מבוא לאסמבלי

מאת אופיר בק

חלקים נרחבים ממאמר זה נכתבו בהשראת הספר "ארגון המחשב ושפת סף" אשר נכתב ע"י **ברק גונן** לתוכנית גבהים של משרד החינוך.

הקדמה

בסדרת המאמרים הקרובה, אנחנו הולכים ללמוד על השפה אסמבלי, על השימוש בה ואף נבנה בעזרה מספר תוכנות קטנות בשביל הכיף. במבוא נסביר קצת על אסמבלי, איך להריץ אותו ונראה את הקוד הראשון שלנו. אך בשלב זה אסביר בעיקר על מבנה המחשב, מידע אשר הכרחי בשביל לדעת אסמבלי.

מהי שפת אסמבלי?

"**שפת סף** אשר נקראת גם **אסמבלי** היא שפת התכנות הבסיסית ביותר והקרובה ביותר לשפת מכונה." - ויקיפדיה.

בשונה מהרבה שפות אחרות שניתן ללמוד, כדי ללמוד אסמבלי צריך לדעת קצת יותר על איך המחשב עובד, אז אנחנו נתמקד בדרישות הידע הכי פשוטות בחלק הזה.

התפתחות שפות התכנות

בתחילה כדי להורות למחשב לבצע חישובים, מפתחים נדרשו לרשום את ההוראות בשפה הבינארית, שפת המכונה, עליה נרחיב בהמשך. כלומר קודים היו נראים ככה:

```
0101 1100 1110 1111 1110 0110 0011...
```

הבעיות הראשיות הן שאי אפשר לקרוא את הקוד ולהבין אותו, וקשה למצוא טעויות.

כאן בדיוק נכנסת שפת האמסבלי. הפקודות של שפת האסמבלי הן באנגלית, קל להבין את משמעותן ולמצוא טעויות בקלות רבה יותר. לכן נקראת שפת האסמבלי שפת סף, כי היא על סף להיות שפת מכונה. הפקודות של השפה דרושות הכרה עמוקה עם המעבד כדי למנוע טעויות קריטיות שעלולות לפגוע במחשב, אך אנו נשתמש ב-Emulator שידמה מערכת הפעלה ישנה שתחסוך לנו את הדאגה.

מעל הרמה הזאת יושבות שפות יותר פשוטות לקריאה ולכתיבה, בנוסף למציאת טעויות, והן נקראות שפות עיליות, וביניהן ניתן למצוא שפות כמו C# ו-Java. השפות האלו בנויות עם מנגנונים שמונעים טעויות של המעבד ולא מריצים את הקוד במידה והם מוצאים טעויות. הן בעצם 'מסתירות' מהתוכניתן את המעבד.

למה לא ללמוד אסמבלי?

כן, אני לא הולך להסתיר מכם את הקשיים שיש בלימוד אסמבלי:

1. מאז שאסמבלי פותחה (בשנת 1949) העולם התקדם ויש שפות תוכנה מודרניות
2. אסמבלי היא שפה שקשה ללמוד
3. מסובך וארוך לכתוב קודים בשפת אמסבלי
4. קשה לבצע דיבאג (למצוא שגיאות ולנפות אותן) באסמבלי, לעומת השפות העיליות

למה כן ללמוד אסמבלי?

בכל זאת, יש יתרונות בלימוד השפה וידיעתה:

1. ניצול מיטבי של משאבי המעבד
2. עבודה מול החומרה של המחשב
3. גודל קובץ קטן מאוד ביחס לשפות אחרות (ראיתם פעם את התיקייה שנוצרת עבור כל קובץ C# קצרצר?)
4. הבנה עמוקה של אופן פעולות המחשב (אין דברים שמוסתרים מהתוכניתן)
5. רכישת מימנויות של סדר וארגון

שיטות ספירה

בני אדם נוהגים לספור בשיטה של 10 ספרות, מ-0 ועד 9, שכאשר אנו מוסיפים ספרה משמאל לספרה הראשונה אנו מגדילים את הערך של הספרה השמאלית פי 10 והספרה הבאה אחריה תהיה גדולה פי 100 שזה בעצם 10^2 וכן הלאה, לדוגמה 16 זה בעצם 1×10+6, והמספר 458 הוא בעצם $4\times10^2+5\times10+8$.

