ריצת התוכנית. וודאו שכל ארבעת הבתים הראשונים ב־DATASEG מקבלים את הערך 100 (בייצוג ההקסדצימלי שלו, כמובן).

העתקה של תא בזיכרון אל רגיסטר

השיטות להעתיק תא בזיכרון לתוך רגיסטר הן בדיוק כמו השיטות להעתיק רגיסטר לתוך תא בזיכרון, בהבדל אחד – האופרנדים הפוכים:

```
mov    register, memory
```

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

```
mov    ax, [1]
mov    ax, [Var]
mov    ax, [bx]
mov    ax, [bx+2]
```

תרגיל 6.7

כהמשך 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]
```

הריצו את התוכנית ב-TD ועיקבו אחרי השינויים ב-al בזמן ריצת התוכנית. וודאו שבכל אחד מארבעת הקטעים al מקבל את הערך 100 (בייצוג ההקסדצימלי 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
```

ההסבר על כך בהמשך.

רגיסטרים 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]
```

תחזיר שגיאה:

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

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

- נשתמש ב־bx כדי לשמור 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־bx ערך קבוע. דוגמאות:

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

- אם נרצה להגיע לכתובות בהיסט משתנה מהכתובת ששמורה על־ידי bx, נוכל להוסיף לו את di או את si, 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־bx 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 ראינו מספר דוגמאות לפקודת mov. בחלקן אופרנד היעד או אופרנד המקור ייצגו כתובת בזיכרון. לדוגמה:

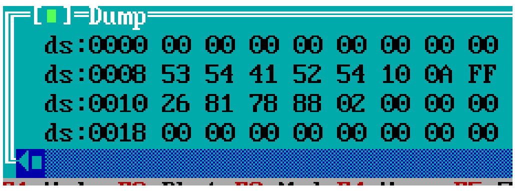
```
mov    [1], ax
mov    [Var], ax
mov    [bx], ax
```

השאלה היא כזו: אנו יודעים שכל כתובת בזיכרון מיוצגת באמצעות 20 ביט. איך בדיוק האסמבלר ממיר את הקבוע "1", את המשתנה var או את הרגיסטר bx (שכזכור, מכיל 16 ביטים), לכתובת בת 20 ביט?

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

השאלה הבאה שעולה היא: איך המעבד מכניס את ax, בגודל 16 ביט, לתוך כתובת בזיכרון בגודל בית אחד? התשובה היא, שהוא יודע להשתמש גם בבית הבא בזיכרון. נסתכל איך נראה זיכרון המחשב לאחר פקודת ה־mov. רק בשביל להפוך את הקריאה ב־DATASEG לברורה יותר, נגדיר בתחילת DATASEG מערך בן שמונה בתים, שמאותחל לאפסים:

```
DATASEG:

ZeroArray      db      8 dup (0)
```



עכשיו בתוך הקוד נריץ את השורות הבאות:

```
mov    ax, 0AABBh

mov    [1], ax
```

ושינינו את DATASEG:

```
ds:0000 00 BB AA 00 00 00 00 00
ds:0008 53 54 41 52 54 10 0A FF
ds:0010 26 81 78 88 02 00 00 00
ds:0018 00 00 00 00 00 00 00 00
```

זוכרים שאמרנו שהאסמבלר תמיד יתייחס לזיכרון כאילו הוא ב-DATASEG, **אלא אם כתבנו אחרת**?

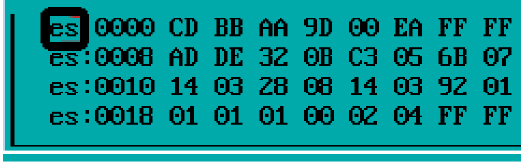
על מנת להתייחס לזיכרון שלא כאילו הוא ב-DATASEG, נחליף את הפקודה:

```
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' בכתובות המתאימות, הפעם בתוך הסגמנט Extended Segment שהרגיסטר ES מצביע עליו:



Little Endian, Big Endian

ראינו שכשאנחנו מעתיקים רגיסטר לזיכרון, הרגיסטר נראה "הפוך" – לדוגמה כשהעתקנו לזיכרון את ax, המעבד שמר את ah בתוך הכתובת הגבוהה יותר בזיכרון. למה דווקא בכתובת הגבוהה ולא בכתובת הנמוכה? הסיבה היא שכך מוגדר למעבד לבצע. אפשר גם היה להגדיר הפוך ולשמור את ah בכתובת הנמוכה.

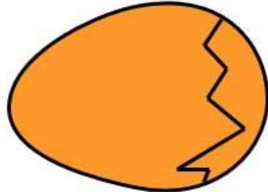
כאשר יש רגיסטר או משתנה כלשהו, בגודל של יותר מבית אחד (לדוגמה ax, bx שגודלם שני בתים וכו'), ואנחנו מעתיקים אותו לזיכרון, ההעתקה לזיכרון יכולה להיות באחת משתי שיטות:

- הבית ה־High Order (כלומר ah, bh וכו') נשמר בכתובת הנמוכה יותר בזיכרון. שיטה זו נקראת Big Endian.
- הבית ה־Low Order (כלומר al, bl וכו') נשמר בכתובת הנמוכה יותר בזיכרון. שיטה זו נקראת Little Endian.

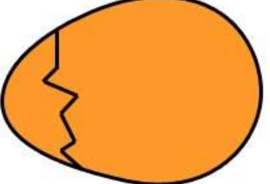
נציין שהשיטות הללו תקפות עבור כל רגיסטרים או משתנה שגודלם יותר מבית אחד.

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

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

מלך ליליפוט חוקק חוק שנוי במחלוקת לגבי האופן בו יש לשבור ביצה

העתקה ממערכים ואל מערכים

לעיתים אחד האופרנדים בפקודת mov הוא איבר במערך. זהו מקרה פרטי של העתקה מזיכרון לרגיסטר או של העתקה מרגיסטר לזיכרון. לדוגמה נתון המערך:

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

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

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

האסמבלר יתרגם את הפקודה הזו לכתובת בזיכרון. אם במקרה Array מוגדר ממש בתחילת DATASEG, האינדקס השני של Array נמצא באופסט 0002 מתחילת DATASEG ותוצאת התרגום לשפת מכונה תהיה:

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

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

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

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

והתוצאה היתה שערכו של al, שבעקבות הפקודה הקודמת הוא שווה לערך של אינדקס מספר 2 במערך, יועתק לתוך האינדקס הראשון במערך.

פקודת offset

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

```
mov     bx, offset Array
```

ואז פקודת ההעתקה ממערך לרגיסטר מקבלת את הצורה:

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

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

נשאל את עצמנו למה טוב השינוי הזה? 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 bx שומר את כתובת ההתחלה של המערך וכל פעם מקדמים את bx כך שהוא מצביע על האיבר הבא במערך.

פקודת LEA

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

```
mov     bx, offset Array
lea     bx, [Array]
```

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

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

ההנחיה word ptr / byte ptr

ניקח את אותו מערך של בתים שהגדרנו 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:

היינו מקבלים הודעת שגיאה של הקומפיילר.

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

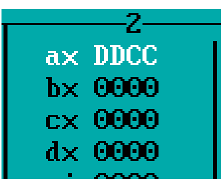
הסיבה היא שאופרנד המקור הוא בגודל בית (כל תו במערך שהגדרנו הינו באורך בית, שכן ציינו db) ואילו אופרנד היעד ax הוא בגודל מילה (שכן הרגיסטר הוא באורך 16 ביט).

אי לכך, עלינו ליידע את האסמבלר שאנחנו מנסים לבצע העתקה של שני בתים – ובכך להמנע מהשגיאה. שימו לב לצורת הכתיבה הבאה:

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

מה בעצם עשינו? הודענו לאסמבלר להתייחס אל המיקום בזיכרון לא בתור byte בודד, אלא בתור word (שני בתים). חישבו – מה יכיל הרגיסטר ax לאחר הרצת שורת הקוד הזו?

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



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

הערה: במקום לכתוב word ptr או byte ptr אפשר לקצר ולכתוב רק word או byte והאסמבלר יתרגם את ההנחיה בצורה נכונה. עם זאת, כדי לשמור על תאימות עם התצוגה ב־TD, נכתוב את הנוסח המלא: word ptr או byte ptr.

אזהרת type override

אם ננסה לבצע פקודת mov שהיא חוקית אך נתונה ליותר מפרשנות אחת, האסמבלר יחזיר לנו אזהרת type override.

לדוגמה, ניקח את צורת ההעברה החוקית הבאה:

```
mov     memory, constant
```

נניח שהיינו רוצים להכניס את הערך 5 לכתובת השמורה על־ידי bx, כך:

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

האסמבלר היה מחזיר לנו אזהרה:

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

מה לא בסדר בפקודה הזו? נתנו לאסמבלר כתובת להכניס אליה את הערך 5, אבל את הערך 5 אפשר לשמור במשתנה בגודל בית אחד, מילה, מילה כפולה... האסמבלר צריך לדעת כמה בתים להקצות לטובת שמירת הערך 5.

הפתרון הוא פשוט להורות לאסמבלר כמה בתים להקצות. שימו לב להגדרות הבאות:

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

ההגדרה הראשונה אומרת לאסמבלר, שאנחנו רוצים לגשת לאזור בזיכרון שגודלו בית יחיד. הקומפיילר ייצור פקודות מכונה שמשמעותן למעבד היא: גש אל הזיכרון בכתובת bx, הכנס לתוכה 00000101 (הייצוג הבינארי של 5 בגודל בית אחד). ההגדרה השניה אומרת לאסמבלר, שאנחנו רוצים לגשת לאזור בזיכרון שגודלו מילה. האסמבלר ייצור פקודות מכונה שמשמעותן למעבד היא: גש אל הזיכרון בכתובת bx, הכנס לתוכה 00000101, גש אל הזיכרון בכתובת bx+1, הכנס לתוכה 00000000.

נשאף להיות מפורשים על מנת להמנע משגיאות. אי לכך, בכל פעם שאנחנו מעתיקים משתנה באופן שגודל ההעתקה נתון לפרשנות, נכוון את האסמבלר באמצעות הסימון byte ptr או word ptr.

פקודת mov - טעויות של מתחילים

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

1. גדלי המקור והיעד אינם זהים. לדוגמה:

```
mov    al, bx
mov    ax, bl
```

פקודות אלו יגרמו לאסמבלר להחזיר שגיאה. שימו לב שבפקודה השניה גודל המקור יותר קטן מהיעד. לכאורה יש ביעד מקום להכניס את תוכן המקור, אך הוראה זו אינה מקובלת על האסמבלר.

2. הכנסת קבוע לתוך רגיסטר סגמנט. לדוגמה:

```
mov    ds, 1234h
```

גם במקרה זה האסמבלר יזכה אותנו 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 באופן ישיר. לכן, כדי להעתיק את var2 לתוך var1, נשתמש ברגיסטר עזר:

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

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

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

פקודה זו תגרום אף היא להודעת שגיאה של האסמבלר. רק bx, si ו־di משמשים לגישה אל הזיכרון (למעט גישה לזיכרון של המחסנית, עליה נלמד בהמשך).

5. העתקת מידע לגודל לא מוגדר בזיכרון. לדוגמה:

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

הקומפיילר לא יודע אם להעתיק את הערך 5 לזיכרון 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 לזיכרון – כולל זיכרון הקוד – ואנחנו יכולים לעשות בו כרצוננו. בואו נראה איך עושים את זה. נתחיל בהסבר תיאורטי קצר:

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

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

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

היזכרו בהסבר על שיטת הסגמנט והאופסט. הפעולה הראשונה שהאסמבלר עושה היא להניח שאתם מתכננים לפנות למקום שנמצא בסגמנט הנתונים. כיוון ש-ds מכיל את כתובת סגמנט הנתונים, הפעולה שהאסמבלר יבצע היא לקחת את ds, להכפיל ב-16 (היזכרו מדוע) ולהוסיף לו 1 (האופסט).

למעשה הפקודה האחרונה זהה לפקודה הבאה:

```
mov [ds:1], al
```

כל ההבדל הוא שהפעם ציינו במפורש מה הסגמנט שאליו אנחנו כותבים, בעוד שלפני כן נתנו לאסמבלר להניח באיזה סגמנט מדובר.

כעת ננסה משהו קצת משוגע:

```
mov [cs:1], al
```

מה עשינו כרגע? הורינו לאסמבלר להעתיק את הערך שיש ב-al לתוך הזיכרון אליו מצביע cs, כלומר סגמנט הקוד. אפשרות זו פותחת בפנינו הזדמנויות מעניינות. מה אם נשנה ערכים בסגמנט הקוד, כך שהערכים החדשים יהיו זהים לפקודות בשפת מכונה? ומה אם נשנה פקודות שהמעבד צפוי להריץ?

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

תרגיל 6.8: תרגיל אתגר- שינוי תוכנית תוך כדי ריצה

לפניכם קטע קוד. העתיקו אותו לתוך התוכנית שלכם בשלמותו:

```
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 ישנן מספר מגבלות:

- יש להעתיק את קטע הקוד בשלמותו ובלי להוסיף שורות קוד באמצע (מותר לפני ואחרי)
- אין להשתמש בפקודות קפיצה או בתוויות
- יש להשתמש אך ורק בפקודות mov

ב. כעת גירמו לכך שבסיום הריצה של קטע הקוד, ערכו של ax יהיה 3 וערכו של bx יהיה גם הוא 3. כל הכללים מסעיף א' תקפים גם כעת.

סיכום

בפרק זה למדנו:

- להגדיר משתנים מסוגים שונים ולתת להם ערכים התחלתיים
- לעבוד עם מערכים
- לבצע העתקות מידע שונות, שכוללות רגיסטרים ומקומות בזיכרון, בעזרת שימוש בפקודת mov:
  - העתקה מרגיסטר לרגיסטר
  - העתקה של קבוע לרגיסטר
  - העתקה מהזיכרון לרגיסטר
  - העתקה רגיסטר לזיכרון
  - העתקה של קבוע לזיכרון

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

פרק 7 – פקודות אריתמטיות, לוגיות ופקודות הזזה

מבוא

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

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

הפקודות האלו יהיו הבסיס לכל תוכנית שנרצה לכתוב.

נתחיל מפקודות אריתמטיות. הפקודה הראשונה שנלמד – חיבור.

פקודות אריתמטיות

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

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

בסעיפים הבאים נדון בפירוט בכל פקודה.

פקודת 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:

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

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. הריצו את התוכנית ב-TD, מיצאו את המערך בזיכרון וצפו בשינוי ב-al.

ב. בתרגיל הקודם שפתרתם, עלול להיות מצב שבו al אינו מתאים לשמירת התוצאה (מתי לדוגמה?) שנו את התוכנית, כך שסכום האיברים ייכנס לתוך ax.

ג. הגדירו שלושה משתנים:

- var1 בגודל בית
- var2 בגודל בית
- sum

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

פקודת SUB

הפקודה sub (קיצור של subtract) מחסרת את הערך של אופרנד המקור source מערך אופרנד היעד destination, ושומרת את התוצאה באופרנד היעד. כפי שאנחנו מצפים, צורות הכתיבה זהות לצורות הכתיבה של add. האופרנדים יכולים להיות מסוג רגיסטר, משתנה או קבוע. מבין כל האפשרויות, הפקודות הבאות הן חוקיות:

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

תרגיל 7.2: פקודת sub

א. הגדירו 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. חיסור המערכים הוא (3,1,-1,-3)

פקודות INC / DEC

פקודה inc (קיצור של increase) מעלה את ערכו של אופרנד היעד ב־1. פקודה dec (קיצור 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 שהוספת 1 או חיסור 1 הן פעולות נפוצות ביותר, הוקדשו להן פקודות נפרדות. הפקודות הבאות הן חוקיות:

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

פקודות MUL / IMUL

פקודה mul (קיצור של multiply) מבצעת הכפלה 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, האסמבלר יעתיק את התוצאה לתוך ax. במקרה של מכפלה באיבר בגודל 16 ביט, האסמבלר 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. תוצאת הכפל ביניהם היא 0AB0h. האסמבלר דואג שהתוצאה תועתק לתוך ax. כלומר ah=0Ah, al=0B0h.

לדוגמה, מכפלה של שני רגיסטרים בני 16 ביטים: ax=0AB0h, bx=1010h. תוצאת הכפל ביניהם היא 0ABAB00h. האסמבלר דואג שהתוצאה תחולק בין ax ו־dx. ax מקבל את שני הבתים הנמוכים ax=0AB00h ואילו dx מקבל את שני הבתים הגבוהים dx=0ABh.

להלן דוגמאות לפקודות חוקיות:

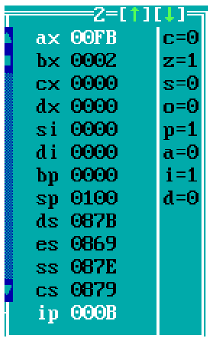
הפקודה	דוגמה	תוצאה
mul register (8 bit)	mul bl	ax = al * bl
mul register (16 bit)	mul bx	dx:ax = ax * bx
mul memory (8 bit)	mul [ByteVar]	ax = al * ByteVar
mul memory (16 bit)	mul [WordVar]	dx:ax = ax * WordVar

בפעולת הכפלה עלינו להגיד למעבד אם אנחנו עוסקים במספרים signed או unsigned, כיוון שהתוצאה משתנה. ניקח לדוגמה את המספר מינוס חמש. מספר זה מיוצג באמצעות הרצף 11111011 בשיטת המשלים ל־2. זהו ייצוג זהה לייצוג של 251. אבל אם נכפול כל אחד מהמספרים בשתיים, לדוגמה, התוצאות חייבות להיות שונות.

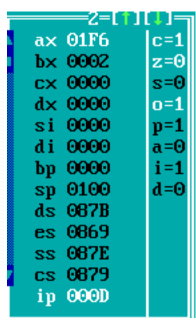
הפקודה mul מבצעת הכפלה של מספרים unsigned (לא מסומנים), ואילו הפקודה imul מבצעת הכפלה של מספרים signed (מסומנים). נבחן את ההבדל ביניהן.

הקוד הבא טוען לתוך al את 11111011 (ניתן לפירוש כ -5 או +251), לתוך bl את 00000010. הקוד מבצע ביניהם שתי הכפלות. פעם אחת unsigned על-ידי הפקודה mul ופעם נוספת הכפלה signed באמצעות הפקודה imul (כמובן שבין ההכפלות מאפסים את ax). שימו לב לערכו של ax לאחר כל אחת מההכפלות:

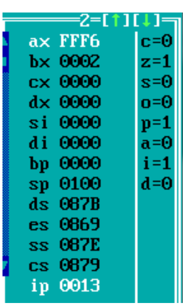
```
CODESEG:
        mov     ax, 0
        mov     bl, 00000010b
        mov     al, 11111011b
        mul     bl
        mov     ax, 0
        mov     al, 11111011b
        imul    bl
```



הרגיסטרים ax, bx לפני פעולות ההכפלה



בהכפלה על־ידי פקודת mul, האסמבלר מתייחס לערכו של al בתור 251 ולכן תוצאת הפעולה היא 01F6h, שווה ל־502



בהכפלה על-ידי פקודת imul, האסמבלר מתייחס לערכו של al בתור מינוס חמש, ולכן תוצאת הפעולה היא 0FFF6, שווה למינוס עשר

תרגיל 7.3: פקודת mul

א. הגדירו שני משתנים בגודל byte, הכניסו לתוכם ערכים בתחום 0-255, כיפלו אותם זה בזה והכניסו את התוצאה למשתנה שלישי (חישבו: באיזה גודל צריך להיות המשתנה השלישי?)

ב. הגדירו שני משתנים בגודל byte, הכניסו לתוכם ערכים בתחום +127 עד -128. כיפלו אותם זה בזה והכניסו את התוצאה למשתנה שלישי (חישבו: באיזה גודל צריך להיות המשתנה השלישי?)

ג. הגדירו שני מערכים, בכל מערך 4 ערכים מסוג בית, signed. שימו בהם ערכים התחלתיים. בצעו כפל של המערכים והכניסו את התוצאה לתוך sum. לטובת הפשטות, הניחו שהתוצאה נכנסת לתוך word. הדרכה: אם שמות המערכים הם a ו־b, אז

sum = a[0]*b[0]+a[1]*b[1]+…

פקודות DIV, IDIV

פקודת 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. תוצאת החילוק ביניהם היא 3 עם שארית 1. לאחר ביצוע הפעולה, ah=1, al=3.

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

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

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

מתי עוד תתעורר בעיה? באחד מן המקרים הבאים:

- חלוקה באפס

- התוצאה אינה נכנסת ברגיסטר היעד (האם תוכלו לחשוב על דוגמה מספרית כזו?)

דוגמאות לפקודות חוקיות:

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

הריצו את התוכנית הבאה ועיקבו אחרי ערכם של הרגיסטרים לפני ואחרי ביצוע פעולת 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 idiv זהה לפקודת div, פרט לכך שהיא מבצעת חילוק של מספרים מסומנים (signed), בניגוד ל-div שמחלקת מספרים לא מסומנים (unsigned).

תרגיל 7.4: פקודת div

א. הגדירו שני משתנים בגודל byte, התייחסו אליהם כ־unsigned, הכניסו את תוצאת החלוקה שלהם למשתנה שלישי ואת השארית למשתנה רביעי.

ב. הגדירו שני משתנים בגודל byte, התייחסו אליהם כ־signed, הכניסו את תוצאת החלוקה שלהם למשתנה שלישי ואת השארית למשתנה רביעי.

ג. הגדירו שני משתנים בגודל word, התייחסו אליהם כ־unsigned, הכניסו את תוצאת החלוקה שלהם למשתנה שלישי ואת השארית למשתנה רביעי.

פקודת NEG

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

```
neg     dest
```

תבצע את החישוב הבא:

```
dest = 0 - dest
```

דוגמאות לשימוש חוקי בפקודה:

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

פקודות לוגיות

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

למה בכלל נרצה להתעסק עם ביטים בודדים? בגלל הצפנות, לדוגמה. כשמצפינים 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 00000001 00000001 00000000 00000000 00000001 00000000

את אותו המידע אפשר לשמור בבית אחד בצורה דחוסה:

01110010

בצורה דחוסה אנחנו יכולים להאיץ את מהירות ההצפנה והפענוח, יחסית למהירות שניתן להשיג כשעובדים בצורה לא דחוסה בה מוקצה בית נפרד לשמירת כל ביט.

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

00100101

מי שלא יודע מה הפעולה שעשינו, לא יוכל לשחזר את הערכים המקוריים ולפענח את המסר המוצפן. בשביל לבצע פעולה כזו, צריך לדעת "לשחק" עם ביטים בודדים. כעת נראה איך עושים את זה.

קיימות ארבע פקודות לוגיות – and, or, xor, not. 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 בדיוק באותה צורה כמו and.

הפקודה not יכולה להיכתב באחת מבין הצורות:

```
not     register
not     memory
```

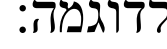
פקודת AND

מבחינה לוגית, and מקבלת כקלט שני ביטים. אם שניהם שווים 1, התוצאה תהיה 1. אחרת, התוצאה תהיה 0. נעשה היכרות עם מונח שנקרא "טבלת אמת". בשורה העליונה הערכים האפשריים לביט הראשון, בטור השמאלי הערכים האפשריים לביט השני. בתוך הטבלה – תוצאת ה־and על הביטים האלה.

AND	1	0
1	1	0
0	0	0

טבלת אמת של פעולת and

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


לדוגמה:

```
0000 0111 and
1001 0110
---------------
0000 0110
```

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

כדי לבדוק אם מספר כלשהו הוא זוגי, אנחנו יכולים לבדוק רק את הביט האחרון בייצוג הבינארי שלו. אם ביט זה הוא 0 – המספר הוא זוגי. אם הביט הוא 1 – המספר הוא אי זוגי.

לצורך הבדיקה, נגדיר אמצעי חדש שנקרא ***מסכה* (MASK)**. המסכה היא אוסף של ביטים, שמאפשר לנו לבודד ולפעול על ביטים מסוימים. מסכה יכולה להיות כל אוסף של ביטים, אבל אנחנו נגדיר אוסף מיוחד של ביטים: 00000001.

נעשה and בין המספר שאנחנו רוצים לבדוק לבין המסכה. כיוון שהגדרנו את שבעת הביטים העליונים כאפס, תוצאת ה־and שלהם תהיה 0 בכל מקרה. אי לכך, אנו מתעלמים למעשה מהערך של שבעת הביטים האלו. כתוצאה מכך, תוצאת ה־and של הביט השמיני תלויה רק במספר שאת הזוגיות שלו אנחנו בודקים – אם הביט השמיני שלו הוא 1, ה־and שלו יחד עם הביט השמיני במסכה, שהוא גם 1, יהיה שווה ל־1. אחרת, התוצאה תהיה 0.

תרגיל 7.5: פקודת and

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

פקודת OR

טבלת האמת של פקודת or נראית כך:

OR	1	0
1	1	1
0	1	0

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

פקודת or שימושית כשרוצים "להדליק" ביט כלשהו בלי לגעת בשאר הביטים. נניח שיש רכיב שמנהל תקשורת עם שמונה התקנים חיצוניים – הוא יכול לשלוח ולקבל מהם מידע, או להפסיק את התקשורת איתם (בהמשך, כשנלמד על פסיקות, נלמד על רכיב כזה). מצב התקשורת עם כל רכיב חיצוני מיוצג על־ידי ביט – 1 קובע שיש תקשורת עם הרכיב, 0 קובע שהרכיב מושתק. מצב התקשורת עם כל שמונת הרכיבים נשמר בבית מסוים.

כעת, אנחנו רוצים לאפשר תקשורת עם התקן מספר 4 (ההתקנים ממוספרים מ־0 עד 7). נקרא את זיכרון הרכיב. נניח שקיבלנו:

1100 0100

ביט מספר 4, אשר מייצג את התקן מספר 4, כבוי ומסומן בצהוב. כעת, ברצוננו להדליק את הביט הזה.

על מנת לעשות זאת, נבצע or עם המסכה 0001 0000 (הביט במיקום 4 דולק) ונקבל:

1101 0100

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

את התוצאה נעתיק חזרה לתוך רכיב התקשורת והתוצאה היא שאיפשרנו תקשורת עם התקן חיצוני מספר 4.

תרגיל 7.6: פקודת or

א. בדוגמה שנתנו, באיזו מסכה צריך להשתמש כדי לאפשר תקשורת עם רכיב מספר 2? עם רכיבים 2 ו־4 ביחד (בפקודת or אחת)?

ב. איך אפשר להורות לרכיב להפסיק את התקשורת עם התקן מספר 4?

פקודת XOR

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

XOR	1	0
1	0	1
0	1	0

מבחינה מתמטית, xor שקול לפעולת חיבור ומודולו 2 (חלוקה בשתיים, ולקיחת השארית בלבד).

פקודת xor היא פקודה שימושית מאוד, לדוגמה כשמבצעים הצפנה. הסיבה היא התכונה הבאה שלה:

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

התכונה המיוחדת הזו תשרת אותנו ביצירת הצפנה. נניח שאנחנו 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 פעולת xor. התוצאה שתתקבל היא:

1001 0011 xor

0101 0100

1100 0111

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

1100 0111 xor

0101 0100

1001 0011

המיוחד בשיטת הצפנה זו הוא, שאפילו אם מישהו מכיר את השיטה שבה השתמשנו בשביל להצפין את המידע שלנו, הוא לא יצליח לשבור את הצופן ולפענח את המידע. רק מי שיודע מה היה מפתח ההצפנה יכול לשחזר את המידע.

אגב, ה־opcode של xor קצר יותר משל mov, ולכן מעתה ואילך כשנרצה לאפס רגיסטר נעשה זאת על־ידי xor של ערך הרגיסטר עם עצמו. מכאן שבמקום לבצע:

```
mov     ax, 0
```

נכתוב:

```
xor     ax, ax
```

שתי הפעולות גורמות לכך שב־ax יהיה הערך 0, אך הפעולה השנייה לוקחת פחות מקום מהראשונה, ולכן נעדיף להשתמש בה.

תרגיל 7.7: פקודת xor

א. הסבירו מדוע התוצאה של xor של אוסף ביטים עם אוסף ביטים זהה – היא תמיד אפס.

ב. הגדירו מערך בשם msg שהוא אוסף של תווי ASCII. לדוגמה 'I LIKE ASSEMBLY$'. הגדירו מפתח הצפנה בן 8 ביטים, לבחירתכם. הצפינו את המידע בעזרת מפתח ההצפנה (תו ה־$ יסייע לכם לדעת שהגעתם לסוף המערך אותו יש להצפין, וכן להדפסה). לכו ל־DATASEG במקום שהגדרתם את המערך והסתכלו איך המחשב מתרגם את המערך לתווי ASCII לפני ואחרי פעולת ההצפנה שלכם. פענחו את המידע שהצפנתם בעזרת מפתח ההצפנה. חיזרו על הבדיקה ב־DATASEG וודאו שקיבלתם חזרה 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 פשוט הופכת את כל הביטים באופרנד. טבלת האמת של not:

	NOT
0	1
1	0

את צורות הכתיבה של not סקרנו בתחילת הסעיף. פקודת not מאפשרת גמישות נוספת בעבודה עם ביטים בודדים.

פקודות הזזה

פקודות הזזה מקבלות אופרנד ו"מזיזות" את הביטים שלו. לדוגמה, הביט במיקום 0 מוזז למיקום 1, הביט במיקום 1 מוזז למיקום 2 וכן הלאה. מיד נפרט איך בדיוק עובדת ההזזה.

יש אוסף של פקודות הזזה, שבעקרון די דומות אחת לשניה. אנחנו נתמקד בשתי פקודות ההזזה השימושיות ביותר: shr (shift right) ו־shl (shift left).

פקודת SHL

פקודת ההזזה שמאלה, shl, מקבלת אופרנד יעד וכמות ביטים להזזה. האופרנד יכול להיות משתנה בזיכרון או רגיסטר, וכמות הביטים להזזה יכולה להיות מספר קבוע או הרגיסטר cl. הצורות הבאות תקינות:

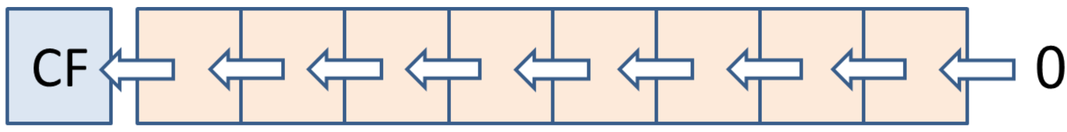
```
shl     register, const
shl     register, cl
shl     memory, const
shl     memory, cl
```

פעולת ה־shl משפיעה באופן הבא:

- על כל הזזה, הביטים שבאופרנד היעד זזים מקום אחד שמאלה.
- על כל הזזה נכנס 0 אל הביט הימני ביותר (אם ההזזה היא של ביט אחד, נכנס 0 אחד. אם ההזזה היא של n ביטים, נכנסים n אפסים).
- הביט האחרון שיצא מצד שמאל נכנס אל דגל הנשא CF.
- דגל הגלישה OF יקבל ערך 1 במקרה שהביט הכי שמאלי השתנה. כלל זה נכון רק עבור הזזה של ביט יחיד.
- דגל האפס ZF יקבל ערך 1 אם לאחר ההזזות ערכו של האופרנד הוא אפס.

- דגל הסימן יהיה שווה לערך של הביט הכי שמאלי.
- דגל הזוגיות יקבל ערך 1 אם יש כמות זוגית של אחדות ב־8 הביטים הנמוכים של האופרנד.

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



פעולת shl

פקודת SHR

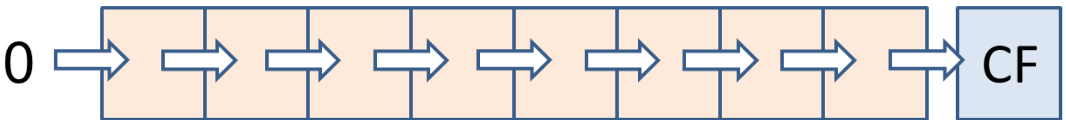
פקודת ההזזה ימינה, shr, כמובן דומה בכתיבה שלה ובפעולה שלה ל־shl. הפקודה מקבלת אופרנד יעד וכמות ביטים להזזה. האופרנד יכול להיות משתנה בזיכרון או רגיסטר, וכמות הביטים להזזה יכולה להיות מספר קבוע או הרגיסטר cl. הצורות הבאות תקינות:

```
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 ביט אחד, נכנס 0 אחד. אם ההזזה היא של n ביטים, נכנסים n אפסים).
- הביט האחרון שיצא מצד ימין נכנס אל דגל הנשא CF.
- דגל הגלישה OF יקבל את ערך הביט הכי שמאלי של האופרנד לפני ההזזה. כלל זה נכון רק עבור הזזה של ביט יחיד.
- דגל האפס ZF יקבל ערך 1 אם לאחר ההזזות ערכו של האופרנד הוא אפס.
- דגל הסימן יהיה שווה לערך של הביט הכי שמאלי, שהוא תמיד אפס.
- דגל הזוגיות יקבל ערך 1 אם יש כמות זוגית של אחדות ב־8 הביטים הנמוכים של האופרנד.

כך נראית פעולת 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 shr

שימושים של פקודות הזזה

פקודות הזזה שימושיות במיוחד לביצוע פעולות:

- ביצוע כפל וחילוק בקלות בחזקות של שתיים
- ביצוע פעולות עזר הקשורות להצפנה
- דחיסה של מידע ופריסה (הפעולה ההפוכה מדחיסה)
- גרפיקה

כפל וחילוק: בבסיס עשרוני, כשרוצים לכפול בעשר, כל מה שצריך לעשות זה להוסיף אפס בתור הספרה הכי ימנית (כלומר, להזיז את הספרות ספרה אחת שמאלה). בבסיס שתיים, אם נוסיף אפס בתור הספרה הכי ימנית, נקבל כפל בשתיים. כלומר כל הזזה שמאלה בביט אחד כופלת את הערך בשתיים. כיוון שערכים שמורים במשתנים עם גודל מוגבל, באיזשהו שלב נחרוג מהגודל של המשתנה ואז התכונה הזו תאבד, אבל אם נעבוד נכון ונדאג לגדלים נכונים של המשתנים, נוכל להשתמש בהזזה כדי לבצע כפל בשתיים, וכן הלאה – כפל בארבע באמצעות הזזת שני ביטים, בשמונה וכו'. כמו שההזזה שמאלה היא כפל בשתיים או בחזקות של שתיים, הזזה ימינה היא חילוק בשתיים או בחזקות של שתיים. גם כאן צריך לשים לב שאם הביט הכי ימני שלנו הוא 1, כלומר המספר אי זוגי, אז הזזה ימינה לא בדיוק תחלק אותו בשתיים – התוצאה תהיה חלוקה בשתיים, מעוגלת כלפי מטה.

הצפנה: ישנם סוגים רבים של הצפנה. אחד מסוגי ההצפנה הצבאיים הנפוצים הוא LFSR. בהצפנה זו ישנו מערך של ביטים שבכל פעם דוחפים ביט לצד אחד שלו ומוציאים ביט מצידו השני. בין הביטים במערך מבוצעות פעולות xor שונות, ועם התוצאה עושים xor לביט שאותו רוצים להצפין. אתם מוזמנים לקרוא על LFSR באינטרנט, ויקיפדיה היא מקום טוב להתחיל ממנו:

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

בכל אופן, כדי לממש את הפעולה של הזזת הביטים בצורה יעילה משתמשים בפקודות הזזה.

דחיסה ופריסה של מידע: הסבר מפורט הינו מחוץ להיקף של ספר זה. קיימים אלגוריתמי דחיסה ופריסה רבים, מומלץ להתחיל מאלגוריתם למפל זיו:

http://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch

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

תרגיל 7.8: פקודות הזזה

א. הכניסו ל־al את הערך 3. בעזרת פקודות הזזה, כיפלו את al ב־4.

ב. הכניסו ל־al את הערך 120 (דצימלי). בעזרת פקודות הזזה, חלקו את al ב־8.

ג. הכניסו ל־al את הערך 10 (דצימלי). בעזרת פקודות הזזה וחיבור, כיפלו את al ב־20. הדרכה: התייחסו ל־20 בתור סכום של 16 ו־4. השתמשו ברגיסטרים נוספים כדי לשמור 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 ביט בודד, כגון הדלקת וכיבוי ביט, הצפנה של מידע באמצעות מפתח.
- לבצע פעולות הזזה, שמשמשות לכפל וחילוק בחזקות של שתיים:
  - Shr
  - Shl

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

פרק 8 – פקודות בקרה

מבוא

בפרק זה נלמד ליצור תוכנית עם תנאים לוגיים ("אם מתקיים תנאי זה, בצע פעולה זו...").

כדי לכתוב פקודות שיש להן תנאים, אנחנו צריכים ללמוד כמה נושאים:

- איך קופצים למקום כלשהו בתכנית (פקודת קפיצה jmp על סוגיה השונים).
- איך בודקים אם תנאי כלשהו מתקיים (פקודת השוואה – cmp).
- איך לחזור על ביצוע קטע קוד מוגדר כל עוד מתקיים תנאי שהגדרנו (פקודת לולאה – loop).

באופן רגיל, התכנית מתקדמת שורה אחרי שורה. כפי שלמדנו קודם לכן, לאחר כל שורה, מצביע התכנית, שנמצא ברגיסטר IP, מקודם אל שורת הקוד הבאה בתוך CODESEG. תפקידן של הוראות הבקרה הוא לאפשר קפיצה קדימה או אחורה אל שורות קוד בתוכנית. קפיצה קדימה, כלומר דילוג על שורות בתוכנית, הכרחית כדי לאפשר לנו להריץ קוד כלשהו רק אם מתקיים תנאי מתאים - "אם ערכו של המשתנה שווה ל־1, המשך בביצוע התוכנית. אחרת – דלג על חמש שורות קוד והמשך משם". קפיצה אחורה הכרחית כדי לאפשר לנו לחזור על הרצה של קוד מספר פעמים, אם תנאי כלשהו מתקיים – "אם ערכו של המשתנה אינו אפס, קפוץ עשר שורות קוד אחורה והמשך משם".

הוראות הבקרה מתחלקות לשני סוגים: הוראות בקרה עם תנאי קיום ("אם מתקיים מצב מסויים, אז קפוץ ל...") והוראות בקרה ללא תנאי קיום ("קפוץ ל..."). לכל סוג יש שימוש יעיל במצבים שונים. למען הפשטות, נתחיל בהוראות בקרה ללא תנאי קיום.

פקודת JMP

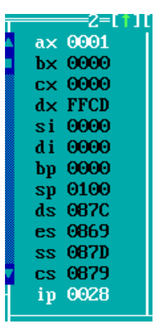
פקודת jmp שולחת את המעבד, ללא תנאי, לנקודה אחרת בתוכנית. פקודת jmp מקבלת כתובת בזיכרון, בתוך CODESEG. לאחר ביצוע פקודת ה-jmp, תועתק אותה הכתובת המובאת לה בתור אופרנד לתוך רגיסטר ה־IP והריצה תמשיך מכתובת זו.

לדוגמה:

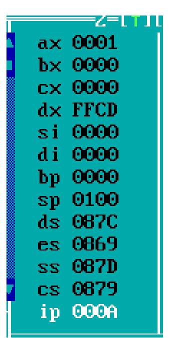
```
DATASEG
        address     dw     000Ah
CODESEG
        mov     ax, @data
        mov     ds, ax
```

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

כך נראים הרגיסטרים לפני ביצוע פקודת ה־jmp:



כך נראים הרגיסטרים לאחר ביצוע פקודת ה־jmp, שימו לב לערכו החדש של ip:



והתכנית תמשיך את הריצה מהבית העשירי ב־CODESEG.

קפיצות NEAR ו־FAR

בדוגמה שהשתמשנו בה כדי להסביר את פקודת ה־jmp, מסרנו לאסמבלר רק את האופסט שהוא צריך לקפוץ אליו – 000Ah – בעזרת שימוש במשתנה address. בזמן שהתכנית הגיעה לשורת ה־jmp שלנו, היא רצה מתוך סגמנט הקוד ופקודת ה-jmp שלחה אותה לאופסט אחר באותו סגמנט. מצב זה, של קפיצה בתוך אותו סגמנט (במקרה הזה מסגמנט הקוד אל חלק אחר בסגמנט הקוד), נקרא קפיצה **near**. כדי לבצע קפיצות מסוג near, מספיק שניתן לפקודת ה־jmp את האופסט – הסגמנט הרי לא משתנה.

קפיצות מסוג **far** הן במקרים שחלק מהקוד שלנו נמצא בסגמנט אחר. עקרונית, האסמבלר מאפשר לנו לחלק את הקוד שלנו לסגמנטים שונים, אם הוא גדול מכדי להיכנס בסגמנט אחד. במקרה זה, הכרחי להודיע לאסמבלר לאיזה סגמנט קוד אנחנו רוצים לקפוץ. צורת הכתיבה היא:

```
jmp     cs:offset       ; for example cs:000A
```

כאשר לפני הקפיצה יש לוודא ש-cs מצביע על כתובת הסגמנט.

אין סיבה להיות מוטרדים מנושא זה – לא סביר שנכתוב קוד שסגמנט קוד אחד לא יספיק עבורו. מצד שני, הבנת התיאוריה של נושא קפיצות ה־near וה־far חשובה להמשך.

תרגיל 8.1: פקודת jmp

נתונה התכנית הבאה (העתיקו אותה לתוך ה-CODESEG שבתכנית base.asm):

```
xor     ax, ax
add     ax, 5
add     ax, 4
```

בעזרת הוספת פקודת jmp לתכנית, גירמו לכך שבסוף ריצת התכנית ax=4.

תוויות LABELS

ברוב הפעמים לא נעביר ל־jmp ממש כתובת בזיכרון. הסיבה היא, שכל שינוי בתכנית שלנו ישנה את מיקום הפקודות ב־CODESEG ואז המעבד יקפוץ למקום לא נכון בתכנית. לא נרצה לעדכן את הכתובות בכל פעם שנבצע שינוי בתכנית שלנו. על מנת להקל עלינו, שפת אסמבלי מאפשרת לנו לתת **תווית (label)** לשורה בקוד, ובמקום לתת לפקודת ה-jmp את כתובת השורה, אפשר לתת 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 ירוץ אינסוף פעמים, ובכל פעם יקדם את הערך הנמצא ברגיסטר ax באחד.

שימו לב לאינדנטציה – הרווחים בתחילת כל שורה. כל ה- labelים תמיד נמצאים בעמודה השמאלית ביותר של הקוד, ופקודות נמצאות טאב אחד ימינה.

אנחנו יכולים לקבוע כל רצף של תווים בתור label, כל עוד הוא אינו מתחיל בספרה ואינו כולל רווח. כמו כן האסמבלר אינו מפריד בין אותיות גדולות לקטנות - מבחינתנו 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 עושה הקוד.

דוגמאות לשמות לא מוצלחים הם: Label1, MyLabel, Shooki וכו'. אלו שמות כלליים וחסרי משמעות. האסמבלר כמובן לא יעיר עליהם, היות שהם תקניים מבחינה טכנית, אבל הבעיה תהיה שלנו (ושל מי שינסה להבין את הקוד שכתבנו).

אבל יש סוג נוסף של שמות ל־label, שהאסמבלר יתריע עליהם כי ישנה שגיאה. אסור לקרוא ל־label בשם של רגיסטר (למשל: ax) או בשם של פקודת אסמבלי (למשל: MOV), ואין להשתמש בשמות מהשמות המופיעים להלן – אלו שמות שמורים.

