תקציר מספר 3

התיחסות לזכרון (כתובות).

בתור "התיחסות לזכרון" כאן הכוונה גישה לכתובת מסוימת בזכרון.
לשם הפשטות נתיחס כשלב זה רק למעבד 8086.

השימוש של תוכנית בזכרון אפשר לחלק ל-2 חלקים:

1. קריאה של פקודות מכונה.

2. קריאה / כתיבה של מידע (Data).

לגבי 1., קריאה של פקודות התוכנית מהזכרון, ראינו בקווים כלליים איך הדבר נעשה: הפקודה הבאה לביצוע נקבע ע"י זוג האוגרים CS והאוגר הבלתי נראה IP (מצוין CS:IP). זוג האוגרים הללו מתעדכן בעת ביצוע כל פקודה - אם לפקודה העוקבת בזכרון או (במקרה של פקודות בקרה) לכתובת אחרת לגמרי.

לגבי 2., שימוש בזכרון בתור Data, הדבר נעשה בעיקר ע"י פקודות 2 אופרנדים, כמו MOV. למשל הפקודה הבאה:

```
MOV  AX,DS:[BX]
```

היא פקודת קריאה מהזכרון של מילה (2 בתים) לתוך AX. הגודל של הזכרון הנקרא נקבע במקרה זה ע"י היעד (AX).
הכתובת שממנו קוראים את המילה בדוגמא הזו נקבעת ע"י הנקודה שמציעה עליה אוגר הסגמנט DS + הסט מנקודה זו הנקבע ע"י הערך של BX ברגע שהפקודה הזו מתבצעת בזמן ריצה. ביצועים שונים של אותה פקודה עשויים להתיחס לנקודות שונות בזכרון. נהוג לציין זאת DS:BX. כפי שצוין קודם, כל התיחסות לזכרון הוא זוג: תוכן אוגר סגמנט + היסט.

ההיסט ב-8086 הוא תמיד מספר 16 ביט. משמעות הדבר הוא שהמרחק המירבי של נקודת הזכרון מהכתובת שמתיחסת אליו מספר הסגמנט 65535 = 1 - 16**2, או 1 - 64*1024 או 64k. המושג 64k, המשמש מגבלה על כמויות זכרון בתוכניות רבות אפילו עד היום, נובע מהעובדה הזו.

הנושא שנבחן עכשיו הוא:

כיצד מפורש צמד: תוכן אוגר סגמנט + היסט לכתובת פיזית (אבסולוטית) בזכרון.