לעומת זאת, המחשב סופר בבסיס 2, כלומר, הוא משתמש רק בערכים 1 ו-0. גם פה ייצוג המספרים הגדולים מכמות הספרות נעשים באמצעות הוספה של מספר משמאל למספר השמאלי ביותר. לדוגמה כדי לייצג את המספר שמוכר לנו כ-2 אנו נרשום 10 בבסיס הבינארי (2), מכיוון ש 1×2+0 = 2.

נהוג לרשום את הקוד הבינארי בבלוקים של ארבעה תווים כדי להקל עלינו לקרוא אותו ולרשום בסיום b. כך לדוג' אנו נרשום את המספר 2 בבסיס 2 בתור 0010b, ואת המספר 15 בתור 1111b. כל תו בינארי נקרא סיבית - סיפרה בינארית.

עם זאת, כדי להקל על התוכניתנים, המחשב יודע לקרוא גם בבסיס הקסדצימלי (16). בבסיס הזה אנו משתמשים בספרות ובנוסף להן בתווים A עד F, כדי להשלים את ששת הספרות החסרות. בבסיס הזה נהוג לרשום את המספר עם האות h בסופו או 0x בתחילה.

כדי לתת למחשב את המספר הדצימלי, פשוט לא מוסיפים אף סימן בתחילה או בסוף, וכך הוא מניח שמדובר בשיטת הספירה שאנו משתמשים.

שימו לב! ניתן להמיר בין הבסיסים הקסדצימלי והבינארי בקלות, מכיוון שכל 4 ביטים מייצגים תו הקסה אחד, כך ש 0001b=1h ו- 1111b=Fh.

כאשר אנו עובדים עם בית אחד, הערכים נעים בין 0 ל-255, אך כאשר אנו רוצים לייצג מספר שלילי, הם נעים בין 128- ל-127. המחשב לא באמת מבדיל בין המספרים, אך בעזרת פעולות מסויימות אנו מגדירים לו כיצד להתייחס אליהם.

השיטה להתייחסות לקוד בינארי כשלילי נקראת "משלים ל-2". כדי לייצג מספר בשיטה הזו, אנו מציגים אותו בפן החיובי, הופכים את כל הביטים ומוסיפים אחד. כלומר, המספר 1- בשיטה הזו יתואר בצורה הבאה:

1) תחילה נביע את המספר 1 באופן הבא: 0000 0001.
2) נהפוך את כל הביטים: 1111 1110.
3) נוסיף 1: 1111 1111.

וככה אנו מביעים את המספר 1-.

בשיטה הזו כל מספר שהביט השמאלי שלו הוא 1, הוא שלילי.

מבנה המחשב

כמו שהזכרנו קודם, אסמבלי עובד בצורה הדוקה מול החומרה של המחשב, ולכן יש מספר יתרונות בהכרת המחשב והמבנה שלו:

1. אופטימיזציה של הקוד
2. הקטנת גודל הזיכרון של חלקים בקוד (בהמשך נגלה למה זה משמעותי)
3. מאפשר דיבאג יותר טוב של הקוד.

אנחנו נעבוד עם מעבד 8086 של אינטל, שיצא בשנת 1978 והיה הראשון בסדרה 80x86. הסיבה שעבודה איתו היא רלוונטית היא בגלל עיקרון שנקרא "תאימות לאחור", לפיו גם מעבדים חדשים יותר של אינטל מסוגלים לעבוד עם הקוד שנועד למעבד ישן יותר.

המעבד הנ"ל בנוי בארכיטקטורה שנקראת "ארכיטקטורת פון ניומן", על שם היוצר של המעבד, ג'ון פון ניומן. לפי הארכיטקטורה, ישנם שלושה 'פסים' במערכת, שניתן לפנות רק אל אחד מהם בכל פעם. הפסים האלו הם הבקרה, המענים, והמידע. והם מחוברים בנפרד לרכיבי הקלט והפלט, למעבד ולזיכרון.