נסיים בטעות נפוצה של מתחילים: אין לתת לשתי שורות קוד שונות את אותו שם label. האסמבלר פשוט לא יבין לאן אתם רוצים לקפוץ.

```
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       .XCREF      .XLIST      XOR

In addition to the above words, TASM also reserves these:

ARG         %BIN        CODESEG     %CONDS      CONST       %CREF
%CREFALL    %CREFREF    %CREFUREF   %CTLS       DATAPTR     DATASEG
%DEPTH      DISPLAY     DP          EMUL        ERRIF       ERRIF1
ERRIF2      ERRIFB      ERRIFDEF    ERRIFDIF    ERRIFDIFI   ERRIFE
ERRIFIDN    ERRIFIDNI   ERRIFNB     ERRIFNDEF   EVENDATA    FARDATA
FARDATA?    GLOBAL      IDEAL       %INCL       JUMPS       LARGE
%LINUM      %LIST       LOCALS      %MACS       MASM        MASM51
MODEL       MULTERRS    %NEWPAGE    %NOCONDS    %NOCREF     %NOCTLS
NOEMUL      %NOINCL     NOJUMPS     %NOLIST     NOLOCALS    %NOMACS
NOMASM51    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 התכנית שכתבתם בתרגיל ה־jmp, כך שפקודת ה־jmp תהיה אל label. תנו ל־label שם משמעותי.

פקודת CMP

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 שווים. הדלקה של דגלים אחרים (גלישה, נשא, סימן) מאפשרת למעבד לבדוק איזה אופרנד גדול יותר. בכך דומה פקודת cmp לפקודת sub, בהבדל אחד – היא לא משנה את ערכם של האופרנדים.

צורות כתיבה חוקיות של פקודת cmp:

הפקודה	דוגמה	תוצאה
cmp register, register	cmp al, bl	שינוי מצב הדגלים בהתאם ליחס בין האופרנדים
cmp register, memory	cmp ax, [WordVar]	
cmp memory, register	cmp [WordVar], cx	
cmp register, constant	cmp ax, 5	
cmp memory, constant	cmp [ByteVar], 5	

שינוי מצב הדגלים באמצעות הוראת cmp הוא נושא חשוב – בעוד כמה שורות נראה מה השימוש שלו.

בטבלה הבאה יש דוגמה למספר פקודות, שמורצות אחת אחרי השניה, והשפעתן על מצב הדגלים:

Code	CF	ZF	SF
mov al, 3h	?	?	?
cmp al, 3h	0	1	0
cmp al, 2h	0	0	0
cmp al, 5h	1	0	1

- בשורה הראשונה אנו פוקדים לטעון לתוך al את הערך 3. כיוון שפקודת mov אינה משנה את המצב הדגלים, בשלב זה איננו יודעים מה מצב הדגלים.
- בשורה השניה, אנו משווים את al ל־3. כפי שראינו, cmp מדמה חיסור של שני האופרנדים. תוצאת 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 al ל־2. התוצאה של al פחות 2 היא חיובית ולכן דגל האפס כבה.

- בשורה הרביעית אנו משווים את al ל־5. תוצאת פעולת החיסור al פחות 5 היא בעלת ביט עליון שערכו '1' לכן נדלק דגל הסימן, כמו כן כיוון שהמחסר גדול מהמחוסר יש צורך בנשא שלילי לחישוב התוצאה ולכן נדלק דגל הנשא.

קפיצות מותנות

פקודות של קפיצות מותנות הן כלי עבודה בסיסי ביצירת לולאות (קטע קוד שחוזר על עצמו מספר פעמים) והוראות מותנות ("אם מתקיים... אז בצע..."). באופן רחב יותר אפשר להגיד שקפיצות מותנות מאפשרות לנו לבנות תוכנית מעניינות – תוכניות שיש בהן קבלת החלטות, לוגיקה וטיפול במצבים שונים.

קפיצות מותנות עובדות בדרך הבאה:

- לפני פקודת הקפיצה, מבצעים בדיקה השוואתית של שני אופרנדים באמצעות פקודת cmp. תוצאת פעולת ה־cmp תהיה קביעת הדגלים לפי היחס בין האופרנדים, כפי שראינו קודם.

- פקודת הקפיצה בודקת אם דגל כלשהו, או כמה דגלים, מקיימים תנאי מוגדר. לדוגמה, האם דגל האפס שווה ל־1.

- אם התנאי מתקיים, הקוד קופץ לכתובת שהוגדרה על־ידי המשתמש. בדרך כלל כתובת זו תצויין באמצעות label.

- אם התנאי לא מתקיים, המעבד ממשיך את ביצוע הפקודות לפי הסדר (כלומר, עובר לפקודה הבאה שלאחר פקודת הקפיצה).

הערה: השלב הראשון, ביצוע פקודת cmp, אינו תמיד הכרחי מבחינה תיאורטית. יש קפיצות מותנות שבודקות ישירות מצב של דגל כזה או אחר. עם זאת, שילוב פקודת cmp מקל על התכנות ועל הקריאות של הקוד.

יש מגוון לא קטן של תנאים לקפיצה, כאשר כל תנאי בודק יחס אחר בין האופרנדים שהתייחסנו אליהם בפקודת ה־cmp. הדבר הראשון שצריך לשים אליו לב, הוא שאין דין השוואה של אופרנדים signed כדין השוואה של אופרנדים unsigned.

להלן שאלה למחשבה:

איזה מספר יותר גדול – 1b או 10000001b?

התשובה היא – כמו שבטח ניחשתם בשלב זה – שתלוי איך בודקים. אם מתייחסים אל שני המספרים בתור unsigned, הרי ש־10000001b (ששווה 129 בבסיס עשרוני) גדול מ־1b. לעומת זאת, בהשוואת מספרים signed , 10000001b מיתרגם למינוס 127, ולכן קטן מ־1b.

ברור, אם כך, שכדי לבצע קפיצות מותנות בתוצאות של cmp, אנחנו חייבים להודיע לאסמבלר איזה סוג השוואה ביצענו, או במילים אחרות – אילו דגלים לבדוק. לכן התפתחו שני סטים מקבילים של פקודות קפיצה, שבנויות כך:

כל פעולות הקפיצה מתחילות באות J, קיצור של Jump. פקודות קפיצה שבאות אחרי cmp של מספרים unsigned יכילו את האותיות B או A – קיצורים של Below ושל Above. לעומתן, פקודות קפיצה שבאות אחרי cmp של מספרים signed יכילו את האותיות L או G – קיצורים של Less ושל Greater. בנוסף לאות J ולאותיות שכבר הזכרנו, יכולות להתווסף האותיות N בשביל Not ו־E בשביל Equal.

ריכוז האפשרויות נמצא בטבלה הבאה. שימו לב להגדרת סדר האופרנדים:

cmp Operand1, Operand2

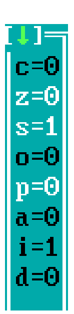
משמעות הפקודה	מספרים Signed	מספרים Unsigned
קפוץ אם האופרנד הראשון גדול מהשני	JG - Jump if Greater	JA - Jump if Above
קפוץ אם האופרנד הראשון קטן מהשני	JL - Jump if Less	JB - Jump Below
קפוץ אם האופרנד הראשון והשני שווים	JE - Jump Equal	
קפוץ אם האופרנד הראשון והשני שונים	JNE - Jump Not Equal	
קפוץ אם האופרנד הראשון גדול או שווה לאופרנד השני	JGE - Jump if Greater or Equal	JAE - Jump if Above or Equal
קפוץ אם האופרנד הראשון קטן או שווה לאופרנד השני	JLE - Jump if Less or Equal	JBE - Jump if Below or Equal

תיאור בדיקת הדגלים (הרחבה)

כל פקודת קפיצה מבוססת על בדיקה ותנאים לוגיים של דגל אחד או יותר. רק לטובת הבנה כללית, נסתכל לדוגמה על פקודת ה־cmp בין שני המספרים שהזכרנו בתחילת הסעיף:

```
mov     al, 10000001b
mov     bl, 1b
cmp     al, bl
```

פעולת cmp בין מספרים אלו תעביר את הדגלים למצב הבא:



פעולת JA, שמיועדת למספרים unsigned, צריכה להחזיר תשובה חיובית (129 גדול מ־1). התנאים שייבדקו הם אם CF שווה לאפס ואם ZF שווה לאפס. תוצאת הבדיקה תהיה חיובית, ולכן תתבצע קפיצה. פעולת JG, שמיועדת למספרים signed, צריכה להחזיר תשובה שלילית (מינוס 127 אינו גדול מ־1). התנאים שייבדקו הם אם SF=OF או ש־ZF שווה לאפס. התוצאה תהיה שלילית ולכן לא תתבצע קפיצה.

תרגיל 8.3: הוראות בקרה (השוואה וקפיצה)

הערה: אלא אם נכתב אחרת, הכוונה למשתנים בגודל בית.

א. כיתבו תוכנית שבודקת אם ax גדול מאפס (יש להתייחס לערך של ax בייצוג שלו כ-signed כמובן), ואם כן מורידה את ערכו באחד.

ב. כיתבו תוכנית שבודקת אם ax=bx, ואם לא – מעתיקה את bx לתוך ax.

ג. כיתבו תוכנית שבודקת אם המשתנה Var1 גדול מ-Var2 (שימו לב – יש להתייחס אל הערכים במשתנים כ-unsigned). אם כן- ax=1, אחרת ax=0.

ד. כיתבו תוכנית שמבצעת את הפעולה הבאה: מוגדרים שני משתנים בגודל בית, var1 ו-var2. אם שני המשתנים שווים – 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 כמות 'x'ים כערכו של TimesToPrintX. הדרכה: החזיקו ברגיסטר כלשהו את כמות הפעמים ש־'x' כבר הודפס למסך. צרו label שמדפיס x למסך ומקדם את הרגיסטר ב־1. לאחר מכן בצעו השוואה בין הרגיסטר ל־TimesToPrintX, ואם לא מתקיים שוויון – קפצו ל־label.

מה יקרה אם TimesToPrintX יאותחל להיות מספר שלילי? או אפס? טפלו במקרים אלו.

פקודת LOOP

לעיתים קרובות אנחנו צריכים לבצע פעולה כלשהי מספר מוגדר של פעמים. לדוגמה, בתרגיל האחרון היינו צריכים להדפיס למסך 'x' כמות של פעמים שנמצאת במשתנה TimesToPrintX. כדי לבצע את הפעולה הזו, נדרשתם להחזיק ברגיסטר כלשהו את כמות הפעמים ש־'x' כבר הודפס למסך, להשוות את כמות הפעמים הזו ל־TimesToPrintX, ואם לא מתקיים שוויון – לקפוץ אחורה למיקום בו מודפס עוד 'x' למסך. פעולה זו – של ביצוע פעולה מוגדרת מספר פעמים – נקראת **לולאה** (**Loop**). כיוון שפעולה זו נפוצה למדי, ישנה פקודה מיוחדת שעושה בשבילנו 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 שהגדרנו.

הפקודה הזו:

```
loop    SomeLabel
```

זהה לקוד הבא:

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

דוגמה לשימוש בפקודת loop לביצוע התכנית שמדפיסה 'x' 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 קצה

נקודה למחשבה: העתיקו את הקוד שלמעלה והריצו אותו, עבור ערך חיובי כלשהו של TimesToPrintX. האם אתם מזהים מקרה כלשהו, שבו התכנית לא תבצע בדיוק את מה שרצינו?

פתרון:

אתחלו את TimesToPrintX לאפס.

כמו שאמרנו, הפקודה loop קודם כל מפחיתה את ערכו של cx ואז משווה אותו לאפס. לכן, אם לפני פקודת ה־loop הערך של cx היה אפס, לפני ההשוואה עם אפס בפעם הראשונה, ערכו של cx הוא כבר 65,535 (הייצוג של מינוס 1 כ-unsigned). בפעם הבאה cx כבר יהיה שווה 65,534 וכך הלאה – לאחר 65,536 פעמים ערכו של cx יתאפס ורק אז תנאי העצירה יתקיים.

כדי שהתוכנית תעבוד בכל מקרה, צריך להוסיף לה בדיקה לפני הכניסה ללולאה (מודגש בצבע):

```
xor     cx, cx
mov     cl, TimesToPrintX       ; we use cl, not cx, since TimesToPrintX is byte long
cmp     cx, 0
je      ExitLoop
PrintX:
        ...                     ; Some code for printing 'x'
        Loop    PrintX
ExitLoop:
        …
```

לולאה בתוך לולאה Nested Loops (הרחבה)

לעיתים נרצה לבצע לולאה בתוך לולאה – קוד שרץ מספר מוגדר של פעמים, ובתוכו רץ 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 שהיא בתוך LoopA לרוץ 5 פעמים על LoopB. בסך הכל, LoopB צריך להיות מבוצע 50 פעמים. מה לא תקין בקוד הזה?

הבעיה היא ששתי הוראות ה־loop משנות את ערכו של cx. בסיום ריצת LoopB ערכו של cx הוא אפס. הקוד מגיע לסוף LoopA ומוריד 1 מערכו של cx, מה שגורם ל־LoopA לרוץ 65,536 פעמים. אך בכל ריצה כזו cx מתאפס... אי לכך, התכנית לעולם לא תיגמר.

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

```
        mov     bx, 10
LoopA:
        mov     cx, 5
```

```
LoopB:
        …       ; Some code for LoopB
        loop    LoopB
        …       ; Some code for LoopA
        dec     bx
        cmp     bx, 0
        jne     LoopA
```

תרגיל 8.4: לולאות

א. סידרת פיבונצ'י: סדרת פיבונצ'י מוגדרת באופן הבא – האיבר הראשון הוא 0, האיבר השני הוא 1, כל איבר הוא סכום שני האיברים שקדמו לו (לכן האיבר השלישי הוא 0+1=1, האיבר הרביעי הוא 1+1=2 וכו'). צרו תוכנית שמחשבת את עשרת המספרים הראשונים בסדרת פיבונצ'י ושומרת אותם במערך בעל 10 בתים. בסיום התוכנית המערך צריך להכיל את הערכים הבאים:

0,1,1,2,3,5,8,13,21,34

ב. שורות הקוד הבאות קולטות תו מהמשתמש:

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

צרו תוכנית שקולטת 5 תווים מהמשתמש ושומרת אותם לתוך מערך. בידקו את התוכנית על־ידי הכנסת התווים HELLO ובדיקה שהמערך מכיל תווים אלו.

ג. צרו תוכנית שמחשבת את המכפלה Var1*Var2, משתנים בגודל בית שיש להתייחס אליהם כ־unsigned, אך מבצעת זאת על־ידי פעולת חיבור. הדרכה: יש לבצע פעולת חיבור של sum=Var1+sum ולחזור עליה על־ידי loop Var2 פעמים.

ד. שאלת אתגר: צרו תוכנית שמקבלת שני מספרים מהמשתמש ומדפיסה 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
```

הדרכה:

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 התו 'x':

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

מעבר שורה – הפקודות הבאות גורמות להדפסה של מעבר שורה:

```
mov     dl, 0ah
mov     ah, 2h
int     21h
```

קפיצה מחוץ לתחום

סקרנו מספר פקודות שמקפיצות אותנו למקום אחר בקוד:

- פקודת jmp
- פקודות קפיצה מותנית כגון ja, jg, jb וכו'
- פקודת loop

במקרים מסויימים, שימוש בפקודת קפיצה מותנית או בפקודת loop עלול לגרום לשגיאה הבאה בזמן ביצוע האסמבלר: Error- Relative jump out of range. כעת נבין את מקור השגיאה ואיך ניתן לעקוף את הבעיה.

האסמבלר מתרגם כל פקודה ל-opcode. בעוד שלפקודת jmp האסמבלר מקצה 16 ביט לייצוג טווח הקפיצה (כלומר טווח של מינוס 32768 עד פלוס 32767 בתים מהמיקום הנוכחי), האסמבלר מקצה רק 8 ביט לייצוג טווח הקפיצה של פקודות קפיצה מותנית ושל loop (כלומר טווח של 127 בתים קדימה עד 128 בתים אחורה בזיכרון). לכן, אם ננסה לכתוב פקודת קפיצה מותנית לתווית שנמצאת במרחק גדול מהטווח שניתן לכתוב ב-8 ביט, נקבל שגיאת relative jump out of range.

ישנן מספר דרכים להתמודד עם בעיה זו- כולן סוגים של מעקפים:

1. ההתמודדות הכי פשוטה היא לא לייצר קפיצות מותנות שחורגות מהטווח וכך להימנע מהבעיה.
2. שיטה פשוטה אך לא אלגנטית- לקפוץ למקום קרוב וממנו מיד לקפוץ למיקום הבא, עד שמגיעים למקום הנכון בקוד.
3. להמיר את פקודות הקפיצה המותנית שגורמות לשגיאה, בפקודת קפיצה בלתי מותנית.

נדגים את השימוש בהמרת קפיצה מותנית בקפיצה בלתי מותנית. כתבנו את הקוד הבא, נניח שפקודת הקפיצה ja גרמה לשגיאה:

```
cmp ax, bx
ja my_label
…. ; more than 127 bytes in code memory
my_label:
…. ; some code here
```

כעת נבצע המרה 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 מנגנון שמחליף את הקפיצה המותנית בקפיצה בלתי מותנית. בואו נראה כיצד מתנהג הקוד החדש שכתבנו. בקוד המקורי התרחשה קפיצה ל-my_label אם התקיים התנאי bx>ax (הפקודה ja משמעה "קפוץ אם גדול ממש", והיא הפקודה ההפוכה ל- jbe, שמשמעה "קפוץ אם קטן או שווה"). בקוד החדש, כאשר bx>ax התנאי jbe אינו מתקיים, כתוצאה מכך התוכנית מגיעה לשורת הקוד jmp my_label והקפיצה מתקיימת.

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 סוגים שונים של קפיצות מותנות ("קפוץ אם ערך גדול מ...", "קפוץ אם ערך קטן או שווה...", "קפוץ אם ערך שווה אפס...").
- עמדנו על ההבדלים בין השוואה של מספרים signed לבין השוואת מספרים unsigned וראינו שישנן פקודות שונות לאלה ולאלה.
- ראינו איך מגדירים לולאות תוך שימוש בפקודת loop וברגיסטר cx (או cl).

בסוף הפרק הגענו לרמה שמאפשרת לנו לתכנת תוכניות שכוללות אלגוריתמים, תנאים לוגיים וחזרות על קטעי קוד. בשלב זה הקוד שאנו יוצרים מתאים לתוכניות קצרות, ועדיין לא לביצוע תוכניות ארוכות ומורכבות. חסרה לנו עדיין היכולת לכתוב קטעי קוד שמקבלים פרמטרים ומבצעים פעולות שונות – פרוצדורות. על כך נלמד בפרק הבא.

פרק 9 – מחסנית ופרוצדורות

מבוא

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

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

בשפת אסמבלי אין הבדל מעשי בין פרוצדורה לפונקציה ולכן נתייחס לכול בתור פרוצדורות. כשאנחנו כותבים קוד מודולרי – קוד שבנוי מפרוצדורות עם קוד שמקשר ביניהן – אנחנו מתכננים מראש איך לחלק את מה שהתוכנית שלנו צריכה לעשות לכמה פרוצדורות, כאשר לכל פרוצדורה יש תפקיד מוגדר. לעבודה בשיטה הזו יש כמה יתרונות:

יתרון ראשון – הקוד שלנו קצר יותר. נניח שחלק מהתוכנית שלנו דורש שנקלוט סיסמה מהמשתמש, ואנחנו צריכים לעשות את הפעולה הזו לעיתים קרובות. זו לא פעולה מסובכת, אבל עדיין – צריך להגדיר לולאה שרצה כמה פעמים, להפעיל בכל פעם פקודה של קליטת תו ולדאוג לשמור את התווים שנקלטו. אם נגדיר פרוצדורה שיודעת לקלוט תווים מהמשתמש, במקום לכתוב את כל הקוד של קליטת התווים, נוכל פשוט לקרוא לפרוצדורה שלנו. בסוף הפרוצדורה התוכנית תקפוץ חזרה למיקום האחרון שהיא הייתה בו לפני הקריאה. כך חסכנו את כתיבה חוזרת של קוד קריאת התווים.

רגע, למה שלא פשוט נגדיר label, נקרא לו, לדוגמה – ReadPassword, ובכל פעם שנצטרך לקלוט סיסמה נעשה אליו jmp? אנחנו בהחלט יכולים לעשות את זה, וזה גם יעבוד, אבל לאן נמשיך אחרי קריאת הסיסמה? אם לא ניתן לתכנית הוראה אחרת, היא פשוט תמשיך אל שורת הקוד הבאה. אולי נפתור את זה באמצעות פקודת jmp בסוף קריאת הסיסמה? זה יעבוד. אבל רגע – מה אם יש יותר ממקום אחד בקוד שדורש להקליד סיסמה? דבר זה בעייתי, שכן בכל פעם נרצה לקרוא לקוד שיקרא את הסיסמה ולהמשיך במקום אחר בתכנית. נגיד שאנחנו צריכים לכתוב קוד שקולט סיסמה לגישה למחשב, ואחר כך סיסמה לחשבון המייל. אחרי קריאת הסיסמה, איך נדע לאן לקפוץ?

המחשת הבעיה של שימוש ב־label שאפשר לקרוא לו מיותר ממקום 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 לבעיה זו יש פתרון – אנחנו יכולים להגדיר משתנה שקובע לאן צריך לקפוץ ולבדוק אותו על־ידי פקודת cmp – אבל כל הבדיקות האלה מתחילות להיות יותר ארוכות מאשר לכתוב את הקוד של קליטת הסיסמה בכל המקומות שאנחנו צריכים אותו... פרוצודורות מהוות פיתרון אלגנטי ויעיל לסוגיה הזו.

יתרון שני – אנחנו יכולים לבדוק חלקים מהקוד שלנו ולדעת שהם עובדים בצורה טובה. נגיד שהגדרנו פרוצדורה ReadPassword. כתבנו אותה, בדקנו שהיא עובדת כמו שאנחנו רוצים. זהו. לא צריך לגעת בה יותר. אם יש לנו באג בתוכנית, אז נחפש אותו במקומות אחרים. אם במקרה מצאנו באג בפרוצדורה ReadPassword, מספיק שנתקן אותו פעם אחת – והוא יתוקן עבור כל הקריאות בתוכנית. העובדה שיש לנו אוסף של קטעי קוד בדוקים מקטינה משמעותית את המאמץ שלנו לגלות באגים.

יתרון שלישי – קל יותר לקרוא את הקוד שלנו. במקום לכתוב הרבה קוד, אנחנו פשוט כותבים פקודה שנקראת call ואת שם הפרוצדורה. כך, לדוגמה:

```
call    ReadPassword
```

יתרון רביעי – יכולת לשתף קוד גם בין תוכניות וגם בין אנשים. נניח שכתבנו קטע קוד שמבצע משהו שימושי ונפוץ, לדוגמה, קליטת סיסמה מהמשתמש. אם נשים את הקוד בתוך פרוצדורה ואת הפרוצדורה נשמור בקובץ נפרד, נוכל להשתמש בה בכל תוכנית שנכתוב וגם לשתף בה אחרים. העיקרון שלא צריך "להמציא את הגלגל מחדש" בכל פעם שאנחנו רוצים לתכנת משהו, עומד בבסיס התוכנות מרובות הקוד שקיימות היום.

מושלם, לא? ובכן, עכשיו נראה קצת חסרונות של פרוצדורות.

חיסרון ראשון – היתרון של שיתוף קוד הוא פתח לבעיות. אם יש באג בקוד שמישהו כתב 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 כמות גדולה של משתמשים, שנדרשים לעדכן את התוכנה שברשותם.

חיסרון שני – כתיבת פרוצדורה דורשת השקעה מסוימת מצד המתכנת. יותר קל פשוט לכתוב בתוכנית הראשית את הקוד מאשר להגדיר אותו בתוך פרוצדורה, במיוחד בתור מתכנתים מתחילים.

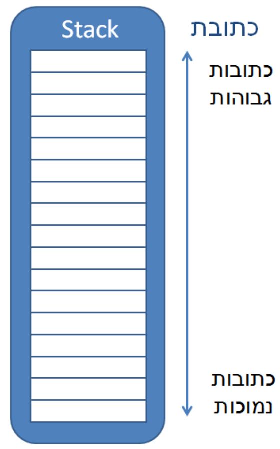
חיסרון שלישי – הקריאה לפרוצדורות והחזרה מהן דורשת משאבים של המחשב. לדוגמה, ביצוע פעולות קפיצה וחזרה. לדוגמה, שימוש בזיכרון. מכאן שקריאה לקוד בתוך פרוצדורה היא "יקרה" יותר עבור המחשב מאשר הרצה של הקוד שלא מתוך פרוצדורה.

חרף החסרונות שסקרנו, התועלת שבכתיבת קוד מודולרי, שמבוסס על פרוצדורות, עולה על החסרונות שלו. לכן נקפיד לכתוב את הקוד שלנו בצורה זו.

המחסנית STACK

המחסנית (STACK), היא סגמנט בזיכרון. המחסנית משמשת לאחסון לזמן קצר של משתנים, קבועים וכתובות ולכן היא כלי חשוב בעבודה עם פרוצדורות ויצירת תוכניות מודולריות.

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



תיאור סכמטי של ה־Stack Segment

הגדרת מחסנית

כמו כל סגמנט בזיכרון, המחסנית היא אזור בזיכרון שמתחיל בכתובת כלשהי ותופס גודל מוגדר של זיכרון. גודל האזור בזיכרון שמוקצה למחסנית נקבע על־ידי המתכנת בתחילת התוכנית. הקצאת המקום נעשית בדרך הבאה:

```
STACK          number of bytes
```

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

```
STACK          100h
```

למחסנית יש שני רגיסטרים שקשורים אליה. עשינו איתם היכרות קצרה בעבר, כשסקרנו את כלל הרגיסטרים שיש למעבד:

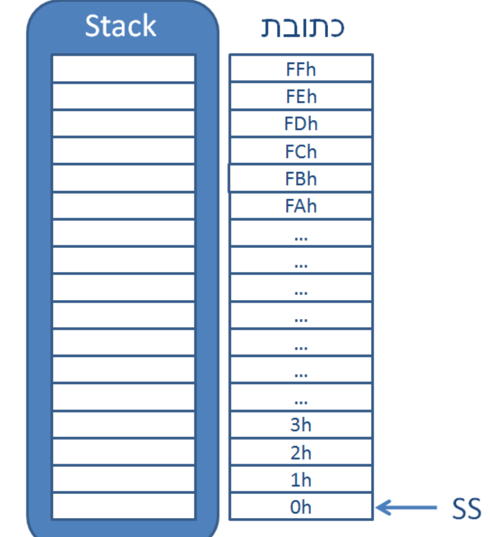
- ss – stack segment הוא סגמנט המחסנית. מצביע על הכתובת הנמוכה ביותר במחסנית.

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

מצביע המחסנית, sp, הוא רגיסטר שמחזיק אופסט בזיכרון. בכך אין הוא שונה מהרגיסטר bx. כמו שאפשר להגיע לכל כתובת ב־DATASEG בעזרת הצירוף ds:bx, כך אפשר להגיע לכל כתובת ב־STACK בעזרת הוספת האופסט הנתון ב-sp לכתובת הסגמנט הנתונה ב-ss.

בתחילת התוכנית, הרגיסטר sp שווה לגודל המחסנית. בדוגמה שנשתמש בה, מחסנית בגודל 100h, לפני שהכנסנו ערך כלשהו למחסנית sp מאותחל להיות שווה 100h.

שימו לב שבמחסנית בגודל 100h, כלומר 256 בתים, מרחב הכתובות אינו מגיע עד 100h אלא הוא בין 0 ל־0FFh. כלומר, הערך ההתחלתי של sp, 100h, מצביע על כתובת שהיא בדיוק בית אחד מעל קצה המחסנית.



מחסנית בגודל 100h. הרגיסטר ss מצביע על תחילת סגמנט המחסנית. בתחילת התוכנית הרגיסטר sp מאותחל לערך 100h, כלומר לערך שנמצא מחוץ למחסנית.

העובדה ש־sp מאותחל להצביע לא על תחילת המחסנית אלא על הקצה שלה נראית כרגע קצת מוזרה, אבל היא קשורה לדרך שבה המחסנית מנוהלת. המחסנית מנוהלת בשיטת LIFO – Last In First Out, כלומר הערך שנכנס אחרון הוא הראשון לצאת מהמחסנית. לפני הכנסה של נתונים למחסנית, ערכו של sp <u>יורד</u> ולאחר הוצאה של נתונים מהמחסנית ערכו של sp <u>עולה</u>. נבין זאת כאשר נלמד על הפקודות המשמשות אותנו בעת הכנסה והוצאה של נתונים מהמחסנית, ונראה דוגמאות.

אילו פקודות מאפשרות הכנסה והוצאה של נתונים מהמחסנית?

פקודת PUSH

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

```
push    operand
```

מה יבוצע?

- ערכו של sp יירד בשתיים: sp=sp-2.
- ערכו של האופרנד יועתק למחסנית, לכתובת ss:sp.

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

דוגמאות לשימוש בפקודת push:

```
push    ax
push    10
push    var
```

הפקודה הראשונה תדחוף למחסנית את ax.

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

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

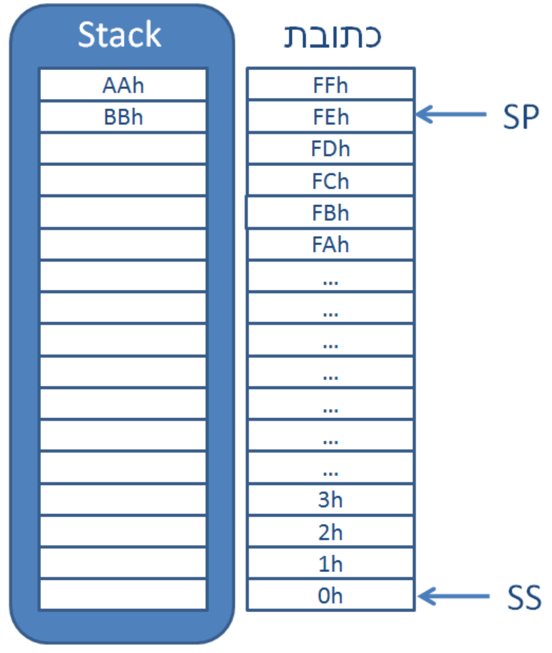
האם הפקודה הבאה היא חוקית?

```
push    al
```

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

נחזור למחסנית שהגדרנו, בגודל 100h. מה יהיה מצב המחסנית לאחר ביצוע הפקודות הבאות?

```
mov     ax, 0AABBh
push    ax
```



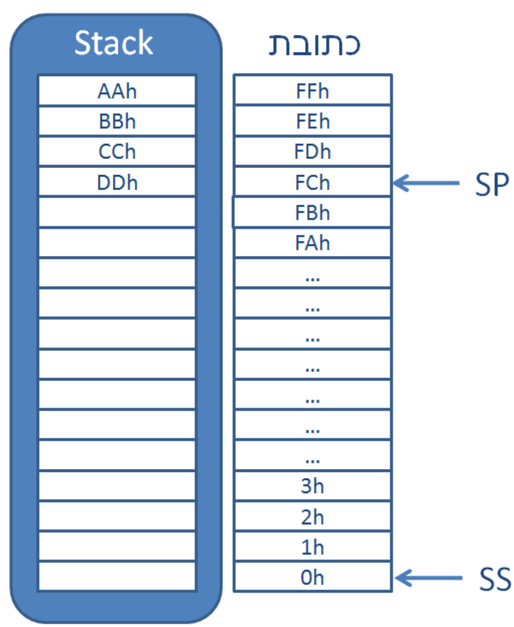
פקודת push גורמת לירידת ערכו של sp ב־2

והוא מצביע על כתובת נמוכה יותר במחסנית

שימו לב לכך שהביטים הגבוהים של ax, אלו השמורים ב־ah, נדחפו למחסנית ראשונים בכתובת גבוהה יותר.

אנחנו יכולים להמשיך לדחוף ערכים למחסנית לפי הצורך. כל דחיפה כזו תוריד עוד 2 מערכו של sp. לדוגמה:

```
push    0CCDDh
```



פקודת POP

פקודת pop היא הפקודה ההפוכה ל-push. פקודה זו גורמת להוצאה של מילה (שני בתים) מהמחסנית והעתקה שלה לאופרנד:

```
pop     operand
```

מה יבוצע?

- מילה מראש המחסנית תועתק לאופרנד.
- ערכו של sp יעלה ב-2.

דוגמאות לשימוש בפקודת pop:

```
pop     ax
pop     [var]
```

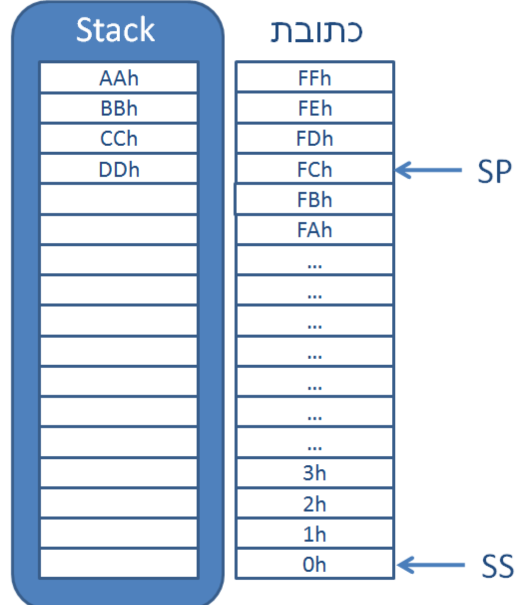
הפקודה הראשונה תעתיק לתוך ax את המילה שבראש המחסנית.

הפקודה השניה תעתיק לתוך המשתנה var את המילה שבראש המחסנית (כמובן שכדי לא לקבל שגיאת קומפילציה, var צריך להיות מטיפוס מילה).

```
pop     al
```

זוהי כצפוי 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 0CCDDh, ערכו של sp יעלה בשתיים ויהיה שווה 0FEh.

שאלה: לאחר ביצוע הפקודה הקודמת, בוצעה פקודת ה־pop הבאה:

```
pop     var
```

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

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

מה קרה לערכים שבתוך המחסנית? האם הם נמחקו?

לא! הערכים שהכנסנו למחסנית עדיין קיימים בתוכה, אך כעת אין למעבד דרך לגשת אליהם כיוון ש־sp אינו מצביע עליהם יותר. תיאורטית אנחנו עדיין יכולים לגשת אליהם אם היינו מבצעים את הפקודה הבאה:

```
sub     sp, 4
```

אך מעשית לא מקובל "לשחק" עם ערכו של sp, מסיבות שנעמוד עליהן בהמשך.

תרגיל 9.1: מחסנית, push ו־pop

א. הגדירו מחסנית בגדלים שונים. 10h, 20h. 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 שבמחסנית ומיצאו את הערך שדחפתם.

ג. בצעו pop למחסנית לתוך ax. איך השתנה sp? הסתכלו על הזיכרון שבמחסנית – האם הערך 1234h נמחק?

ד. בצעו push לערך 5678h. האם עכשיו נמחק הערך 1234h?

ה. העתיקו את ax לתוך bx בעזרת המחסנית ללא שימוש בפקודת mov.

פרוצדורות

הגדרה של פרוצדורה

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

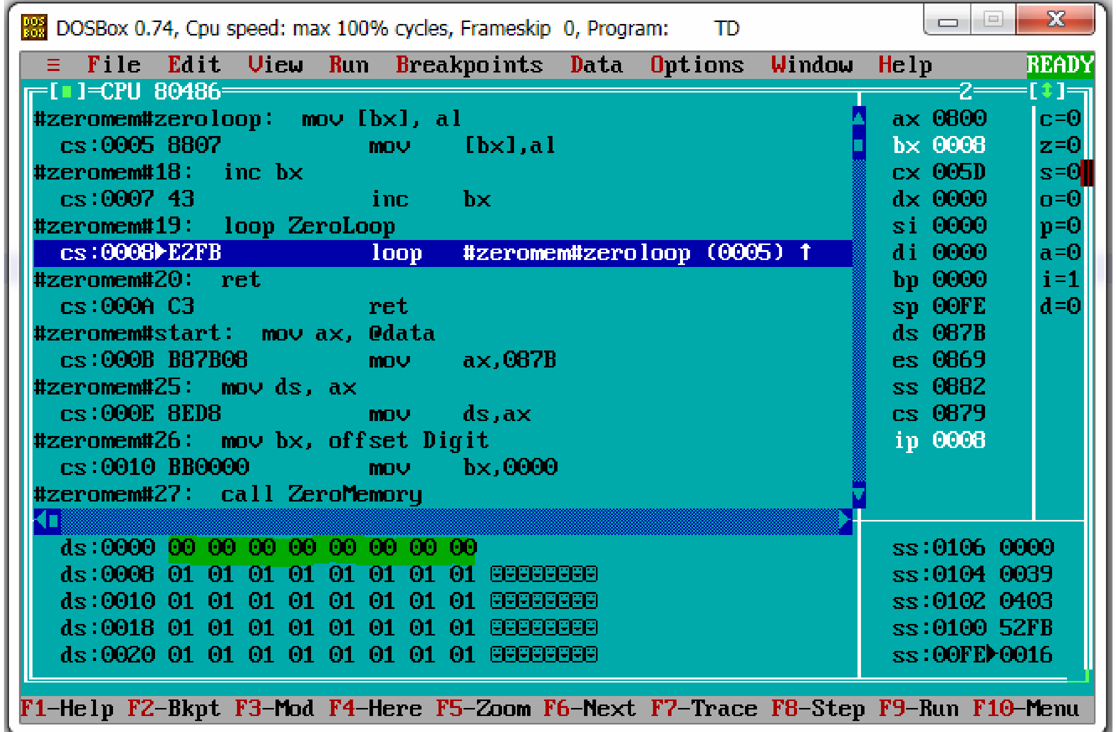
- קוראים לפרוצדורה באמצעות הפקודה call.
- אפשר להעביר פרמטרים לפרוצדורה. לדוגמה, פרוצדורה שמחברת שני מספרים ומחזירה את סכומם – אפשר להעביר לה כפרמטרים שני משתנים – num1, num2.
- לפרוצדורה יש מנגנונים להחזרת תוצאות העיבוד. כלומר, num1+num2 לא רק יחושב, אלא גם יועבר חזרה לתוכנית שזימנה את הפרוצדורה.
-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

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

```
proc        ProcedureName
    ...             ;Code for something that the procedure does
    ret             ; Return to the code that called the procedure
endp        ProcedureName
```

דוגמה לפרוצדורה– להלן תוכנית שכוללת פרוצדורה בשם ZeroMemory, פרוצדורה שמאפסת 10 בתים מתחילת ה־DATASEG (כלומר, הופכת את ערכם ל־0). שימו לב למיקום הפרוצדורה בתוך CODESEG, להגדרת הפרוצדורה ולקריאה לפרוצדורה:

```
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
```



DATASEG לאחר שהפרוצדורה הספיקה לאפס 8 בתים ראשונים (מודגש בצהוב)

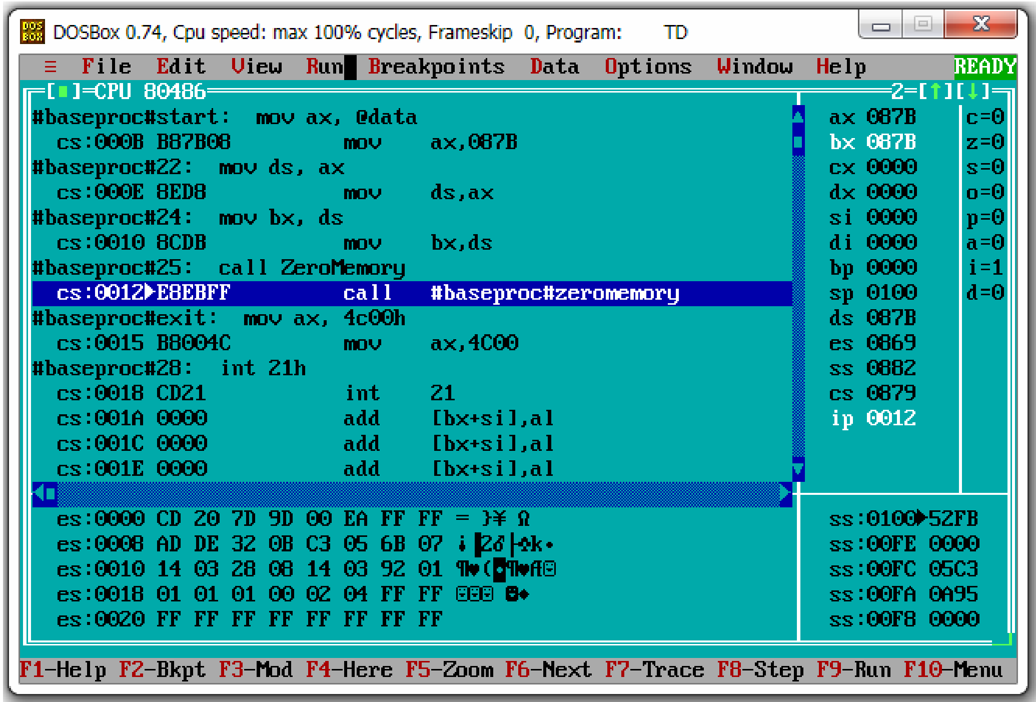
העתיקו את קוד הדוגמה של ZeroMemory. הריצו אותו ב־TD שורה אחרי שורה וצפו בשינויים שמתרחשים ב־ip וב־sp בשלבי התוכנית השונים

פקודות CALL, RET

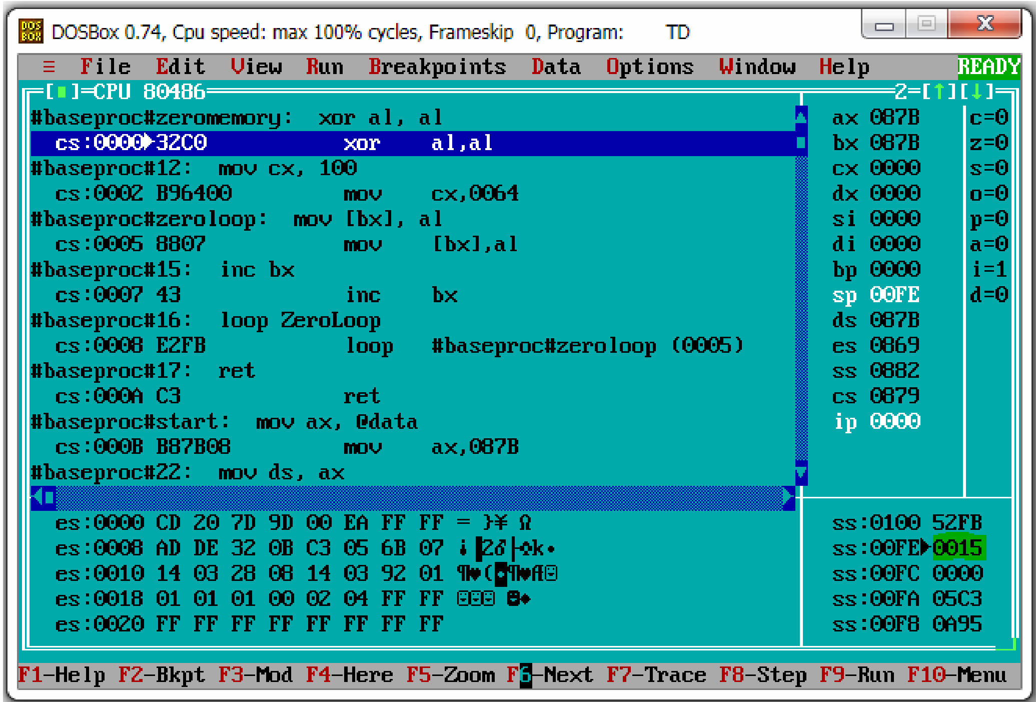
בדוגמה ראינו שפקודת call משמשת לזימון הפרוצדורה. נראה בדיוק מה מבצעת הפקודה:

```
call    ZeroMemory
```

לפני פקודת ה־call לפרוצדורה, הרגיסטר ip=12, כלומר מצביע על שורה 12h ב־CODESEG, שהיא השורה שאחרי פקודת ה־call (בדוגמה שלנו: ...).