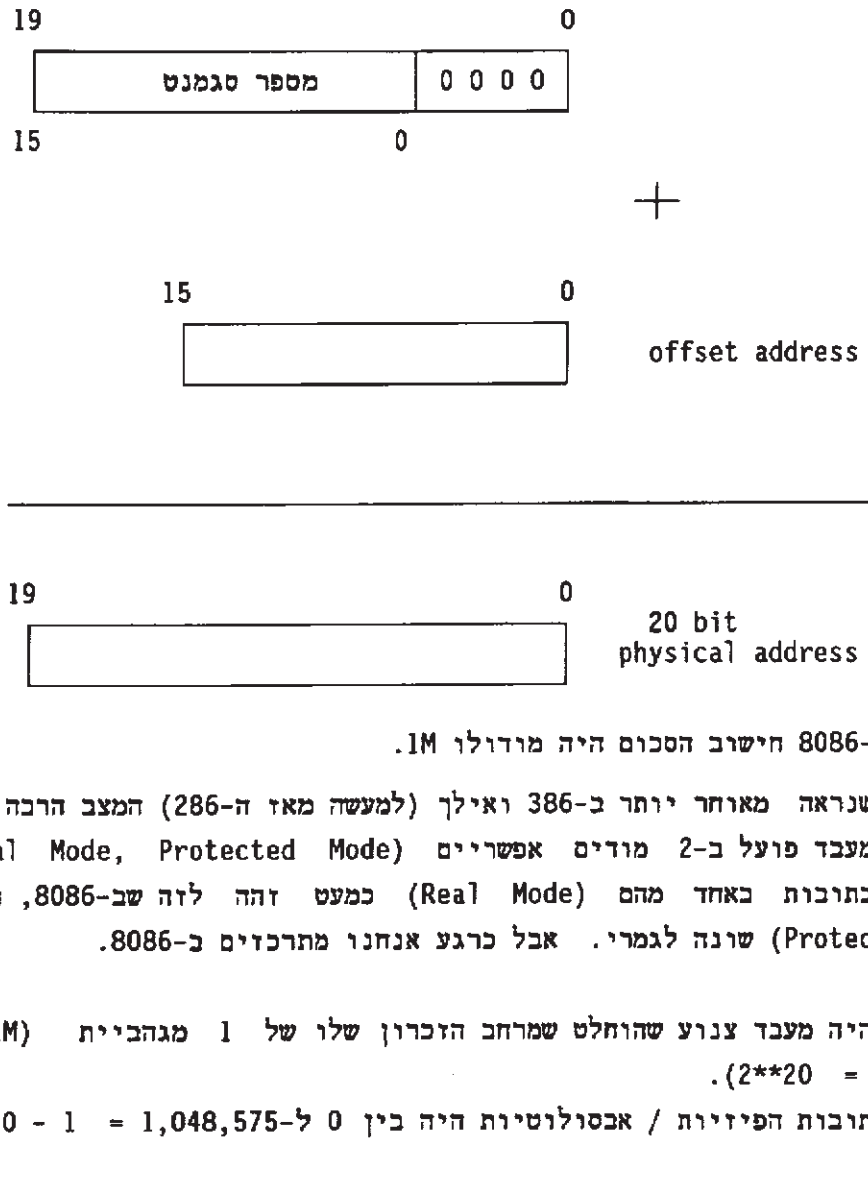
ב-8086 התשובה לשאלה הזו היא פשוטה:

תוכן אוגר הסגמנט (16 ביט) מוכפל ה-16 ולתוצאה מתווספת ההיסט (שהוא גם, תמיד 16 ביט).

לדוגמא, אם מספר הסגמנט הוא 3 וההיסט הוא 273 אזי הכתובת הפיזית היא 3*16+273 = 273 + 48 = 321. במילים אחרות:

ב-8086 החליטו שכל האוגרים יהיו 16 ביט והתיחסות לזכרון יהיה מורכב משני מספרים 16 ביט - מספר סגמנט והיסט - שיורכבו למספר 20 ביט ככתובת אבסולוטית. החישוב הוא כאמור הכפלת מספר הסגמנט ב-16 והוספת ההיסט. מאחר והכפלה ב-16 (שהוא 2**4) הוא כמו הזזת מספר ב-4 ביטים שמאלה (וריפוד באפסים) המשמעות של החישוב הזה הוא למעשה:

<u>יצירת כתובת פיזית של 20 ביטים:</u>



ב-8086 חישוב הסכום היה מודולו 1M.

כפי שנראה מאוחר יותר ב-386 ואילך (למעשה מאז ה-286) המצב הרבה יותר מורכב: המעבד פועל ב-2 מודים אפשריים (Real Mode, Protected Mode), שחישוב הכתובות באחד מהם (Real Mode) כמעט זהה לזה שב-8086, והשני (Protected Mode) שונה לגמרי. אבל כרגע אנחנו מתרכזים ב-8086.

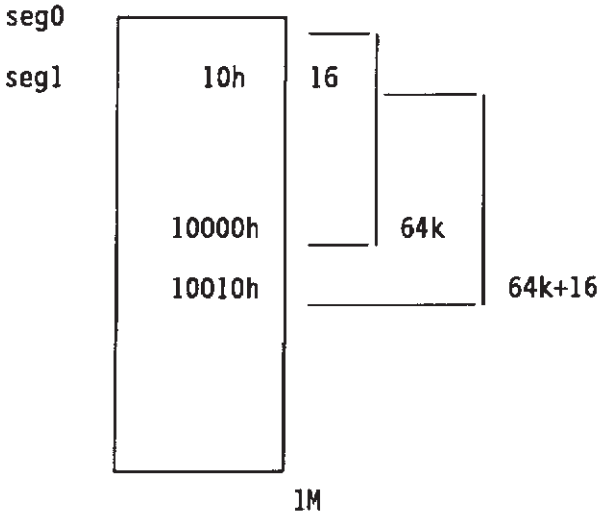
ה-8086 היה מעבד צנוע שהוחלט שמרחב הזכרון שלו של 1 מגהבייט (1M או 2**20 = 1,048,576).