פס הבקרה במחשב אחראי לומר למעבד האם אנו מבצעים פעולה של קריאה או של כתיבה, והאם לפנות לקלט-פלט או לזיכרון. הערכים של הקריאה והכתיבה הם בד"כ על 1, וכאשר read=0 מתבצעת קריאה, וכאשר write=0 מתבצעת כתיבה.

פס המענים מודיע לאיזו כתובת בזיכרון המעבד מכוון. בזיכרון כל מקום הוא בגודל של בית (8 ביט), והוא מסודר כך שאם נכניס אליו את הערך 1234h הוא יסודר כך שקודם כל יישמר בזיכרון 34h ורק לאחריו 12h. שיטה זו קרויה little-endian.

פס הנתונים מעתיק נתונים ממקום למקום. מעבד עם פס נתונים רחב יותר יכול להעביר מידע מהר יותר. פס הנתונים של המעבד 8086 מסוגל להעתיק 16 ביטים בהעתקה אחת.

צריך לשים לב שמבחינה עקרונית, זיכרון לא יכול להיות ריק, לעיתים הוא עם מידע שאנו הכנסנו לו, ואז הוא שמיש מבחינתנו, אך לעיתים מדובר במידע זבל שאינו אמין, וקריאה שלו עלולה לגרום לבעיות בהמשך.

בנוסף לכך, מכיוון שלמעבד ה-8086 יש מרחב כתובת של 20 ביטים, הוא משתמש בשיטה של segment (קטע בזיכרון) ו-offset (מיקום בקטע) ופונה לזיכרון במיקום שבנוי בתור segment:offset.

המעבד

המעבד 8086 בנוי ממספר חלקים, ואנו נפרט עבור אלו שחשובים לתכנות האסמבלי שלנו:

- **רגיסטרים - Registers**: הרגיסטרים מתחלקים לכמה סוגים, וכשנתחיל לכתוב תוכניות אנו גם נפרט עליהם.
- **יחידה אריתמטית לוגית - Arithmetic Logic Unit:** היחידה האריתמטית לוגית אחראית על ביצוע פעולות מתמטיות ולוגיות עבור המעבד.
- יחידת בקרה.
- **יחידת קלט/פלט - I/O Ports:** ניתן להשתמש ביחידה הזו כדי לקבל קלט מהמשתמש.
- **שעון - Timer:** מאפשר שימוש בפונקציות זמן מסוגים שונים.

אוגרים

האוגרים, או כפי שנכנה אותם בד"כ, רגיסטרים, הם רכיבי חומרה שצמודים למעבד, והם מאפשרים לנו לבצע מגוון פעולות. בשונה מהזיכרון, אל האוגרים המעבד יכול לפנות באופן ישיר, וללא שום המתנה, בשל מיקומם הפיזי. הכמות והגודל של הרגיסטרים משתנה בין דורות של מעבדים, אבל אנחנו נשתמש ברגיסטרים של מעבד ה-8086.

לרגיסטרים שונים יש שמות שונים ומטרות רשמיות שונות, אך בעיקרון, את מרבית הפעולות רובם יכולים לבצע:

- **AX** - Accumulator: הרגיסטר המתמטי. הוא משמש לרוב הפעילויות האריתמטיות והלוגיות, והוא בד"כ יעיל יותר בביצוען.
- **BX** - Base: בדרך כלל משמש לשמירת כתובות בזיכרון, מכיוון שהוא בין היחידים שיש להם גישה לזיכרון.
- **CX** - Counter: רגיסטר שהוא ייעודי לספירה, עבור לולאות, כמות תווים בקובץ או במחרוזת וכדומה.
- **DX** - Data: שומר מידע עבור גישות מיוחדות לזיכרון או עבור פעולות מיוחדות, ועבור חלק מהפעולות האריתמטיות הוא משמש כרגיסטר נוסף.
- **SI** ו-**DI** - Source Index ו Destination Index: משמשים לגישה לזיכרון עם BX, ולאחסון נוסף ברגיסטרים.
- **BP** - Base Pointer: משמש לגישה לזיכרון של הסגמנט של המחסנית (Stack Segment).
- **SP** - Stack Pointer: שומר את המיקום הנוכחי במחסנית. בד"כ לא נשנה את הערך שלו ידנית, אבל זה עשוי להימצא יעיל לעיתים.