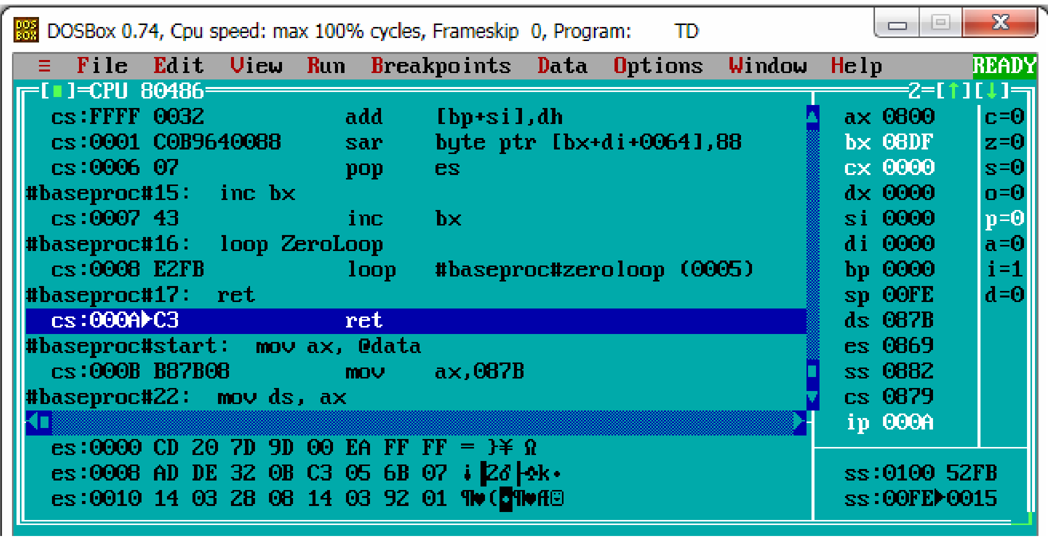
שימו לב לשינוי בערכו של ip מיד לאחר ביצוע הוראת ה־call:



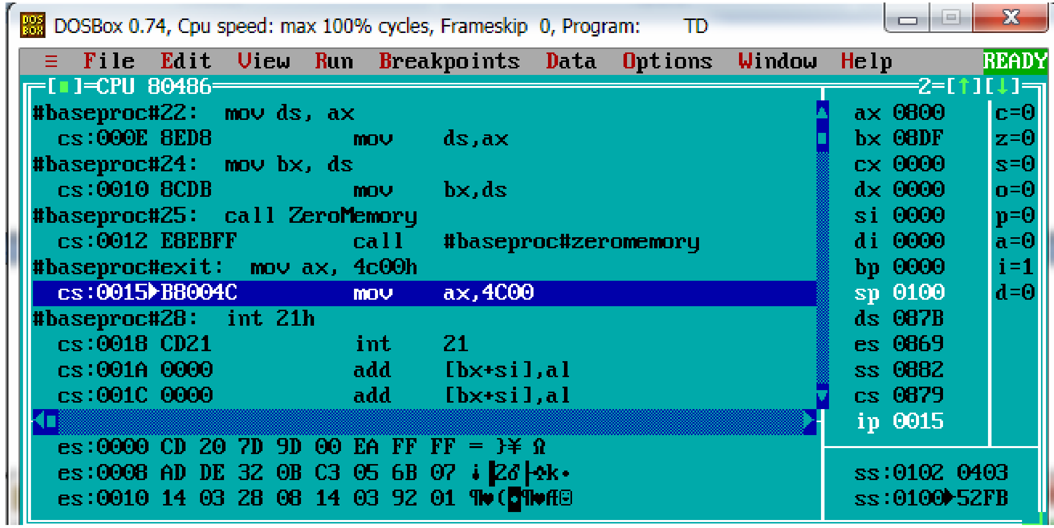
ערכו של ip השתנה ל-0000. זהו המיקום של הפרוצדורה ZeroMemory בזיכרון (זכרו, שמדובר ב־offset בתוך CODESEG, כלומר הפרוצדורה נמצאת ממש בתחילת של ה־CODESEG). כלומר, פקודת call משנה את ערכו של ip כך שהתכנית קופצת אל תחילת הפרוצדורה. אבל את הפעולה הזו אפשר היה להשיג גם באמצעות פקודת jmp. פקודת call עושה משהו נוסף. 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. שימו לב לעוד נתון שהשתנה. בחלק הימני התחתון

של המסך, יש ערכים שונים בסגמנט המחסנית ss. הערך בכתובת ss:00FEh (מודגש בצהוב) הוא 0015h. לפני פקודת ה־call ערכו היה 0000h. מה משמעותו של ערך זה? מיד נראה.

הפרוצדורה כמעט סיימה את ריצתה – היא איפסה עשרה בתים בזיכרון והגיעה לפקודה ret. בנקודה זו הרגיסטר ip=0Ah:



הגיע הזמן לחזור לתוכנית הראשית – לאחר ביצוע פקודת ה־ret, הרגיסטר ip מצביע על שורת הקוד הבאה מיד אחרי שורת הקוד שקראה לפרוצדורה והתכנית ממשיכה ממנה באופן טורי:



התוכנית חזרה מהפרוצדורה, ip קפץ מ־0Ah ל־15h.

... 15h? נכון, זה בדיוק הערך שנכנס למחסנית עם ביצוע פקודת ה־call.

נסכם מה ראינו שפקודות ה־call וה־ret מבצעות:

פקודת call –

1. מורידה את ערכו של sp בשתיים (יש מקרה שבו היא מורידה את ערכו של sp בארבע – כאשר מדובר בפרוצדורת FAR – נראה מקרה זה בהמשך).
2. מעתיקה לתוך הכתובת ss:sp, את הכתובת בזיכרון אליה יש לחזור בסיום ריצת הפרוצדורה.
3. מעתיקה לתוך ip את הכתובת של הפקודה הראשונה של הפרוצדורה (פעולה זו שקולה לביצוע jump אל תחילת הפרוצדורה)

פקודת ret -

1. קוראת מהכתובת ss:sp, את הכתובת בזיכרון אליה יש לחזור.
2. מעלה את ערכו של sp בשתיים (כך במקרה זה; נראה מקרים אחרים בהמשך).
3. משנה את ה־ip אל הכתובת שנקראה מ-ss:sp, ובכך מחזירה את התוכנית לשורת הקוד שבאה מיד אחרי הקריאה לפרוצדורה.

כמו שראינו, זו לא מקריות שבסיום ריצת הפרוצדורה, הערך של ip חזר אל השורה הבאה בתוכנית. פקודות ה־call וה־ret דאגו לשמור את ערכו של ip במחסנית ולשחזר אותו בסיום ריצת הפרוצדורה!

פרוצדורת NEAR, FAR

התוכנית שלנו רצה מתוך ה־CODESEG. בשלב מסויים היא מגיעה לפקודת call אל פרוצדורה שהגדרנו. הקוד עצמו של הפרוצדורה יכול להימצא באחד משני מקומות:

1. בתוך ה־CODESEG, יחד עם שאר הקוד של התוכנית הראשית.
2. מחוץ ל־CODESEG (בסגמנט אחר כלשהו שהגדרנו).

באופן מעשי, כיוון שאנחנו מגדירים מודל זיכרון model small, בעל סגמנט קוד יחיד, כל הפרוצדורות שנגדיר יהיו בתוך ה־CODESEG, אבל יש היגיון להבין גם מה קורה במקרה השני – אחרי הכל, תוכניות יכולות לקבל גם ספריות של פרוצדורות מקומפלות, שאינן נמצאות ב־CODESEG.

אם הפרוצדורה נמצאת בתוך ה־CODESEG, היא נקראת פרוצדורת near. הפרוצדורה ZeroMemory היא דוגמה לפרוצדורה מסוג near. במקרה כזה, פקודת call מכניסה לתוך המחסנית רק את האופסט של כתובת החזרה – אחרי הכל, הסגמנט כבר ידוע – CODESEG. כיוון שהאופסט הוא בגודל שני בתים, הרגיסטר sp משתנה ב־2.

לעומת זאת, אם הפרוצדורה נמצאת מחוץ ל־CODESEG, היא נקראת פרוצדורת far. במקרה כזה, פקודת call מכניסה למחסנית גם את האופסט וגם את הסגמנט של כתובת החזרה, כדי שהמעבד יוכל לחשב בדיוק לאן לחזור. האופסט והסגמנט תופסים ביחד ארבעה בתים, ולכן במקרה זה ערכו של הרגיסטר sp משתנה ב־4.

איך אנחנו יכולים לשלוט בביצוע פקודת call לפרוצדורת near או far?

אנחנו מודיעים לאסמבלר מהו סוג הפרוצדורה בזמן הגדרת הפרוצדורה, על־ידי הוספת המילים near או far. לדוגמה:

```
proc    ProcedureName       near
proc    ProcedureName       far
```

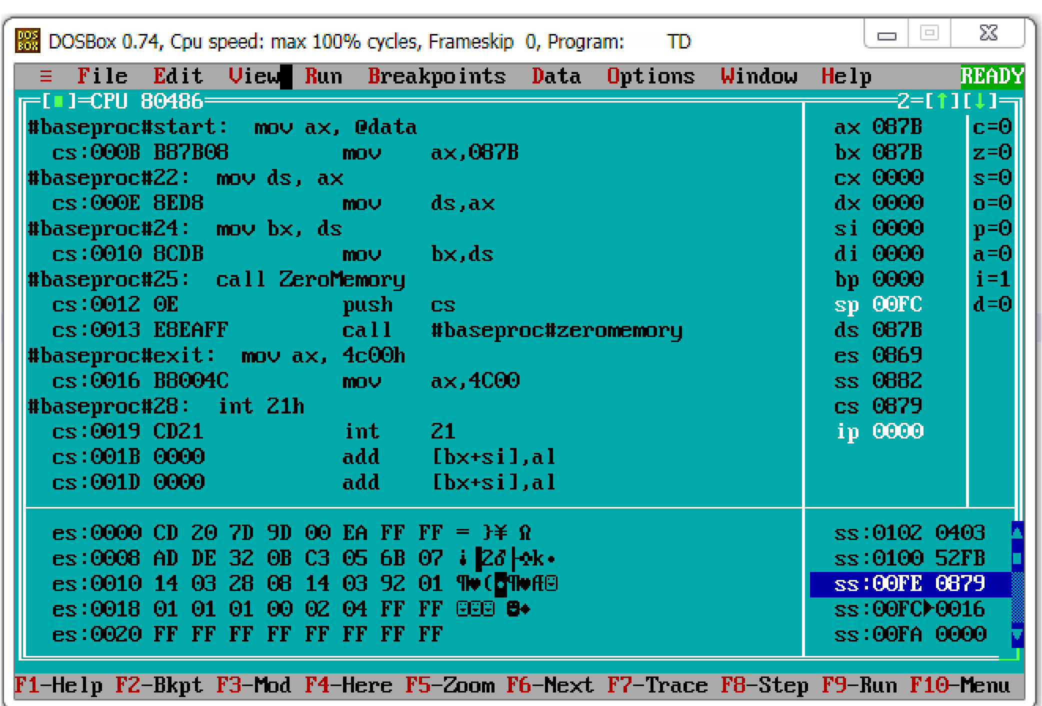
אם אנו לא כותבים לא near ולא far, ברירת המחדל היא near.

נכתוב גירסה חדשה של תוכנית הדוגמה שלנו. כל השינוי הוא הוספת מילה אחת לפרוצדורה ZeroMemory – "far":

```
proc    ZeroMemory          far
```

חוץ מזה, הגרסה החדשה זהה לגרסה הקודמת. להלן צילום מסך של הכניסה לפרוצדורה. שימו לב לדברים הבאים:

- במיקום cs:0012 בתוכנית, האסמבלר הוסיף שורת קוד: "push cs". שורת קוד זו גורמת להעתקת של רגיסטר הקוד cs לתוך המחסנית. כתוצאה מכך לתוך המחסנית נדחף הערך 0879h, שהוא ערכו של רגיסטר ה־cs (מודגש בכחול).
- לאחר מכן נדחף למחסנית הרגיסטר ip, ערכו 16h (ולא 15h כמו בגרסה הקודמת, משום שנוספה שורת הקוד "push cs").
- עם הכניסה לפרוצדורה, ערכו של sp הוא 00FCh (ולא 00FEh, כמו בגרסה הקודמת).



בכל הדוגמאות בהמשך הפרק, נניח שהפרוצדורה היא מסוג near. כלומר אנחנו משמיטים את ההכנסה של cs למחסנית. למעט הבדל זה, אין הבדל בין שימוש בפרוצדורות near ו־far.

שימוש במחסנית לשמירת מצב התוכנית

נתבונן בקוד הבא, שאמור להדפיס שלוש שורות, בכל שורה ארבעה תווי 'X'. בשלב זה נתעלם מהשורות הצבעוניות – 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. במקום זאת, היא לעולם לא תפסיק לרוץ. מה הגורם לבעיה? העתיקו את התוכנית, קמפלו אותה והריצו ב־TD. עיקבו אחרי הערך של הרגיסטר cx.

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

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

נראה איך בעזרת פקודות push ו־pop אפשר לשנות את הפרוצדורה Print10X כך שיתבצע בדיוק מה שאנחנו רוצים. הטכניקה היא פשוטה: בכניסה לפרוצדורה צריך לשמור את הרגיסטרים במחסנית, ביציאה מהפרוצדורה לשלוף את הערכים מהמחסנית ולהחזיר את הרגיסטרים למצבם טרם הפרוצדורה.

```
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 פקודת push ופקודת pop אחת (מודגשות בצהוב). כעת מה שיקרה, הוא שבתחילת הפרוצדורה יועתק הערך של cx למחסנית. רק אז ישונה ערכו של cx ל־4. לאחר שהלולאה PrintXLoop תסתיים, ערכו של cx יהיה שווה לאפס. פקודת ה־pop תעתיק לתוך cx את הערך שנשמר למחסנית בכניסה לפרוצדורה, 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 לשמירת 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 לפרוצדורה (במקומות בהם יש '???') כך שבסיום ריצת הפרוצדורה ערכי הרגיסטרים יישארו כפי שהיו טרם הקריאה לפרוצדורה.

העברת פרמטרים לפרוצדורה

לא תמיד נרצה שפרוצדורה תבצע את אותה הפעולה בכל פעם שאנחנו קוראים לה. ראינו דוגמה לפרוצדורה שמאפסת עשרה בתים בזיכרון. נניח שעכשיו אנחנו רוצים לאפס תשעה בתים בזיכרון – האם אנחנו צריכים עכשיו לכתוב פרוצדורה חדשה? ואם אחר כך נרצה לאפס שמונה בתים בזיכרון, שוב נצטרך לכתוב פרוצדורה חדשה? לא הגיוני לעבוד בצורה זו. למה שלא נבנה פרוצדורה שיודעת לקבל את מספר הבתים שעליה לאפס בתור נתון, וכדי לאפס מספר שונה של בתים פשוט נצטרך

להודיע לפרוצדורה כמה בתים עליה לאפס בזיכרון? לנתון הזה, שהפרוצדורה מקבלת מקוד שקורא לה (למשל התוכנית הראשית), קוראים פרמטר.

פרמטר יכול להיות כל דבר שאנחנו רוצים להעביר לפרוצדורה. לדוגמה, שני מספרים שפרוצדורה צריכה לחבר ביניהם, יכולים לעבור כפרמטרים. מה לגבי תוצאת החיבור? איך מחזירים אותה לתוכנית הראשית? גם את תוצאת החיבור, ובאופן כללי כל תוצאה שהפרוצדורה צריכה להחזיר, אנחנו יכולים להעביר בעזרת פרמטר.

יש יותר משיטה אחת להעביר פרמטרים לפרוצדורה, השיטות הן:

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

השיטה הראשונה, שימוש ברגיסטרים כלליים, היא פשוטה ביותר. הפרוצדורה ZeroMemory, לדוגמה, מקבלת בתוך bx כתובת בזיכרון שממנה עליה להתחיל את איפוס הזיכרון. באותה שיטה היינו יכולים להגדיר גם את כמות הבתים שעליה 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 שאנחנו צריכים לעשות בתוכנית הראשית הוא לדאוג ש־ax יחזיק את כמות הבתים שאנחנו רוצים לאפס.

הערה: אפשר כמובן לבצע את אותה פעולה על־ידי הכנסת הפרמטר ישירות לתוך cx, אך כדי להקל על ההסבר נעשה בדוגמה זו שימוש ב־ax.

למרות פשטות השימוש ברגיסטרים כלליים להעברת פרמטרים לפרוצדורה, לשיטה זו יש חסרונות. ראשית כל, מספר הרגיסטרים מוגבל. מתישהו ax, bx, cx, dx ייגמרו – ואז מה? שנית, מי שכותב את התוכנית הראשית צריך להכיר את הוראות הפרוצדורה ולדעת באילו רגיסטרים היא משתמשת בתור פרמטרים.

השיטה השניה, שימוש במשתנים שמוגדרים ב־DATASEG, פותרת את בעיית המספר המוגבל של רגיסטרים כלליים. מספר המשתנים שאנחנו יכולים להגדיר בזיכרון הוא כמעט בלתי מוגבל (באופן יחסי לצרכים המועטים שלנו בשלב זה). מדוע שלא נעביר פרמטרים לפרוצדורה 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, שלפני כן היה פרמטר שקבע את כמות הבתים שעל התוכנית לאפס. המשתנה MemoryStart מחליף את הרגיסטר bx, שלפני כן היה פרמטר שקבע את המיקום בזיכרון שאנחנו רוצים לאפס. הצלחנו לפנות שני רגיסטרים לטובת משימות אחרות (נכון, אנחנו עדיין משתמשים ב־al וב־bx בפרוצדורה, אבל כרגיסטרים כלליים לחישובי עזר ולא כפרמטרים, וזה אומר שהתוכנית הראשית כבר לא צריכה להיות עסוקה בקביעת הערכים הנכונים לרגיסטרים לפני הקריאה לפרוצדורה).

העברת פרמטרים על המחסנית

גם לשיטה הקודמת ישנם חסרונות – כותב התוכנית הראשית צריך להכיר את הוראות הפרוצדורה, לדעת שצריך להגדיר ב־DATASEG שני משתנים, ולא סתם שני משתנים אלא עם שמות קבועים שאי אפשר לשנות בלי לשנות את הפרוצדורה. התוכנית הופכת לקשה לשיתוף, ויותר מכך – מה אם נרצה להשתמש בשתי פרוצדורות שלקחנו ממקורות שונים, ושתיהן עושות שימוש במשתנים בעלי שמות זהים? ומה אם נרצה לכתוב פרוצדורה שקוראת לעצמה (רקורסיה)?

נלמד כעת איך להעביר פרמטרים לפרוצדורה על גבי המחסנית. זוהי השיטה המקובלת להעברת פרמטרים לפרוצדורות, משום שהיא פותרת את הבעיות שהצגנו בשיטות הקודמות: אין מגבלה מעשית של מקום במחסנית (אפשר להגדיר מחסנית יותר גדולה, ולא סביר שנצטרך יותר מ־64K, מגבלת גודל סגמנט), אין הגבלה על מספר הפרמטרים שאנחנו שולחים לפרוצדורה, אנחנו לא מוגבלים על־ידי כמות הרגיסטרים, ואחרון חביב – מי שקורא לפרוצדורה שלנו לא צריך לדעת איך אנחנו קוראים למשתנים בתוך הפרוצדורה.

ישנן שתי שיטות להעברת פרמטרים אל פרוצדורה:

1. העברה לפי ערך – Pass by Value
2. העברה לפי ייחוס – Pass by Reference

Pass by Value

בשיטה זו מועבר לפרוצדורה ערך הפרמטר. כלומר, על המחסנית נוצר העתק של הפרמטר. לפרוצדורה אין גישה לכתובת בזיכרון שמכיל את המשתנה המקורי, ולכן אין היא יכולה לשנות את ערכו – רק את ערכו של ההעתק. ביציאה מהפרוצדורה, ערכו של הפרמטר שנשלח לפרוצדורה יישאר ללא שינוי.

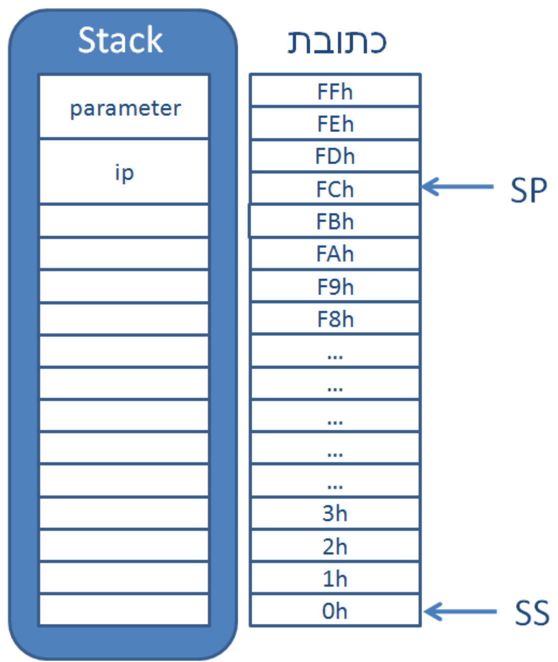
נמחיש זאת באמצעות דוגמה. נדמיין פרוצדורה בשם SimpleAdd, שמקבלת פרמטר אחד ומוסיפה לערכו 2. נניח שהפרוצדורה הזו היא חלק מספריה של פרוצדורות שמישהו כתב, קימפל ומסר לנו. הנתון הזה חשוב, כדי להדגיש שלאסמבלר שקימפל את SimpleAdd לא היה מושג מה הכתובות של המשתנים שאנחנו מגדירים ב־DATASEG של התוכנית שלנו. יכול להיות שהפרוצדורה SimpleAdd נכתבה בכלל לפני שאנחנו כתבנו את התוכנית שלנו.

בתוכנית שלנו מוגדר משתנה, נקרא לו parameter. כעת אנחנו רוצים ש־SimpleAdd תקבל את הערך שנמצא בתוך parameter.

כדי להעביר את ערכו של הפרמטר לפרוצדורה, התוכנית הראשית צריכה לדחוף אותו למחסנית ואז לקרוא לפרוצדורה:

```
push    [parameter]
call    SimpleAdd
```

מצב המחסנית עם הכניסה לפרוצדורה:

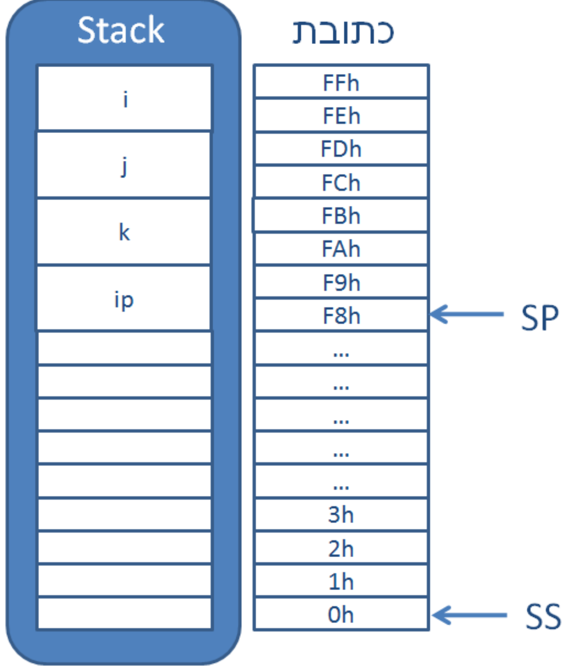


בתוך הפרוצדורה המשתנה parameter אינו מוכר. כל מה שהפרוצדורה מכירה זה את הערך שלו, שנמצא על המחסנית. הפרוצדורה לא יודעת ש־parameter נמצא בכתובת כלשהי ב־DATASEG. מבחינתה, היא מכירה רק את ההעתק שלו שנמצא בכתובת כלשהי על המחסנית. לאחר שהפרוצדורה תוסיף 2 לערכו של ההעתק, ערכו של parameter "המקורי" לא ישתנה כלל.

למרות שפרוצדורה שמקבלת ערכים בשיטת Pass by Value לא יכולה לשנות אותם, לפעמים שיטה זו מתאימה עבורינו. בדוגמה הבאה נראה איך פרוצדורה משתמשת בפרמטרים שהועברו אליה בשיטת Pass by Value. נניח שיש פרוצדורה בשם SimpleProc, שמקבלת שלושה פרמטרים: i, j ו–k ומחשבת בתוך ax את i+j-k. קוד אסמבלי מתאים לקריאה לפרוצדורה (בשיטת Pass by Value) יכול להיות:

```
push    [i]
push    [j]
push    [k]
call    SimpleProc
```

עם הכניסה לפרוצדורה, המחסנית תראה כך (בהנחה שלא הכנסנו למחסנית מידע נוסף):



מצב המחסנית לאחר דחיפת i, j, k

בתוך הפרוצדורה SimpleProc אנחנו יכולים להוציא את הפרמטרים שהועברו אליה בעזרת פקודת pop. יש רק עניין אחד – ראש המחסנית מצביע על כתובת החזרה מהפרוצדורה, ip, שנדחף למחסנית עם פקודת ה-call. כלומר ה-pop הראשון שנעשה יוציא את כתובת החזרה מהמחסנית.

אנחנו נראה שיטה פשוטה לטיפול בכתובת החזרה. זו **אינה שיטה שמשתמשים בה בפועל** ואנחנו מדגימים אותה בקצרה, רק כדי שנוכל להבין איך המחסנית עובדת. בהמשך נלמד שיטה שונה, שהיא השיטה המקובלת לפניה לפרמטרים שהועברו על המחסנית.

נשמור את הכתובת בצד ונדחוף אותה למחסנית לאחר ביצוע ה-pop לכל הפרמטרים שהעברנו לפרוצדורה.

```
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
```

ReturnAddress הוא משתנה כלשהו שהגדרנו ב־DATASEG. אם נרצה, אפשר להחליף אותו ברגיסטר. בשיטה זו הצלחנו לקרוא את הפרמטרים שהועברו במחסנית אל הפרוצדורה ועדיין לחזור אל המקום הנכון בתוכנית הראשית. כאמור- בקרוב נחליף את השיטה הזו בשיטה נכונה יותר.

תרגיל 9.4: Pass by Value

א. צרו פרוצדורה שמקבלת פרמטר אחד בשיטת pass by value ומדפיסה למסך מספר 'X'ים לפי הנתון בפרמטר. דגשים: יש לבדוק קודם לכן שערך הפרמטר חיובי! בסיום הריצה יש לשחזר את ערכי הרגיסטרים שנעשה בהם שימוש. עזרה – הדפסת תו למסך:

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

ב. צרו פרוצדורה שמקבלת שני פרמטרים 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 לשחזר את הערכים המקוריים של הרגיסטרים.

ג. צרו תוכנית שמוגדרים בה ארבעה מספרים קבועים בתחילת התוכנית, ומשתנים בשם max ו־min. צרו פרוצדורה שמקבלת כפרמטרים pass by value את ארבעת המספרים הקבועים ומכניסה למשתנה max את הערך המקסימלי מביניהם ולמשתנה min את הערך המינימלי מביניהם.

Pass by Reference

בשיטה זו מועברת לפרוצדורה הכתובת של הפרמטר בזיכרון. כלומר, על המחסנית לא נוצר העתק של הפרמטר אלא רק הכתובת שלו בזיכרון מועתקת למחסנית. הפרוצדורה לא יודעת מה ערכו של הפרמטר שהועבר אליה אבל יש לה גישה לכתובת בזיכרון שמכילה את המשתנה, ולכן היא יכולה לשנות את ערכו – היא פשוט צריכה לגשת למיקום בזיכרון. ביציאה מהפרוצדורה, ערכו של הפרמטר שנשלח לפרוצדורה עשוי להשתנות.

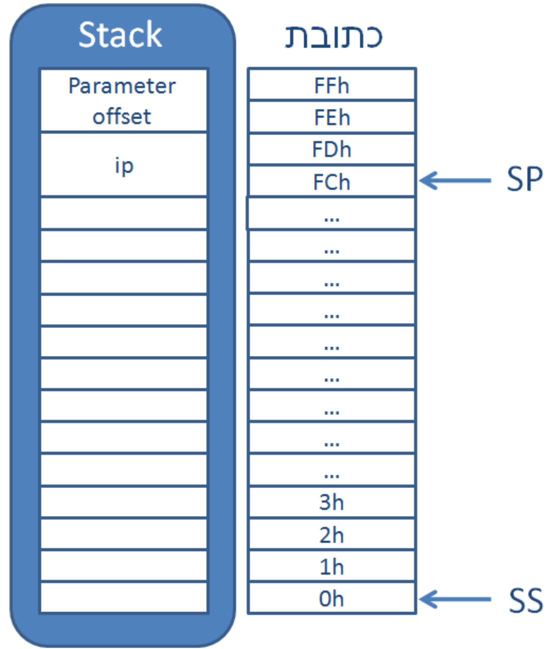
נמחיש בעזרת דוגמה: שימו לב לפקודה חדשה שמופיעה בה. אנחנו הולכים לקרוא לפרוצדורה SimpleAdd, אבל הפעם בשיטת Pass by Reference:

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

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

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

בחזרה למחסנית שלנו. מצב המחסנית עם הכניסה לפרוצדורה:



כעת הפרוצדורה 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 פרוצדורה שמקבלת פרמטר בשיטת pass by reference ומעלה את ערכו ב-1.

ב. צרו פרוצדורה שמקבלת ארבעה פרמטרים בשיטת pass by reference ומאפסת אותם.

ג. צרו פרוצדורה שמקבלת שני פרמטרים בשיטת pass by reference, ומחליפה ביניהם (לדוגמה – לפני הפרוצדורה var1=4, var2=5. אחרי הפרוצדורה var1=5, var2=4).

שימוש ברגיסטר BP

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

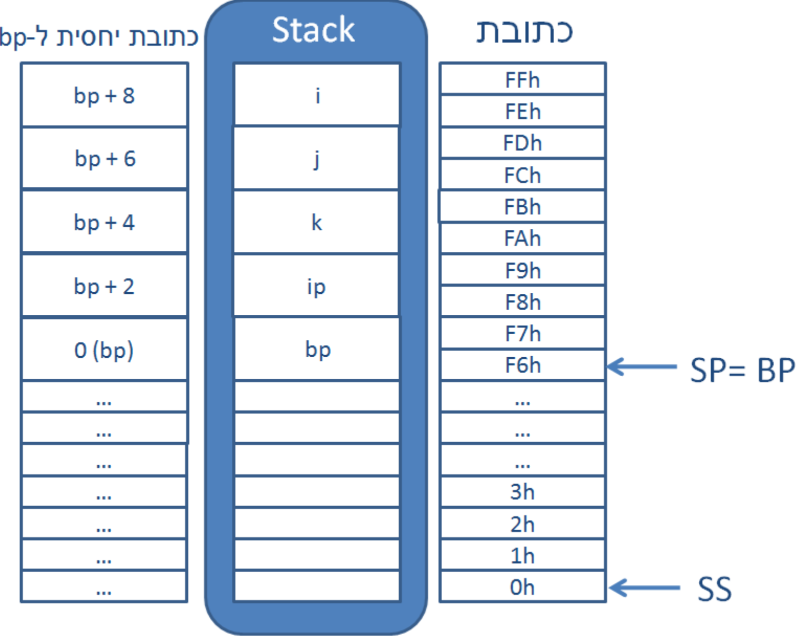
הרגיסטר bp, קיצור של Base Pointer, מסייע לנו לגשת לפרמטרים שהתוכנית הראשית הכניסה למחסנית מבלי להתעסק עם הרגיסטר ip. שימו לב לשורות הקוד שאנחנו מוסיפים לפרוצדורה (מודגשות):

```
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 מנגנון, ששומר את ערכו ההתחלתי של sp. מעכשיו, גם אם sp ישתנה כתוצאה מדחיפה או הוצאה של ערכים מהמחסנית, bp נשאר קבוע ותמיד מצביע לאותו מקום. מנגנון זה פותח לנו אפשרות לקרוא לכל ערך במחסנית לפי הכתובת היחסית שלו ל-bp.



כעת המשתנה i, שדחפנו אותו ראשון למחסנית, נמצא בכתובת שהיא 8 בתים מעל bp. המשתנים i ו־k נמצאים בכתובות שהן 6 ו־4 בתים, בהתאמה, מעל bp. מה שחשוב הוא שהמרחקים מ־bp נשארים קבועים למשך כל חיי הפרוצדורה.

נראה איך משתמשים במסקנה האחרונה. נדגים זאת שוב על הפרוצדורה SimpleProc, שמבצעת את הפעולה ax=i+j-k. 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 עכשיו בפקודה של אסמבלי, שתשדרג את הקוד שלנו. הוראת equ אומרת לקומפיילר שבכל 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]
```

עכשיו הפרוצדורה שלנו הפכה לא רק פשוטה, אלא גם פשוטה לקריאה:

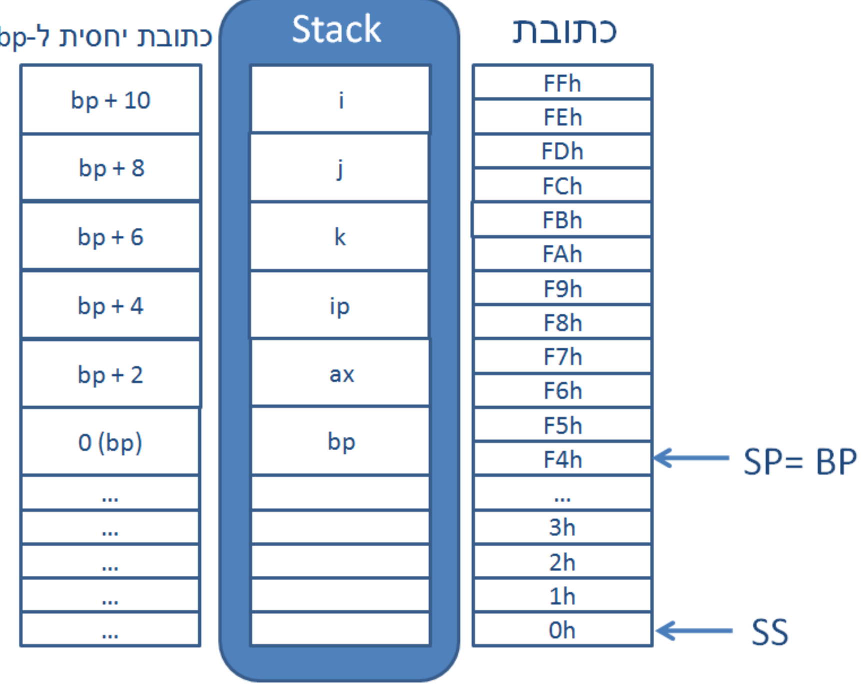
```
proc    SimpleProc
        push    bp
        mov     bp, sp
        ; Compute I+J-K
        xor     ax, ax
        add     ax, iParm
        add     ax, jParm
        sub     ax, kParm
        pop     bp
        ret     6
endp    SimpleProc
```

טיפ: השימוש ב-equ נוח במיוחד כאשר עובדים בשיטת pass by value. כך, לכל משתנה שדחפנו למחסנית יש שם מוגדר בפרוצדורה. לו היינו עובדים בשיטת pass by reference, היינו מרוויחים פחות משימוש ב-equ, משום שבשיטת pass by reference לא פונים ישירות אל הערך שנשמר במחסנית, אלא מעתיקים אותו לתוך רגיסטר שמשמש לפניה לזיכרון (לדוגמה- מעבירים מערך ומעתיקים את כתובת תחילת המערך ל-bx. במקרה זה אין תועלת לתת שם מיוחד לתחילת המערך משום שאנחנו תמיד ניגשים אל המערך דרך bx).

שימו לב: כדי שהמבנה שיצרנו יעבוד, ההעתקה של bp למחסנית ופקודת mov bp, sp חייבות להיות הפקודות הראשונות בפרוצדורה.

הסיבה היא, שאם בטעות נבצע פקודת push או pop לא מאוזנות (דחיפה והוצאה של כמות לא שווה של בתים) לפני פקודות אלו, כל המבנה שיצרנו, של יצירת משתנים במרחקים מוגדרים מ־bp, היה משתבש. הנה דוגמה לשימוש לא נכון ב־bp:

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



המחסנית של WrongBP

בעקבות ההכנסה של ax לפני bp, מבנה המרחקים מ־bp שיצרנו אינו נכון. עכשיו bp+8 מצביע על j במקום על i וכולי.

נסיים בהסבר על פקודת ה־ret 6 שבסוף הפרוצדורה.

ראינו שפקודת ret מבצעת את התהליך ההפוך מאשר פקודת call: פקודת ret מוציאה את כתובת החזרה מהמחסנית, מעלה את ערכו של sp ב־2, וקופצת אל כתובת החזרה. הפקודות הבאות שקולות לפקודת ret (למעט העובדה ש־ret אינה משנה את bx):

```
pop     bx      ; pop increments sp by 2
jmp     bx
```

כשאנו מוסיפים מספר ליד פקודת ה־ret, לאחר ביצוע ה-pop, נוסף ל-sp הערך שרשמנו ליד ה־ret. הפקודות השקולות ל־ret 6 הן:

```
pop     bx      ; pop increments sp by 2
add     sp, 6   ; sp is incremented by a total of 8
jmp     bx
```

השימוש בספרה ליד פקודת ה־ret נועד לשחרר מקום במחסנית שתפסנו באמצעות פקודות push לפני הכניסה לפרוצדורה. בדוגמה שלנו עשינו push לשלושה משתנים בגודל 2 בתים כל אחד, כלומר הכנסנו בסך הכל 6 בתים למחסנית. הפקודה ret 6 מחזירה את sp למצבו המקורי טרם הכניסה לפרוצדורה ובכך "משחררת" את הזיכרון שבמחסנית. באותה מידה היינו

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

סיכום ביניים של יתרונות השימוש ב־bp:

1. בתחילת הפרוצדורה לא צריך לעשות pop לכתובת החזרה ולשמור אותה.
2. לא צריך לעשות pop לכל הערכים שדחפנו למחסנית. פשוט ניגשים ישר אל הכתובת שלהם במחסנית בעזרת bp.
3. אפשר לייצג כל תא בזיכרון המחסנית על־ידי שם קריא ובעל משמעות.

יתרון נוסף של bp, שנראה אותו מיד, הוא ש-bp עוזר לנו לגשת למשתנים מקומיים.

תרגיל 9.6: פרוצדורה ושימוש ב־bp

א. כיתבו פרוצדורה שמקבלת שני משתנים, בשיטת pass by reference, ומחליפה ביניהם (לדוגמה – לפני הפרוצדורה var1=4, var2=5. אחרי הפרוצדורה var1=5, var2=4). בתוך הפרוצדורה יש להתייחס למשתנים רק בעזרת bp.

ב. כיתבו פרוצדורה שמקבלת שלושה משתנים: var1, var2, max. המשתנה max נשלח בשיטת pass by reference ויתר המשתנים בשיטת pass by value. ביציאה מהפרוצדורה, max יכיל את הערך הגבוה מבין var1, var2.

שימוש במחסנית להגדרת משתנים מקומיים בפרוצדורה (הרחבה)

משתנה מקומי הוא משתנה שהפרוצדורה מגדירה, והוא אינו מוכר מחוץ לפרוצדורה. משתנים אלו הם בשימוש של הפרוצדורה בלבד, ואין להם משמעות מחוץ לפרוצדורה. במקרה של משתנה מקומי, הפרוצדורה מקצה עבור המשתנה זיכרון על המחסנית וגם דואגת לשחרר את הזיכרון לפני החזרה לתוכנית שקראה לה.

איך זה מתבצע בפועל? כמו שראינו, כל הקצאת זיכרון על המחסנית מורידה את ערכו של sp. גם כאן, כדי להקצות משתנים מקומיים הפרוצדורה פשוט מורידה את ערכו של sp. אם, לדוגמה, הפרוצדורה רוצה להקצות למשתנים מקומיים 6 בתים, ההקצאה תתבצע באמצעות הפקודה:

```
sub     sp, 6
```

כמובן שלפני היציאה מהפרוצדורה צריכה להתבצע הפעולה ההפוכה, כדי לשחרר את הזיכרון (וכדי להביא את sp לערך הנכון – שיצביע על המקום במחסנית בו שמור ip):

```
add     sp, 6
```

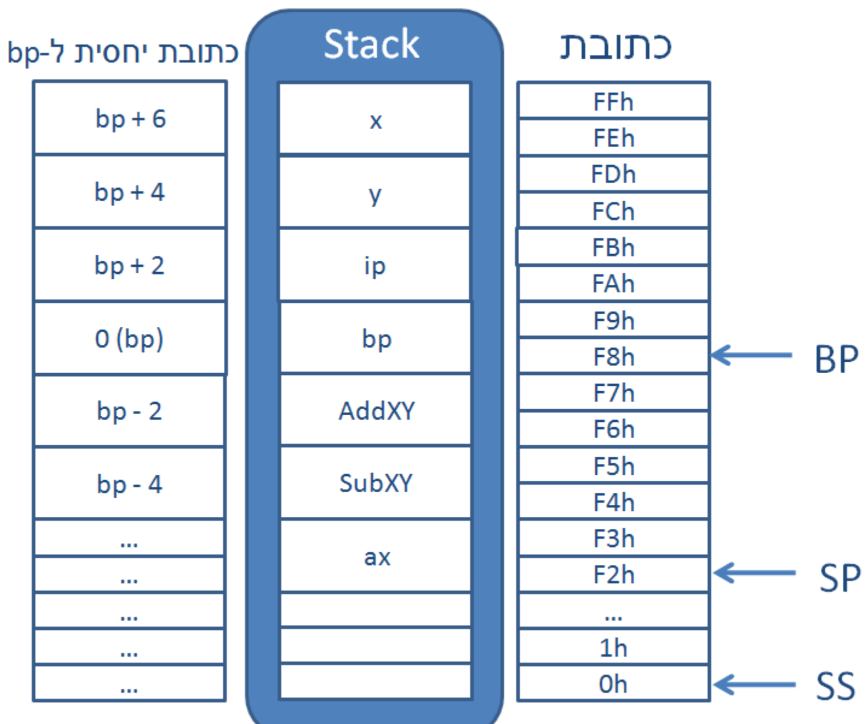
נראה דוגמה לפרוצדורה שעושה שימוש במשתנים מקומיים.

נניח פרוצדורה שמקבלת שני פרמטרים – x,y. הפרוצדורה מגדירה שני משתנים מקומיים AddXY ו־SubXY ומכניסה לתוכם את הסכום וההפרש של x ו־y, בהתאמה.

```
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 המחסנית לאחר ביצוע הפקודה push ax:



אם לדוגמה אנחנו קוראים ל־XY בתוכנית הראשית באמצעות שורות הקוד הבאות:

```
push    5
push    4
call    XY
```

כך נראית המחסנית, לאחר ביצוע החישובים ושמירת הערכים בתוך AddXY ו־SubXY. השורה המודגשת בכחול היא השורה עליה מצביע bp:

דחפנו למחסנית את הערך 5 ואת הערך 4. עשינו הקצאה של מקום לשמור את הסכום ואת ההפרש של שני הערכים. ואמנם כפי שאפשר לראות, המחסנית מכילה הן את הסכום (0009) והן את ההפרש (0001), במקומות בזיכרון בו הגדרנו את המשתנים המקומיים AddXY ו־SubXY, בהתאמה.

שימו לב לדמיון שבין צילום הזיכרון במחשב לבין האיור שמתאר את מצב המחסנית: במיקום ss:00F6, שהוא [bp-2], נמצא הסכום של שני הפרמטרים. במיקום ss:00F4, שהוא [bp-4] נמצא החיסור של שני הפרמטרים.

תרגיל 9.7: יצירת משתנים מקומיים במחסנית

כיתבו תוכנית ראשית, שדוחפת למחסנית שלושה ערכים כלשהם וקוראת לפרוצדורה בשם XYZ. הפרוצדורה כוללת שלושה משתנים מקומיים – LocalX, LocalY, LocalZ. הפרוצדורה תעתיק כל ערך לתוך משתנה מקומי אחר. הריצו את התוכנית ובידקו במחסנית שההעתקה בוצעה כנדרש.

שימוש במחסנית להעברת מערך לפרוצדורה

לעיתים נרצה שפרוצדורה תקבל אוסף של איברים, הדוגמה הקלאסית היא מערך. השיטה הבסיסית ביותר להעביר מערך לפרוצדורה היא פשוט לדחוף למחסנית את האיברים של המערך אחד אחד, בשיטת pass by value. חסרונות השיטה הזו:

- מייגעת, אם המערך ארוך.
- תופסת הרבה מקום במחסנית.
- הפרוצדורה לא יכולה לשנות את הערכים של המערך המקורי.