מרחב הכתובות הפיזיות / אבסולוטיות היה בין 0 ל-1,048,575 = 2**20 - 1.

מהם 640K הוקצו לתוכנה (Conventional Memory) ו-384k נשמרו למערכת (Reserved). עוד נדבר על כך.

כלומר שכתובת אבסולוטית ב-8086 היה מספר בן 20 ביט.

הגישה של חישוב כתובות מזוג מספרים 16 ביט ע"י הכפלה ב-16 של מספר הסגמנט והוספת ההיסט יצרה חלוקה של הזכרון ל-64k יחידות זכרון (לא זרות) בגודל 64k כל אחת, המתחילות בכל אחת מהכתובות בזכרון המתחלקות ב-16. יחידות הזכרון האלו אינן זרות כאמור: לכל סגמנט יש 4k סגמנטים שונים ממנו שחופפים לו חלקית, כאשר גודל השטח החופף הוא בין 16 בתים ל- 64k-16 בתים. היחידות האלו נקראות <u>סגמנטים</u>.



<u>חישוב כתובות במעבדי ה-386 ואילך.</u>

מאז ה-286 מעבדי האינטל יכולים לעבוד בשני מודים:

- Real Mode
- Virtual Addressing Protected Mode (או בקיצור Protected Mode).

בכדי לעבוד ב-Protected Mode דרושות פעולות הכנה של התוכנה (בדרך כלל מערכת ההפעלה). לכן כאשר מדליקים מחשב כלשהוא הוא תמיד עובד ב-Real Mode.
עד DOS 3.3 כולל, מחשבי האינטל עבדו ב-Real Mode. החל מ-DOS 4.0 עד DOS 6.23 רוב המחשבים 386 ואילך עובדו תוך שימוש ב-EMM386.EXE. תוכנה זו אומנם מעבירה את המחשב ל-Protected Mode, אך דואגת שההתנהגות של המעבד דומה מאד להתנהגות המעבד ב-Real Mode, אם אפשרות למספר שיפורים (שעיקרם ניצול יעיל של זכרון). מערכות הפעלה מתקדמות כמו Windows ו-Linux עובדות

ב-Protected Mode מלא (כלומר לא המוד המצומצם תחת ה-EMM386.EXE).

שני המודים האילה שונים גם (ואולי בראש ובראשונה) בשיטת חישוב הכתובות.

ב-Real Mode המעבדים 386 ואילך מחשבים כתובות באופן כמעט זהה ל-8086. יש שני הבדלים שידועים לי: ההיסט יכול להיות 32 ביט - עד 4 גיגה (4096M), וחישוב הכתובות אינו מודולו 1M (יש טיפול שונה במקצת לגבי חריגות). אגב, השימוש בהיסטים גדולים מ-64k ב-Real Mode נדיר מאד מסיבות שונות, שאולי נעמוד עליהן.

כאשר מעבד 286 או מתקדם יותר עובד ב-Protected Mode, שיטת המיעון שונה לגמרי. מספר הסגמנט עובר <u>המרה</u> למספר אחר, כתובת פיזית, לפי ערכים הנקבעים על ידי <u>תוכנה</u> (בדרך כלל מערכת ההפעלה) שנשמרים בטבלאות בזכרון. ב-286 תוצאת ההמרה הזו היתה 24 ביט, והחל מה-386, 32 ביט. מסיבה זו מערכות הפעלה כמו Windows ו-Linux יכול לנצל עד 4 גייגה במחשבי 386 (24 מגה ב-286). מעבר לכך ההיסט מתווסף ההיסט כמו קודם. השימוש בהיסטים 32 ביט נפוץ בעיקר במערכות הפעלה 32 ביט, כמו Windows 95, NT ו-Linux.