כדי לאפשר גישה לקטעי זיכרון יותר קטנים מ-16 ביט, אפשר לפנות רק לחצי רגיסטר, כאשר עבור AX, BX, CX, DX אנו נהפוך את ה-X ל-L עבור החצי התחתון של הרגיסטר ו-H עבור החלק העליון, מלשון High ו-Low. עבור הרגיסטרים SI ו-DI ניתן להוסיף בסוף L או H, מבלי למחוק את האות I.

בנוסף לרגיסטרים הכלליים שהזכרנו קודם, ישנם גם רגיסטרים מיוחדים שנקראים רגיסטרי מקטע, או Segment Registers. ישנם ארבע רגיסטרים כאלה:

1. **DS**, או Data Segment, מצביע על המקום בזיכרון בו שמורים המשתנים שלנו.
2. **CS**, או Code Segment, מצביע על המקום בו שמור הקוד שלנו בזיכרון המחשב.
3. **SS**, או Stack Segment, מצביע על איזור המחסנית בזיכרון המחשב.
4. **ES**, או Extra Segment, מצביע על אזור נוסף בזיכרון, שבמידה ואנו חורגים ממגבלות הקוד של הסגמנטים (64KB כל אחד) אנו יכולים להשתמש בו.

הרגיסטרים האחרונים הם IP, או ה-Instruction Pointer, שמצביע על המיקום בזיכרון של השורה הבאה להרצה, ו-FLAGS, שחשוב מאוד עבור ביצוע תנאים לוגיים ולולאות.

הכנות לאסמבלי

אנחנו נשתמש ב-notepad++ החינמי בתור סביבת העבודה שלנו, ובשביל להתאים את זה לשפת האסמבלי, ניגשים לכפתור Language בתפריט העליון ותחת האות A בוחרים באפשרות Assembly.

בנוסף לכך, כדי להריץ את הקבצים אנו נשתמש באימולטור DOSBox, על השימוש בו נסביר בהמשך. תוספת אחרונה שלה אנו נזדקק היא TASM ו-TLINK. TASM (Turbo Assembler) הוא האסמבלר הבסיסי שבו נשתמש, והוא לא מורכב כמו רבים מהאחרים, כמו NASM, עליהם אולי ארחיב בהמשך. אנו משתמשים בהם (ובקבצים נוספים) כדי להפוך את קוד האסמבלי שלנו לקובץ מסוג EXE (Executable), שנוכל להריץ ב-DOSBox.

קישור להורדה מסודרת: https://drive.google.com/open?id=0B7fBWiSzrcHTam96NjB6RmtpdzQ

חלצו את הקבצים, התקינו את DOSBox ואת ++Notepad, והעבירו את התיקיה BIN לכונן C.

Base.asm

פתחו את הקובץ base.asm שנמצא בתיקייה BIN, בעזרת ++notepad. הוא הבסיס הקבוע יחסית לכל קבצי האסמבלי שנכתוב, ועכשיו נסביר כל שורה ממנו. שימו לב, כל הפקודות באמסבלי הן לא Case Sensitive, כלומר, אפשר לרשום גם באותיות גדולות וגם באותיות קטנות. עם זאת, נהוג לרשום באותיות גדולות את ההוראות למהדר (האסמבלר) ובאותיות קטנות את הקוד עצמו:

- **IDEAL** - זאת הוראה לאסמבלר שלנו, TASM, שמודיע לו שאנו עובדים במצב אידיאלי. אנו לא נשתמש במצבים אחרים במסגרת הזו, אבל תוכלו למצוא עוד הרבה אופציות באינטרנט.
- **MODEL SMALL** - אנו נשתמש רק במודל הזה, בו יש segment אחד של קוד, ו-segment אחד של מידע (בנוסף לאחד של המחסנית, בו ניגע בהמשך).
- **STACK 100h** - הצהרה על גודל המחסנית. כל מקום במחסנית הוא בית אחד, כלומר שמונה ביטים, שהם 2 תווים הקסדצימליים. אז בכתיבת הפקודה הזו, בעצם הצהרנו על 50 מקומות במחסנית. שימו לב שתמיד יש צורך להצהיר על גודל המחסנית, גם אם אתם לא מתכננים להשתמש בה.
- **DATASEG** - זה בעצם אזור המידע בקוד שלנו. כאן אנו מצהירים על המשתנים שאנו רוצים לשמור בזיכרון. תכף נדבר גם על איך עושים את זה.
- **CODESEG** - אזור הקוד בזיכרון שלנו. פה אנחנו רושמים את הקוד עצמו.

- **start:** - הצהרה על תחילת הקוד. השם לא באמת משנה, אבל נהוג לקרוא לו start. הוא חייב להתאים גם לתגית הסיום END, שמופיע בסוף הקוד בתור END start.
- **mov as, @data** - אנחנו בעצם מעבירים לרגיסטר ax את המיקום שבו מתחילה שמירת הנתונים שלנו. צריך לבצע את זה בתחילת כל קובץ אסמבלי. השימוש בסימן ה-@ נועד כדי לקרוא למילה השמורה data, ולא למשתנה בשם הזה אם היינו מחליטים לבנות אחד שכזה.
- **mov ds, ax** - מעבירים לרגיסטר שאחראי על המשתנים את המיקום שבו הם שמורים.
- **exit:** - מלבד התגית הראשונה, שלה קראנו start, האסמבלי משתמש בשיטה שקוראים לה Labeling, ובעצם מתבססת על כך שנקרא לכל קטע קוד תחת איזושהי תגית, כדי שנוכל לקפוץ ביניהם בהמשך. כרגע לא נעמיק במשמעות השורות הבאות בקוד, מלבד END start, אותה כבר הסברנו קודם.

בקובץ יש שיטה של אינדנטציה, או בעברית - הזחה. כל מה שנמצא תחת DATASEG נמצא TAB אחד פנימה יותר, וכן כל מה שנמצא תחת התגיות הפנימיות של שב-CODESEG. האסמבלי לא קורא את ההזחות בכל מקרה, אך הן מקלות על המשתמש את ההבנה של הקוד.

DOSBox

המקור של האימולטור DOSBox הוא בעצם במערכת ההפעלה DOS שהייתה פעילה גם לאחר יציאת Windows, בשל מדיניות ה-"תאימות לאחור" שהזכרנו קודם לכן.

האימולטור נדרש מכיוון שלמרות מערכת ההגנה של Windows, שהייתה מונעת מאיתנו לפגוע במחשב בעזרת אסמבלי בדרך כלל, הוחלט שלא לתמוך באסמבלי 16 ביט, מה שמשאיר אותנו עם מערכת DOS חביבה. כשפותחים את ה-DOSBox הוא מוביל אותנו לכונן Z, שהוא כונן וירטואלי, אך כדי להקל את השימוש אנחנו נשתמש באוסף הפקודות הבא:

- mount c: c:\ - בקשת מעבר לכונן C:, בו נמצאת התיקייה BIN.
- c: - מעבר לכונן C:.
- cycles = max - העלת מספר סיבובי המעבד למקסימום (אל תדאגו, זה רק המעבד המדומה, ומרבית המעבדים היום יעמדו בזה בקלות רבה).
- cd bin - מעבר לתיקייה bin.

זהו! עכשיו אנו נמצאים בתיקייה בה נשמור את קבצי העבודה. אם תרצו פירוט על פקודות נוספות או על אלו שהזכרנו, תוכלו למצוא את זה באינטרנט.

השלב הבא הוא חשוב לא פחות - ההפיכה של קובץ ה-asm שלנו לקובץ שהדוס יוכל להריץ. את זה אנו נעשה בשני שלבים:

1. tasm /zi base.asm - או כל שם קובץ אחר שנבחר. זה בעצם אסמבלר, והוא מתרגם את שפת האסמבלי שלנו לשפת מכונה, כלומר קובץ עם סיומת obj.
2. tlink /v base.obj - או כל שם קובץ אחר שנבחר. את הפעולה הזאת מבצע לינקר שיודע לאחד עבורנו כמה קבצים לתוכנית אחת, אך מכיוון ששלנו היא רק קובץ אחד בכל מקרה, השימוש הוא פשוט יחסית. עכשיו נוצר לנו קובץ exe.