בשיטת pass by reference, לעומת זאת, מעבירים לפרוצדורה רק את:

- הכתובת של האיבר הראשון במערך
- מספר האיברים במערך

כעת נראה דוגמה. נגדיר מערך בתוך DATASEG:

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

בתוך CODESEG, לפני הקריאה לפרוצדורה, נדחוף למחסנית את כתובת האיבר הראשון במערך ומספר האיברים במערך:

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

בתוך הפרוצדורה אפשר להתייחס לכל אלמנט במערך על-ידי כתובת הבסיס שלו, שהועתקה אל המחסנית, בתוספת ההיסט של האלמנט מתחילת המערך.

שימו לב- בתרגילים הבאים, אין להשתמש בתוך הפרוצדורות במשתנים שהוגדרו בסגמנט הנתונים. את כל המידע הדרוש לפרוצדורה יש להעביר על גבי המחסנית.

א. צרו פרוצדורה שמקבלת מערך ואת המשתנה sum ומכניסה לתוך sum את סכום האיברים במערך. לדוגמה, עבור המערך 2,2,3,4,5 התוצאה תהיה sum=16.

ב. כיתבו פרוצדורה SortArray שמקבלת מצביע למערך ומספר איברים במערך, וממיינת את המערך מהאיבר הקטן לגדול. לדוגמה עבור המערך 3,6,5,2,1 הפרוצדורה תגרום למערך להכיל את הערכים: 1,2,3,5,6.
הדרכה לכתיבת הפרוצדורה:

- כיתבו פרוצדורת עזר FindMin שמקבלת מצביע למערך, מוצאת את האיבר הקטן ביותר ומחזירה את האינדקס שלו.
- כיתבו פרוצדורת עזר Swap שמקבלת שני פרמטרים באמצעות שיטת pass by reference ומחליפה את הערכים שלהם.
- הפרוצדורה SortArray תרוץ בלולאה על המערך ותקרא ל־FindMin. לאחר מכן תקרא ל־Swap עם שני פרמטרים: האינדקס שהחזירה FindMin והאינדקס הראשון במערך.
- לאחר שהפרוצדורה SortArray העבירה את האיבר הקטן ביותר לאינדקס הראשון, היא תקרא ל־FindMin כאשר המצביע למערך הוא על האינדקס השני במערך. לאחר מכן, SortArray תקרא ל-Swap עם שני פרמטרים: האינדקס שהחזירה FindMin והאינדקס השני במערך.
- הפרוצדורה תמשיך בשיטה זו עד לסיום מיון המערך.

ג. כיתבו פרוצדורה שנקראת Sort2Arrays, שמקבלת מצביעים לשני מערכים ומספר איברים בכל מערך, ומצביע נוסף למערך יעד sorted אליו יש להכניס את תוצאת המיון של שני המערכים, כאשר ערכים כפולים מסוננים החוצה. לדוגמה:

Array1 = 4,9,5,3,2

Array2 = 3,6,4,1

לאחר הרצת הפרוצדורה:

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

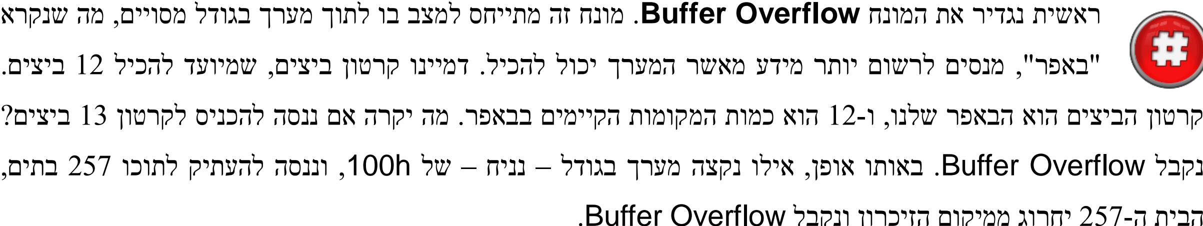
הדרכה לכתיבת הפרוצדורה:

- כיתבו פרוצדורה בשם Merge שמקבלת מצביעים לשני מערכים ומעתיקה אותם למערך אחד, ללא מיון או סינון.
- קיראו ל־SortArray עם המערך שיצרה Merge.
- כיתבו פרוצדורה בשם Filter שעוברת על המערך הממוין ומאפסת את כל הערכים שמופיעים יותר מפעם אחת.

גלישת מחסנית- 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 ממה שתוכנן (ולכן בהיבט של אבטחת מידע מדובר בבעיה קשה). איננו מעודדים כלל שינוי קוד מקור של תוכנות והדבר אף אינו חוקי כמובן. מאידך מובא לפניכם הרקע התיאורטי לנושא, מהסיבות הבאות:

1. מודעות לבעיות אבטחה עשויה תעודד אתכם לתכנת קוד ברמה גבוהה יותר, שאינו חשוף לבעיות אבטחה קלאסיות.
2. הבנה של הנושא תדרוש מכם חזרה והתעמקות בחומר הלימוד בפרק זה, שהינו אחד הפרקים החשובים ביותר להבנת אופן הפעולה של תוכנות מחשב.



ראשית נגדיר את המונח **Buffer Overflow**. מונח זה מתייחס למצב בו לתוך מערך בגודל מסויים, מה שנקרא "באפר", מנסים לרשום יותר מידע מאשר המערך יכול להכיל. דמיינו קרטון ביצים, שמיועד להכיל 12 ביצים. קרטון הביצים הוא הבאפר שלנו, ו-12 הוא כמות המקומות הקיימים בבאפר. מה יקרה אם ננסה להכניס לקרטון 13 ביצים? נקבל Buffer Overflow. באותו אופן, אילו נקצה מערך בגודל – נניח – של 100h, וננסה להעתיק לתוכו 257 בתים, הבית ה-257 יחרוג ממיקום הזיכרון ונקבל Buffer Overflow.

Buffer Overflow הוא השם הכללי ביותר לכל גלישת זיכרון מחוץ לבאפר. באם הבאפר שגלש הוגדר על המחסנית, הגלישה נקראת באופן ספציפי **Stack Overflow**.

הבה נראה איך עשויה להתרחש Stack Overflow.

התבוננו בתוכנית הבאה. עיברו על שורות הקוד ובררו לעצמכם, מה התוכנית עושה?

```
; --------------------------------------------------------------------------------
; Program StackOF – demonstration of stack overflow
; Author: Barak Gonen 2015
; --------------------------------------------------------------------------------
IDEAL
MODEL small
STACK 100h
DATASEG
        msg1   db 'Please enter your name, press enter to finish',13,10,'$'
```

```
        msg2  db 13,10,'Program finished$'
        msg3  db 13,10,'Here be dragons$'
CODESEG
proc GetName
        ; Get user input and store it on the stack
        push    bp
        mov     bp, sp
        sub     sp, 10     ; Allocate a buffer of 10 bytes on the stack
        mov     di, sp
        mov     ah, 1
        xor     bx, bx
get_char:
        int     21h
        cmp     al, 13            ; Is it the 'enter' key?
        je      quit_proc
        mov     [ss:di+bx], al    ; Copy user input to the buffer on the stack
        inc     bx
        jmp     get_char
quit_proc:
        add     sp, 10            ; De-allocate buffer
        pop     bp
        ret
endp GetName

start:
        mov     ax, @data
        mov     ds, ax
        mov     ah, 9
        mov     dx, offset msg1
        int     21h
```

```
        call    GetName
        mov     ah, 9
        mov     dx, offset msg2
        int     21h
exit:
        mov     ax, 4c00h
        int     21h
        ; This code should not be reached at all, as the program should have
        ; already exited
        nops    db 20E8h dup (90h)   ; Fill a part of the memory with NOP (90h)-
                                     ; NOP - a command which does nothing (No Operation)
        mov     ah, 9
        mov     dx, offset msg3
        int     21h
        jmp     exit
END start
```

הסבר אודות התוכנית: התוכנית מדפיסה למסך בקשה לקלוט את שם המשתמש. לאחר שהמשתמש סיים עליו להקיש enter. אז התכנית מדפיסה למסך הודעה Program finished. כך:

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

Program finished
```

את קליטת שם המשתמש מבצעת פרוצדורה, ששומרת את הקלט על המחסנית, בבאפר בגודל 10 בתים.

שימו לב שלמרות שלמחסנית אפשר לעשות push רק בכפולות של שני בתים, בגישה ישירה לזיכרון אפשר להעתיק אל המחסנית גם ערכים בגודל בית יחיד.

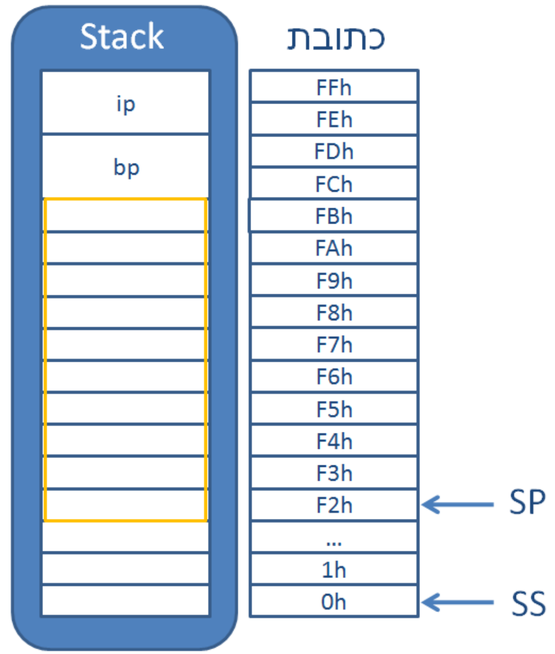
באופן לא שגרתי, בסיום קטע הקוד שמסיים את ריצת התוכנית, יש קטע קוד נוסף, שמדפיס הודעה שונה למסך. זהו קטע קוד מוזר, מכיוון שהוא לא אמור להיות מורץ. אין בתכנית שום פעולה שמקפיצה את ip כך שיגיע אל קטע הקוד הזה.

אם כך, איך אפשר להסביר את ההרצה הבאה?

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

Here be dragons
```

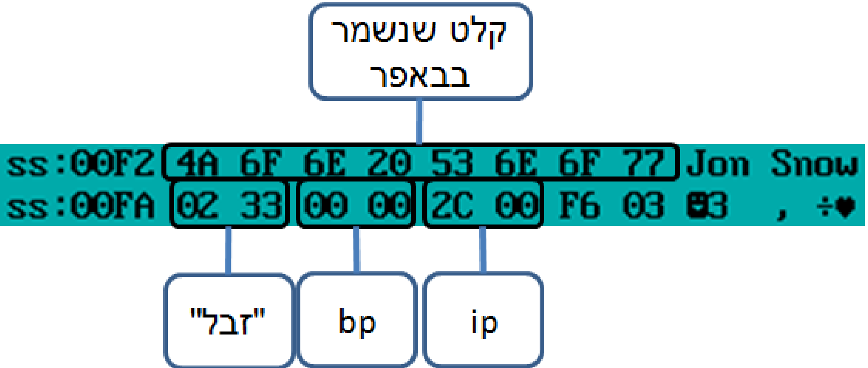
לאחר שנוכחנו שיש כאן התנהגות מוזרה, נסביר מה התרחש כאן שלב אחרי שלב.

הפרוצדורה GetName מגדירה באפר בגודל 10 בתים על המחסנית. בתוך הפרוצדורה, המחסנית נראית כך:



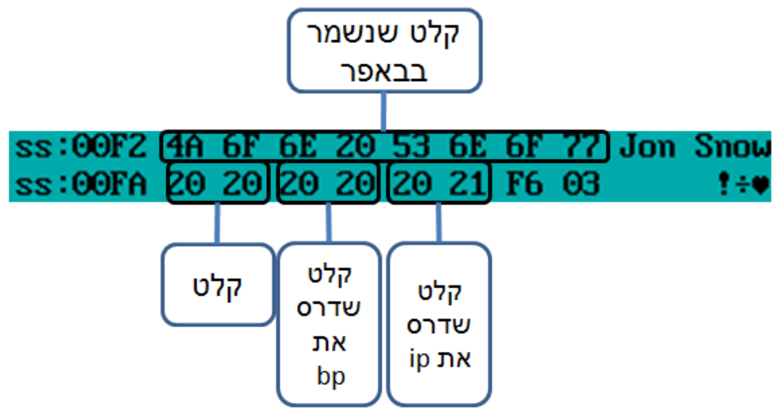
המחסנית בתוך GetName. בצהוב- הבאפר בגודל 10 בתים.

בהרצה הראשונה המשתמש הזין את שמו, Jon Snow, סך הכל 8 תווים (כולל הרווח בין המילים). ערכי ה-ASCII של התווים נכנסו למחסנית בזה אחר זה והמחסנית נראתה כך:



נבחן את מצב המחסנית. הקלט הראשון – האות J (קוד ASCII: 4Ah)– נכנסה לתחילת הבאפר במיקום ss:00F2. יתר האותיות שנקלטו הועתקו לזיכרון עד מיקום ss:00F9, כולל. כיוון שהבאפר הוגדר בגודל 10 בתים אך הועתקו אליו רק 8 בתים, נשארו בו שני בתים של "זבל", ערכים לא מאותחלים, שהיו בזיכרון טרם ריצת התוכנית. מעל ה"זבל" נמצא bp ומעליו ip, כתובת החזרה.

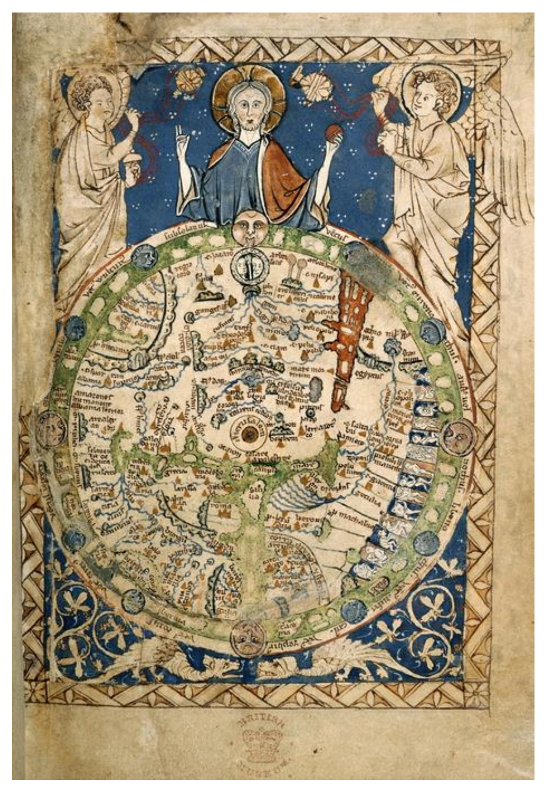
כעת נניח שהמשתמש שלנו ממשיך להזין תווים. לאחר שהוא כתב את המחרוזת 'Jon Snow', המשתמש מזין 5 תווי רווח ואז סימן קריאה. כתוצאה מהעתקה של 14 בתים לתוך באפר שגודלו 10 בתים, נוצר מצב של Stack Overflow. בשלב זה המחסנית נראית כך:



ארבעת הבתים האחרונים שנכנסו למחסנית, דרסו את bp ואת כתובת החזרה, הערכים המקוריים הוחלפו בערכי ה-ASCII של התווים שהוקלדו: ערך ה-ASCII של תו הרווח הוא 20h ואילו של סימן קריאה- 21h. פעולה זו לא גרמה למעבד לשגיאה, כיוון שלא התבצעה פעולה לא חוקית, ההעתקה לזיכרון בוצעה באופן תקין מבחינת שפת אסמבלי. עם זאת, בפעולה זו שוכתבה כתובת החזרה של הפרוצדורה. חישבו- מה יהיו ההשלכות של מצב זה?

התו הבא שהמשתמש מזין הוא enter ובשלב זה הפרוצדורה מסתיימת ומגיעה לקוד היציאה. בשלב הראשון נמחק הבאפר שהוקצה בזיכרון. בשלב השני מבוצע pop ל-bp. כמובן, שהערך שנכנס לתוך bp אינו הערך המקורי שנשמר במחסנית אלא הערך החדש- 2020h. לבסוף מבוצעת פקודת ה-ret. הערך 2120h מועתק לתוך ip ומבוצעת קפיצה לכתובת 2120h (מדוע לא 2021h? היזכרו – הזיכרון פועל בשיטת little endian).

הקפיצה לכתובת 2120h מביאה את ip הישר אל קטע הקוד שמדפיס את הודעת הסיום 'Here be dragons'. המשפט לקוח ממפות עתיקות והמשמעות הרווחת שלו היא "אזור לא מוכר, שאין לדעת מה נמצא בו. צפו לסכנות...".



מפת Psalter מ-1265. בתחתית המפה, מחוץ לטריטוריה המוכרת לאדם, דרקונים.

לסיכום סעיף זה, ראינו שבעזרת שימוש ב-Stack Overflow ניתן להקפיץ את התוכנית להוראות שהמתכנת לא תכנן שיתבצעו. אמנם במקרה הזה הפקודות שאליהן קופצים נמצאות בתוכנית, אולם באמצעות טכניקות שונות (שאינן בהיקף של ספר זה) ניתן לשתול פקודות חדשות בתוך הקוד.

תרגיל 9.8: מניעת אפשרות של Stack Overflow

שפרו את התוכנית, כך שלא ניתן יהיה לבצע בה Stack Overflow. הדרכה: העתיקו את קוד התוכנית והוסיפו שורות קוד מתאימות. עליכם למנוע לא רק את האפשרות שתודפס המחרוזת 'Here be dragons' אלא גם כל אפשרות ל- Stack Overflow אחר, לדוגמה כזה 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 לקוד שבשפת המכונה מתבצע על ידי הלינקר- אותו הכרנו בפרק אודות סביבת העבודה – אשר מקשר בין מספר קבצים ליצירת קובץ הרצה יחיד בשפת מכונה. כתוצאה מכך, יש צורך בתאימות בין הקוד בשפה העילית לקוד בשפת האסמבלי. מהי תאימות של קוד? נתקלנו כבר במקרה שבו נדרשה תאימות. היזכרו ב- endians (פרק בנושא הגדרת משתנים ופקודת mov). המעבד והתוכנה יכולים לעבוד או בשיטת little endian או בשיטת big endian- אין הדבר משנה כלל, כל עוד גם המעבד וגם התוכנה מבצעים את כל הפעולות בשיטה אחת. עירוב של שיטות יגרום לכך שהתוכנה לא תתפקד.

קונבנציה Convention – מוסכמה. התנהגות רווחת.

כפי שלמדנו על קונבנציות שקשורות לזיכרון, ישנן גם קונבנציות של הפעלת פרוצדורות, ואלו נקראות Calling Conventions. תחילה 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 הפרוצדורה MyProc? אפשר לקבוע שהיא מקבלת שני פרמטרים מטיפוס integer ומחזירה ערך מטיפוס integer. מה בדיוק עושה הפרוצדורה אינו משנה כרגע.

קריאה לפרוצדורה יכולה להיות לדוגמה:

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

קטע הקוד שקורא לפרוצדורה ("האבא") נקרא Caller ואילו קטע הקוד שנקרא, כלומר הפרוצדורה עצמה ("הבן"), נקרא Callee.

כעת הבה נדמיין שהפרוצדורה MyProc נכתבה בשפת אסמבלי ואילו הקריאה לפרוצדורה נכתבה כחלק מקוד בשפת C, אשר קומפיילר ממיר לשפת אסמבלי. חישבו- מה יכול להשתבש בין ה-Caller ל-Callee?

1. העברת הפרמטרים על גבי המחסנית:

הקומפיילר יכול להמיר את הקריאה לפרוצדורה ביותר מדרך אחת. אפשרות א':

```
push 1
push 2
call MyProc
```

אפשרות ב':

```
push 2
```

```
push 1
call MyProc
```

אפשרות א' נקראת העברה משמאל לימין Pass Left to Right ואילו אפשרות ב' נקראת העברה מימין לשמאל Pass Right to Left. בהמשך נכיר דוגמאות של קונבנציות נפוצות.

אילו ה-caller יעביר את הפרמטרים למחסנית בשיטת העברה שונה מאשר השיטה שבה ה-callee קורא את הפרמטרים מהמחסנית, ערכיהם של הפרמטרים יוחלפו. המסקנה היא, שכדי למנוע תקלות, ה-caller וה-callee חייבים לעבור לפי אותה הקונבנציה.

2. החזרת ערך מהפרוצדורה:

הדרך הנוחה ביותר להחזיר את הערך שחישבה MyProc אל הקוד שקרא לה, היא להשתמש ברגיסטר. כלומר, הפרוצדורה MyProc תעתיק את תוצאת 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 מתואמים לגבי הרגיסטר שמשמש להחזרת הערך. אין זה משנה באיזה רגיסטר כללי יוחזר הערך, רק שה-caller וה-callee יתייחסו לאותו הרגיסטר.

3. ניקוי המחסנית:

כזכור, כאשר מעבירים פרמטרים לפרוצדורה על גבי המחסנית, יש צורך לנקות את המחסנית בסיום ריצת הפרוצדורה. פעולה זו מתבצעת על ידי העלאת ערכו של sp בהתאם למספר הבתים שהועתקו למחסנית. למדנו שישנן שתי דרכים לעשות זאת. האחת, היא להוסיף לפקודת ה-ret קבוע. לדוגמה, ניקוי 4 בתים מהמחסנית:

```
ret 4
```

הדדרך השניה היא לקדם את ערכו sp:

```
add sp, 4
```

ה-caller וה-callee צריכים להיות מתואמים לגבי מי מנקה את המחסנית. אם ה-callee מבצע את הניקוי, בתוך הפרוצדורה תהיה פעולת ret עם קבוע. אם ה-callee מבצע את הניקוי, לאחר החזרה מהפרוצדורה יבוצע הניקוי. לדוגמה כך:

```
call MyProc
add sp, 4
```

גם במקרה זה אפשר לעבוד בכל שיטה, העיקר שה-caller וה-callee יעבדו לפי אותה קונבנציה.

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 שתיים נפוצות. ישנן קונבנציות נוספות, שניתן לקרוא עליהן בויקיפדיה (https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions) ובמגוון אתרי אינטרנט, אך לצורך המחשת הנושא נסתפק בפירוט של CDECL ו-STDCALL.

לפי קונבנציית CDECL:

- ערכים מועברים למחסנית Right to Left.
- רגיסטר החזרה הוא ax (או רגיסטר ax מורחב, אם מדובר על מעבד של למעלה מ-16 ביט)
- ה-caller הוא שאחראי על ניקוי המחסנית.

לפי קונבנציית STDCALL:

- ערכים מועברים למחסנית Right to Left.
- רגיסטר החזרה הוא ax (או רגיסטר ax מורחב, אם מדובר על מעבד של למעלה מ-16 ביט)
- ה-callee הוא שאחראי על ניקוי המחסנית.

היתרון של CDECL על פני STDCALL, הוא שאפשר לשלוח לפרוצדורה כמות לא קבועה של פרמטרים. מדוע? משום שה-caller אחראי על ניקוי המחסנית, כלומר מי שמבצע את פעולת עדכון sp הוא ה- caller. כיוון שה-caller יודע כמה פרמטרים הוא העביר לפרוצדורה הוא גם יכול לבצע את הניקוי. ה-callee, לעומת זאת, אינו יכול לבצע את הניקוי כיוון שמספר הפרמטרים אינו קבוע מראש.

מתי נרצה להעביר לפרוצדורה כמות לא קבועה של פרמטרים? לדוגמה, כאשר אנחנו קוראים לפרוצדורת print בשפה עילית, אנחנו רוצים גמישות לקרוא לה עם כמה פרמטרים שאנחנו צריכים להדפיס. כל פרמטר יועבר לפרוצדורה print, ולכן לא ניתן לכתוב את print כך שהיא תנקה את המחסנית בסוף הריצה.

מהו היתרון של STDCALL על פני CDECL? כאשר כמות הפרמטרים היא קבועה וידועה מראש, הפרוצדורה יכולה לנקות את המחסנית על ידי פקודת ret עם קבוע. זאת לעומת המצב שבו הפרוצדורה מבצעת ret ואז ה-caller מבצע פקודה שמעלה את sp. כלומר חסכנו פקודה. אמנם הסכון של פקודה אחת נראה כמו משהו צנוע, אך כשמכפילים את החיסכון בכמות הקריאות לפרוצדורות עשוי להתקבל חיסכון משמעותי.

לקריאה נוספת: http://www.codeproject.com/Articles/1388/Calling-Conventions-Demystified

סיכום

פרק זה הוקדש ללימוד של כמה נושאים מתקדמים, שהם הכרחיים כדי להבין איך בנויות תוכניות מודולריות. כשתעבדו בעתיד עם שפות תוכנה עיליות, חלק מהמנגנונים שלמדנו בפרק זה יהיו נסתרים מעיניכם. לדוגמה, כדי להגדיר משתנים מקומיים לא תצטרכו להקצות מקום על גבי המחסנית – אולם הרקע שניתן בפרק זה יאפשר לכם להבין לעומק את פעולות המעבד, וההבנה העמוקה היא שתיתן לכם את הכלים הנדרשים לעבודה בעולם הסייבר.

פתחנו את הפרק עם סקירה של המחסנית, אופן הפעולה שלה, הרגיסטרים שקשורים אליה ופקודות pop ו־push.

המשכנו בהסבר על פרוצדורות – איך מגדירים פרוצדורה, מה המעבד מבצע עם הקריאה לפרוצדורה, ההשפעה של פרוצדורה על מצב המחסנית. למדנו על פקודות call ו־ret. לאחר מכן למדנו איך אפשר לשלוח פרמטרים לפרוצדורה. 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, ואיך 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 הראשון, Stack Overflow, השתמשנו באבני בניין אותן למדנו בתחילת הפרק וראינו כיצד ניתן לנצל את הידע החדש שלנו על מנת להבין בעיות אבטחה. כעת נוכל להשתמש בידע זה על מנת לכתוב קוד מאובטח יותר. הנושא השני, Calling Conventions, מאפשר לנו להבין את הממשקים שבין קוד בשפה עילית לקוד בשפת אסמבלי.

בפרק הבא נלמד על פסיקות. בפרקי הלימוד ראינו כמה דוגמאות קוד שלא הבנו – למשל, השתמשנו בקוד של הדפסת תו למסך או קריאת תו מהמשתמש, אך לא הבנו איך הקוד פועל. לאחר שנלמד פסיקות נבין את קטעי הקוד הללו ונדע לכתוב בעצמנו קטעי קוד דומים.

פרק 10 – CodeGuru Extreme (הרחבה)

מבוא

CodeGuru Extreme הינה תחרות ארצית יוקרתית, בה קבוצות מתחרות זו בזו על כתיבת תוכנה שתשלוט בזירה וירטואלית. התוכנות נכתבות בשפת אסמבלי ונקראות "שורדים". התחרות מתקיימת אחת לשנה וההשתתפות בה מתבצעת בקבוצות של 2-5 תלמידים.

לוגו תחרות CodeGuru Extreme

אתר התחרות: http://www.codeguru.co.il/xtreme/

השתתפות בתחרות היא בעלת יתרונות רבים. ראשית, זוהי הזדמנות מעולה לחדד את כישורי התכנות שלכם בשפת אסמבלי. שנית, זוהי הזדמנות להתנסות במחקר תוכנה על ידי Reverse Engineering. שלישית, השתתפות בתחרות היוקרתית היא פרט שניתן להוסיף אותו לקורות החיים ולציין אותו בראיונות עבודה ובמיונים לתפקידים שונים. אחרון- התחרות היא חוויה כיפית ומהנה, גם מי שלא זוכים בה זוכרים את החוויה.

הסבר קצר על חוקי התחרות:

כל קבוצה מגישה לתחרות קוד אסמבלי, שנקרא שורד. השורדים שהגישו כל הקבוצות נטענים אל זירה וירטואלית, קטע זיכרון שגודלו 64 קילו בתים בסך הכל. כל שורד מוגרל למיקום כלשהו בזירה. בכל תור, כל שורד מקבל אפשרות להריץ פקודת אסמבלי אחת. באמצעות פקודת האסמבלי השורד יכול לבצע דברים שונים, לדוגמה- לנסות לפגוע בקטע קוד של שורד יריב. אם שורד מנסה להריץ פקודה לא חוקית, הוא נפסל ומנוע המשחק מסלק אותו מהזירה. השורד שנשאר אחרון בזירה מוכרז כמנצח.

נוסף על השורדים, צוות התחרות מכניס לזירה "זומבים". אלו הן תוכנות זדוניות, שתופסות חלק מזירת המשחק. הן כוללות חידה תכנותית או מתימטית, שפתרון שלה מאפשר השתלטות על הזומבי ושימוש בו נגד שורדים מתחרים.

נסתפק בהסבר קצר זה, כיוון שכל הפרטים וההדרכה על התחרות מצויים באתר קודגורו אקסטרים ובמיוחד בחוברת ההדרכה ובמצגת, שבקישורים הבאים:

חוברת הדרכה: http://www.cyber.org.il/assembly/codeguru-guide.pdf

מצגת: http://www.cyber.org.il/assembly/codeguru-slides.pdf

כמו כן מומלץ להעזר בפורום שבאתר קודגורו אקסטרים, שם גם מפורסמים עדכונים שוטפים 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 ההדרכה ובמצגת, אך הם עשויים לסייע לכם רבות בהכנה לתחרות. הנושא הראשון הוא מספר פקודות שימושיות באסמבלי. הנושא השני הוא ביצוע Reverse Engineering לזומבים על מנת להשתלט עליהם ולשורדים מתחרים על מנת לאתר נקודות חולשה שלהם.

פקודות אסמבלי שימושיות

פקודות אלו אינן הכרחיות לטובת יחידת המעבדה באסמבלי, כיוון שניתן לכתוב תוכנית אסמבלי עובדת היטב גם ללא שימוש בהן. עם זאת הן יכולות לבצע פעולות שבשביל לבצע אותן בדרך אחרת יידרשו מספר רב של שורות קוד ולכן רצוי להכיר אותן כדי לכתוב קוד יעיל לשורד.

1. XCHG: פקודת XCHG מקבלת שני רגיסטרים, או רגיסטר ותא בזיכרון, ומחליפה בין הערכים שלהם

   לדוגמה:

```
xchg ax, bx
```

   מחליפה בין הערכים של ax ו-bx. פעולה זו חוסכת מספר פעולות mov.

2. XLAT: על מנת להבין את פקודת XLAT נבין קודם מהו Look Up Table, או בקיצור LUT. LUT היא טבלה שמחזיקה אוסף של ערכים, כאשר יש קשר בין הערך לאינדקס שלו. לדוגמה- אפשר להחזיק ב-LUT את ערכי סדרת פיבונצ'י. כיוון שסדרת פיבונצ'י היא

0, 1, 1, 2, 3, 5, 8, 13 …

   אז לדוגמה הערך באינדקס 6 יהיה 8, הערך באינדקס 7 יהיה 13 וכך הלאה (שימו לב- האינדקסים מתחילים מאינדקס אפס).

בשביל מה זה טוב? נניח שיש לנו תוכנית שכוללת המרה מסט אחד של ערכים לסט אחר של ערכים. לדוגמה- אנחנו רוצים להצפין טקסט על ידי צופן החלפה (צופן בו כל תו מוחלף על ידי תו אחר. לדוגמה a מוחלף ב-m, b מוחלף ב-f וכו'). דרך אחת היא ליצור פרוצדורה שיש בה הרבה מאד תנאים (אם האות היא a החזר m, אם האות היא b החזר f וכך הלאה...). דרך יעילה הרבה יותר היא לשמור ב-LUT את צופן ההחלפה כך שכל אינדקס ב-LUT מצביע על הערך אותו יש להחזיר. לדוגמה, קוד ה-ASCII של a הוא 97 ואנחנו רוצים להחליף אותה עם האות m, שקוד ה-ASCII שלה הוא 109. את האות b, שקוד ה-ASCII שלה הוא 98, אנחנו רוצים להחליף באות f, שהקוד שלה הוא 102. 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. כעת בתוך סגמנט הקוד נכתוב:

```
mov bx, offset Cipher

mov al, 'a'

xlat
```

ו-al יכיל את ערך ה-ASCII של m, כפי שרצינו.

3. NOP: זהו קיצור של No Operation, כלומר פקודה שאומרת למעבד "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: הפקודה STD מדליקה את דגל הכיוון ואילו CLD מכבה אותו. לשם מה זה נחוץ? בשביל פקודת MOVSW.

5. MOVSW: פקודה זו מסייעת לנו להעתיק בצורה יעילה מידע מאזור זיכרון אחד לאחר. לדוגמה, יש לנו מחרוזת של 200 איברים, ואנחנו מעוניינים להעתיק אותה למקום אחר בזיכרון. דרך אפשרית היא ליצור לולאה שתרוץ על המחרוזת הראשונה תוך כדי שהיא מקדמת בכל פעם אינדקס שמשמש לקריאה מהזיכרון, ובעזרת פקודת mov תעתיק את המחזורת למיקום החדש, תוך קידום אינדקס שמשמש לכתיבה לזיכרון. זה יעבוד, אולם פקודת MOVSW עוזרת לפשט את התהליך.

פקודה זו מעתיקה מילה מכתובת ds:di אל הכתובת es:si, תוך עדכון ערכי הרגיסטרים di, si. כלומר, נחסכה מאיתנו המתכנתים פעולת עדכון רגיסטר המקור ורגיסטר היעד. אך כדי לעדכן את ערכי הרגיסטרים di, si צריך

לדעת האם להעלות אותם או להוריד אותם. לשם כך קבענו את ערכו של דגל הכיוון בפקודת STDאו CLD טרם ההעתקה.

6. REP: כדי להעתיק מספר תאים בזיכרון ניתן להוסיף לפני הוראת MOVSW את הקידומת REP, קיצור של Repeat. ראשית מאתחלים את ערכו של cx כך שיכיל מספר הפעמים שהוראת ההעתקה צריכה להתבצע ואז כותבים

```
rep movsw
```

קוד זה שקול לפעולות הבאות:

```
my_label:
        movsw
        dec     cx
        jnz     my_label
```

פקודת אסמבלי נוספות ניתן למצוא בנספח א'.

Reverse Engineering

נושא ה-Reverse Engineering, או בקיצור RE, הוא עמוק ורחב מכדי לסכם אותו בפרק אחד. לכן לצערנו אין ביכולתנו ללמוד לבצע RE במסגרת לימודי האסמבלי. מאידך, כן נלמד איך מבצעים RE לשורדים ולזומבים של קודגורו אקסטרים. זוהי משימה בהיקף מצומצם, שמצד אחד ניתן לכסות בפרק יחיד ומצד שני מספקת טעימה מיכולת מעניינת זו.

הזומבים שבפרק זה נמצאים בקישור:

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

שימו לב- בסעיפים הבאים נלמד כיצד לחקור קוד של תוכנה. מחקר של תוכנה הוא חוקי כל עוד הוא נעשה על תוכנה שנמסרה לנו בכוונה שנחקור אותה. מחקר של תוכנה מסחרית, בכוונה לפרוץ סיסמאות או לשנות דברים בתוכנה, הוא אסור לפי החוק. היזהרו והישארו בצד הטוב של החוק.

duck.com

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

אפשרות אחת היא להפעיל את קובץ ההרצה. מתוך חלון ה-cmd הקישו duck ואז enter. התוכנית רצה אך אינה מגיבה... הבעיה 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+si],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 במיקום cs:100h. הפקודה הראשונה היא jmp 100h. לאחר מכן יש פקודות שונות שלא ברור מה הקשר ביניהן. מיד נבין.

נתחיל בהרצת התוכנה באמצעות f7. כאשר נריץ את התוכנה נגלה שהיא נשארת באותה שורת קוד. הסיבה לכך היא שפקודת jmp 100h נמצאת בכתובת 100h. למעשה, אפשר להבין שהקוד שיצר את התוכנית היה:

```
start:
        jmp     start
end start
```

כעת, מהן אותן שורות קוד שמופיעות החל מכתובת cs:102h? זהו התרגום לשפת מכונה של התאים הלא מאותחלים שישנם בזיכרון המחשב, כלומר זיכרון "זבל". זהו, סיימנו את ניתוח הזומבי הראשון שלנו.

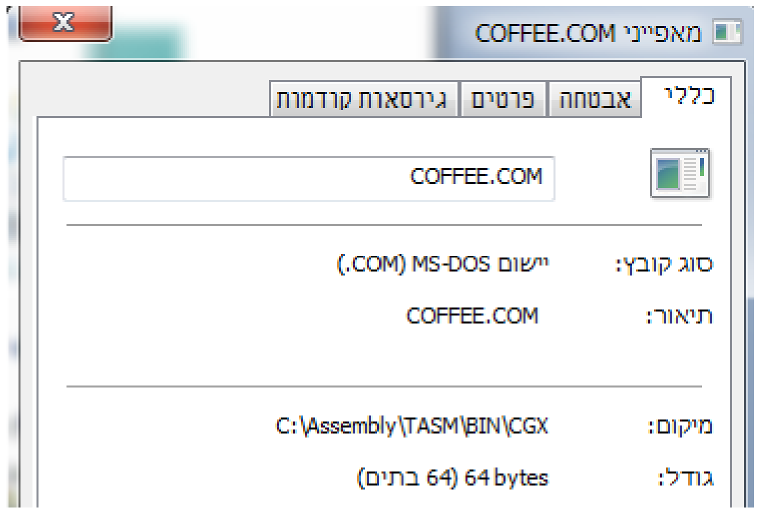
coffee.com

לאחר שהבנו את העקרון הכללי של RE, נעבור לזומבי הבא, שמו coffee.com. הפעם גם נבצע RE כדי להבין מה הזומבי מבצע וגם נלמד איך אפשר לגרום לו להריץ קוד שאנחנו כתבנו.

```
[■]=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 חוזר של פקודות mov-cmp-jne ואז פקודת push ו-ret. פקודת ה-ret היא מועמדת לא רעה לציין את סיום הקוד של הזומבי. גם אילו היינו רואים את פקודות היציאה מ-dos (int 21h ,ax=4C00h) היינו יכולים לדעת שמדובר בסיום הקוד. לאחר פקודת ה-ret נפסק רצף הפקודות ההגיוניות. לדוגמה, הפקודה push 0000 מופיעה פעמיים, סביר להניח שזה אינו קוד שמישהו כתב.

כדי למצוא בוודאות את מיקום סיום הקוד, ניגש אל המאפיינים (properties) של הקובץ:



כפי שרואים, גודלו של הקובץ הוא 64 בתים, או 40h. כיוון שהקובץ מתחיל בכתובת cs:0100h, הרי שהפקודה המתחילה ב-cs:0140h כבר אינה חלק מהקוד של הקובץ. כך וידאנו באופן סופי את ההשערה שלנו.

כעת – מה עושה הזומבי coffee?

שלוש הפקודות הראשונות צריכות להיות מוכרות לכל מי שקרא את המדריך- פקודת int 87 היא הפצצה חכמה. הזומבי נפטר מהפצצה החכמה שלו.

ב-cs:0104 מועתק בית ממקום 0000h בזיכרון לתוך dl. בפקודה הבאה הערך של dl מושווה ל-43h ואם אין שוויון, מתצעת חזרה על ההעתקה מהזיכרון ופעולת ההשוואה. כלומר, כל עוד מקום 0000h אינו מכיל את הערך 43h (קוד ה-ASCII של האות c). אם השוויון מתקיים, התוכנית ממשיכה לבדיקה הבאה- האם הבית הבא בזיכרון מכיל את הערך 30h (קוד ה-ASCII של הספרה 0). המשיכו מכאן- אילו תנאים צריכים להתקיים כדי שהתוכנית תגיע אל פקודת ה-ret?

נניח שאנחנו רוצים להשתלט על זומבי זה, כלומר לגרום לו להריץ את הקוד שלנו. איך אפשר לבצע זאת? נצטרך לשלוח אותו ל-ip בו נמצא הקוד שלנו. פקודת ה-ret שבכתובת cs:013Fh יכולה לסייע לנו בביצוע התהליך. אנחנו צריכים לדאוג לשני תנאים. הראשון, שהזומבי יגיע להריץ את שורת ה-ret. ראינו שעל ידי הזנת ערכים מתאימים למיקומים מוגדרים בזיכרון, אנחנו יכולים "לשחרר" אותו ולגרום לו להגיע לפקודת ה-ret. התנאי השני, הוא שעם ההגעה לפקודת ה-ret בראש המחסנית יהיה ה-ip של הקוד שלנו. לעזרתנו נמצאות הפקודות שמעתיקות את מיקום 0006h אל bx ואז דוחפות את bx לראש המחסנית. אם נדאג לכך שבזמן שהזומבי מריץ את שורות אלה בכתובת 0006h נמצאת כתובת של פקודה בקוד שלנו, הזומבי יסיים את הריצה ויקפוץ אליה. משם הוא ימשיך ויריץ את הקוד שלנו.

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

codeguru.com

כעת נטפל בזומבי אמיתי מתחרות CodeGuru Extreme 2015. כנהוג בתחרות, הזומבי מכיל בתוכו חידה מתימטית 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 לפני כן, ניתן לזהות את סיום קוד הזומבי בפקודת ה-ret שבמיקום cs:011Ch.

ב. שלושת הפקודות הראשונות נפטרות מהפצצה החכמה.

ג. ארבעת הפקודות הבאות (cs:0104h עד cs:010Ah) מעתיקות לתוך cx את ax, כאשר ax שווה בקודגורו אקסטרים למיקום ההתחלתי של הזומבי ואילו bx מאותחל לקבוע 1Dh.

ד. לאחר מכן נטען לתוך ax המילה שמתחילה במיקום 2015h בזיכרון

ה. לאחר ביצוע xor בין ax ו-cx מתבצעת פעולת xlat. ניזכר במה מבצעת xlat: היא מעתיקה לתוך al את הערך שנמצא במיקום ds:bx+al. נציין כי בניגוד לתוכניות שאיתן עבדנו בפרקים הקודמים, בקודגורו אקסטרים מודל הזיכרון מוגדר כך ש-ds ו-cs מצביעים על אותו מקום בזיכרון. אם כך, על איזה זיכרון מצביע ds:bx? נבדוק

מהו ערכו של bx ערך זה הוגדר להיות מיקום תחילת התוכנית ועוד הקבוע 1Dh. כיוון שהתוכנית מתחילה במיקום 100h, ערכו של bx יהיה 11Dh. כלומר ds:bx יהיה שווה ל-cs:011Dh.

ו. נחקור את המיקום בזיכרון cs:011Dh. זהו המיקום הראשון שנמצא אחרי פקודת ה-ret. נמצאת שם פקודת add משונה. גם יתר הפקודות הן צורות משונות של add. מדוע מתבצע xlat למיקומים אלו בזיכרון? נניח al=0. הפקודה xlat תחזיר את הערך שבמיקום cs:011Dh. ערך זה הוא 0. נניח al=1, הפקודה xlat תחזיר את הערך שבמיקום cs:011Eh 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 לגלות את חוקיות הסדרה. אם אינכם מצליחים, לא נורא- הריצו חיפוש בגוגל על סדרת המספרים. אם הצלחתם למצוא את החוקיות בלי עזרה- הרשמו בהקדם לאולימפיאדה למתמטיקה לנוער.

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

ט. פעולת ה-xlat חוזרת על עצמה פעמיים, פעם עבור al ופעם עבור ah.

י. לאחר מכן מחושב הסכום שלהם- אם הוא אינו עולה על 6, מתבצעת יציאה על ידי ret. איזה ערך יקבל ip?

יא. מהו הערך שיש בראש המחסנית בשלב זה? הערך האחרון שנדחף למחסנית הוא ax, מיד לאחר שהועתק אליו ה-word שמתחיל במיקום 2015h.

לסיכום, אנחנו מבינים שעל מנת להוציא את הזומבי מהלולאה בה הוא נמצא ולהגיע לפקודת ה-ret, יש צורך להכניס ערך מסויים ל-word במיקום 2015h. אותו הערך הוא גם ה-ip שהזומבי יקפוץ אליו בסיומו. הבנת הערך ה"נכון" תלויה בהבנת חוקיות הסדרה המתימטית שאיתרנו.

תרגיל: Make it – Break it – Fix it

בתרגיל זה נכתוב תוכנית שמבקשת סיסמה מהמשתמש ובודקת אם הסיסמה "נכונה". אם כן, תודפס למסך הודעה מתאימה: "Access granted". בשלב הראשון, ננסה לכתוב תוכנה מסובכת ככל הניתן לפענוח. בשלב השני נמסור את התוכנה לחברינו לכיתה, שינסו לגרום להדפסת ההודעה "Access granted" על ידי פענוח הסיסמה. בשלב השלישי נפיק לקחים ממה שחברינו עשו ונשפר את התוכנה כך שמציאת הסיסמה תהיה קשה יותר.

שלב א'- Make it

בשלב זה אנחנו מעוניינים לכתוב תוכנית שקולטת סיסמה מהמשתמש, אם הסיסמה נכונה היא מדפיסה את הודעת ההצלחה למסך. כתיבה של תוכנית כזו דורשת שני דברים שטרם למדנו: הראשון, קליטה של תו מהמשתמש. השני, הדפסה של מחרוזת למסך.

קליטה של תו מהמשתמש: מכניסים לתוך ah את הערך 1, כותבים את הפקודה int 21h, התו שהמשתמש מקליד נכנס לתוך al. כך:

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

הדפסה של מחרוזת למסך:

כדי להדפיס מחרוזת למסך אנחנו צריכים קודם כל ליצור מחרוזת, שמסתיימת בתו '$' (זה הסימן ל-ISR להפסיק את ההדפסה למסך – אחרת יודפס כל הזיכרון...). התווים 13,10 מורים לרדת שורה בסוף ההדפסה. לדוגמה:

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

לאחר מכן טוענים לתוך dx את האופסט של המחרוזת:

```
mov     dx, offset message
```

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

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

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

להלן תוכנית פשוטה, שמבקשת סיסמה מהמשתמש ועל סמך בדיקת הסיסמה מדפיסה הודעת הצלחה או כישלון. שקולטת תווים, משווה אותם לסיסמה ואם יש שוויון מדפיסה הודעה מתאימה. כפי שאפשר לראות קל למצוא סיסמה מתאימה. אתם יכולים להתבסס על התוכנית הזו ולשפר אותה כך שמציאת הסיסמה תהיה משימה 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 הזו להיות יותר חסינה לנסיונות מציאת הסיסמה?

הדרכה:

1. בחנו את השורות בהן מתבצעת הבדיקה של הסיסמה. האם תוכלו לפתח מנגנון יותר מורכב, שכולל בדיקות נוספות?
2. ביצוע RE לקוד שיש בו השוואה לקבוע, לדוגמה 'X', מסגיר מיד מהו הקבוע המבוקש. נסו לבצע את ההשוואות על ידי רגיסטרים.

3. השתמשו בפעולות על ביטים כדי להקשות עוד יותר את מציאת הסיסמה הנכונה מתוך הקוד בשפת מכונה.
4. מכאן המשיכו הלאה בכוחות עצמכם. גלו יצירתיות!

שלב ב'- Break it

קחו את התוכנית שכתבו חבריכם (כמובן, רק את קובץ ההרצה- לא את שורות הקוד באסמבלי). השתמשו בידע שצברתם כדי לפצח את סיסמת הכניסה אל התוכנית של חבריכם. הסבירו לחבריכם איך מצאתם את הסיסמה לתוכנית שלהם.

שלב ג'- Fix it

לאחר ששמעתם כיצד חבריכם פיצחו את סיסמת הכניסה לתוכנית שלכם, מיצאו פיתרון שיקשה על חבריכם למצוא את הסיסמה באותה השיטה.

סיכום

בפרק זה סקרנו בקצרה את תחרות קודגורו אקסטרים וקיבלנו מספר כלים חשובים להצלחה בתחרות. הכלי הראשון הינו ידיעת פקודות אסמבלי מיוחדות. הכלי השני הוא יכולת מחקר של קוד שנכתב על ידי מישהו אחר באסמבלי, הכלי השלישי הוא הבנת טכניקת ההשתלטות על זומבים. יתר הידע הנדרש מרוכז בחומרי ההדרכה- המשיכו מכאן בכוחות עצמכם.

איחולי הצלחה בתחרות והנאה מההשתתפות ומהתרגול לקראת התחרות!

פרק 11 – פסיקות

מבוא

פסיקה (Interrupt) היא אות המתקבל במעבד ומאפשר לשנות את סדר ביצוע הפקודות בתוכנית שלא על-ידי פקודות בקרה מותנית (פעולות השוואה וקפיצה – כגון cmp ו־jmp).

נסביר למה הכוונה. משתמשים בפסיקה כשרוצים לשנות את סדר ביצוע הפקודות באופן בלתי צפוי מראש (אחרת היינו משתמשים בפקודות בקרה וקפיצה). למה בכלל רוצים לשנות את סדר ביצוע הפקודות?

לעיתים נרצה שהתוכנה שלנו לא תעבוד בדיוק באותו אופן כל הזמן – לדוגמה, תמצא את האיבר הגדול ביותר במערך – אלא תגיב לאירועים שונים. דמיינו משחק מחשב, שבו השחקן משתמש במקלדת כדי להפעיל דמות שזזה על המסך. התוכנה צריכה להגיב לפקודות השחקן – כל לחיצה על המקלדת צריכה לגרום לתוכנה להריץ קטע קוד אחר. גם התזמון של לחיצות המקלדת לא ידוע מראש – התוכנה לא יכולה להניח שברגע מסויים השחקן יקיש על המקלדת. הצורך בגמישות, ביכולת לשנות את אופן ריצת התוכנה, גורם לכך שנצטרך מנגנון שיודע לשנות את סדר ביצוע הפקודות במעבד באופן דינאמי.

במשפחת ה־80x86 יש שלושה סוגים של אירועים שנכנסים 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 לאירוע 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 אירוע חיצוני שדורש טיפול. המעבד עוצר את ביצוע הקוד שרץ, משרת את רכיב החומרה וחוזר לתוכנית למקום שעצר בה.

בהמשך נדון בהרחבה בכל אחת מסוגי הפסיקות.

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

```
int     operand
```

לדוגמה, הפעלת פסיקה מספר 1:

```
int     1h
```

בפרקים הקודמים נתקלנו בפקודה זו. כאשר רצינו לבצע פעולות של קריאת תו מהמקלדת או הדפסת תו למסך, למשל, השתמשנו בפקודה:

```
int     21h
```

במשך השנים חברות שונות פיתחו קוד שכולל אוסף של פסיקות תוכנה וחומרה שימושיות. לדוגמה, חברת מיקרוסופט, פיתחה מערכת הפעלה בשם DOS – קיצור של Disk Operating Systems. מערכת ההפעלה DOS מספקת

למתכנתים פסיקות שימושיות, שנתקלנו בהן בפרקים קודמים, כגון קריאת תו מהמקלדת והדפסת תו למסך. דוגמה נוספת היא חברת אינטל, יצרנית מעבדי ה־80x86, שפיתחה קוד שנקרא BIOS – קיצור של Basic Input Output System. ה־BIOS הוא קוד שנמצא בזיכרון מיוחד, ונטען למעבד מיד עם הפעלתו. תפקידו של קוד ה־BIOS הוא לבדוק את תקינות החומרה ולטעון את קוד מערכת ההפעלה. כמו כן כולל ה־BIOS מספר פסיקות שמפשטות את השימוש בהתקני חומרה, לדוגמה קליטת תווים מהמקלדת, נושא שנגיע אליו בהמשך.

כתוצאה מכך שיש לנו פסיקות זמינות לשימוש ממקורות שונים, ישנן לעיתים דרכים שונות לבצע את אותה פעולה. לדוגמה, תקשורת עם המקלדת:

- עם המקלדת אפשר לתקשר באמצעות פסיקה מספר 9h.
- אפשר לתקשר עם המקלדת גם דרך פסיקה מספר 16h, שהיא פסיקת BIOS. פסיקה זו למעשה "עוטפת" את פסיקה 9h עם קוד נוסף.
- אפשר לתקשר עם המקלדת גם דרך פסיקה מספר 21h, של DOS. קוד ה־DOS "עוטף" את הקוד של BIOS.

דוגמה נוספת, תקשורת עם שעון המערכת (טיימר):

- פסיקה מספר 8h עובדת מול הטיימר.
- פסיקה מספר 1Ch היא פסיקת BIOS, שגם עובדת מול הטיימר. פסיקה זו למעשה "עוטפת" את פסיקה 8h עם קוד נוסף.
- פסיקה מספר 21h של DOS, עובדת גם היא מול הטיימר. קוד ה־DOS "עוטף" את הקוד של BIOS.

יתרונות וחסרונות של שימוש בפסיקות מסוגים שונים: באופן כללי, כל קוד שעוטף את הפסיקה המקורית עושה את השימוש בה יותר פשוט, אבל אנחנו מפסידים גמישות ולעיתים גם ביצועים.

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

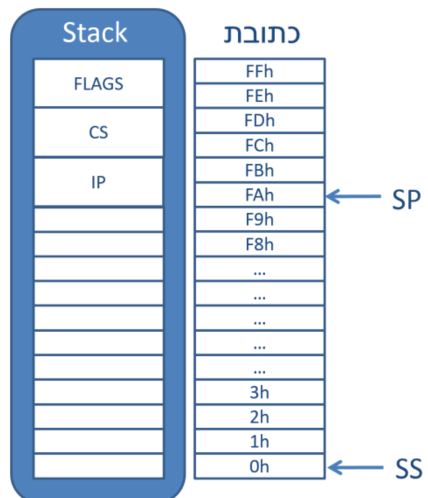
כל פסיקה, יהיה אשר יהיה המקור שלה, מפעילה קוד מיוחד לטיפול בפסיקה. הקוד הוא כמובן שונה בין פסיקה לפסיקה. קוד זה נקרא בשם כללי Interrupt Service Routine או בקיצור ISR.

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

שלבי ביצוע פסיקה

תהליך ביצוע פסיקה מורכב מהשלבים הבאים:

1. המעבד מסיים את ביצוע ההוראה הנוכחית. כלומר – אם לדוגמה היינו באמצע הוראת mov ax,5 בזמן שהמשתמש לחץ על מקש במקלדת, המעבד יעתיק לתוך ax את הערך 5 ואז יתפנה לשרת את הפסיקה מהמקלדת.
2. המעבד שומר במחסנית את תוכן רגיסטר הדגלים ואת כתובת ההוראה הבאה לביצוע בתום ביצוע תוכנית הפסיקה (ה-ISR). כזכור, כתובת ההוראה הבאה היא צירוף של הרגיסטרים cs ו-ip.
3. לפני הכניסה ל-ISR המעבד "מנקה" (מאפס) את דגל הפסיקות (Interrupt Flag) ואת דגל המלכודת (Trap Flag). הסיבה לכך – בהמשך.
4. המעבד מחשב את כתובת ה-ISR ומעתיק אותו לרגיסטרים cs ו-ip. תהליך חישוב כתובת ה-ISR ראוי להסבר בפני עצמו ונתייחס אליו בהמשך.
5. המעבד מבצע את ה-ISR.
6. בתום ביצוע ה-ISR, המעבד מוציא מהמחסנית את הערכים שנדחפו אליה (ראו סעיף ב') ומשחזר את הערכים המקוריים של רגיסטר הדגלים, cs ו-ip.
7. המעבד ממשיך בביצוע התוכנית ממקום cs:ip.



מצב המחסנית עם הכניסה לפסיקה

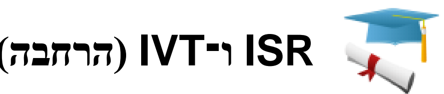
(האיור מניח שהמחסנית בגודל 100h וכמו כן שהמחסנית היתה ריקה לפני הפסיקה)

נתייחס למשמעות איפוס הדגלים Interrupt Flag ו־Trap Flag:

איפוס דגל המלכודת גורם לכך שהפקודות יבוצעו ללא הפסקה גם אם אנחנו בתוך הדיבאגר. נדמיין מה היה קורה אם דגל זה לא היה מאופס אוטומטית. אנחנו נמצאים בתוך הדיבאגר, מתקדמים בתוכנית שורה אחרי שורה. כל 55 מילישניות, כפי שנלמד בהמשך, מגיעה פסיקה לעדכון השעה. התוכנית שלנו קופצת אל ה־ISR שאחראי לעדכון השעה, אבל מיד עוצרת – אנחנו צריכים ללחוץ על מקש ה־F7 כדי להריץ את התוכנית לשורה הבאה. פעולה זו חוזרת עצמה כל 55 מילישניות... עצם הלחיצה על מקש ה־F7 מפעיל ISR שאחראי לטיפול במקלדת, כך שלמעשה הפעולה אף פעם לא מסתיימת...

איפוס דגל הפסיקות מונע מפסיקות נוספות להגיע תוך 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 שסיים לטפל בפסיקה הקודמת.

לאחר היציאה מה־ISR, המעבד משחזר מהמחסנית את רגיסטר הדגלים, ובין היתר את ערכו של הדגל if, כך שפסיקות חומרה שוב מאופשרות. המונח המקצועי של פעולה זו הוא enable interrupts.



ISR ו־IVT (הרחבה)

המבנה הכללי של ISR הוא כזה:

```
proc    ISRname far
        …
        iret
endp    ISRname
```

פקודת ה־iret היא כמו פקודת ret – היא דואגת לשחזור כתובת החזרה אל המקום שבו היתה התוכנית לפני הקריאה ל־ISR. ההבדל בין iret ל־ret הוא ש־iret עושה pop נוסף, כדי לשחזר גם את רגיסטר הדגלים. כפי שאפשר לראות, יש דמיון רב בין הגדרה של ISR להגדרה של פרוצדורה, ואמנם אופן הפעולה שלהם דומה.

השאלה שאנחנו רוצים לענות עליה היא: המעבד קיבל פסיקה. עכשיו הוא צריך להפסיק את ריצת התוכנית ולקפוץ למקום כלשהו בזיכרון, שם אמרנו 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 ל־255 דצימלי, אך נהוג לקרוא לפסיקה לפי הערך ההקסדצימלי שלה. לדוגמה, כשכתבנו את הפקודה:

```
int     21h
```

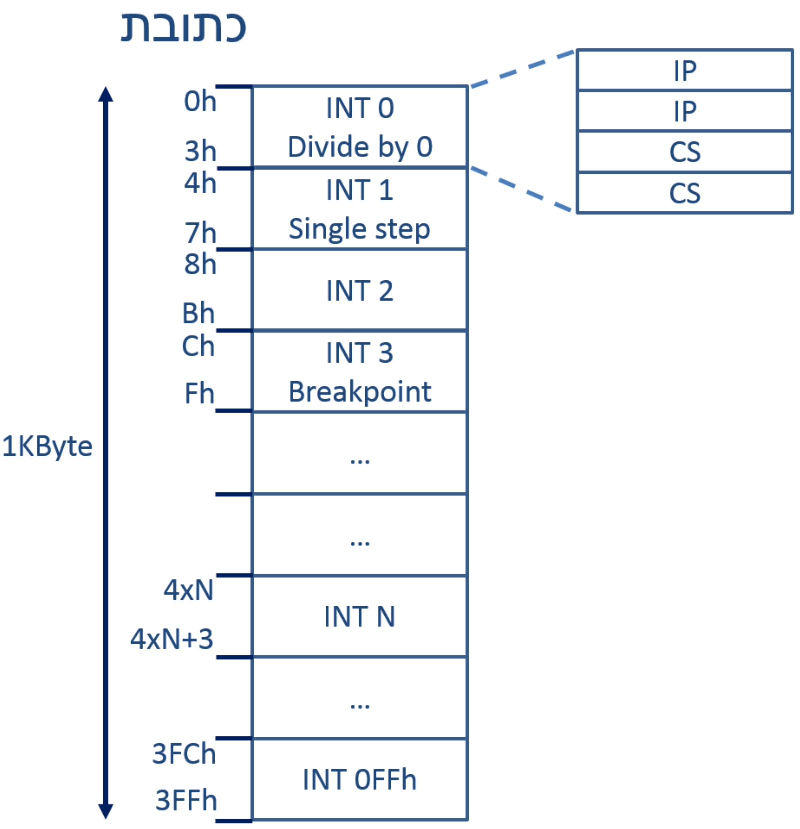
הודענו למעבד שהוא צריך לשרת את פסיקה מספר 21h.

כעת המעבד ניגש למלאכת תרגום מספר הפסיקה לכתובת בזיכרון. לטובת פעולת התרגום, המעבד פונה לטבלת תרגום, ששמורה אצלו מראש בזיכרון. טבלה זו נקראת Interrupt Vector Table, או IVT.

ה־IVT הינה טבלה בגודל של 256 Double Words, כלומר 1024 בתים. תחילת ה־IVT היא בסגמנט 0 ובאופסט 0 בזיכרון, או במילים אחרות – ה־IVT נמצא ממש בתחילת הזיכרון ותופס את הכתובות שבין 0 ל־1023 (3FFh בתים בסך הכל).

איך ה־IVT עוזר למעבד לתרגם את מספר הפסיקה לכתובת בזיכרון? ב־IVT נמצאות הכתובות של 256 פרוצדורות שמטפלות בפסיקות, 'ISR'ים. כל כתובת מורכבת משתי מילים. המילה הראשונה היא האופסט של ה־ISR והמילה השניה היא הסגמנט של ה־ISR. כך לדוגמה, ארבעת הבתים הראשונים ב־IVT הם הכתובת של ISR של פסיקה מספר 0h. ארבעת הבתים הבאים אחריו הם הכתובת של ISR של פסיקה מספר 1h וכן הלאה.

כלומר, כדי לדעת מה הכתובת של ISR כלשהו בטבלה, לוקחים את מספר הפסיקה, כופלים ב־4 והתוצאה היא הכתובת הפיזית של ה־ISR שהמעבד צריך לקפוץ אליה. דוגמה: פסיקה 21h מיתרגמת למקום 84h בזיכרון. המעבד ניגש למקום 84h, שנמצא כמובן ב־IVT, וקורא ארבעה בתים, שמורים לו לאיזו כתובת לקפוץ לביצוע ה־ISR של פסיקה 21h.



מבנה ה־Interrupt Vector Table בזיכרון

פסיקות DOS

מערכת ההפעלה DOS, קיצור של Disk Operating System, נכתבה על־ידי חברת מיקרוסופט. מערכת הפעלה זו שלטה בעולם מערכות ההפעלה עד אמצע שנות התשעים, אז הוחלפה על־ידי מערכת ההפעלה Windows.

מערכת ההפעלה מספקת לנו שירותים שונים. בין היתר היא מקשרת בין רכיבי החומרה במחשב לבין אפליקציות – תוכנות ומשחקים שרצים על המחשב שלנו. כך היא חוסכת עבודה למי שמתכנת אפליקציות, שבמקום לגשת ישירות לרכיבי החומרה, מקבל שירות ממערכת ההפעלה. יכולת זו גם מונעת התנגשות בין אפליקציות שצריכות להשתמש באותם רכיבים באותו זמן. לכן, אחד מהדברים שמערכת הפעלה מודרנית כוללת הוא אוסף של 'ISR'ים, שיודעים לעבוד מול רכיבי חומרה שונים.

ל־'ISR'ים של DOS הוגדרו מספר מקומות שמורים ב־IVT – טווח הפסיקות 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 שמוקצות ל־DOS בתוך ה־IVT

אחת מהפסיקות היא 21h, שמוגדרת כפסיקת שירות של מערכת ההפעלה. כעת, כל פעם שנרצה שירות של מערכת ההפעלה נוכל לבצע זאת על־ידי השורה int 21h. איך אפשר בעזרת פסיקת תוכנה אחת לבצע את כל השירותים שמערכת ההפעלה נותנת לנו? התשובה היא שלפני שאנחנו קוראים ל־int 21h אנחנו שמים ברגיסטרים פרמטרים שקובעים מה מערכת ההפעלה תבצע בשבילנו.

ספציפית, הרגיסטר ah מחזיק את סוג השירות שאותו אנחנו 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.

קליטת תו מהמקלדת – AH=1h

כדי לקבל בעזרת int 21h שירות של קריאת תו מהמקלדת, נשים בתוך ah את הקוד "1". כך:

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

התו שייקרא ייטען לתוך al.

יש לציין ש־al יכיל את ערך ה־ASCII של התו שהוקלד. לדוגמה, אם המשתמש הקליד "2", al לא יכיל 2, אלא 32h (כפי שאפשר לראות בטבלה למטה), שהוא ערך ה־ASCII של התו "2".

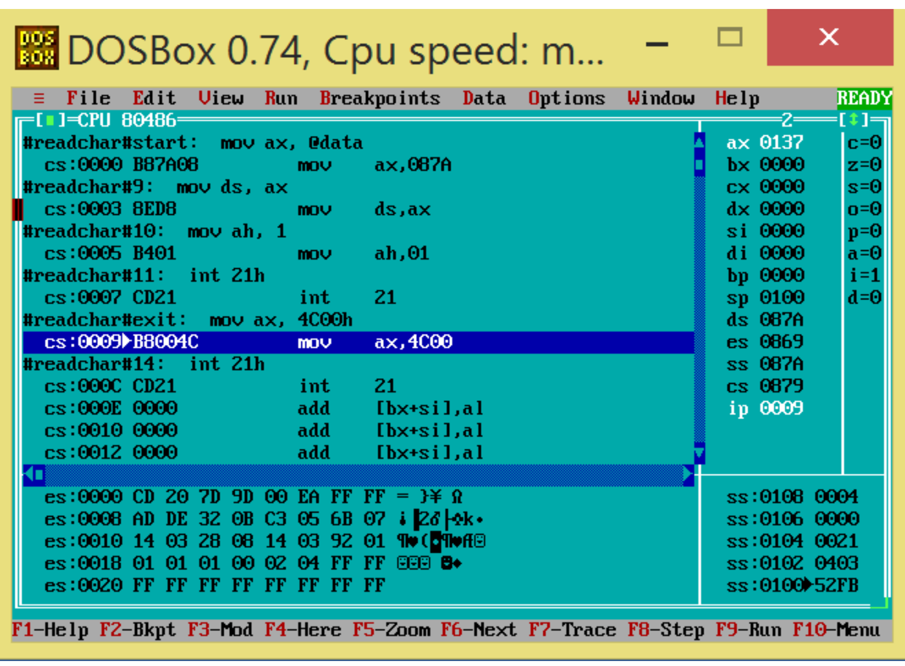
Regular ASCII Chart (character codes 0 - 127)																	
000		(nul)	016	►	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001	☺	(soh)	017	◄	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	☻	(stx)	018	↕	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	♥	(etx)	019	‼	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	♦	(eot)	020	¶	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005	♣	(enq)	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	↓	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010		(lf)	026		(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	♂	(vt)	027	←	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012	♀	(np)	028	∟	(fs)	044	,	060	<	076	L	092	\	108	l	124	¦
013		(cr)	029	↔	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	♫	(so)	030	▲	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	☼	(si)	031	▼	(us)	047	/	063	?	079	O	095	_	111	o	127	⌂

כדי ש־al יכיל ממש 2, או כל ספרה אחרת שהמשתמש הקליד, הטכניקה המקובלת היא להחסיר ממנו את ערך ה-ASCII של התו "0" (ערך זה הוא 30h).

```
sub     al, 30h
```

דוגמה לתוכנית שקולטת תו 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 תו מהמקלדת

א. כיתבו תוכנית שקולטת שני תווים ובודקת לאיזה תו יש ערך ASCII גדול יותר.

ב. כיתבו תוכנית שקולטת תו מהמקלדת ובודקת אם מדובר בספרה (כלומר ערך ה-ASCII נמצא בטווח ערכי ה-ASCII שבין 0 ל-9).

ג. כיתבו תוכנית שקולטת עשרה תווים מהמקלדת, ואם כולם ספרות התוכנית מחשבת את סכום הספרות לתוך הרגיסטר DL (טיפ: השתמשו בלולאה).

ד. אתגר: כיתבו תוכנית שקולטת מספר בן 4 ספרות. התוכנית תקלוט אותו ספרה אחרי ספרה ותשמור 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 פחות מ-30 שורות קוד.

הדפסת תו למסך – AH=2h

כדי לקבל בעזרת int 21h שירות של הדפסת תו 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 תשנה את ערכו של al לערך התו האחרון שנשלח להדפסה.

קיימים שני קודי ASCII שימושיים, שאינם מדפיסים תווים למסך אלא נותנים הוראות בקרה:

1. קוד 10, או Line Feed – 0Ah– מורה להתחיל שורה חדשה, הסמן נמצא באותה עמודה בה היה בשורה הישנה.
2. קוד 13, או Carriage Return – 0Dh– מורה לחזור לתחילת השורה.

הצירוף של שני הקודים האלה מאפשר לנו לרדת שורה ולהמשיך את ההדפסה מתחילת השורה.

דוגמה לתוכנית שמדפיסה 'X', יורדת שורה ומדפיסה 'Y':

```
IDEAL
MODEL  small
STACK       100h
DATASEG
CODESEG
start:
        mov     ax, @data
        mov     ds, ax
        ;print x
        mov     dl, 'X'
        mov     ah, 2
        int     21h
        ;new line
        mov     dl, 10
        mov     ah, 2
        int     21h
        ;carriage return
        mov     dl, 13
        mov     ah, 2
        int     21h
        ;print y
        mov     dl, 'Y'
        mov     ah, 2
        int     21h
exit:
        mov     ax, 4C00h
        int     21h
END     start
```

DOSBox 0.74, Cpu speed: ...

```
C:\TASM\BIN>tasm /zi putchar.asm
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland International

Assembling file:   putchar.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  466k

C:\TASM\BIN>tlink /v putchar.obj
Turbo Link  Version 7.1.30.1. Copyright (c) 1987, 1996 Borland International

C:\TASM\BIN>putchar
X
Y
C:\TASM\BIN>_
```

תרגיל 11.2: הדפסת תו למסך

א. הדפיסו למסך את התו 'A'.

ב. הדפיסו למסך את התו 'a'.

ג. הדפיסו למסך את המילה 'HELLO', תו אחר תו.

ד. הדפיסו למסך את המילה 'HELLO', תו אחר תו, בשורה אחת, ואת המילה 'WORLD', תו אחר תו, בשורה מתחתיה.

ה. כיתבו תוכנית שקולטת שתי ספרות ומדפיסה למסך את הספרה הגדולה מביניהן.

ו. כיתבו תוכנית שקולטת שתי ספרות ומדפיסה למסך את החיסור 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 שמתקבלת. לדוגמה – הספרה הראשונה היא 5 והשניה 7. התוכנית תמצא שהמספר הראשון יותר קטן מהשני, תדפיס מינוס ואחר כך את ההפרש בין המספר השני לראשון.

הדפסת מחרוזת למסך – AH=9h

כדי להדפיס מחרוזת למסך אנחנו צריכים קודם כל ליצור מחרוזת, שמסתיימת בתו '$' (זה הסימן ל-ISR להפסיק את ההדפסה למסך – אחרת יודפס כל הזיכרון...). לדוגמה:

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

לאחר מכן טוענים לתוך dx את האופסט של המחרוזת:

```
mov     dx, offset message
```

נותר לנו רק לתת את הקוד הנכון ולהפעיל את הפסיקה:

```
mov     ah, 9h

int     21h
```

ירידת שורה בסוף המחרוזת:

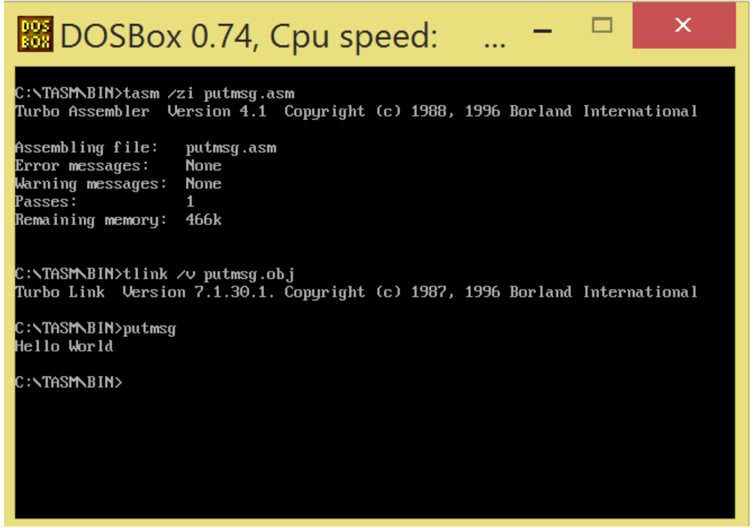
שיטה נוחה לכתוב מחרוזת שבסוף ההדפסה שלה מתחילה שורה חדשה, היא להגדיר את תווי הבקרה בתוך המחרוזת:

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

תוכנית דוגמה:

```
IDEAL

MODEL  small

STACK          100h

DATASEG

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


CODESEG

start:

        mov     ax, @data

        mov     ds, ax

        push    seg message

        pop     ds

        mov     dx, offset message
```

```
        mov     ah, 9h
        int     21h
exit:
        mov     ax, 4C00h
        int     21h
END     start
```



תרגיל 11.3: הדפסת מחרוזת למסך

א. כיתבו תוכנית שמדפיסה למסך את ההודעה: 'Enter a digit' ואז קולטת ספרה מהמשתמש.

ב. הרחיבו את התוכנית שכתבתם בסעיף הקודם, כך שאם המשתמש הזין תו שאינו ספרה בין 0 ל-9, יודפס למסך: "Wrong input".

ג. כיתבו תוכנית שמדפיסה למסך הודעה כלשהי, ובשורה הבאה כותבת את שמכם. לדוגמה:

I like to write assembly code

Barak

קליטת מחרוזת תווים – AH=0Ah

כדי לקלוט מחרוזת תווים מהמקלדת, צריך קודם כל להכין בזיכרון מקום לקליטת התווים- מקום זה נקרא "באפר" (Buffer). לתוך הבית הראשון של הבאפר צריך להכניס את מספר התווים המקסימלי שאנחנו מאפשרים לקלוט. לאחר מכן מכניסים לתוך dx את האופסט של הבאפר (יחסית ל-ds) ואז מפעילים את Int 21h כאשר ah שווה לקוד 0Ah. כשהמשתמש סיים

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

גם קוד ה-ASCII של Enter נכנס אל הבאפר.

דוגמה לתוכנית שקולטת עד 20 תווים מהמשתמש:

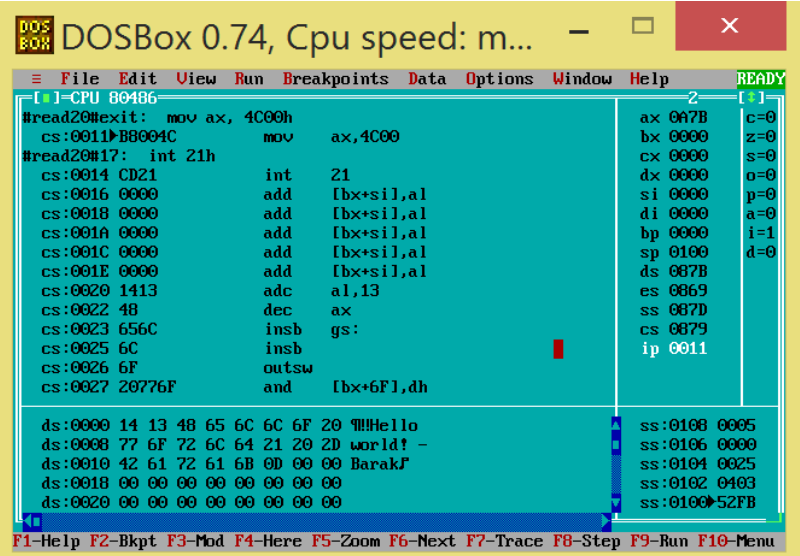
(שימו לב - אנחנו מקצים 23 בתים, כיוון ששני בתים בתחילת הבאפר תפוסים על-ידי מספר התווים המקסימלי ומספר התווים בפועל, והבית האחרון תפוס על-ידי קוד ה-ASCII של Enter):

```
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
```

להן צילום מסך של הטקסט שהוכנס על ידי המשתמש:

```
DOSBox 0.74, Cpu speed: m...
C:\TASM\BIN>tasm /zi read20.asm
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland International

Assembling file:   read20.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  466k


C:\TASM\BIN>tlink /v read20.obj
Turbo Link  Version 7.1.30.1. Copyright (c) 1987, 1996 Borland International

C:\TASM\BIN>td read20
Turbo Debugger Version 5.0 Copyright (c) 1988,96 Borland International
Hello world! -Barak
```

הטקסט נשמר במשתנה message, כפי שאפשר לראות בתוך ds. הבית הראשון, 14h, הוא גודל ההודעה המקסימלית שהתוכנית שלנו מאשרת. הבית השני, 13h, הוא כמות התווים שהוכנסו בפועל, כולל ה־enter. הבית האחרון בהודעה, שנראה כמו סימן של צליל, ערכו 0Dh והוא קוד ה־ASCII של enter.



תרגיל 11.4: קליטת מחרוזת תווים

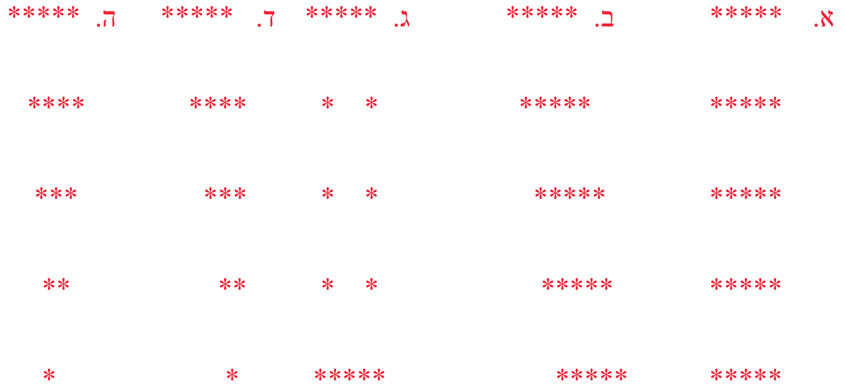
כיתבו תוכנית שקולטת עד 10 תווים מהמשתמש, באותיות קטנות (abc) ומדפיסה למסך את התווים באותיות גדולות (ABC).

תרגילים מסכמים- קלט פלט (קרדיט: פימה ניקולייבסקי)

11.5: כיתבו תכנית המציגה על המסך את שם המשפחה ואת שמכם הפרטי.

11.6: כיתבו תכנית שקולטת תו בודד מהמקלדת ומציגה אותו 10 פעמים בשורה חדשה.

11.7: כיתבו תכנית המציגה על המסך את הצורות הבאות (כל צורה – תכנית חדשה!)



11.8. כיתבו תכנית המציגה על המסך את רצף של תווים הבאים:

ABCDEFGE….Z

11.9: כיתבו תכנית ובה פעולות:

א. לקליטת מספר שלם מן המקלדת. סוף הקלט לחיצה על מקש אנטר.

ב. להצגת מספר שלם על המסך.

ג. קליטת מספר שלם שמתאר את מספר הנתונים שיש לקלוט למערך וקליטת מספרים למערך הזה.

ד. להצגה של ערכי מערך על המסך.

11.10: כיתבו תוכנית הקולטת תו אחרי תו עד הלחיצה על המקש אנטר מהמקלדת ועבור כל תו מציגה הודעה מתאימה:

Small letter, Capital letter, Number, Other

11.11: כיתבו תכנית הקולטת רצף של 5 תווים מהמקלדת "קוד סודי", במקום כל תו שנקלט מוצג תו "*". על התכנית לבדוק האם הקוד הוא: "12345" ואם כן - להציג הודעה מתאימה. אחרת, להודיע על הטעות ולאפשר עוד 2 נסיונות.

יציאה מהתוכנית – AH=4Ch

קריאה ל־int 21h עם קוד 4Ch גורמת לשחרור הזיכרון שתפוס על־ידי התוכנית וסגירת כל הקבצים שנפתחו על־ידי התוכנית. כמו כן הפסיקה מעבירה למערכת ההפעלה את מה ששמור בתוך al בתור קוד חזרה. נהוג לקבוע את ערכו של al כאפס במידה והתכנית הסתיימה בהצלחה, ולכן יציאה סטנדרטית מתוכנית נראית כמו שורות הקוד המוכרות לכם מ־base.asm:

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

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

לצד המעבד קיים רכיב חומרה שתפקידו לתת תזמונים למעבד – הטיימר. הטיימר שולח למעבד אות חשמלי 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, כחלק מ- int 21h, ולשלוח לו את הקוד 2Ch:

```
mov     ah, 2Ch
int     21h
```

תוצאת הקריאה תהיה:

- השעות יועתקו לתוך ch.
- הדקות יועתקו לתוך cl.
- השניות יועתקו לתוך dh.
- מאיות השניה יועתקו לתוך dl.

שימו לב לכך שערך מאיות השניה מתעדכן רק כל 55 מילישניות. כלומר אם קראנו את השעון פעמיים, ובין הקריאות האלו עברו פחות מ־55 מילישניות, יכול להיות שנקבל 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 הטיימר לפי הפרמטרים ברגיסטרים השונים:

- ch יועתק לתוך השעות.
- cl יועתק לתוך הדקות.
- dh יועתק לתוך השניות.
- dl יועתק לתוך מאיות השניה.

תוכנית לדוגמה – קריאת הטיימר והדפסת השעה למסך:

התוכנית הבאה משתמשת ב-int 21h כדי לקרוא את ערכו של הטיימר. לאחר מכן נעשה שימוש בפרוצדורה שממירה את הערכים שברגיסטרים לקוד ASCII של ספרות ומדפיסה אותם למסך.

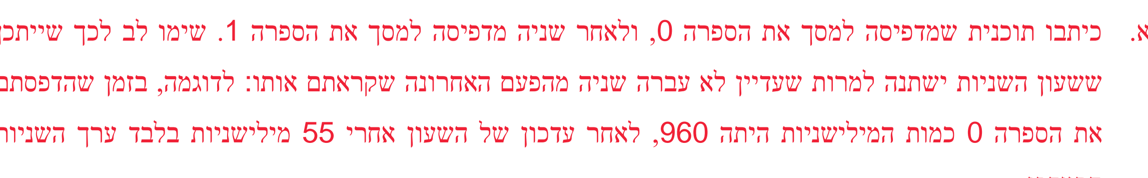
```
; --------------------------------------------------------------------------------
; Print time to screen
; Author: Barak Gonen 2014
; Credit: www.stackoverflow.com (printing-an-int, by Brendan)
; --------------------------------------------------------------------------------
IDEAL
MODEL       small
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- hundreths 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. כיתבו תוכנית שמדפיסה למסך את הספרה 0, ולאחר שניה מדפיסה למסך את הספרה 1. שימו לב לכך שייתכן ששעון השניות ישתנה למרות שעדיין לא עברה שניה מהפעם האחרונה שקראתם אותו: לדוגמה, בזמן שהדפסתם את הספרה 0 כמות המילישניות היתה 960, לאחר עדכון של השעון אחרי 55 מילישניות בלבד ערך השניות התעדכן.

ב. כיתבו תוכנית שמדפיסה למסך את השעה, הדקה והשניות. המסך יעודכן בכל מעבר של שניה. התוכנית תסתיים לבד לאחר פרק זמן מוגדר כרצונכם.

פסיקות חריגה – Exceptions

פסיקות החריגה נמצאות בתחילת ה־IVT. פסיקה מסוג exception מתרחשת אוטומטית כתוצאה מאירוע תוכנתי שהמתכנת לא יזם אותו – בדרך כלל אירוע חריג, ולכן 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
```

האסמבלר יאפשר לכם לקמפל את התוכנית בלי בעיה, כיוון שמבחינתו כל אחת מהפקודות שרשמתם היא חוקית. אך בשלב ההרצה התוכנית תיעצר ותופעל פסיקת חריגה של חילוק באפס.

פסיקה זו, של חילוק באפס, נקראת פסיקה מספר אפס או int 0h.

דוגמה שניה לפסיקת חריגה היא כאשר אנחנו ב־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 ברגיסטר הדגלים. המעבד יודע שאם דגל זה מופעל, עליו לעצור את הריצה עד לקבלת פקודת "המשך".

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

דוגמה נוספת היא השימוש ב־**breakpoints** בתוך ה־debugger. כל breakpoint גורם למעבד להריץ את הפקודות בלי עצירה עד להגעה לשורת קוד שיש בה breakpoint ואז מתבצעת עצירה עד לקבלת פקודת "המשך".

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

פסיקות תוכנה – Traps

בניגוד ל־exceptions, שמתרחשות אוטומטית, מי שיוזם פסיקות תוכנה הוא מי שכותב את התוכנית. הדרך שבה מופעלת פסיקת תוכנה היא פשוטה ביותר, ועקרונית לא צריך לדעת הרבה בשביל לעבוד עם פסיקות תוכנה:

כותבים פקודת int ואחריה אופרנד, שהוא מספר הפסיקה:

```
int     operand
```

לדוגמה:

```
int     80h
```

מכאן והלאה התהליך של ביצוע פסיקת התוכנה הוא זהה לתהליך שתואר עד כה – שמירת המיקום והדגלים במחסנית, חישוב כתובת ה־ISR בעזרת ה־IVT וקפיצה למיקום ה־ISR.

בשביל מה בכלל צריך פסיקות תוכנה? הרי אם המתכנת רוצה שבנקודה ידועה בתוכנית המעבד יקפוץ למקום אחר, יבצע את הקוד שכתוב שם ויחזור- אפשר פשוט להשתמש בפרוצדורה.

התשובה היא, שיש מצבים שבהם לא מעשי להשתמש בפרוצדורה ובמצבים אלו פסיקת תוכנה היא שימושית. פסיקות תוכנה הן דרך נוחה לתת לתוכנה שלנו להריץ קוד של תוכנות שרצות במקביל לתוכנית שלנו – הדוגמה הבולטת ביותר היא מערכת ההפעלה. כשהתוכנה שלנו רוצה להשתמש בשירותים של מערכת ההפעלה, היא לא יודעת את הכתובות של הפרוצדורות של מערכת ההפעלה. תיאורטית היינו יכולים לפתור את הבעיה הזו אם היינו מקמפלים את התוכנה שלנו יחד עם הקוד של מערכת ההפעלה, אבל אופציה זו לא מעשית. לכן בפועל מערכת ההפעלה מגדירה את הפרוצדורות שלה כפסיקות ומכניסה ל־IVT את המיקום בזיכרון אליו צריך לקפוץ. כעת כל מה שנשאר לתוכנה שלנו לעשות הוא לקרוא לפסיקת תוכנה וה־IVT כבר ידאג "להקפיץ" את הקוד שלנו למיקום הנכון.

בסעיף הבא נלמד לכתוב Trap.

כתיבת ISR (הרחבה)

כדי להדגים את שלבי כתיבת ה־ISR נכתוב ISR פשוט מסוג Trap, שכל מה שהוא עושה זה להדפיס למסך 'Hello World'.

שלב א' – כתיבת ה-ISR.

נתחיל מ־ISR ריק:

```
proc    SimpleISR far
        …
        iret
endp    SimpleISR
```

נוסיף ל־ISR שלנו את ההודעה אותה אנחנו רוצים להדפיס.

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

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

נוסיף קריאה ל־int 21h. הקוד שמדפיס מחרוזת הוא ah=9h, ולפני הקריאה צריך לדאוג לכך שב־ds יהיה הסגמנט של המחרוזת וב־dx יהיה האופסט 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 הריצה מצב הרגיסטרים יהיה כמו לפני הקריאה. גם בלי פקודות אלו ה־ISR יעבוד, אבל התוכנית שקוראת לו עלולה לא לתפקד כמו שצריך אחרי שהשליטה תחזור 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 יעבוד, הוא צריך להיות מוכר על-ידי ה-IVT. כלומר, אנחנו צריכים להוסיף את כתובת ה-ISR לתוך ה-IVT. נבחר לשתול את כתובת ה-ISR במקום האחרון ב-IVT, מקום 255 (0FFh). הסיבה שבחרנו דווקא במקום זה, היא שאנחנו רוצים לשתול את ה-ISR שלנו במקום שאינו תפוס על-ידי כתובת של ISR שבשימוש, ובסוף ה-IVT יש מקומות פנויים (חישבו- מה היה קורה אילו היינו שותלים את ה-ISR שלנו במקום 21h?)

כדי לשתול ב-IVT את הכתובת של ה-ISR החדש שלנו, נשתמש בשירות נוסף שמבצע int 21h, שירות שזה הזמן המתאים לסקור אותו. קוד AH=25h מכניס כתובת לתוך ה-IVT. מספר הפסיקה צריך להיות בתוך al, וכתובת ה-ISR בתוך ds:dx. שורות הקוד הבאות מבצעות את הפעולות הדרושות:

```
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. כיתבו ISR שמקבל בתוך al ערך ASCII של תו, מעלה אותו באחד ומדפיס למסך את התו החדש. גירמו לכך שהוא יופעל על־ידי הפקודה int 0FEh.