עכשיו אנו מגיעים לצומת, מכיוון שיש שתי דרכים להריץ את הקובץ, האחת עם Turbo Debugger, שייתן לנו את האופציה לעקוב אחרי פעולת התוכנית, ואחת רגילה.

כדי להריץ תחת הדיבאגר, רושמים:

```
td base
```

וזה יפתח את הקובץ בצורה המתאימה. אחרת, רושמים רק את שם בקובץ, במקרה שלנו:

```
base
```

התהליך נשמע מתיש בשביל כל הרצה של קוד, נכון? לכן יש לנו את הקישור הבא:

https://drive.google.com/open?id=0B7fBWiSzrcHTbzh3elJDZXNjYzg

בקישור תמצאו RAR שמכיל שני קבצי batch, כלומר, בעלי סיומת bat. תעתיקו אותם אל התיקייה bin, ומעכשיו, כשתרצו להריץ קובץ בלי הדיבאגר, תוכלו לרשום run base, ובמידה ותרצו להשתמש בדיבאגר, תרשמו rund base, ולאחר בדיקת טעויות הקובץ ירוץ אם הוא לא נתקל בשגיאה.

IP ו-FLAGS

כבר הזכרנו את שניהם כשדיברנו על מבנה המחשב, אבל הפעם אנחנו ניכנס עמוק יותר לתוך השימוש של כל אחד מהם.

IP, או Instruction Pointer הוא המצביע על הפקודה הבאה בקוד. הפקודות מגיעות בגדלים שונים, בד"כ בין בית אחד לשלושה, וה-IP אמור לוודא שהמעבד יקלוט את הפקודות כראוי, ולא יחבר בין כמה ביטים מפקודה אחת וכמה מאחרת, ויקבל הבנה שגויה של הפקודה שלנו.

FLAGS - בשונה משאר הרגיסטרים, ב-FLAGS יותר קשה לעשות שימוש חופשי, ובפועל יש שם שימוש רק בחלק מהביטים לצורך תנאים וכד'.

הביטים השונים נותנים לנו מידע על מצב המעבד לאחר הפקודה האחרונה שהרצנו, ואנחנו נתמקד בארבעה מהדגלים, הנקראים דגלי בקרה:

1. **Zero Flag** - הערך של דגל הזה הופך ל-1 במידה ולאחר הרצת הפקודה האחרונה האופרנד התאפס. האופרנד הוא בעצם מקום בזיכרון, או רגיסטר, שבמקרה הזה הם אלו שקיבלו את תוצאת הפעולה. לדוגמה, אם נחבר 1 ו-1- נקבל 0, ואז הדגל יקבל את הערך 1. גם חיבור של המספר 255 עם המספר 1 יהפוך לאפס, מכיוון שהוא חוצה את גבול המספרים שניתן לייצג בשמונה ביטים (זה לא יקרה לרגיסטרים של 16 ביט לדוגמה).
2. **Carry Flag** - הדגל הזה מקבל את הערך 1 אם תוצאת הפעולה חורגת מתחום המספרים הרגילים (ללא התייחסות לשליליות). עבור 8 ביט החריגה היא מהתחום של 0-255, ועבור 16 התחום הוא 0-65535
3. **Overflow Flag** - הדגל הזה מקבל את הערך 1 אם תוצאת הפעולה חורגת מתחום המספרים המסומנים, שעבור 8 ביט החריגה היא מ-128- עד 127+ . עבור 16 ביט התחום הוא בין 32768- לבין 32767+.
4. **Sign Flag** - דגל הסימן. מקבל את הערך 1 כאשר הביט השמאלי בתוצאה הוא 1 (כמו שהזכרנו קודם, במקרה כזה המספר הוא שלילי לפי שיטת המשלים ל-2).