ב. כיתבו ISR שמקבל שני רגיסטרים ax, bx ומדפיס למסך:

- 'ax' אם ax גדול מ-bx.
- 'bx' אם bx גדול מ-ax.

- 'SAME' אם הם שווים.

גירמו לכך שהוא יופעל על־ידי הפקודה int 0F0h.

תרגיל אתגר- פיצוח צופן הזזה

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

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

1. כיתבו פרוצדורה שמקבלת מחרוזת ומרחק הזזה, ומצפינה את המחרוזת לפי מרחק ההזזה. תווי רווח יש להשאיר כמו שהם.

2. לפניכם העמוד הראשון מתוך הספר הנפלא Anna Karenina מאת טולסטוי. כדי להקל על תהליך ההצפנה הכתוב מובא רק עם אותיות קטנות וללא סימני פיסוק. הצפינו אותו בעזרת צופן הזזה. שימו לב לכך שבסוף העמוד ישנו סימן '$'. סימן זה נועד גם להקל עליכם למצוא את סוף המחרוזת וגם להקל עליכם בהדפסה של התוצאה. הדפיסו למסך את הטקסט המוצפן.

```
all happy families resemble one another every unhappy family is unhappy in its own way
all was confusion in the house of the oblonskys
the wife had discovered that her husband was having an intrigue with a french governess who had been in
their employ and she declared that she could not live in the same house with him
this condition of things had lasted now three days and was causing deep discomfort not only to the husband
and wife but also to all the members of the family and the domestics
all the members of the family and the domestics felt that there was no sense in their living together and
that in any hotel people meeting casually had more mutual interests than they the members of the family
and the domestics of the house of oblonsky
the wife did not come out of her own rooms
the husband had not been at home for two days
the children were running over the whole house as if they were crazy
the english maid was angry with the housekeeper and wrote to a friend begging her to find her a new place
the head cook had departed the evening before just at dinner time
the kitchen maid and the coachman demanded their wages$
```

3. כיתבו פרוצדורה שמקבלת תו ומחרוזת, ומחזירה כמה פעמים מופיע התו בתוך המחרוזת.

4. באמצעות הפרוצדורה שכתבתם, סיפרו כמה פעמים מופיעה כל אות באלף בית הלועזי בטקסט המוצפן. פעולה זו נקראת ניתוח תדירויות. הדפיסו למסך את התוצאה עבור כל אות.

5. צרו פרוצדורה שמקבלת את ניתוח התדירויות שעשיתם, ובאמצעות טבלה של ניתוח תדירויות שבשפה האנגלית מחליטה מה היתה האות המקורית שהוצפנה. מצורפת טבלה של ניתוח תדירויות אותיות 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 מהסבר על הצורך בפסיקות, ועל כך שלא תמיד אפשר לצפות מראש מתי התוכנה תצטרך לבצע פעולה מסויימת. לאחר מכן למדנו על ה־ISR, הפרוצדורה לטיפול בפסיקות, ועל ה־IVT, המערך בזיכרון שמכיל את המידע לגבי מיקום כל ה-'ISR'ים.

למדנו על פסיקה 21h של מערכת ההפעלה DOS. סקרנו את הקודים השימושיים לביצוע פעולות שונות כגון קליטה של תו ומחרוזות, הדפסה של תו ומחרוזות וקריאת השעה מהשעון הפנימי של המחשב.

ראינו איך כותבים ISR ואיך שותלים אותו ב־IVT.

עסקנו בצורות שונות של פסיקות:

- פסיקות תוכנה (Traps)
- פסיקות חריגה (Exceptions)

על הסוג האחרון של הפסיקות, פסיקות חומרה (Interrupts), נפרט בפרק הבא.

פרק 12 – פסיקות חומרה (הרחבה)

מבוא

כשעסקנו בנושא הפסיקות ראינו שקיימות פסיקות DOS שמבצעות פעולות שונות מול רכיבי חומרה, לדוגמה קריאת תו מהמקלדת. כמו כן למדנו להשתמש בכמה פסיקות DOS שימושיות. עם זאת, לא עסקנו בשאלה איך המידע מהתקני החומרה מגיע אל המעבד? איך, לדוגמה, הקשה על המקלדת גורמת להופעת תו בזיכרון המחשב? בפרק זה נבין יותר לעומק את הדרך שבה המעבד מתקשר עם התקני החומרה.

כדי להבין זאת אנו נלמד:

- תיאוריה של פסיקות חומרה – מדוע צריך פסיקות חומרה ובאיזו דרך מגיעות פסיקות החומרה אל המעבד.
- פורטים I/O Ports – תאים מיוחדים בזיכרון שמשמשים לעבודה מול התקני קלט / פלט.

לאחר מכן, נעבור לדוגמה מעשית על עבודה מול התקני חומרה ונפרט אודות פעולת המקלדת, תוך סקירת הדרכים השונות לעבודה מולה:

- קליטת המידע דרך הפורטים של המקלדת
- שימוש בפסיקות BIOS
- שימוש בפסיקות DOS

פסיקות חומרה – Interrupts

המעבד שלנו מקושר להתקני חומרה שונים כגון מקלדת, עכבר, שעון המערכת (טיימר) ועוד. התקנים חיצוניים אלו מייצרים אירועי חומרה. ארוע חומרה יכול להיות לחיצה על המקלדת, הזזת העכבר או עדכון של הטיימר. אירועים החומרה האלו דורשים שירות מצד המעבד – לדוגמה, לקחת את התו שהוקלד במקלדת ולהדפיס אותו למסך, להזיז את הסמן של העכבר על המסך או לעדכן את השעה שמוצגת על המסך. מבחינת המעבד, כל האירועים האלו בלתי צפויים מראש – המעבד לא יכול לתכנן מראש מתי תהיה לחיצה על המקלדת ובאיזה קצב המשתמש יקליד תווים. גם מי שכותב את התוכנה לא יכול לדעת מראש באיזה תזמון יקרה אירוע חומרה ולהתכונן אליו. אם כך, איך המידע שמגיע מהתקנים חיצוניים מועבר לתוכנה בזמן ריצה?

קיימות שתי גישות לפתרון 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

התקן חומרה שצריך שירות. המקלדת צריכה שירות? אם כן – המעבד מטפל בה, אם לא – ממשיך לבדוק אם העכבר צריך שירות. כך המעבד עובר הלאה על כל ההתקנים עד שחוזר אל ההתקן הראשון.

היתרון של Polling הוא שזו גישה פשוטה יחסית למימוש, ולא צריך רכיבי חומרה נוספים שיעזרו לנו בתהליך (להבדיל מפסיקות חומרה, שנגיע אליהן מיד). החיסרון של Polling הוא שהתקן החומרה צריך להמתין שיפנו אליו. כלומר 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 בעייתי, מקרים שבהם הפעולה התקינה של התוכנה שלנו תלויה בזמן שלוקח לה לשרת אירועים מהתקן חומרה כלשהו. לדוגמה, במערכות שמבצעות מסחר אלקטרוני במניות, יכולים להיות הבדלים של פרקי זמן קצרים ביותר בין הוראות קניה ומכירה והתוכנה חייבת לדעת מה הסדר הנכון של ההוראות. לכן, עליה לשמור על קצב גבוה של טיפול בפניות מהטיימר. דוגמה אחרת – נניח שיש מערכת שמשגרת טיל ברגע שמישהו לוחץ על כפתור במקלדת, במקרה זה מובן שלמהירות הטיפול באירוע הלחיצה על המקלדת יש חשיבות רבה. פתרון פשוט הוא להעלות את קצב התשאול של התקני החומרה. נניח שבמקום לשאול את המקלדת 10 פעמים בשניה אם יש מקש לחוץ, התוכנה תישאל את המקלדת 1000 פעמים בשניה אם יש מקש לחוץ. שינוי זה יוריד את הזמן שהמקלדת ממתינה לשירות מעשירית שניה לאלפית שניה. הבעיה היא, שככל שהתוכנה שלנו מתשאלת את רכיבי החומרה לעיתים קרובות יותר, כך היא מבזבזת זמן רב יותר בבדיקה של התקנים, שלרוב לא יהיה להם צורך בשירות. כלומר התוכנה שלנו תגרום לכך שהמעבד יקצה חלק משמעותי מהזמן לבדיקת התקני חומרה ולא לפעולות אחרות שאנחנו רוצים שיבצע.

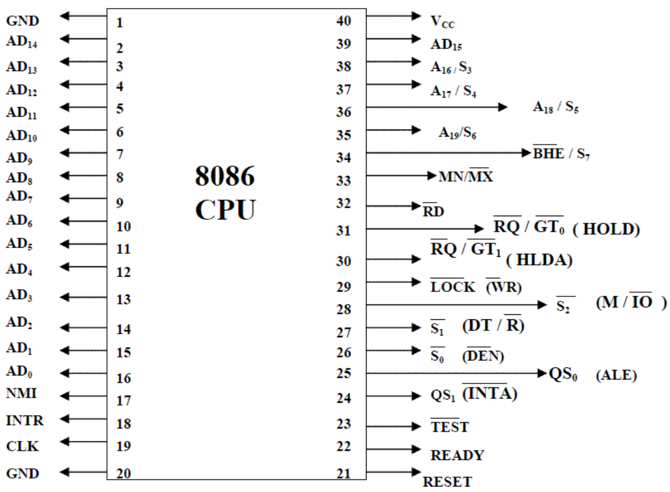
גישת Polling – אחת לזמן מוגדר, המורה שואלת כל תלמיד אם יש לו שאלה. כל עוד המורה לא פונה אליהם, התלמידים מחכים.

הגישה השניה היא פסיקת חומרה, שנקראת **אינטרפט (Interrupt)**. מעכשיו כשנכתוב "אינטרפט" נתכוון לפסיקת חומרה. כשיש להתקן החומרה צורך בשירות של המעבד, התקן החומרה שולח אות חשמלי שמגיע אל המעבד וגורם לו לעצור לאחר סיום הפעולה הנוכחית שלו ולבצע קוד, ISR, שנכתב מראש לטיפול באירוע החומרה. בסיום ה-ISR התכנית תמשיך בריצתה. היתרון של Interrupt הינו שה-Interrupt גם שומר על שיהוי נמוך (בקשות של התקני חומרה מטופלות עם שיהוי נמוך יחסית) וגם חסכוני במשאבי מעבד – כל עוד לא מגיע אינטרפט, המעבד פשוט מריץ את הקוד שנקבע לו להריץ בלי לדאוג לגבי התקני החומרה.

גישת Interrupt – תלמיד שיש לו שאלה מרים יד. המורה עוצרת את השיעור, עונה לתלמיד וממשיכה את השיעור מהמקום שהפסיקה אותו.

כמו שתלמיד שיש לו שאלה מרים יד, רכיב חומרה שצריך טיפול שולח אות חשמלי. למעבד יש "רגליים", או "פינים" – חוטי מתכת דקים שמחברים אותו אל העולם החיצון. שינוי המתח החשמלי על אחת הרגליים גורם למעבד להניע תהליך של אינטרפט.

Pin Diagram of 8086



דיאגרמת פינים של מעבד ה-8086

פין מספר 18, שמסומן "INTR", מקבל אינטרפטים מרכיבי חומרה חיצוניים

בקר האינטרפטים – PIC

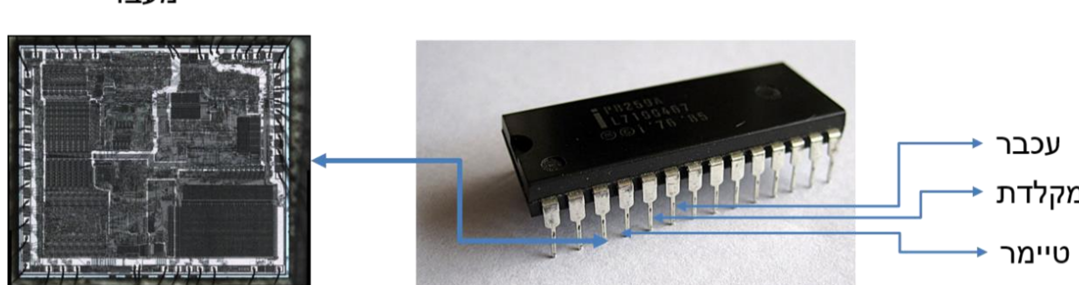
כפי שראינו באיור של מעבד ה־8086, יש למעבד רגל אחת שמיועדת לאינטרפטים. בעזרת הרגל הזו המעבד מקבל אינטרפטים מכל ההתקנים שקשורים אליו. איך בעזרת רגל אחת המעבד יכול להיות קשור למספר רב של התקני חומרה?

שימוש באינטרפטים מצריך רכיב חומרה נוסף. רכיב זה הוא **בקר האינטרפטים**, **Programmable Interrupt Controller**, או בקיצור **PIC**.

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

כפי שאפשר לראות בתמונה, ל־PIC יש אוסף של רגליים. הרגליים שמסומנות IR0 עד IR7, בסך הכל 8 רגליים, כאשר כל אחת מהן יכולה להיות מקושרת לרכיב חומרה בודד. כלומר PIC מדגם 8259A מסוגל לעבוד עם עד 8 רכיבי חומרה במקביל. לדוגמה – רגל IR0 מחוברת לטיימר, רגל IR1 מחוברת למקלדת, רגל IR6 מחוברת לכונן דיסקטים, ורגל IR2 מחוברת לעכבר (לא באופן ישיר, אבל לצורך הפשטות נכון להגיד שאינטרפטים מהעכבר מגיעים ל-IR2).

ל־PIC יש גם רגל שנקראת INT. רגל זו מחוברת חשמלית אל הרגל של המעבד שנקראת INTR. כעת אפשר להתבונן במסלול שעושה אינטרפט. ניקח בתור דוגמה אינטרפט של המקלדת. האות החשמלי עובר מהמקלדת אל רגל IR1 של ה־PIC. כתגובה, ה־PIC מוציא אות חשמלי מרגל INT, שמגיעה לרגל INTR של מעבד ה-8086.



עד עכשיו תיארנו את ה־PIC כגורם מקשר בין מספר רכיבי חומרה למעבד. אולם ה־PIC לא רק מקשר, אלא גם קובע עדיפויות לטיפול בהתקני החומרה. דמיינו מערכת שמקושרת לשני התקני חומרה: מקלדת, וכור גרעיני. המשתמש הקיש על המקלדת והכור הגרעיני מדווח על בעיה... במי המעבד צריך לטפל קודם?

הטיפול לא מתבצע בשיטת כל הקודם זוכה. ה־PIC מחזיק תור של אינטרפטים שמחכים לטיפול. אם הגיע אינטרפט חדש, הוא לא יידחף לסוף התור אלא ייכנס לתור בהתאם לעדיפות שלו. ככל שהאינטרפט הגיע מרגל בעלת מספר נמוך יותר, כך העדיפות שלו גדלה.

עם קבלת אינטרפט ה־PIC מבצע את הפעולות הבאות:

- שולח למעבד אות חשמלי שמסמן שיש אינטרפט.
- שולח מידע שקשור לאינטרפט לאזור מיוחד בזיכרון המעבד, אזור שנקרא I/O Port או בקיצור – פורט.
- מפסיק לשלוח אינטרפטים למעבד.
- מחכה לאות חשמלי end of interrupt מהמעבד, ובינתיים שומר בתור אינטרפטים חדשים שמגיעים אליו מרכיבי חומרה.
- חוזר לשלוח אינטרפטים למעבד.

אובדן אינטרפטים

כזכור, כשהמעבד מטפל בפסיקות הוא מבצע פעולה של disable interrupts על־ידי איפוס דגל הפסיקות. פעולה זו היא בעלת משמעות מיוחדת עבור התקני חומרה. בניגוד לפסיקות תוכנה, שהתוכנית מפעילה באופן צפוי מראש (כיוון שהמתכנת קבע אותן בקוד), פסיקות חומרה מגיעות בזמנים לא צפויים מראש. מה קורה אם בזמן שהמעבד מטפל באינטרפט מגיע אינטרפט נוסף? לדוגמה – לחצנו על המקלדת והעכבר בהפרש זמן קטן ויצרנו אינטרפט אחד מהמקלדת ואינטרפט אחד מהעכבר. האינטרפט מהעכבר מתרחש בזמן שבו המעבד כבר מטפל באינטרפט של המקלדת וחוסם אינטרפטים נוספים. האם האינטרפט של העכבר "יילך לאיבוד"?

ובכן, כפי שציינו קודם, ה־PIC מרכז את האינטרפטים של כל רכיבי החומרה ושומר אותם אצלו בתור. בדוגמה שלנו, עם סיום האינטרפט של המקלדת, ה־PIC יעביר למעבד את האינטרפט של העכבר. התור של ה־PIC עוזר במצב שבו מספר

אינטרפטים הגיעו בהפרש זמנים קצר. לעומת זאת, אם המעבד לא מצליח לעמוד בקצב הפסיקות, התור של ה־PIC ילך ויתארך עד לנקודה מסויימת בה לא יהיה ל־PIC מקום לשמור אינטרפטים חדשים והם ילכו לאיבוד.

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

זיכרון קלט / פלט – I/O Ports

בפרק אודות מבנה המחשב סקרנו את הפסים (buses) השונים שיש למעבד, ושמשמשים אותו לתקשורת עם הזיכרון ועם רכיבי הקלט / פלט. בין היתר, סקרנו את פס המענים – address bus – שמתרגם כל בית בזיכרון לכתובת.

למעשה, למעבד יש שני פסי מענים. נוסף על פס המענים שמשמש לגישה לזיכרון, משפחת ה־80x86 כוללת פס מענים מיוחד ונפרד בגודל 16 ביט, שמשמש לתקשורת עם רכיבי חומרה. את הזיכרון שמשמש לתקשורת עם רכיבי חומרה מכנים גם זיכרון קלט / פלט, או I/O (קיצור של Input / Output).

התקשורת עם זיכרון I/O עובדת בצורה דומה מאוד לתקשורת עם הזיכרון הרגיל, למעט מספר הבדלים:

- כתובת של זיכרון I/O נקראת **פורט (Port)**.
- כתובת של זיכרון I/O מיוצגת על-ידי 16 ביט בלבד (כלומר יש בסך הכל 64K פורטים).
- במקום פקודת mov, משתמשים בפקודות in ו־out.
- פס הבקרה דואג שפקודות in ו־out יעבירו כתובות על פס המענים של ה־I/O (לעומת פקודות mov, שמתייחסות לפס המענים של הזיכרון).

פקודות in, out: פקודת in משמשת לקריאה מפורט, ופקודת out משמשת לכתיבה לפורט. פקודת in מעתיקה זיכרון בגודל מילה או בית מפורט מסויים אל ax או al. פקודת out מעתיקה זיכרון בגודל מילה או בית מ־ax או al אל פורט.

- יש שתי שיטות כתיבה של מספר הפורט בפקודות in, out:
  - ישירות: רושמים את מספר הפורט, בתנאי שהוא בין 0 ו־255 בלבד.
  - בעקיפין: אם אנחנו רוצים לתקשר עם פורט בתחום שמעל 255, במקום פורט 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 in / out 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
```

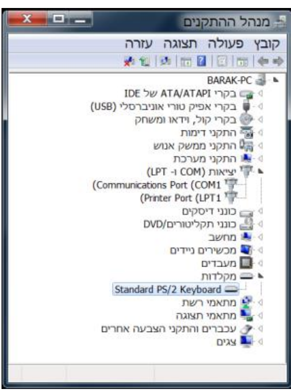
משמעות הביטים ששינינו איננה חשובה כרגע, (נגיע אליה בהמשך, כשנעסוק בפרוייקטי סיום). החשיבות היא בתהליך – אנחנו קוראים באמצעות פקודת in את הסטטוס של פורט כלשהו, משנים בו כמה ביטים וכותבים חזרה את הסטטוס אל הפורט על־ידי פקודת out.

דוגמה לעבודה מול פורט שמספרו גדול מ־255:

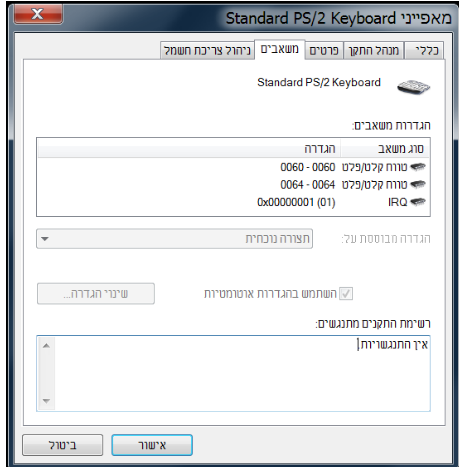
```
mov     dx, 300h
in      al, dx
```

כיוון שמספר הפורט גדול מ־255, לא ניתן לפנות אליו ישירות אלא נדרש שימוש ב-dx לשמירת מספר הפורט.

דוגמה – מציאת פורט התקשורת עם המקלדת: ניכנס למנהל ההתקנים (חיפוש תכנית-> Device manager).



בתוך מנהל ההתקנים, נסמן את המקלדת ונקיש על עכבר ימני – נבחר "מאפיינים" (Properties). בתוך מאפיינים, נבחר "משאבים". ניתן לראות שהמעבד מתקשר עם המקלדת דרך פורט 60 ופורט 64.



המקלדת

הקדמה

בחלק זה נפרט את שרשרת הפעולות שמתבצעת מרגע ההקשה 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 ושחרור של מקשים. מידע זה נקרא Scan Codes.
2. המקלדת מקושרת לזיכרון ה-I/O של המעבד והמידע שמגיע ממנה מועתק אל פורט קבוע בזיכרון, פורט 60h.
3. פסיקת חומרה, אינטרפט, אוספת את ה־scan codes מפורט 60h ומעתיקה אותם אל באפר מיוחד שמוגדר לשמירת המידע מהמקלדת. מספרו של האינטרפט הוא 9h והשם של הבאפר של המקלדת הוא Type Ahead Buffer.

כמו שאמרנו במבוא לנושא הפסיקות, למי שמתכנת באסמבלי יש יותר מדרך אחת לעבוד מול חומרה. ספציפית לגבי המקלדת יש שלוש דרכים לעבוד מולה:

1. לעבוד ישירות מול הפורט של המקלדת בשביל לקבל את המידע על המקשים שהוקשו.
2. להשתמש בפסיקה של BIOS (תזכורת – BIOS היא חבילת תוכנה של אינטל, שנסקרה בקצרה בראשית הפרק הקודם. אחד מתפקידיה הוא קישור אל רכיבי חומרה). ל־BIOS יש פסיקה מספר 16h שנותנת לנו לבצע פעולות שונות מול המקלדת – כמו קליטת תו – בפשטות יחסית.
3. להשתמש בפסיקה של DOS, פסיקה 21h, שכבר הכרנו חלק מהשירותים שהיא נותנת 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, כשאנחנו 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.

הסימון הזה מבוצע על־ידי טבלת קודים, שנקראת scan codes. בטבלה יש לכל מקש במקלדת שני קודים – קוד ללחיצה down וקוד לשחרור up. כמו שאפשר לראות בטבלה הבאה, ההבדל בין scan code של לחיצה לשחרור הוא 80h, כלומר ביט מספר 7 של ה־scan code מקבל 0 בלחיצה ו־1 בשחרור, יתר הביטים נשארים קבועים בין לחיצה ושחרור של אותו תו (אך משתנים כמובן 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	B0	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 המקש ESC תגרום לשליחת קוד 1h, שחרור של מקש ה־ESC יגרום לשליחת קוד 81h. באופן כללי, כל פעולה של לחיצה או שחרור של מקש גורמת לשליחת scan code יחד עם אינטרפט. להלן התהליך שמתרחש עם הלחיצה על המקלדת:

1. רכיב חומרה שנמצא במקלדת מעביר לפורט 60h את ה־scan code של הלחיצה / שחרור של המקש.
2. ה־PIC מקבל דרך IR1 אינטרפט מהמקלדת.
3. ה־PIC שולח למעבד אינטרפט, שאומר למעבד שיש מידע בפורטים של המקלדת. האינטרפט שמופעל הוא int 9h.
4. כתגובה לאינטרפט, המעבד מריץ ISR שמטפל באינטרפט 9.

5. ה־ISR מטפל בהעתקת ה־scan code אל מקום מוגדר בזיכרון (ה־Type Ahead Buffer שהזכרנו בפתיח).
6. בסיום ריצת ה־ISR, המעבד שולח ל־PIC סימן end of interrupt והתהליך מגיע לסיום.

באפר המקלדת Type Ahead Buffer

ה־ISR שמופעל על־ידי האינטרפט 9h לטיפול במקלדת לוקח את ה־scan code, שהגודל 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 ה־ASCII בגודל בית. עכשיו יש לנו גם scan code וגם ASCII code, שתופסים בסך הכל שני בתים. ה־ISR לוקח את שני הבתים האלו ומעתיק אותם אל אזור קבוע בזיכרון בשם באפר המקלדת, Type Ahead Buffer. הבאפר מוגדר בצורה הבאה:

- מיקום 0040:001A – מצביע על ראש הבאפר
- מיקום 0040:001C – מצביע על זנב הבאפר
- מיקום 0040:001E – 16 מילים (words)

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

כמו שיש מנגנון שדואג להכניס נתונים לבאפר, יש מנגנון שדואג להוציא נתונים מהבאפר. זוהי פסיקת BIOS, שמעתיקה את המידע מהבאפר במיקום שראש הבאפר מצביע עליו, ומעלה את ערכו של מצביע ראש הבאפר ב־2.

כך הראש של הבאפר "רודף" אחרי הזנב של הבאפר במסלול מעגלי שגודלו 16 מילים. יש פסיקת מקלדת שדואגת לקדם את הערך של הזנב ויש פסיקת BIOS שדואגת לקדם את הערך של הראש. מה קורה אם הראש "משיג" את הזנב? המשמעות של המקרה הזה היא שהבאפר מלא, כלומר אם נעתיק לתוכו ערך חדש אנחנו נדרוס הקלדה על המקלדת שעדיין לא טופלה. שמתם פעם לב לכך שאם התוכנית שלכם תקועה, ואתם מנסים ללחוץ על המקלדת הרבה פעמים, מתישהו המחשב ישמיע לכם צליל מעצבן עם כל הקלדה של המקלדת? זה בגלל שמילאתם את באפר המקלדת, הקשתם 15 פעמים בדיוק על המקלדת (בהקלדה הבאה הראש יעקוף את הזנב) ובגלל שהמחשב תקוע, הפסיקה של ה־BIOS לא מרוקנת את הבאפר.

הסיבה שנתנו את כל הסקירה הזו, היא כדי להגיע למסקנה שניקוי הבאפר הוא פעולה הכרחית וצריך לדאוג לכך שהבאפר לא יהיה מלא – אחרת אנחנו מאבדים את היכולת לקלוט תווים חדשים מהמקלדת. זה מה שצריך לזכור, ומיד נשתמש במידע הזה.

עכשיו, אחרי שסקרנו את התיאוריה של שרשרת הפעולות שגורמות להקשות מקלדת להפוך לקוד ASCII זמין לשימוש, נראה איך מבצעים שתי פעולות עיקריות:

- קריאה של תו מהמקלדת
- ניקוי הבאפר של המקלדת

נראה את הפעולות האלו בשלוש שיטות שונות – ישירות דרך הפורטים של המקלדת, בעזרת פסיקת BIOS ובעזרת פסיקת DOS.

שימוש בפורטים של המקלדת

בקצרה, המעבד עובד מול שני רכיבי חומרה, microcontrollers. אחד נמצא בתוך המקלדת, שני נמצא על לוח האם של המעבד. ישנם שלושה פורטים שמעורבים בעבודה עם המקלדת:

1. פורט 60h משמש לתקשורת בין המעבד לבין ה־microcontroller של המקלדת. עוברים ביניהם נתונים מסוגים שונים, הנתון שחשוב לנו הוא ה-scan codes שמגיעים מהמקלדת.
2. פורט 64h משמש לתקשורת בין המעבד לבין ה־microcontroller של לוח האם. הם מעבירים ביניהם הודעות בקרה, שדרכן אפשר לדעת אם יש scan code בפורט 60h.
3. פורט 61h הוא פורט בקרה כמו פורט 64h, אבל ישן יותר. מקלדות מודרניות לא עושות שימוש בפורט זה, אבל ישנם רכיבי חומרה אחרים שעדיין עושים בו שימוש ולכן נגיע אליו בהמשך (כשנרצה לעבוד מול כרטיס הקול).

קוד שבודק בפורט 64h אם הגיע תו חדש מהמקלדת (הביט השני מחזיק את הסטטוס):

```
in      al, 64h             ; Read keyboard status port
cmp     al, 10b             ; Data in buffer ?
```

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

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

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

```
in      al, 60h
```

לגבי השלב האחרון, ריקון הבאפר, ניתן לבצע אותו על־ידי קידום מצביע ראש המקלדת ב-2.

תוכנית דוגמה – קליטת הקלדות מקלדת ובדיקה האם מקש ה־ESC נלחץ. שימו לב שבדוגמה זו אנחנו לא מנקים את באפר המקלדת אחרי כל הקלדה. אחרי 15 תווים הבאפר כבר מלא – בתו הבא הראש כבר ישיג את הזנב. כתוצאה מכך בסוף התוכנית יודפסו למסך התווים שהקלדנו לפני ה־ESC, כל עוד מדובר בפחות מ־15 תווים. התווים ה־16 והלאה כבר "הלכו לאיבוד" ואינם קיימים יותר בזיכרון. גם אם הבאפר מלא, בגלל שאנחנו עובדים ישירות מול פורט 60h אנחנו עדיין מסוגלים לקרוא תווים חדשים, אבל הם כבר לא נכנסים לבאפר.

```
; --------------------------------------------------------------------
; 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 שהוקלדו יותר מ־15 תווים, רק 15 תווים נשמרו בבאפר המקלדת

והודפסו למסך עם היציאה מהתוכנית.

לעיתים אנו צריכים לזהות לא רק איזה מקש הופעל במקלדת, אלא גם אם הפעולה היא לחיצה על המקש או שחרור. לדוגמה, כאשר אנחנו רוצים לנגן תו בפסנתר- הנגינה מתחילה עם הלחיצה על המקש ומסתיימת עם שחרור המקש.

על מנת לעשות זאת נתבסס על התוכנית האחרונה אך בשינוי קטן- נוסיף בדיקה האם ה-scan code הוא מעל 80h (כלומר שחרור המקש). אם הערך הוא מעל 80h, אז ביצוע הפקודה

```
and al, 80h
```

יוביל לתוצאה שאינה אפס. אחרת 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 a new key in buffer
        in      al, 64h
        cmp     al, 10b
        je      WaitForKey
        in      al, 60h
        ;check if ESC key
        cmp     al, 1
        je      exit
        ;check if the key is same as already pressed
        cmp     al, [saveKey]
        je      WaitForKey
        ;new key- store it
        mov     [saveKey], al
        ;check if the key was pressed or released
        and     al, 80h
        jnz     KeyReleased
KeyPressed:
        ;print "Start"
        mov     dx, offset msg1
        jmp     print
KeyReleased:
        ;print "Stop"
        mov     dx, offset msg2
print:
        mov     ah, 9h
        int     21h

        jmp     WaitForKey
```

```
exit:
        mov     ax, 4c00h
        int     21h
END start
```

שימוש בפסיקת BIOS

פסיקת BIOS מספר 16h נותנת לנו כלים נוחים לבדיקת מצב המקלדת, קריאת התו שהוקלד (אם הוקלד) ו"ניקוי" באפר המקלדת (כלומר שינוי מצביע הראש ומצביע הזנב של באפר המקלדת כך שמצב הבאפר יהיה ריק, אין נתונים).

כדי לקרוא את התו הבא מתוך באפר המקלדת, מפעילים את 16h עם קוד ah=0h. הפסיקה מחזירה בתוך al את קוד ה־ASCII של התו שנמצא בראש הבאפר ובתוך ah את ה־scan code שלו. בנוסף, הפסיקה "מנקה" את התו מהבאפר על־ידי קידום ערכו של ראש הבאפר ב־2.

הבעיה היחידה עם הפסיקה הזו, היא שאם אין תו שממתין בבאפר – הפסיקה תחכה לתו. כתוצאה מכך, אם אנחנו רוצים לתכנת משחק שלא עוצר בהמתנה לפעולה של השחקן, הפסיקה הזו לבדה לא מתאימה.

הפתרון הוא לשלב את הפסיקה 16h עם קוד ah=1. במקרה זה, הפסיקה מחזירה את הסטטוס של המקלדת – 0 אם יש תו מוכן לקריאה, 1 אם אין תו מוכן. אם יש תו מוכן, al ו-ah יקבלו את ערכי ה־ASCII וה־scan code של התו.

שילוב הפסיקות מאפשר לנו:

- לדעת מתי יש מידע מהמקלדת (בלי לעצור את הריצה ולחכות למשתמש).
- לקלוט את המידע.
- לנקות את באפר המקלדת.

דוגמה:

```
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 ניקוי 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 בפסיקת DOS

פסיקה 21h עם קוד ah=0Ch מנקה את הבאפר של המקלדת, ואז מבצעת טריק נחמד – היא לוקחת הערך ששמנו ב-al כפרמטר, ומריצה int 21h עם הקוד הזה. אנחנו נראה דוגמה בה al=7h, כלומר – לאחר ניקוי באפר המקלדת, הפסיקה תעבור לקוד 7h, שהוא קוד של קליטת תו מהמקלדת ללא הדפסת התו על המסך. באופן זה אנחנו מבצעים שתי פעולות:

- ניקוי באפר המקלדת

- קליטת תו חדש מהמשתמש

בסיום הריצה al יכיל את קוד ה-ASCII של התו שהוקלד.

הקוד הבא מבצע את הפעולות הנ"ל:

```
; Clear keyboard buffer and read key without echo
mov     ah,0Ch
mov     al,07h
int     21h
```

לעבודה בשיטה זו יש יתרון ברור מבחינת פשטות תכנות. עם זאת ישנם שני חסרונות:

החיסרון ראשון, היותר ברור מאליו, הוא שהתוכנה עוצרת בזמן שהיא מחכה לקלט מהמשתמש. זה יכול להיות בעייתי אם אנחנו מריצים משחק וכו'.

החיסרון השני, הוא שישנם מקשים שונים שקוד ה-ASCII שלהם הוא לא קוד נורמלי בגודל בית אחד, אלא קוד ASCII מורחב בגודל שני בתים. מקשים אלו דווקא מאוד שימושיים למשחקים, כמו לדוגמה מקשי החיצים. במקרה זה, אם נרצה לעשות תנאי השוואה ובדיקה על קוד ה-ASCII – נהיה בבעיה.

תרגיל 12.1: מקלדת

א. ה-ISR של המקלדת מעתיק את ה-scan code אל ה-Type Ahead Buffer שנמצא במיקום 0040:001Eh בזיכרון המעבד.

כיתבו תוכנית שקוראת תו מהמקלדת (השתמשו ב-int 21h עם הקוד המתאים), הריצו את התוכנית ב-TD במצב step by step, וצפו בשינוי בזיכרון במיקום של ה-type ahead buffer. בתור קלט, הכניסו את התו 'a' ומצאו ב-type ahead buffer את ה-scan codes שלו.

ב. במשחקי מחשב שונים, המקשים wasd משמשים לתזוזת 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 עוצרת את ריצת התוכנית בהמתנה לקלט.

ג. הקוד הבא גורם לכרטיס הקול להשמיע צליל:

```
in      al, 61h
or      al, 00000011b
out     61h, al
mov     al, 0b6h
out     43h, al
mov     ax, 2394h
out     42h, al
mov     al, ah
out     42h, al
```

הקוד הבא גורם לכרטיס הקול להפסיק את השמעת הצליל:

```
in      al, 61h
and     al, 11111100b
out     61h, al
```

כיתבו תוכנית שברגע שנלחץ מקש כלשהו מוציאה צליל, ועם שחרור המקש מפסיקה את השמעת הצליל.
הדרכה: התוכנית תשתמש בפסיקה 16h כדי לבדוק אם יש מידע חדש מהמקלדת. אם יש מידע חדש, התוכנית תבדוק בעזרת פורט 60h אם ה-scan code מתאים ללחיצה או לשחרור ובהתאם יופעל קטע הקוד שמשמיע צליל או קטע הקוד שמפסיק את השמעת הצליל.

סיכום

התחלנו את הפרק בלימוד התיאוריה של אינטרפטים: למה בכלל יש צורך בפסיקות לטיפול בהתקני חומרה ואיך ה־PIC, בקר האינטרפטים, מבצע את עבודת הקישור בין התקני החומרה לבין המעבד.

למדנו על פורטים, אותם 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 התהליך שגורם לכך שבסופו של דבר התו שנלחץ במקלדת מופיע בזיכרון המחשב, ב־Type Ahead Buffer. ראינו דוגמאות לעבודה מול המקלדת במגוון שיטות:

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

בפרק הבא נעסוק בנושאים שימושיים לכתיבת פרוייקטי הסיום.

פרק 13 – כלים לפרוייקטים

מבוא לפרוייקטי סיום

זהו, כיסינו בפרקים הקודמים את כל החומר התיאורטי שנכלל בתוכנית הלימודים. כמובן שיש עוד חומר תיאורטי רב שקשור לשפת אסמבלי, אבל מדובר בעיקר על הרחבת נושאים שדנו בהם בקצרה. עכשיו אנחנו מתקדמים אל העבודה המעשית, אל הפרוייקטים. פרוייקט הסיום הוא ההזדמנות שלכם לכתוב תוכנה "אמיתית" שמשלבת בין הדברים שלמדתם לדברים שמעניינים אתכם, להרגיש שאתם מבינים את החומר היטב ולהוכיח את זה. הפרק הזה מיועד לקריאה וללימוד עצמי. הוא אינו מתיימר לכסות באופן מלא את הנושאים אלא רק לתת הסבר ראשוני ומקורות ללימוד עצמי. קיימות לכך שתי סיבות עיקריות. הראשונה, סיבה מעשית: במסגרת פרק אחד אין אפשרות לכסות לעומק נושאים, שלרוב מוקדש להם פרק משל עצמם או אפילו ספר. הסיבה השניה היא הרצון להכין אתכם לצורת העבודה בה תתנסו בעתיד – המידע שאתם צריכים קיים אי שם, עליכם לחפש אותו, להגיע אליו ולהבין אותו בכוחות עצמכם.

בחירת פרוייקט סיום

האם כבר בחרתם פרוייקט סיום? מומלץ שתבחרו פרוייקט לפני קריאת פרק זה, כך שתוכלו להתמקד בדיוק בדברים שאתם צריכים כדי לעבוד על הפרוייקט ותשקיעו פחות זמן בעיסוק בנושאים אחרים. כמובן, שאם יש לכם עניין ללמוד את כל הנושאים – דבר זה מומלץ וצפויה לכם הנאה לאורך הדרך.

קיים מגוון לא קטן של פרוייקטי סיום לבחירתכם. רוב הפרוייקטים הם בסדר גודל של 1000 עד 2000 שורות קוד. כאשר מדובר בפרויקט שכולל מתחת ל־1000 שורות קוד, ייתכן שאתם לא ממצים את מלוא היכולות שלכם. אם הפרויקט הוא הרבה מעל ל־2000 שורות קוד, ייתכן שאתם לוקחים על עצמכם משימה שתיקח זמן רב מדי. אפשר לסווג את הפרוייקטים לשלוש קטגוריות עיקריות:

- משחקים – סנייק, פונג או כל דבר שעולה על דעתכם. היתרון בבחירת פרוייקט שהוא משחק, הוא שהעבודה שצריך לעשות בפרוייקט די מוגדרת ובדרך כלל הפעלת התוכנה די אינטואיטיבית, דבר שמקל על תכנון הפרוייקט.

- אפליקציה קטנה – לדוגמה כלי נגינה (הקשה על המקלדת או העכבר גורמים להפעלת הכלי, השמעת צלילים ושינויי גרפיקה), מכונת מוזיקה (בחירת שירים שהוקלטו מראש), צייר (הזזת העכבר תוך כדי לחיצה גורמת לציור על המסך) או כל דבר שעולה על דעתכם. היתרון בפיתוח אפליקציה הוא שיש מקום רב ליצירתיות ואפשר לעשות מה שמעניין אתכם.

- בעיות אלגוריתמיות – לדוגמה מימוש של קודים לתיקון שגיאות, פתרון אוטומטי של סודוקו או פתרון של בעיות מתמטיות שונות. בפרויקטים מסוג זה ישנו סיכון – אם אתם מתכננים לפתור בעיה כזו, תניחו מראש שאת כל הידע התיאורטי תצטרכו ללמוד בכוחות עצמכם. אי לכך, כדאי שתיכנסו לנושאים כאלה רק אם אתם נלהבים ללמוד את

כל הידע בעצמכם מהאינטרנט. היתרון בפרוייקט כזה הוא שמדובר בנושאים מאוד מעניינים והידע האלגוריתמי שתרכשו עשוי להיות רלבנטי לדברים שתעשו בעתיד.

הכלים שנלמד בפרק זה הם:

- קבצים:
  - עבודה עם קבצים, קריאה מקובץ ושמירה לקובץ
- גרפיקה:
  - יצירת גרפיקה בעזרת תווי ASCII
  - יצירת גרפיקה בעזרת הדפסת פיקסלים למסך
  - יבוא של תמונות בפורמט BMP
- צלילים:
  - השמעת צלילים בתדרים שונים
- שעון:
  - שימוש בשעון למדידת זמן (השהיה)
  - שימוש בשעון ליצירת מספרים אקראיים
- ממשק משתמש:
  - טיפים לעבודה עם מהמקלדת
  - קליטת פקודות מהעכבר
- שיטות דיבוג

עבודה עם קבצים

פתיחת קובץ

הדבר הראשון שאנחנו צריכים לעשות כדי לקרוא מקובץ, הוא לפתוח אותו לקריאה (כמו ספר – אי אפשר לקרוא אותו כשהוא סגור, קודם צריך לפתוח את הכריכה...). הדרך הפשוטה לבצע פתיחת קובץ היא באמצעות פסיקת DOS, עם קוד ah=3Dh. הפרמטרים שצריך לשלוח לפסיקה הם:

AL – מטרת הפתיחה

- 0: קריאה בלבד
- 1: כתיבה בלבד
- 2: קריאה וכתיבה

DS:dx – מצביע על שם הקובץ.

שימו לב שהמחרוזת של שם הקובץ צריכה להסתיים ב־0. לדוגמה:

```
Filename	db	'file.txt',0
```

בסיום הריצה, ax יכיל את ה־filehandle שהוקצה לו על־ידי מערכת ההפעלה DOS. אם היתה שגיאה, יודלק דגל CF, ו־ax יכיל את אחד הערכים הבאים:

- 2: הקובץ לא נמצא.
- 5: יותר מדי קבצים פתוחים.
- 12: אין הרשאה לפתיחת הקובץ.

מומלץ לאחר הפתיחה לבדוק אם הפתיחה היתה תקינה, ואם לא – להדפיס הודעת שגיאה. אם ננסה להמשיך לעבוד עם קובץ שהפתיחה שלו נכשלה, סביר שהתוכנה תקרוס.

```
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 שצריך לשלוח לפסיקה הם:

bx – filehandle שקיבלנו מ־DOS בשלב הפתיחה.

cx – כמות הבתים שאנחנו מבקשים לקרוא.

dx – מצביע על באפר (מערך) שאליו יועתק המידע מהקובץ.

שימו לב: הגודל של dx חייב להיות גדול או שווה לכמות הבתים שאנחנו רוצים לקרוא, אחרת יידרס הזיכרון שאחרי הבאפר.