הגדרת משתנים

בתכנות נהוג להשתמש במשתנים כדי לשמור מספרים, והמקור לכך הוא באסמבלי, שבו משתמשים במשתנים כדי לחסוך את המבט לזיכרון בכל פעם שרוצים להשתמש בערך שמור. כמו שכבר הזכרנו קודם, את המשתנים אנו יוצרים ב-DATASEG של הקובץ שלנו באופן הבא:

Name size value - כלומר כדי ליצור משתנה בשם age עם בגודל של בית אחד, עם הערך ההתחלתי 16, אנו נרשום ב-DATASEG:

```
age db 16
```

הפירוש של DB הוא Define Byte, אך כדי ליצור משתנים בגדלים אחרים, צריך להחליף את הצירוף. ניתן ליצור משתנים בגודל מילה (WORD), שהיא שני בתים בעזרת הצירוף DW, וליצור משתנה בגודל מילה כפולה (DWORD) בעזרת הצירוף DD. כדי לפנות למשתנים בקוד עצמו, אנו עוטפים אותם בסוגריים מרובעים משני הצדדים.

לדוגמה, כדי להעביר למשתנה age את הערך 17, אנו נשתמש בפקודה mov, שמזיזה ערכים בין מיקומים, באופן הבא:

```
mov [age], 5
```

ולא:

```
mov 5, [age]
```

הסיבה לכך היא שלפקודות באסמבלי יש עד שני אופרנדים, ועבור הפקודה mov, בשונה מרוב הפקודות האחרות, האופרנד הראשון שאנו מכניסים, במקרה הזה mov הוא אופרנד היעד, שאליו מועתק הערך מהאופרנד השני, שנקרא גם אופרנד המקור.

כדי להגדיר מחרוזת, כלומר אוסף תווים אנו יכולים לרשום אותם החל מהתו הראשון באופן הבא:

```
string db 'HELLO'
```

במקרה כזה כל אחד מהתווים יישמר בבית נפרד, אחד אחרי השני, ברצף הנכון. באסמבלי יש גם הגדרה של מערכים, אך היא שונה במקצת:

```
ArrayName SizeOfElement N dup(?)
```

- ArrayName - שם המערך
- SizeOfElement - גודל כל תא (DB, DW...)
- N - מספר הפעמים לביצוע ההעתקה של הערך (או רצף הערכים) בסוגריים.
- dup - פקודת העתקה (duplicate).

גם באסמבלי יש אינדקסים במערך, שעוזרים לנו למצוא את המיקום של תא מסוים. האינדקס הראשון הוא 0. כדי לקבוע את כל הקפיצות בין אינדקסים, צריך לספור את גודל התא, מכיוון שהזיכרון בנוי מבתים, ולכן במקרה של מערך מסוג WORD לדוגמה, האינדקס של התא השני יהיה 2 ולא אחד, מכיוון ש WORD מבוטא בגודל של שני בתים.

לסיכום

למדנו את הבסיס לאסמבלי, לדוגמה שיטות ספירה (בינארית והקסדצימלית), מבנה המחשב, וגם יצירת משתנים ומערכים. בפרק הבא נעסוק בפקודות בסיסיות ונכתוב קודים בסיסיים באמסבלי.

על המחבר

שמי אופיר בק, בן 16 מפתח תקווה. אני לומד בתכנית גבהים של מטה הסייבר הצה"לי וב-C Security, לאחר שסיימתי את לימודי המתמטיקה והאנגלית בכיתה י'. קשה למצוא חומר מעודכן בעברית, ולאחר שהמגזין הזה היווה עבורי מקור מידע נגיש, רציתי לתרום חזרה. זה המאמר הראשון שלי במגזין ובכלל. ניתן גם ליצור איתי קשר בכתובת ophiri99@gmail.com.

קישורים לקריאה נוספת

- טורבו דיבאגר:

  https://en.wikipedia.org/wiki/Borland_Turbo_Debugger

- מבנה המחשב:

  http://www.rup.co.il/sites/default/files/%20%D7%9E%D7%91%D7%A0%D7%94%20%D7%94%D7%9E%D7%97%D7%A9%D7%91.docx