```
proc ReadFile
; Read file
        mov     ah,3Fh
        mov     bx, [filehandle]
        mov     cx,NumOfBytes
        mov     dx,offset Buffer
        int     21h
        ret
endp ReadHeader
```

ביציאה, ax יחזיק את כמות הבתים שנקראו מהקובץ, או קוד שגיאה אם היתה בעיה.

שימו לב כי יש לקרוא מהקובץ רק לאחר שפתחנו אותו בהצלחה.

כתיבה לקובץ

פעולת הכתיבה לקובץ נראית בדיוק כמו פעולת הקריאה – רק להיפך. הקוד לכתיבה לקובץ הוא ah=40h. פרמטרים:

bx – filehandle שקיבלנו מ-DOS בשלב הפתיחה.

cx – כמות הבתים אותם אנחנו מבקשים לכתוב. הערה: אם cx=0 אז כל המידע שבקובץ אחרי filehandle, יימחק.

dx – מצביע על באפר (מערך) שממנו יועתק המידע אל הקובץ.

ביציאה, ax יחזיק את כמות הבתים שנכתבו לקובץ, או קוד שגיאה אם היתה בעיה.

שימו לב: אם לא פתחתם את הקובץ במצב שמאפשר כתיבה (cx=1, cx=2) אז יוחזר קוד שגיאה ax=5 שמשמעותו access denied.

```
proc WriteToFile
        mov     ah,40h
        mov     bx, [filehandle]
        mov     cx,12
        mov     dx,offset Message
        int     21h
        ret
endp WriteToFile
```

שימו לב כי יש לכתוב לקובץ רק לאחר שפתחנו אותו בהצלחה.

סגירת קובץ

בזמן היציאה מהתוכנית (פסיקה 21h עם קוד ah=4Ch), משוחרר כל הזיכרון שהתוכנית תפסה ונסגרים כל הקבצים שהתוכנית פתחה. אם כך, מדוע בכלל לסגור קבצים שאנחנו משתמשים בהם? אחת הסיבות היא שלא תמיד נוכל לסמוך על היציאה מהתוכנית, שתסגור את הקבצים שלנו. לדוגמה, אם התוכנית קרסה בזמן ריצה, היא לא תסגור את הקבצים שפתחנו. זהו גם הרגל טוב לעתיד: לסגור משאבים חיצוניים שאנחנו משתמשים בהם (קבצים, זיכרון, קישורים למחשבים אחרים, תוכנות עזר ועוד). בצורה זו, תוכנות אחרות יכולות לנצל את המשאבים שפינינו והקוד שלנו הופך ליעיל יותר.

סגירת קובץ מתבצעת על־ידי קוד ah=3Eh. הפרמטר היחידי שצריך לתת לפסיקה הוא:

bx – filehandle שקיבלנו מ־DOS בשלב הפתיחה.

```
proc CloseFile
        mov     ah,3Eh
        mov     bx, [filehandle]
        int     21h
        ret
endp CloseFile
```

פקודות נוספות של קבצים

באמצעות קודים שונים ניתן לבצע פעולות נוספות, שנשאיר ללימוד עצמי. עקב ריבוי המקורות והעובדה שמקורות שונים מתמקדים בהסברים ובדוגמאות על פעולות שונות, עדיף שתתנסו לבד בחיפוש אחרי המקור שנדרש לכם, בין שזה 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 – יצירת קובץ

AH=41h – מחיקת קובץ

AH=42h – הזזת המצביע בתוך הקובץ

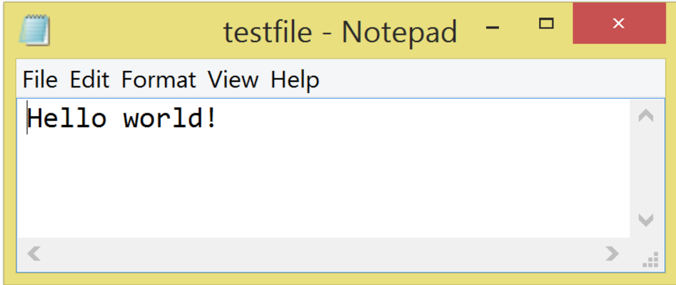
תכנית לדוגמה – filewrt.txt

התוכנית הבאה פותחת קובץ ריק בשם testfile.txt, מעתיקה לתוכו את המחרוזת 'Hello world!' וסוגרת 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 להכניס לתוכנית שלכם אלמנטים גרפיים. לפני הכל, ניתן קצת רקע תיאורטי.

הרעיון הבסיסי בעבודה עם גרפיקה הוא שישנו אזור בזיכרון ה-I/O ששומר את כל המידע שמוצג על המסך. האזור הזה נקרא video memory והוא קשור אל כרטיס המסך. ה־video memory תופס את המיקום בזיכרון שבין A000:0000 לבין B000:FFFF.

כרטיס המסך יכול לעבוד בשני מצבים, שנקראים modes:

- במצב טקסטואלי, או text mode, אנחנו קובעים לכרטיס המסך שעליו לקרוא את המידע מה־video memory במיקום שמתחיל ב־B800:0000 וגודלו 4K (4,096 בתים). במצב זה, כרטיס המסך מדפיס 25 שורות כפול 80 עמודות של תווי ASCII.

- במצב גרפי, או graphic mode, אנחנו קובעים לכרטיס המסך שעליו לקרוא את המידע מה־video memory במיקום שמתחיל ב־A000:0000, וגודלו 64K (65,536 בתים). במצב זה, כרטיס המסך מדפיס 200 שורות כפול 320 עמודות של פיקסלים.

אפשר לפרק את תהליך ההדפסה למסך לשלושה שלבים:

שלב ראשון – קביעת מצב העבודה של כרטיס המסך (כשמפעילים DOSBOX, מצב ברירת המחדל של הגרפיקה הוא מצב טקסטואלי).

שלב שני – תרגום הגרפיקה שלנו לביטים שכרטיס המסך יכול לפענח (מה שנקרא פורמט 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 שצריך לחשב בעצמנו את הכתובת שצריך לפנות אליה.

- אפשר לקרוא לפסיקת BIOS. פסיקות BIOS הוזכרו בקצרה בפרק שדן בפסיקות. להזכירכם, BIOS היא ספריית תוכנה של חברת אינטל. השימוש בפסיקות BIOS נותן לנו ממשק פשוט יחסית להתקני חומרה.

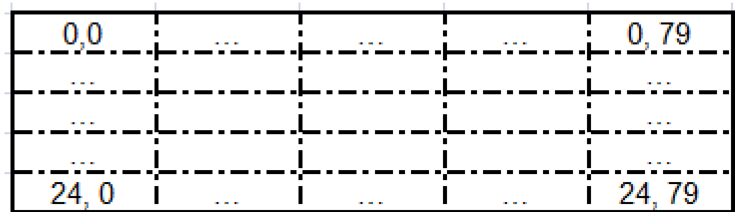
- אפשר להשתמש בפסיקות DOS, שכבר יצא לנו להכיר.

נסביר עכשיו את שלושת השלבים, פעם ל־text mode ופעם ל־graphic mode.

גרפיקה ב־Text Mode

כמו שאפשר להבין, מצב טקסטואלי מיועד להצגת טקסט בעיקר ולכן הוא מוגבל לדברים שאפשר לייצג באמצעות תווי ASCII. עדיין, עם קצת יצירתיות ומשחק עם תווי ASCII אפשר להגיע לדברים נחמדים מאוד.

שלב ראשון – קביעת מצב עבודה text mode. לשמחתנו, זהו מצב העבודה ברירת המחדל של DOS. במצב זה המסך מחולק ל־25 שורות ו־80 עמודות של תווים.



אם אנחנו לא נמצאים ב־text mode, נוכל לעבור אליו באמצעות פסיקת BIOS, int 10h. קיראו לפסיקה באופן הבא:

```
mov ah, 0

mov al, 2

int 10h
```

לביצוע השלב השני והשלישי יש כמה שיטות. נראה עכשיו שתי דרכים – יש יותר מכך, ואפשר גם לבצע שילובים.

שימוש במחרוזות ASCII

שיטה פשוטה אבל בעלת תוצאות מקסימות היא להגדיר ב־DATASEG אוסף של תווים שיוצרים ציור או כתובת. לחילופין, במקום לבצע את ההגדרה ישירות ב־DATASEG, אנחנו יכולים להגדיר קובץ, ואז לעשות לו include לתוך DATASEG.

לדוגמה הקובץ monalisa.asm:

```
monalisa    db "-` .-/v-` .-/v@@@A)                     -)ZdMd@@@@@@@@@@@@@@@@@@\' _\ ",13,10
            db "-` .-/v- ` ,iVJ@@@!                    '-!(K5K@@@@@@@@@@@@@@@@@@@[(/s[.",13,10
            db "-` .-/v- i\G@@@Z-                    ' ! -i55ZZ@@@@@@@@@@@@@@@@@)(4)`  ",13,10
            db "-` .-/v, -|b@@@@!\                     '  ` |-tYG@@@@@@@@@@@@@@@@@XNYZ- ",13,10
            db "-` .-/v  tt@@@@A-,                        '  `)(d@@@@@@@@@@@@@@@@@D)8A[ ",13,10
            db "-` .-/v  )8@@@@@\                         ,-'-/Kd@@@@@@@@@@@@@@@@@@KD@[ ",13,10
            db "-` .-/v]]Z@@@@d|-              ,ii.c,, -.icLZKK@@@@@@8@K@@@@@@@@@(@8[  ",13,10
            db "-` .-/v  KN8@@@@@( .i!vGG_      J4Kb8ZKb@bbK@d@88@@@@@@@b@@@@@@@@@@dK@-",13,10
            db "-` .-/v )/8K@@@K@b@dP~~~T4(    Jd@@7`___s@M@@@@MM8d@@@d@@@@@@@@@@@@LM8[",13,10
            db "-` .-/v\!48@K@@@@8@@d*@@@bVi   bAKLY~~@@@@@@*ff/\NM8@@@@@@@@@@@@@@@@db@[",13,10
            db "-` .-/v,\\Kb@@@d@.~t` !*~!`.  -MA)    '~'.).` `,'K@@@@@@@@@@@@@@@@@AKb[",13,10
            db "-` .-/v,`8M@@@@@@ -`,,gvZ``    A//-  ..c\+\`    i]d@@@@@@M@@@@@@@@@@@8[",13,10
            db "-` .-/vi\@8@K@@@D              \!'               !iZ8@@@8A@@@@@8d@b@@@8M[",13,10
            db "-` .-/ve8d5@@@@@@             '!-               '-)8@@@@@@@@@@@@@@@@@M8i",13,10
            db "-` .-/v8dZ8@M@@@@-             v  ,            ,\tK@@@@@@@@@@@@A@@@@@@Z2|",13,10
            db "-` .-/v@b@AK@@@b@[              //              cctbA@@@AK@@d@@d@@@K@@@bmi",13,10
            db "-` .-/v@@8@M@8@@@P-             -=/.           /iD8d@@@@@@@@@@@@@@A@@@d@@[",13,10
            db "-` .-/v@8@@@MA@@@@\-      .   _)g2i          -((dKK@@@@@d@@@@d@@K@@@@@@K[",13,10
            db "-` .-/v@@@bAK@@K@@)i      'c,,Kb@@bK        )X)Kb@M@@d@@@Mb@@A@d@@@@@@8@[",13,10
            db "-` .-/v@K@b@@@@A@AA/i-      ~M@@@@Mc    .,\c=)D8d@@b@@@d@@@@@@@@@@@8d@@A[",13,10
            db "-` .-/v@@@@Mb@@@@@@('c\`      PPK((,i]v|-\-v)8XNAdMK@@@@@@@b@@MK@A@@@@@[",13,10
            db "-` .-@@8@8@@MK@d@A@L!--c)s_, ,(ZsbLb@\`- .-N]/KM@@@@@@@d@@@A@@@@@@@@@d@[",13,10
            db "-` .@@8d@Kb@@@K@b@@@/-  !''~~Vff*N5f -` -,\))KK@@@@@@@MK@@d@@@M8d@b@@@[",13,10
            db "-` @@8d@b@@@KAK@@@@@@2--     ,,_JJ/i)/- |/v)NK@8d@@@@@@@@@@8@@@@@@@@@M@K[",13,10
            db "-`@8dv@@8d@K@@@b@@@@@d!,    'VV\)\\)\7(-)4Jb@8@A@@@K@d@@@@@@@@8@@@@@@@@@[$"
```

הקובץ הזה הוא בסך הכל מחרוזת ארוכה, שמופרדת לשורות כדי שיהיה יותר נוח לעבוד איתה. שימו לב לסימן ה־$ בסוף המחרוזת – אתם כבר בטח מבינים מדוע הוא שם.

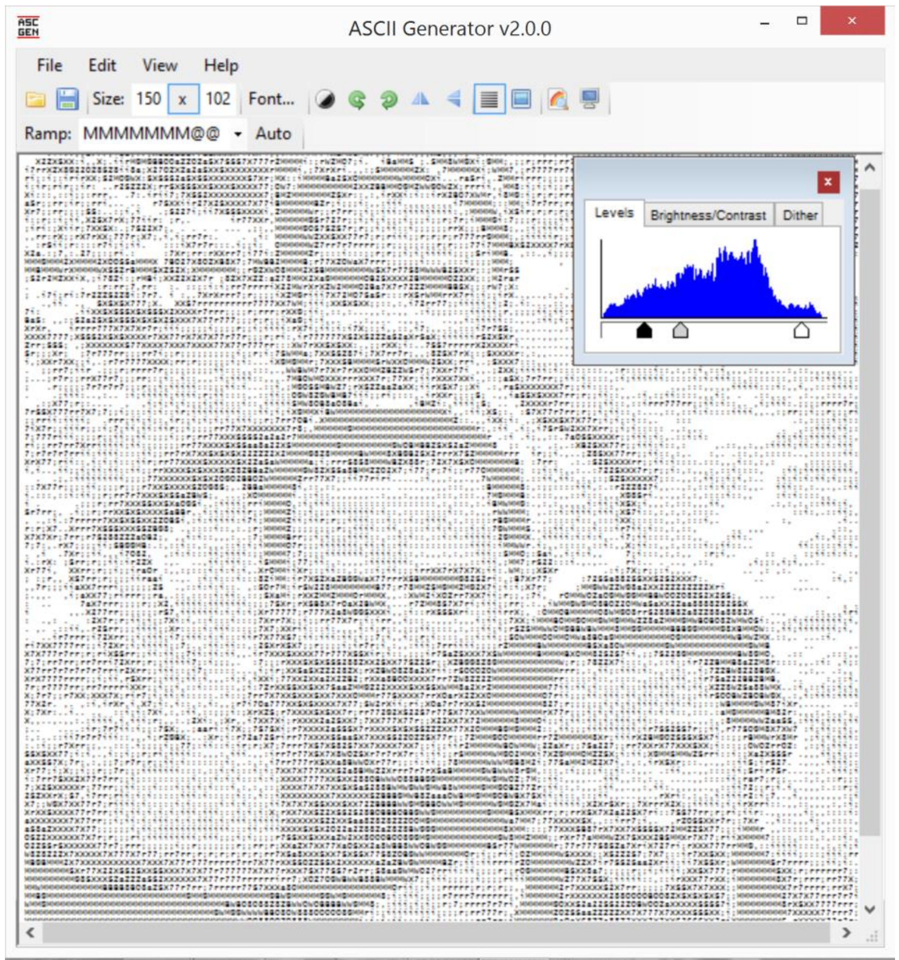
את ההדפסה למסך נבצע בשיטה המוכרת, באמצעות הפקודה של הדפסת מחרוזת למסך int 21h עם קוד ah=9h.

דוגמה לתוכנית קצרה שמדפיסה למסך את הקובץ monalisa.asm (שימו לב לכך שיש להוריד את ההוראה IDEAL בראש התוכנית, על מנת שה־include יעבוד):

```
; -----------------------------------------------------------
; 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 לתווי ASCII באמצעות תוכנה כגון ASCII Generator שניתנת להורדה מ־http://sourceforge.net/projects/ascgen2. לדוגמה:



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/

גרפיקה ב־Graphic Mode

עד כה דיברנו על תווים, עכשיו נעבור לדבר על פיקסלים. פיקסל הוא היחידה הקטנה ביותר שניתן לשנות את הערך שלה במסך. פיקסל הוא כמו אטום, אבל של גרפיקה. כמות הפיקסלים במסך נקראת רזולוציה. אנחנו נדבר על פורמט VGA, קיצור של Video Graphics Array. סקירה של פורמט זה ניתן למצוא במקומות רבים, מומלץ כרגיל לתת את הכבוד הראשון לויקיפדיה:

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

פורמט VGA תומך במספר רזולוציות מסך, אנחנו נתמקד ברזולוציה של 320X200, כלומר 200 שורות כפול 320 עמודות של פיקסלים.

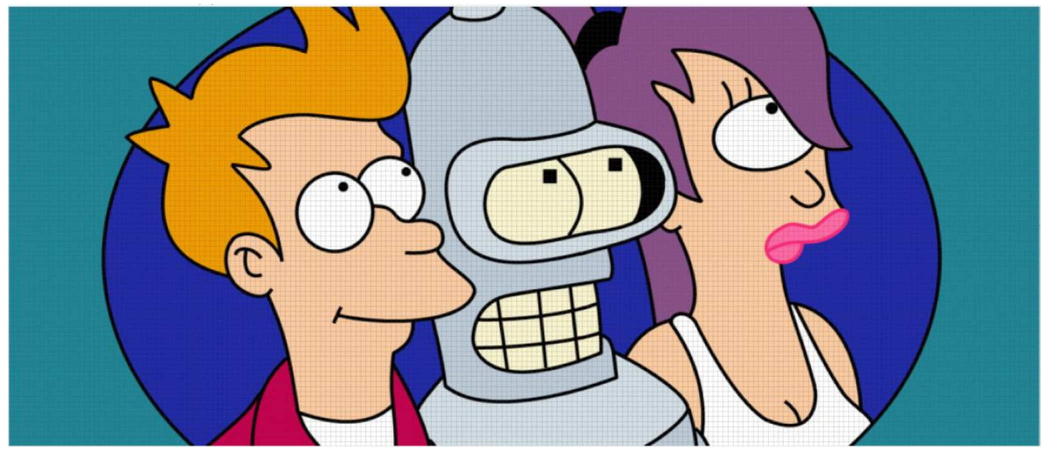
כדי לעבור לפורמט גרפי זה נשתמש בפסיקת BIOS, int 10h שנעשית חביבה עלינו בפרק זה:

```
mov ax, 13h
int 10h
```

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

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

בתוך ה־video memory, באזור שמתחיל ב-A000:0000, שמור כל המידע שקשור לייצוג התמונה. בשביל כל פיקסל אנחנו צריכים לדעת שני דברים: קודם כל, מה המיקום שלו על המסך. אחר כך, מה הצבע של הפיקסל.

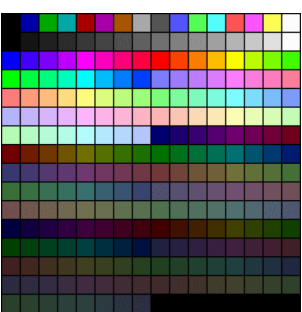


לכל פיקסל יש צבע אחר, שילוב של פיקסלים יוצר תמונה

המיקום של הפיקסל על המסך נקבע על־ידי המיקום שלו בזיכרון: כל פיקסל מיוצג על־ידי בית אחד. כך, הבית A000:0000 קשור לפיקסל 0:0, הבית A000:0001 קשור לפיקסל 0:1 וכך הלאה. כדי להגיע לפיקסל בשורה Y, כופלים את מספר השורה ב־320, מוסיפים את מספר העמודה X ופונים לזיכרון במיקום שחושב.

נעמיק את ההסבר על קביעת הצבע. כאמור, בפורמט VGA שאנחנו עוסקים בו, מספר הצבע מיוצג על־ידי בית אחד, כלומר 8 ביטים. מכאן שאנחנו יכולים לבחור $256=2^8$ אפשרויות של צבעים. בזיכרון המחשב יש טבלת המרה, שממירה כל מספר לשילוב של Red, Green, Blue, או בקיצור – RGB. קיימים הרבה יותר מ־256 שילובים של RGB, אך טבלת ההמרה מכילה רק 256 אפשרויות (כדי שניתן יהיה לייצג צבע באמצעות בית בודד). נניח ששמנו ב־video memory מספר צבע 0. כרטיס המסך יפנה לטבלת ההמרה למיקום 0, שם הוא ימצא – סביר להניח – שערכי ה־RGB הם 0,0,0 – כלומר מייצגים צבע שחור.

בזיכרון המחשב יש טבלת המרה סטנדרטית, שנקראת standard palette. אלו הצבעים שנשמרים ב־ standard palette:



לעיתים יש לנו תמונה, שהגוונים בה די דומים אחד לשני. במקרה זה 256 הצבעים הקיימים ב־standard palette לא מספיקים כדי לייצג את השוני בין הגוונים. עם זאת, אפשר לוותר על חלק מ־256 הצבעים ב־standard palette, היות שאינם מיוצגים בתמונה, ולהחליף אותם בגוונים אחרים שיש להם ייצוג בתמונה. מיד נראה איך ניתן לשנות את ה־palette ברירת המחדל, באמצעות טעינה של קובץ בפורמט bmp שמכיל palette מותאם לקובץ.

הדפסת פיקסל למסך

לאחר שסקרנו את המנגנון שעומד מאחורי הדפסת פיקסלים למסך, נותר לנו רק לתאר את הפקודות שגורמות לכך בפועל. האפשרות הראשונה היא פשוט לגשת ל־video memory ולהכניס לתוכו ערכים בגודל בית באמצעות פקודת out. לפני כן צריך רק לחשב את מיקום התא בזיכרון, כדי שנקלע לקואורדינטות x,y הנכונות.

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

al – צבע

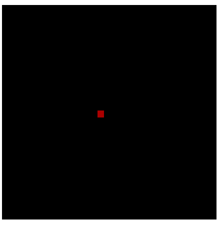
bl – עמוד (צריך להיות 0)

cx – קואורדינטת X

dx – קואורדינטת Y

```
; ------------------------------------------------------------
; Paint a red pixel in the center of the screen
; Author: Barak Gonen 2014
; ------------------------------------------------------------
IDEAL
MODEL       small
STACK       100h
DATASEG
x       dw      160
y       dw      100
color   db      4
CODESEG
start:
        mov     ax, @data
        mov     ds, ax
        ; Graphic mode
        mov     ax, 13h
        int     10h
        ; Print red dot
        mov     bh,0h
        mov     cx,[x]
        mov     dx,[y]
        mov     al,[color]
        mov     ah,0ch
        int     10h
        ; Wait for key press
        mov     ah,00h
```

```
	int	16h
	; Return to text mode
	mov	ah, 0
	mov	al, 2
	int	10h
exit:
	mov	ax, 4c00h
	int	21h
END start
```



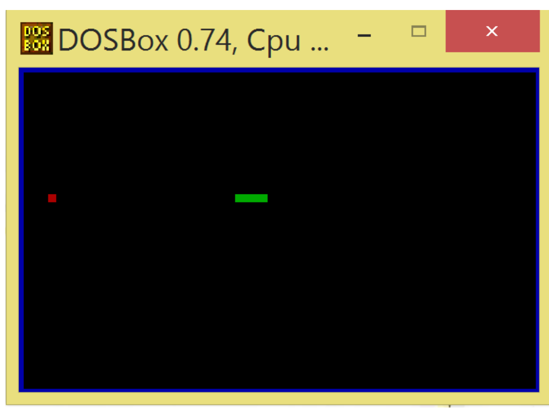
הדפסת פיקסל למסך

קריאת ערך הצבע של פיקסל מהמסך

פסיקת ה־BIOS מאפשרת לנו גם לקרוא את ערך הצבע של פיקסל מהמסך, כאשר מבצעים קריאה עם ah=0Dh. אנחנו נתייחס בקצרה לאפשרות זו, כיוון שהיא יכולה להיות שימושית במשחקים. לדוגמה – נניח שאנחנו כותבים משחק סנייק, ואנחנו רוצים לבדוק האם הנחש שלנו התנגש בקיר. אנחנו יכולים פשוט לתת לקיר צבע ייחודי ולבדוק את ערך הצבע במקום בו נמצא ראש הנחש. אם ערך הצבע שווה לצבע של הקיר – השחקן נפסל.

```
; 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
```



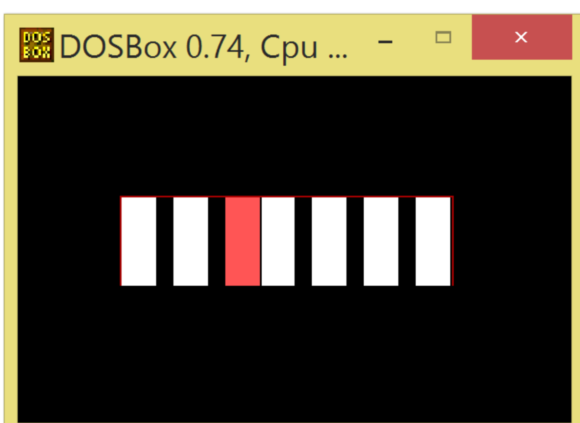
משחק סנייק שתוכנת באסמבלי (קרדיט: נועם אזולאי)

יצירת קווים ומלבנים על המסך

ברגע שאנחנו יודעים ליצור פיקסל על המסך, אנחנו יכולים ליצור גם קווים ומלבנים (למעשה אנחנו יכולים ליצור כל שורה, אבל נתמקד בקווים ובמלבנים לצורך ההסבר).

הרעיון הבסיסי מאחורי יצירת קו, הוא שאנחנו קוראים לפרוצדורה שמדפיסה 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). יצירת קו אלכסוני דורשת שינוי גם בערך ה־X וגם בערך ה־Y. אם אתם רוצים ליצור קו אלכסוני, או מעגל, חפשו בגוגל "Bresenham algorithm" – נושא זה הוא ללימוד עצמי, למעוניינים בכך.

יצירת מלבן מתבצעת באמצעות אוסף קריאות לפרוצדורה שיוצרת קו אופקי, כאשר בכל פעם נקודת ההתחלה בציר ה־Y גדלה ב־1. כדי לייצג כפתור מלבני לחוץ ניתן לקבוע לו צבע אחר, או להקיף אותו במסגרת.



סימולציה של קלידי פסנתר, אחד הקלידים לחוץ (קרדיט: אליזבת לנגרמן)

קריאת תמונה בפורמט BMP

לעיתים, במקום לצייר תמונה בכוחות עצמנו, נרצה לשלב בתוכנה שלנו תמונה נאה שמצאנו. קיימים פורמטים (הגדרות מקובלות לייצוג מידע בקבצים) רבים של תמונות, אנחנו נתמקד בפורמט BMP, קיצור של Bit Map, משתי סיבות. הראשונה היא שהוא פורמט נפוץ, והשניה היא שהוא פשוט. את הנתונים שיש בקובץ BMP אפשר לטעון בלי הרבה שינויים ומשחקים לתוך ה־video memory, זאת בניגוד לפורמטים אחרים, כמו JPG הנפוץ, שמצריך אלגוריתמיקה מורכבת לפתיחה.

נתאר בקיצור רב את פורמט BMP. לפני כן, נקדים ונאמר שאת הלימוד של פורמט ה־BMP מומלץ וקל לעשות על־ידי חיפוש באינטרנט. הסברים מפורטים ניתן למצוא באמצעות חיפוש "read bmp file in assembly". קישורים לדוגמה:

www.brackeen.com/vga/bitmaps/html

www.ragestorm.net/tutorial?id=7

פורמט BMP מורכב מהשדות הבאים:

1. Header – פתיח בן 54 בתים. בתחילת ה־header נמצאים התווים 'BM' שמציינים שהקובץ הוא בפורמט BMP.
2. Palette – 256 צבעים, כל צבע תופס ארבעה בתים (בסך הכל 1,024 בתים). שימו לב, שבעוד שכרטיס המסך מקבל את הצבעים בפורמט RGB (כלומר, אדום–ירוק–כחול), הצבעים שב־palette של ה־BMP שמורים בפורמט BGR (כחול–ירוק–אדום), ולכן נצטרך להפוך את סדר הצבעים לפני שנטען את ה־palette לכרטיס המסך.
3. Data – המידע על כל פיקסל ופיקסל שמור בבית יחיד. הבית הזה מכיל מספר בין 0–255, שהוא מספר הצבע ב־palette. שימו לב, שהשורות שמורות מלמטה למעלה. כלומר, אם ניקח את ה־data ונעתיק אותו כמו שהוא לתוך ה־video memory, נקבל תמונה הפוכה. קבצי BMP יכולים להכיל מספר פיקסלים הרבה יותר גדול ממה שה־video memory של VGA יכול לקבל. אנחנו נעסוק בקבצים שהגודל המקסימלי שלהם הוא 320 עמודות כפול 200 שורות, או 64,000 פיקסלים.

נחבר את כל הדברים שזה עתה למדנו כדי לראות איך הצבעים מועברים למסך. נניח שיש לנו תמונה בגודל 320x200 פיקסלים. ניקח את הפיקסל הראשון בשורה התחתונה (שמוצג במסך למטה מצד שמאל). כיוון ש־BMP שומר את השורות הפוך, בתוך קובץ ה־BMP, זהו הפיקסל שנמצא במקום הראשון ב־data. במילים אחרות, הוא נמצא בבית ה־1079 בתוך קובץ ה־BMP (1079=54+256x4).

נניח שהערך בבית ה־1079 הוא '0'. המשמעות של ערך זה, היא שהצבע של הפיקסל השמאלי בשורה התחתונה הוא מה שנמצא באינדקס 0 ב־palette. אנחנו ניגשים ל־palette, ערכי ה־BGR של הצבע באינדקס 0 נמצאים בתחילת ה־palette, בבתים 54:57.

התוכנית הבאה קוראת ומדפיסה למסך קובץ BMP בגודל 320 על 200. כדי ליצור קובץ זה, אתם יכולים להיכנס ל־Microsoft Paint ולשנות את גודל התמונה כך שהיא תהיה בגודל המתאים. אתם יכולים גם ליצור ולטעון קבצים יותר קטנים, אך תצטרכו לשנות את הגדלים 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 לקריאה.
- לאחר מכן נקרא ה־header.
- לאחר מכן נקרא ה־palette.
- ה־palette נטען לכרטיס המסך, בפורטים 3C8h, 3C9h. התוכנית משנה את סדר הערכים 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.
- שימו לב לחלוקה של התוכנית לפרוצדורות, 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 הפיאצה מכיל מידע איכותי בנושא שימוש בגרפיקה, כולל קטעי קוד וטיפים. בחלון החיפוש אתם יכולים לחפש כל ביטוי. לדוגמה bmp או graphics. הדיון הבא מומלץ במיוחד:

https://piazza.com/class/i98gbkdp1mg15m?cid=20

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

"שלום לכולם

אני רוצה שדיון זה ירכז פרקטיקות טובות לטיפול בגרפיקה, שללא ספק זהו נושא שמעסיק תלמידים רבים.

לדיון זה מוזמנים מאד לענות תלמידים שעשו פרוייקטים עם גרפיקה איכותית, והידע שלהם גם יסייע לאחרים וגם יישמר לשנתונים הבאים.

נא להעלות הסברים וקטעי קוד לנושאים הבאים:

1. איך מעלים תמונת BMP למיקום מוגדר במסך? לדוגמה, ציור של מטרה שגודל ה-BMP הוא 10x10 פיקסלים במיקום 80,100

2. איך מזיזים תמונה על המסך בצורה חלקה בלי ריצודים? לדוגמה, הליקופטר נע מצד לצד.

3. איך מזיזים תמונה על גבי תמונת 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 תשובות, אבל מי שיעלו הסברים טובים, אשמח לשלב אותם בגרסה הבאה של ספר הלימוד + קרדיטים (:

בברכה

ברק"

קרדיטים על תשובות:

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

סעיפים 2,3- **קורן ברנד**, אהל שם רמת גן

סעיף 4- **שני שוורץ**, אורט בנימינה

השמעת צלילים

כפתיח לנושא השמעת צלילים, מומלץ לקרוא קודם כל תיאוריה על צלילים, גלי קול ותדירויות. אתם יכולים למצוא מגוון עצום של מקורות באינטרנט. מומלץ במיוחד להוריד את גרסת ה־pdf החינמית של הספר Art of Assembly ולקרוא אודות הפיסיקה של גלי קול (The Physics of Sound).

נתמקד בחלקים הטכניים של השמעת צלילים מהמחשב.

למחשב יש רכיב חומרה שצמוד אליו – טיימר, שעון, שלמדנו עליו בפרקים הקודמים. הטיימר מחובר לכרטיס הקול של המחשב ומעביר לו תקתוקי שעון, או "טיקים". כל "טיק" של הטיימר גורם לסגירת מעגל חשמלי בכרטיס הקול, שמתרגם אותו לקול שאנחנו מסוגלים לשמוע. כמות ה"טיקים" שמגיעים כל שניה מהטיימר אל כרטיס הקול הם התדר. ככל שמתקבלים יותר "טיקים" – התדר יותר גבוה. בספר המומלץ קיימת טבלה שממירה בין תדרים לצלילים שמוכרים לנו. בעיקרון יש חוקיות מתמטית פשוטה – כל אוקטבה מורכבת מ-12 צלילים. האוקטבה הבאה היא בתדר כפול. בתוך אוקטבה, הצלילים רחוקים זה מזה במכפלות של שורש 12 של 2 (בערך 1.06), כלומר אם התדירות של תו מסוים היא 110 הרץ, אז התדירות של התו הבא אחריו היא 110*1.06, כמעט 117 הרץ, והתדירות של אותו תו באוקטבה הבאה היא 110*2 הרץ, 220 הרץ.

על מנת לנגן תו כלשהו יש להפעיל קודם לכן את הרמקול של המחשב (speaker). אנחנו עושים זאת על־ידי פורט 61h, המקושר לטיימר. צריך לקרוא את הסטטוס שלו, לשנות את שני הביטים האחרונים ל־00, ולכתוב חזרה לפורט 61h. כך:

```
in      al, 61h
or      al, 00000011b
out     61h, al
```

הפסקת פעולת הרמקול:

```
in      al, 61h
and     al, 11111100b
out     61h, al
```

כעת, כאשר הרמקול פועל, עלינו לשלוח את תדר התו שנרצה להשמיע. 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 לשלוח ל־port 42h "מחלק" (divisor) בגודל 16 ביט, עבור התו שאנחנו רוצים להשמיע. המחלק יסמל את תדר התו הרצוי.

כדי לקבל את המחלק יש לחלק את הקבוע 1193180 בתדר של התו הרצוי. כלומר:

$$Divisor = \frac{1193180}{Frequency}$$

port 42h הוא בגודל 8 ביט בלבד, ולכן יש לשלוח את המחלק הרצוי בשני חלקים – ראשית את הבית הפחות משמעותי, ואחריו המשמעותי.

למשל, עבור התו "לה" של האוקטבה הראשונה, שהתדר שלו הוא 440 הרץ, המחלק יהיה 2712, ובבסיס הקסדצימלי 0A98h. לכן השליחה תתבצע 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

הקדשנו לנושא הטיימר סעיף משלו בפרק על הפסיקות – חיזרו על סעיף זה ובידקו שאתם מבינים את התיאוריה לפני שאתם מתקדמים הלאה. כזכור, אפשר לקרוא את השעה באמצעות פסיקה int 21h עם קוד ah=2Ch. בחלק זה אנחנו עוסקים בכלים לפרוייקטים ולכן נתמקד בשני כלים מעשיים שנותן לנו השעון.

הכלי הראשון הוא יכולת למדוד פרקי זמן קצובים שקבענו מראש – נניח שאנחנו רוצים שפעולה כלשהי תתבצע כל זמן קצוב. לדוגמה:

- במשחק סנייק, לקבוע כל כמה זמן הנחש יתקדם צעד אחד.
- כשמנגנים צליל כלשהו, לקבוע למשך כמה זמן ינוגן הצליל.
- במשחק שחמט, לקבוע פרק זמן מקסימלי לצעד.

הכלי השני הוא יכולת ליצור מספרים אקראיים. זהו כלי חשוב אם אנחנו רוצים ליצור משחקים מעניינים, שלא יחזרו על עצמם. לשם כך, אנו צריכים מנגנון שנותן לנו ערכים שמשתנים בכל פעם שאנחנו מריצים את המשחק, ואת הערכים האלה אנחנו יכולים אחר כך לתרגם לדברים אחרים שאנחנו צריכים. לדוגמה:

- במשחק סנייק, הצבה של אוכל במיקום אקראי במסך.
- במשחקי קוביה, כמו סולמות וחבלים, לקבוע את הערך של הקוביה.
- במשחקי פעולה, תזוזה אקראית של דברים על המסך.

מדידת זמן

דרך אפשרית אחת למדוד זמן שעובר היא על־ידי הפסיקה שקוראת את השעון – מבצעים לולאה של קריאה חוזרת לפסיקה, ובכל פעם משווים את השעה עם השעה הקודמת (מספיק להשוות את ערך מאיות השניה, שנשמר לתוך dl – אם הוא לא השתנה אז שאר הערכים בהכרח לא השתנו). הדרך הזו נראית ברורה מאליה ולכן לא נתעמק בה. הנקודה היחידה שצריך לשים לב אליה, היא שלמרות שהשעה משתנה כל 55 מילישניות, השינוי הראשון של השעה לא יהיה בהכרח אחרי 55 מילישניות. זאת מכיוון שהקריאה הראשונה של השעה בוצעה לא בדיוק ברגע השתנות השעה ולכן ייקח פחות מ־55 מילישניות עד שהשעה תשתנה. רק מהמדידה השניה והלאה אנחנו יכולים להניח שהפרש הזמנים הוא מדויק.

הדרך השניה לדעת ש־55 מילישניות עברו היא להשתמש בעובדה שכל פסיקה של הטיימר גורמת לעדכון של מונה שנמצא בכתובת 0040:006Ch. אנחנו לא נדע מה השעה, אבל כל שינוי בערך שנמצא בכתובת הזו בזיכרון יצביע על כך שעברו 55 מילישניות. היתרון של השיטה הזו הוא מהירות. קריאה ישירה תמיד תהיה מהירה יותר מפסיקה (עם כל הפעולות

הנוספות שהיא גוררת, כגון שמירת ערכים במחסנית וביצוע קפיצות). באופן כללי, אם יש קוד שאנחנו מריצים לעיתים קרובות אז עדיף לשים בצד את שיקולי הנוחות ולכתוב אותו בדרך היעילה יותר.

להלן דוגמה לתוכנית שמודדת פרק זמן של עשר שניות (בקירוב רב). הרעיון הוא כזה: בודקים את מצב המונה בכתובת 0040:006Ch. מחכים שהוא ישתנה פעם אחת – כך אנחנו יודעים מתי מתחיל פרק הזמן שאנחנו רוצים למדוד. אנחנו סופרים 182 שינויים של המונה (182x0.055sec=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           equu    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 באמת באמצעות מחשב היא בעיה מעניינת מאוד עם השלכות פילוסופיות. בעיקרון, 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 באמת. כל מספר אקראי שהמחשב יוצר הוא למעשה תוצאה של חישוב שניתן לשחזר אותו. למחשבה פילוסופית זו יש השלכות מעשיות רבות, לדוגמה בעולם ההימורים אונליין.

לכן אנחנו נכיר מושג שנקרא מספרים **פסאודו־אקראיים (Pseudo-random)**. אלו מספרים שניתן לשחזר אותם באמצעות ידיעת האלגוריתם ותנאי ההתחלה, אבל הם עדיין עונים לחוקים סטטיסטיים של אקראיות (הערכים מתפלגים בצורה אחידה, לא חוזרים על עצמם וכו'. אין צורך שנכיר יותר לעומק את המשמעות של מספרים פסאודו אקראיים, תלמידים המתעניינים בנושא מוזמנים לפנות ללימוד עצמי). מעכשיו, בכל פעם שנכתוב "אקראי" נתכוון בעצם "פסאודו אקראי".

קיימות שיטות שונות ליצירת מספרים אקראיים. לא נוכל להתייחס במסגרת ספר זה לכולן ואפילו לא למקצתן. נתאר אפוא שיטה אפשרית אחת, ותלמידים המעוניינים להרחיב את הידע בנושא מוזמנים להמשיך את הלימוד באופן עצמאי.

השיטה מבוססת על מונה הטיימר שהכרנו בסעיף הקודם – הכתובת 0040:006Ch. אנחנו יכולים לקחת חלק מהביטים שלו וכך לקבל מספר אקראי בתחום שאנחנו רוצים.

לדוגמה, אם ניקח את הביט האחרון נקבל מספר אקראי: 0 או 1. קטע קוד ששם מספר אקראי (0 או 1) בתוך al:

```
mov    ax, 40h
mov    es, ax
mov    ax, es:6Ch
and    al, 00000001b
```

אם ניקח את שני הביטים האחרונים נקבל מספר אקראי מבין: 00, 01, 10, 11. כלומר, מספר בין 0–3. נבצע זאת באמצעות שינוי קטן בשורה האחרונה:

```
and    al, 00000011b
```

וכך הלאה. אנחנו יכולים לקבל מספר אקראי בתחום 0–7, 0–15 וכו'.

שימו לב לכך שבשיטה הזו יצירת מספר אקראי בין 0–9, או בכל תחום אחר שאינו חזקה של 2, היא אינה פשוטה. לכאורה אפשר לקבל את התחום בין 0–9 על-ידי חיבור של שלוש פעולות ליצירת מספרים אקראיים:

- יצירת מספר אקראי בתחום 0–1.

- יצירת מספר אקראי נוסף בתחום 0–1.
- יצירת 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 מצירוף אחד שמוביל לתוצאה 7 (0,0,7 או 1,0,6 או 1,1,5 או 1,5,1 או 6,1,0 ועוד...).

פתרון פשוט הוא ליצור מספר אקראי בתחום גדול יותר מהתחום שאנחנו צריכים, ואם המספר שנוצר לנו הוא מחוץ לתחום המבוקש – לזרוק אותו ולבחור במספר אחר. לדוגמה, כדי ליצור מספר אקראי בתחום 0–9, ניצור מספר אקראי בתחום 0–15 ונזרוק את כל המספרים שגדולים מ־9.

הבעיה השניה בשיטה זו, היא קצב היצירה של מספרים אקראיים. קצב זה תלוי בקצב השינוי במונה שבכתובת 0040:006Ch, שכאמור מתעדכן כל 55 מילישניות. כלומר, אם נרצה ליצור כמה מספרים אקראיים בפרק זמן קצר, לא נקבל גיוון בתוצאות, או שנקבל מספרים שיש ביניהם קשר ברור. כלומר, השיטה הזו טובה בעיקר אם רוצים ליצור מספר אקראי פעם אחת לפרק זמן ארוך יחסית לפרק הזמן של עדכון הטיימר.

פתרון אפשרי הוא לבצע פעולה מתמטית כלשהי על הביטים שאנחנו קוראים מהטיימר, כך שגם אם קראנו את אותם הביטים מהטיימר במספר קריאות נפרדות, עדיין נפיק מהם ביטים אחרים בכל פעם. אנחנו יכולים, לדוגמה, לעשות לביטים שקראנו מהטיימר xor עם ביטים במקום כלשהו בזיכרון – בכל פעם עם מקום אחר. לחלופין, ניתן לקחת קובץ כלשהו, לקרוא ממנו בית ולעשות איתו xor. בכל פעם שנקרא מהקובץ, נקרא בית אחר.

להלן דוגמה לתוכנית שיוצרת מספרים אקראיים באמצעות שילוב של קריאת הטיימר ו־xor עם ביטים שנמצאים ב־CODESEG. נסכם את הסעיף על יצירת מספרים אקראיים בכך שקיימות שיטות רבות וטובות מזו ליצירת מספרים אקראיים, ומי שמתעניין בנושא יוכל למצוא עושר של מידע באינטרנט.

```
; --------------------------------------------------------------------------------------------
; Generate 10 random numbers between 0–15
; The method is by doing xor between the timer counter and some bits in CODESEG
; Author: Barak Gonen 2014
; --------------------------------------------------------------------------------------------
IDEAL
MODEL        small
```

```
STACK       100h
DATASEG
Clock       equ   es:6Ch
EndMessage  db    'Done',13,10,'$'
divisorTable db   10,1,0
CODESEG
proc    printNumber
        push    ax
        push    bx
        push    dx
        mov     bx,offset divisorTable
nextDigit:
        xor     ah,ah               ; dx:ax = number
        div     [byte ptr bx]       ; al = quotient, ah = remainder
        add     al,'0'
        call    printCharacter ; Display the quotient
        mov     al,ah               ; ah = remainder
        add     bx,1                ; bx = address of next divisor
        cmp     [byte ptr bx],0 ; Have all divisors been done?
        jne     nextDigit
        mov     ah,2
        mov     dl,13
        int     21h
        mov     dl,10
        int     21h
```

```
        pop     dx
        pop     bx
        pop     ax
        ret
endp    printNumber

proc    printCharacter
        push    ax
        push    dx
        mov     ah,2
        mov     dl, al
        int     21h
        pop     dx
        pop     ax
        ret
endp    printCharacter

start:
        mov     ax, @data
        mov     ds, ax
        ; initialize
        mov     ax, 40h
        mov     es, ax
        mov     cx, 10
        mov     bx, 0
```

```
RandLoop:
        ; generate random number, cx number of times
        mov     ax, [Clock]             ; read timer counter
        mov     ah, [byte cs:bx]        ; read one byte from memory
        xor     al, ah                  ; xor memory and counter
        and     al, 00001111b           ; leave result between 0-15
        inc     bx
        call    printNumber
        loop    RandLoop
        ; print exit message
        mov     dx, offset EndMessage
        mov     ah, 9h
        int     21h
exit:
        mov     ax, 4c00h
        int     21h
END start
```

ממשק משתמש

בסעיף זה נעסוק בשני כלים שכל תוכנית שעובדת עם ממשק משתמש צריכה – מקלדת ו/או עכבר.

קליטת פקודות מהמקלדת

בפרק על הפסיקות כיסינו את התיאוריה והפרקטיקה של פעולת המקלדת ואת הגורמים הקשורים לפעולה של המקלדת:

- Scan Codes
- פורטים שקשורים לעבודה עם המקלדת
- שימוש בפסיקות BIOS לקליטת תווים וניקוי באפר המקלדת
- שימוש בפסיקות DOS לקליטת תווים וניקוי באפר המקלדת

עקרונית, עברנו על כל הטכניקות שצריך בשביל לשלב מקלדת בפרוייקט הסיום. בסעיף זה ניתן דגשים לעבודה עם המקלדת:

1. כאשר אנחנו צריכים לתכנת פרוייקט שמשלב קליטת פקודות מקלדת, אנחנו יכולים להחליט באיזו גישה לעבוד (פורטים / BIOS / DOS). כמו שראינו, פסיקות BIOS מאפשרות לנו לקבל קלט מבלי לעצור את ריצת התוכנה, ופסיקות DOS מאפשרות לנו לחכות למשתמש. כלומר, המימוש המומלץ תלוי בצורה בה אנו רוצים שהתוכנה תפעל.
2. שימוש בפסיקת ה־DOS int 21h מחזיר את קוד ה־ASCII של המקש שהוקלד. ישנם מקשים – ודווקא כאלה שצריך להשתמש בהם במשחקים (מקשי חיצים לדוגמה) – שיש להם קוד ASCII מורחב. בקוד ASCII מורחב, לתוך באפר המחסנית מועתק קוד ה־ASCII במקום של ה־scan code, ובמקום של קוד ה־ASCII מועתק אפס. מבלבל? נכון. מסיבה זו, עדיף תמיד לבצע את פעולות הבדיקה וההשוואה עם ה־scan code. גם אם משתמשים בפסיקת DOS, עדיף להוסיף אחריה:

```
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 בקלות 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.

לפני שנפעיל את העכבר, נעבור למוד גרפי, כפי שלמדנו לעשות מוקדם יותר בפרק זה:

```
mov     ax,13h
```

```
int     10h
```

העכבר נשלט בידי פרוצדורות שונות שמפעילה פסיקה 33h. הקודים נשלחים על גבי ax (שימו לב, ax ולא ah כמו שאנחנו רגילים עם int 21h). כדי להפעיל את העכבר, ראשית יש לאתחל אותו ואת החומרה שלו. הקוד ax=0h מבצע את פעולת האתחול:

```
mov     ax,0h
int     33h
```

כעת נציג את העכבר על המסך, קוד ax=1h:

```
mov     ax,1h
int     33h
```

השלב הבא הוא לקלוט את מיקום העכבר ואת סטטוס הלחיצה עליו:

```
mov     ax,3h
int     33h
```

הפסיקה מחזירה את הערכים הבאים:

- bx – מצב הלחיצה על כפתורי העכבר
  - xxxx xxx1 – הביט במקום 0 שווה '1' – כפתור שמאלי לחוץ
  - xxxx xx1x – הביט במקום 1 שווה '1' – כפתור ימני לחוץ

  כלומר, כל ערך של bx שהביטים הימניים שלו הם '00' – משמעותו שאין כפתור לחוץ.
- cx – מיקום העכבר, שורה בין 0–639 (שימו לב, כשאנחנו עובדים במוד גרפי כמות השורות שיש לנו היא 320 בלבד, לכן צריך לבצע התאמה ולחלק את cx בשתיים כדי להגיע למיקום הנכון).
- dx – מיקום העכבר, עמודה בין 0 –199

התוכנית הבאה משלבת את האלמנטים שסקרנו. לחיצה על המקש השמאלי של העכבר תוביל להופעת נקודה אדומה במיקום העכבר. לאחר מכן, לחיצה על המקלדת תגרום ליציאה מהתוכנית.

```
; --------------------------------------------------------------------------------------
; PURPOSE : Paint a point on mouse location, upon left mouse click
; AUTHOR  : Barak Gonen 2014
; --------------------------------------------------------------------------------------
IDEAL
MODEL small
STACK 100h
DATASEG
color	db 12

CODESEG
start:
	mov	ax,@data
	mov	ds,ax
	; Graphics mode
	mov	ax,13h
	int	10h
	; Initializes the mouse
	mov	ax,0h
	int	33h
	; Show mouse
	mov	ax,1h
	int	33h
	; Loop until mouse click
```

```
MouseLP:
        mov     ax,3h
        int     33h
        cmp     bx, 01h         ; check left mouse click
        jne     MouseLP
        ; Print dot near mouse location
        shr     cx,1            ; adjust cx to range 0-319, to fit screen
        sub     dx, 1           ; move one pixel, so the pixel will not be hidden by mouse
        mov     bh,0h
        mov     al,[color]
        mov     ah,0Ch
        int     10h
        ; Press any key to continue
        mov     ah,00h
        int     16h
        ; Text mode
        mov     ax,3h
        int     10h
exit:
        mov     ax,4C00h
        int     21h
END start
```

שיטות ניפוי Debug

עד עכשיו עסקנו בתוכניות לא ארוכות במיוחד – את כל התרגילים שבספר הלימוד אפשר לבצע עם תוכניות מסדר גודל של מאה שורות קוד, ובדרך כלל הרבה פחות. לעומת זאת, בפרוייקט סיום יש למעלה מאלף שורות קוד. אפילו אם אתם מתכנתים מעולים ויש לכם באג רק אחת למאה שורות קוד, זה עדיין אומר שתיתקלו בעשרות באגים לפני שהתוכנית שכתבתם תעבוד כמו שתכננתם. ואנחנו עוד לא מדברים על באגים שנובעים מבעיה בתכנון, כזו שתאלץ אתכם להעביר חלקי קוד ממקום למקום, או באג באלגוריתם, שעלול לגרום לכם לבלות ערב מול מסך המחשב בעודכם כוססים ציפורניים, נעים בין ייאוש לזעם.

אין הכוונה לייאש אתכם, רק לציין את המובן מאליו – כתיבת תוכנית גדולה דורשת מיומנות תכנות ושימוש בטכניקות דיבוג, שאינן נחוצות בשביל כתיבת תוכנית קטנה כמו התוכניות שנתקלנו בהן עד עכשיו.

אנחנו נעסוק עכשיו בטכניקות שונות שיוכלו, אולי, לעזור לכם לצלוח יותר בקלות את אותם ערבים בהם התוכנה לא עובדת ואין לכם מושג למה.

תיעוד

כתיבת תוכנית קטנה אורכת מעט זמן, אז כשאתם מגיעים לסוף התוכנית אתם עדיין זוכרים מה כתבתם בהתחלה. כתיבת תוכנית גדולה אורכת הרבה זמן. אל תניחו שאחרי שבועיים של כתיבה תזכרו מה הפרמטרים שמקבלת פרוצדורה שכתבתם. תתעדו. רמה נכונה של תיעוד היא כזו:

- בתחילת כל פרוצדורה, הסבר מי נגד מי: ציון פרמטרים ותוצרים.
- אחת לכמה שורות קוד, לרשום הערה מה עושה קטע הקוד הבא.
- אם יש שורה מסובכת, כזו שלא מובן מאליו למה היא צריכה להיות שם, הוסיפו הערת הסבר בצד. לדוגמה, בתוכנית לדוגמה על הפעלת העכבר, איך נזכור למה חילקנו את cx ב־2 אם לא נסביר בצד?

```
shr	cx,1		; adjust cx to range 0-319, to fit screen
```

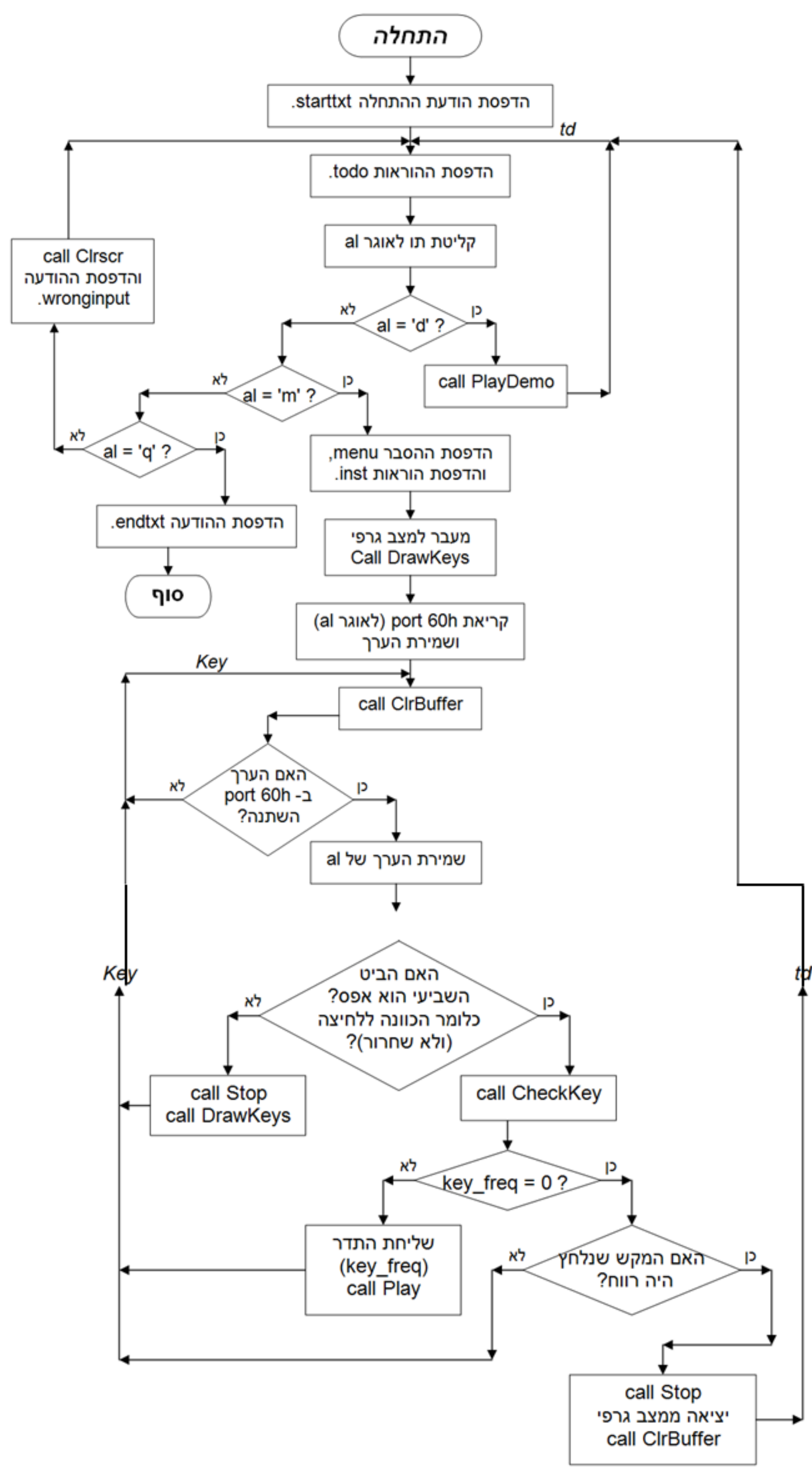
עם זאת, חשוב לא להגזים. מתכנתים מתחילים נוטים לעיתים להוסיף תיעוד רב, שרובו מיותר, או שאינו מסביר למה עשינו את מה שעשינו. דוגמה לתיעוד שלא משרת שום מטרה טובה:

```
mov	ax, 5		; copy ‘5’ into ax
```

תכנון מוקדם – יצירת תרשים זרימה

תרשים זרימה לכלל התוכנית יכול להקל עליכם לעשות סדר בתוכנית המורכבת שלכם ולמנוע טעויות תכנון שיגזלו מכם זמן יקר בהמשך. אל תוותרו על תיאור מסודר של התוכנית שלכם לפני שאתם מתיישבים לתכנת. בכל אופן, גם אם תוותרו לעצמכם, משרד החינוך דורש שלכל תוכנה יתלווה תרשים זרימה, כך שבכל מקרה עליכם להכין אחד... עדיף שתתחילו מתרשים זרימה טוב שיעשה לכם סדר במה שאתם מתכננים לתכנת.

לתרשימי זרימה יש שפה מיוחדת. כדי להקל על הקריאה וההבנה יש טקסט ששמים במלבן, טקסט ששמים בתוך מעוין, טקסט ששמים בעיגול וכו'. תוכלו ללמוד בקלות את שפת תרשימי הזרימה באינטרנט (חפשו "איך לכתוב תרשים זרימה").



דוגמה לתרשים זרימה של פסנתר (אליזבת לנגרמן)

חלוקה לפרוצדורות

קרוב לוודאי שאחרי שיצרתם תרשים זרימה, זיהיתם מספר קטעי קוד שצריכים לרוץ מספר רב של פעמים. אלו הן הפרוצדורות שאתם מיועדים לכתוב. אם אתם רוצים שיהיה לכם סיכוי סביר למצוא באגים, אתם צריכים לחשוב היטב על החלוקה לפרוצדורות. היתרון העיקרי של פרוצדורות הוא בשלב הדיבוג - אם כתבתם פרוצדורה טובה ובדקתם אותה, אתם לא צריכים להיות מודאגים בקשר אליה. אנחנו נציג כרגע כמה דרכים לדבג תוכנית שכוללת פרוצדורות:

- לחפש בשיטת "אריה במדבר" – אם אנחנו רוצים למצוא אריה שמסתתר במדבר, אנחנו נחלק את המדבר לשטחים, נקיף כל שטח בגדר, ואחרי שנסיים לסרוק את השטח המגודר 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 מריצים אותה והתמונה לא עולה. פשוט כלום לא קורה. מה עושים? בשיטת "אריה במדבר" נכניס את הפרוצדורות 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
```

נריץ את התוכנית, שכרגע כל מה שהיא עושה זה רק לפתוח קובץ. נבדוק שהקובץ נפתח בצורה תקינה (לא מוחזר קוד שגיאה). הכל תקין? המשכנו הלאה.

- לקרוא זיכרון תוך כדי ריצה. ה־TD מספק לנו תמיכה טובה בבדיקת מצב הזיכרון תוך כדי ריצה. נניח שהוצאנו את הקריאה ל־ReadHeader מהערה, ואנחנו רוצים לבדוק אם הפרוצדורה עובדת בצורה תקינה. בתוך DATASEG הגדרנו מערך ששומר את ה־header. נוכל לראות איך header נראה בזיכרון מיד בסיום

הקריאה. אנחנו יודעים ששני הבתים הראשונים שלו צריכים להיות 'BM' – אם קיבלנו ערך אחר, יש לנו בעיה בקריאה.

- להחליף פרוצדורות בקטעי קוד קבועים, שאנחנו יודעים מה הם צפויים לבצע. נניח שיש לנו פרוצדורה שבודקת איזה מקש נלחץ על־ידי המשתמש, והיא מעבירה את המידע לפרוצדורה אחרת שעושה שימוש במידע הזה, נניח כדי להזיז אלמנט גרפי על המסך. 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 תידרס ובמקומה יועתק ערך קבוע. אם הבעיה ממשיכה, כנראה שיש בעיה בפרוצדורה שמזיזה את הגרפיקה על המסך.

שימו לב, שכדי שתוכלו לדבג בצורה יעילה, 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 שחוזרים על עצמם בתוך הפרוצדורה שלכם, ואם הפרוצדורה לא ארוכה מדי.

מעקב אחרי מונים

מונים, כאלה שסופרים בשבילנו כמה פעמים לולאה שכתבנו צריכה לרוץ, הם מקור נפוץ לשגיאות ובעיות. לכן, אם יש בעיה בקוד שלכם, שווה לבצע בדיקה של המונים שלכם ואם לא לגמרי במקרה הכנסתם לתוכם ערך שהם לא היו אמורים לקבל (לדוגמה, הרצתם לולאה ואז קראתם לפרוצדורה שמשנה את cx).

אם אתם לא שמים לב לבעיה, מומלץ להכניס לתוכנית שלכם קטעי קוד שמדפיסים את הערכים של המונים למסך או אפילו לקובץ. כך תוכלו לוודא שהמונים שלכם מקבלים רק ערכים בתחום הצפוי וההגיוני. יעזור לכם אם תכתבו פרוצדורה או מקרו, שמקבלים כפרמטר רגיסטר ומדפיסים את הערך שלו למסך או לקובץ, במקום לממש את הלוגיקה הזו בכל פעם מחדש.

העתקות זיכרון

עוד באג נפוץ מאוד הוא העתקה למקום לא נכון בזיכרון. אין הכוונה להעתקה למקום לא נכון במערך, בעיה נפוצה בפני עצמה, אלא להעתקה של מידע מחוץ למערך שהגדרתם. העתקה כזו עלולה לדרוס לכם ערכים אחרים ב־DATASEG, או – אם ממש יש לכם מזל – לדרוס ערכים ב־CODESEG ולסיים בריסוק התוכנית.

נקודה נוספת שקשורה להעתקות לזיכרון, היא שלעיתים אנחנו צריכים לשנות את הערכים של רגיסטרי הסגמנט. לאחר השינוי צריך להחזיר אותם למצב המקורי, אחרת אנחנו מסתכנים בתוצאות בלתי צפויות בכל פניה לזיכרון.

הודעות שגיאה של האסמבלר

לעיתים האסמבלר יגרום לכם להרגיש חסרי מזל במיוחד, כשתתקלו בהודעת שגיאה שלא נתקלתם בה בעבר, לדוגמה: A2034: must be in segment block. מה הפירוש של זה? אל דאגה – בטוח שמישהו נתקל בהודעה הזו לפניכם. העתיקו את קוד השגיאה למסך החיפוש, קרוב לודאי שתגיעו לפורום מתכנתים (לדוגמה StackOverflow) ותמצאו שמישהו כבר נתקל בבעיה שלכם וקיבל עזרה.

תרגיל הכנה לפרוייקטי הסיום

בתרגיל זה נתרגל שימוש במספר אבני בניין שנחוצות לצורך פרוייקטי הסיום

א. כיתבו תוכנית, שעם לחיצה על מקש כלשהו במקלדת תדפיס למסך הודעה "Key pressed" ועם שחרור המקש תדפיס הודעה "Key released". לחיצה על ESC תסיים את ריצת התוכנית. שימו לב- לחיצה ארוכה נחשבת לחיצה אחת. לכן, הדרך הנכונה לבצע את הבדיקה היא לשמור את ה-scan code האחרון שהגיע מהמקלדת ורק אם הוא השתנה אז להדפיס הודעה בהתאם.

ב. שדרגו את התוכנית, כך שלחיצה על המקלדת גם תשמיע צליל כלשהו. הצליל יופסק רק עם שחרור המקלדת.

ג. שדרגו את התוכנית, כך שבמקום להדפיס הודעה למסך יצוייר ריבוע אדום 5X5 פיקסלים במרכז המסך. הריבוע יופיע עם לחיצה על מקש וייעלם עם שחרור המקש. טיפים:

a. מומלץ לצייר את הריבוע באמצעות לולאה ופרוצדורות, ולא באמצעות שכפול קוד של ציור פיקסל 25 פעמים

b. "מחיקת" הריבוע האדום מתבצעת על ידי ציור ריבוע בצבע הרקע במקום הריבוע האדום

ד. שדרגו את התוכנית, כך שמיקום הריבוע במסך יהיה אקראי, כל פעם ייבחר מיקום חדש

סיכום

בפרק זה עסקנו במגוון נושאים רלבנטיים לכתיבת פרוייקט סיום. למדנו איך לשלב אלמנטים גרפיים בתוכנית שלנו – ראינו איך אפשר בקלות להוסיף גרפיקת ASCII שמשדרגת את הפרוייקט ואיך משלבים תמונות בפורמט bmp הנפוץ. למדנו עבודה עם קבצים, עבודה עם עכבר, יצירת מספרים אקראיים. סיימנו בטיפים לכתיבת פרוייקט ובאופן כללי כיצד יש לגשת לפרוייקט כתיבת תוכנה בהיקף משמעותי.

סיימנו את לימוד שפת האסמבלי ומבנה המחשב, אנו מוכנים לאתגרים הבאים שלנו.

לתלמידים: זיכרו, הודות לאינטרנט, ידע אינסופי נמצא במרחק כמה הקשות מקלדת. זוהי מתנה שלא היתה קיימת בדורות קודמים. כשתלמדו למצוא לבד מקורות לימוד, תרתמו את כח הידע של אחרים לטובתכם. מכאן והלאה, דבר לא יוכל לעצור את הסקרנות שלכם. בהצלחה!

נספח א' – רשימת פקודות חובה לבגרות בכתב באסמבלי

לפניכם רשימת פקודות החובה לבגרות בכתב באסמבלי. שימו לב- אלו אינן פקודות חובה עבור תלמידים שלומדים אסמבלי במסגרת יחידת המעבדה (כמו ב"גבהים") אלא רק עבור תלמידים שניגשים לבגרות במחשבים בפרק בחירה באסמבלי.

את רוב הפקודות הכרנו כבר, בפרק זה נפרט אודות הפקודות שטרם הכרנו.

ADD
AND
CALL
CLC
CLI
CMP
DEC
DIV
IDIV
IMUL
IN
INC
INT
IRET
JA
JAE
JB
JBE
JC
JCXZ
JE
JG
JGE
JL
JLE
JMP

JNA
JNAE
JNB
JNBE
JNGE
JNL
JNLE
JNO
JNP
JNS
JNZ, JNE
JO
JP
JPE
JPO
JS
JZ
LAHF
LEA
LOOP
LOOPE
LOOPNE
LOOPNZ
LOOPZ
MOV
MUL

NEG
NOP
NOT
OR
OUT
POP
POPF
PUSH
PUSHF
RCL
RCR
RET
ROL
ROR
SAL
SAR
SBB
SHL

פקודות שקובעות מצבי דגלים:

CLC- הורדת דגל הנשא

STC- הדלקת דגל הנשא

CLI- הורדת דגל הפסיקות

פקודות קפיצה:

מספרים Unsigned	מספרים Signed	תיאור התנאי
JA (JNBE)	JG (JNLE)	קפוץ אם גדול ממש
JB (JNAE)	JL (JNGE)	קפוץ אם קטן ממש
JAE (JNB)	JGE (JNL)	קפוץ אם גדול או שווה
JBE (JNA)	JLE (JNG)	קפוץ אם קטן או שווה
JE	JE	קפוץ אם שווה
JNE	JNE	קפוץ אם שונה

JCXZ- קפוץ אם cx=0

פקודות קפיצה שבודקות ישירות את מצב הדגלים:

הוראה	תיאור
JC	קפוץ אם דגל הנשא דלוק
JNC	קפוץ אם דגל הנשא כבוי

קפוץ אם דגל האפס דלוק	JZ
קפוץ אם דגל האפס כבוי	JNZ
קפוץ אם דגל הסימן דלוק	JS
קפוץ אם דגל הסימן כבוי	JNS
קפוץ אם דגל הגלישה דלוק	JO
קפוץ אם דגל הגלישה כבוי	JNO
קפוץ אם דגל הזוגיות דלוק	JP / JPO
קפוץ אם דגל הזוגיות כבוי	JNP / JPE

פקודות רגיסטר דגלים:

LAHF- מעתיק את 8 הביטים הנמוכים של רגיסטר הדגלים ל-ah

PUSHF- מעתיק את רגיסטר הדגלים למחסנית

POPF- מעתיק את תוכן ראש המחסנית אל רגיסטר הדגלים

פקודות לולאות נוספות:

נוסף ל-LOOP שאנחנו מכירים, ישנן הפקודות הבאות-

LOOPE (LOOPZ)- כמו LOOP, אם cx לא שווה לאפס מתבצעת קפיצה לתווית. אך יש עוד תנאי לקפיצה-נדרש שדגל האפס יהיה דולק. לכן שימושי לבצע פקודת השוואה cmp לפני ביצוע LOOPE או LOOPZ.

LOOPNE (LOOPNZ)- כמו LOOP, אם cx לא שווה לאפס מתבצעת קפיצה לתווית. אך יש עוד תנאי לקפיצה-נדרש שדגל האפס יהיה כבוי.

פקודות הזזה נוספות:

נוסף על SHL, SHR שאנו מכירים, ישנן הפקודות הבאות-

ROL- כמו SHL, כל הביטים זזים שמאלה והביט השמאלי ביותר מועתק לדגל הנשא, בהבדל אחד- הביט הכי שמאלי מועתק גם אל הביט הכי ימני. כלומר אם ניקח את ah לדוגמה ונבצע לו ROL שמונה פעמים, הוא יחזור למצבו המקורי.

ROR- כמו SHR, כל הביטים זזים ימינה והביט הימני ביותר מועתק לדגל הנשא, בהבדל אחד- הביט הכי ימני מועתק גם אל הביט הכי שמאלי.

RCL- כמו SHL, אך דגל הנשא מועתק אל הביט הכי ימני.

RCR- כמו SHR, אך דגל הנשא מועתק אל הביט הכי שמאלי.

SAL- זהה ל-SHL

SAR- זהה ל-SHR

פקודת חיסור מיוחדת:

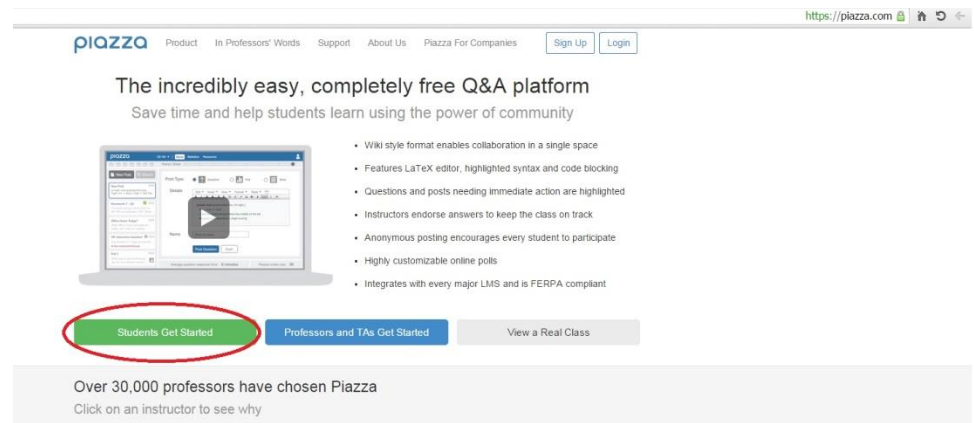
SBB- כמו SUB, אבל מוסיפה לתוצאה את ערכו של דגל הנשא.

נספח ב' – מדריך לתלמידים: כיצד נכנסים לפורום האסמבלי הארצי

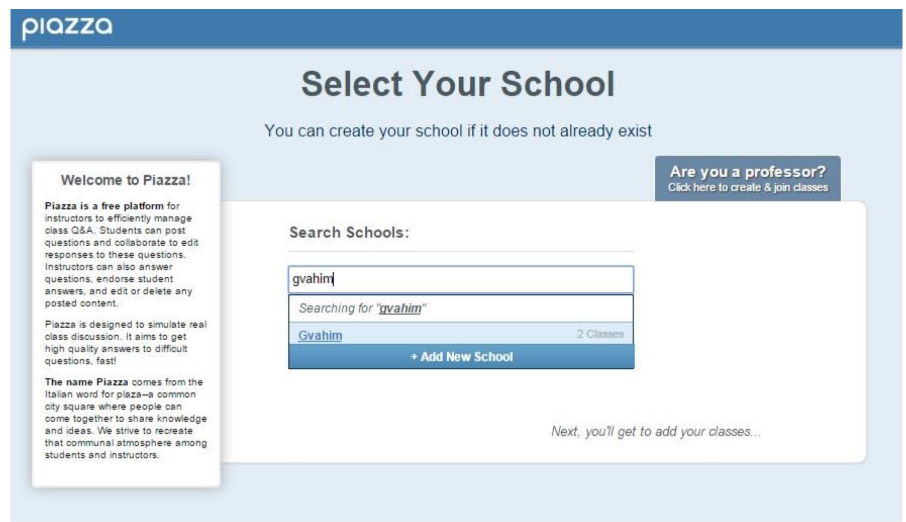
Piazza היא פורום שאלות ותשובות שמיועד לתלמידים בכל הארץ. ניתן לפתוח פורום בכל נושא, לכתוב שאלה ולענות תשובות. כיוון שתלמידים בכיתות גבהים בכל הארץ נתקלים בבעיות דומות, עומדת לרשותכם "אוניברסיטה" וירטואלית, שם תוכלו לסייע אחד לשני. בנוסף, התשובות שלכם יסייעו לתלמידים בשנים הבאות!

לכניסה ל-Piazza:

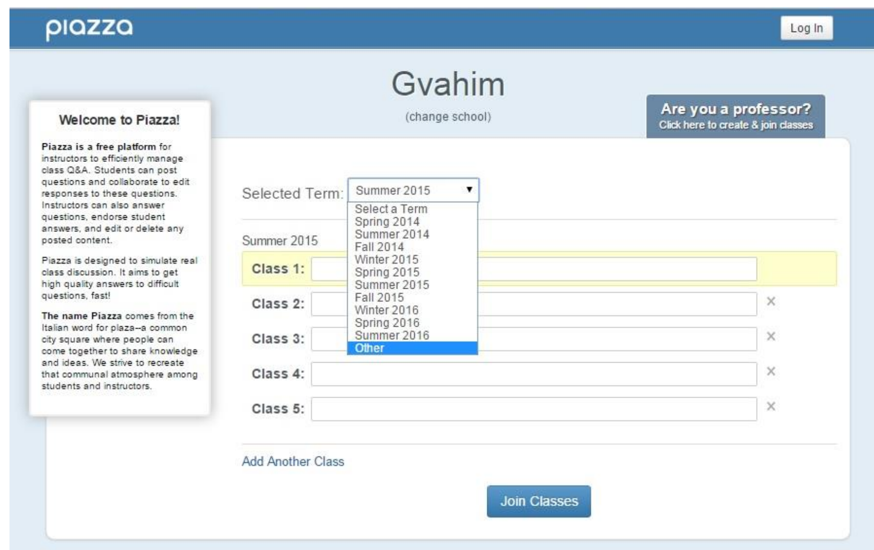
1. היכנסו לאתר www.piazza.com ובחרו "students get started"



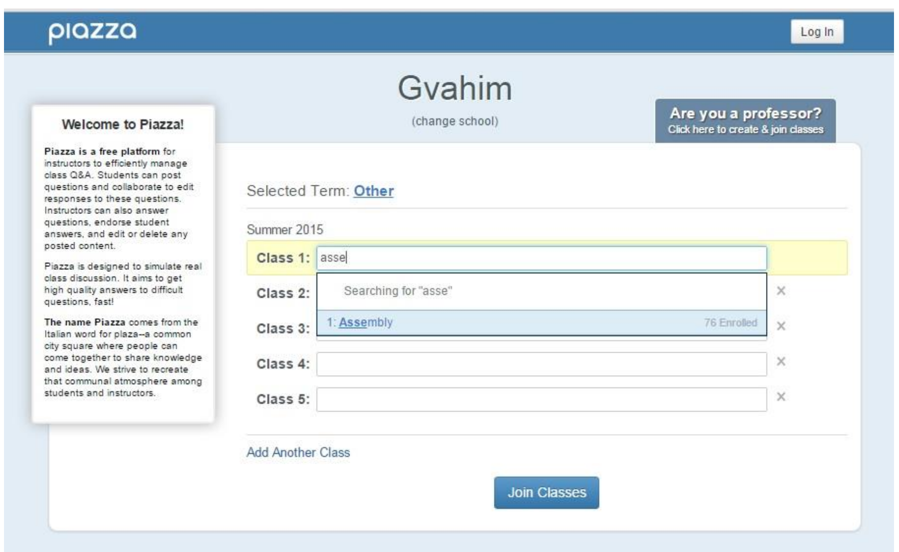
2. בחרו באוניברסיטת "gvahim"



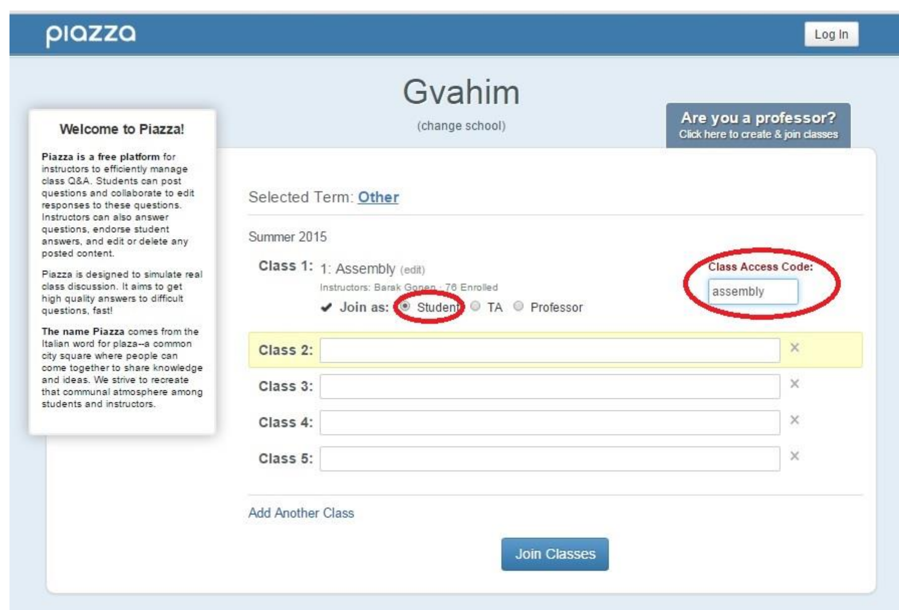
3. בחרו סמסטר "other"



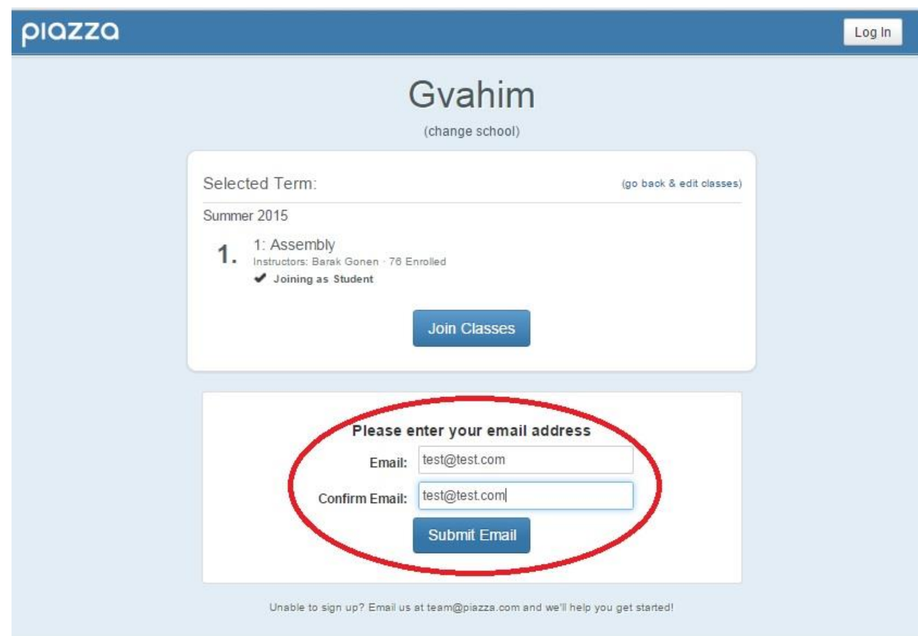
4. בחרו קורס assembly



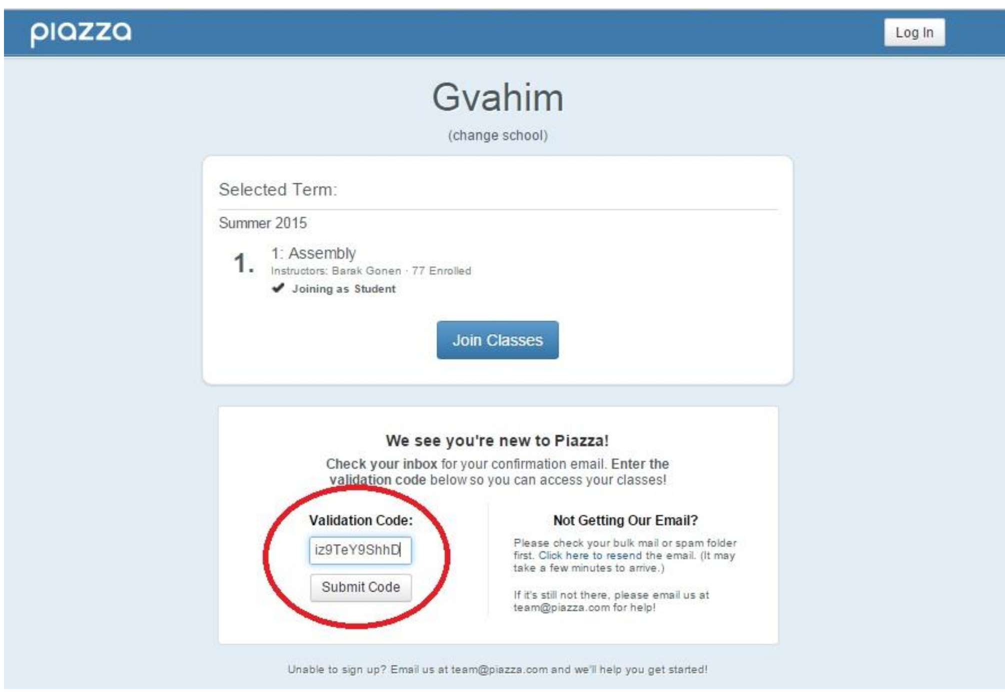
5. בחרו להצטרף כ-student והכניסו סיסמה "assembly"



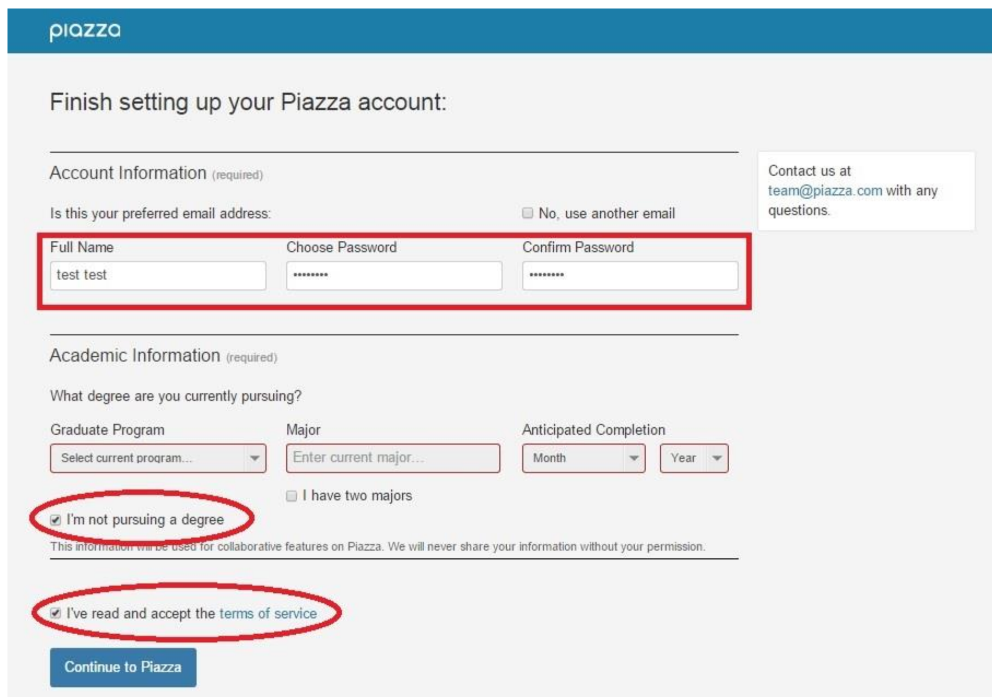
6. הכניסו כתובת מייל, אליה יישלח קוד הפעלה של האתר, וליחצו על submit email



7. הכניסו את הקוד שקיבלתם ולחצו על submit



8. בחרו שם משתמש וסיסמה. למטה בחרו באופציה I am not pursuing a degree ואשרו שקראתם את תנאי השימוש. לאחר מכן לחצו על Continue- זהו, סיימתם. ברוכים הבאים.



<u>הכנסת שאלה חדשה</u>

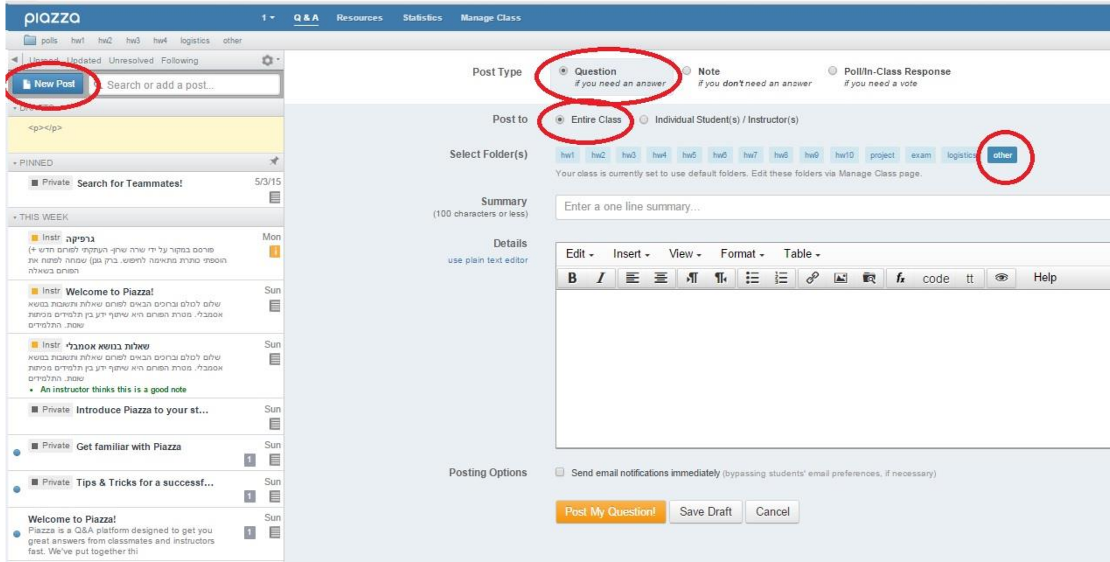
על מנת לשאול שאלה, הקישו על new post. בחלון שייפתח בחרו

- Post type = Question

- Post to = Entire class

- Folder = Other

אל תשכחו לתת כותרת לשאלה, ולבסוף הקישו על Post my question



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/