התיחסות לזכרון כמידע (Data).

כפי שנאמר קודם, התיחסות לזכרון בתור פקודות מכונה של תוכנית נעשה ע"י CS:IP (CS:EIP ב-386 ואילך). עכשיו נראה איך מתיחסים לזכרון בתור מידע (data).

שוב נחזור איך הדברים היו ב-8086.

התיחסות לזכרון נעשה בעיקר ע"י פקודות 2 אופרנדים:

שם הפקודה אופרנד היעד, אופרנד המקור

דוגמאות לכך יכולים להיות MOV, ADD, SUB, AND, OR וכו'.

בפקודת 2 אופרנדים, אופרנד המקור יכול להיות אוגר, התיחסות לזכרון או קבוע. אופרנד היעד יכול להיות אוגר או התיחסות לזכרון. רק צירוף אפשרי אחד אינו נתמך: אי אפשר שבפקודה גם אופרנד המקור וגם אופרד היעד יהיו שניהם התיחסויות לזכרון. אבל כל צירוף אחר הוא אפשרי. לפיכך הצירופים האפשריים הם:

אוגר, אוגר,
אוגר, זכרון,
זכרון, אוגר,
קבוע, אוגר,
קבוע, זכרון.

אגב, המקום שבו אופרנד קבוע מאוכסן בזכרון הוא ביחד עם הפקודה. קוד הפקודה מציין שהאופרנד הוא קבוע, הנמצא בכתובת העוקבת שלאחר הקוד בזכרון. הערך של הקבוע נשלף ביחד עם קוד הפקודה - הערך של האופרנד הוא למעשה חלק מהפקודה - והגודל של הפקודה הוא בהתאם.

למשל, הפקודות הבאות הם חוקיות:

(1)	קריאה מהזכרון	MOV AX,[BX+7] <=====
(2)	כתיבה לזכרון	MOV [SI-2],BL <=====
(3)	כתיבה של קבוע לתוך הזכרון	MOV WORD PTR [BP],8 <=====
(4)	AX = AX + תוכן זכרון	ADD AX,[BX+7] <=====
(5)	תוכן זכרון = תוכן זכרון + BX	ADD [SI+2],BX <=====

ADD BYTE PTR [BP],8 <===== תוכן זכרון = תוכן זכרון + 8 (6)

שימו לב שאין ציון של אוגר סגמנט כאן, במקרים שבהם אין ציון של אוגר סגמנט יש כללים של ברירות מחדל. כפי שנראה בהמשך, בשני הפקודות 1 ו-2 יבחר אוגר הסגמנט DS, ואילו בפקודה 3 אוגר הסגמנט הוא SS. בצורה דומה, DS הוא אוגר הסגמנט עבור הפקודות 4, 5 ו-SS הוא אוגר הסגמנט עבור פקודה 6.

מה שמצוין בין הסוגריים המרובעות הוא <u>ההיסט</u> (Offset) מהנקודה שעליה מצביע אוגר הסגמנט.

ב-8086 ההיסטים היו תמיד 16 ביט. באופן כללי ביותר ההיסט היה מהצורה:

BX / BP או + SI / DI או + קבוע

כאשר הקבוע נקרא הזזה (Displacement).

במילים אחרות: בכדי לבנות היסט להתייחסות לכתובת, אפשר היה לבחור 2 אוגרים ולהוסיף להם קבוע. הקבוע יכול להיות שלילי. שני האוגרים היו יכולים בחירה מתוך BX או BP או אף אחד משניהם, ובחירה מתוך SI או DI או אף אחד משניהם.

לפיכך הקומבינציות האפשריות של אוגרים הם

{BX}, {BP}, {SI}, {DI}, {BX,SI}, {BX,DI}, {BP,SI}, {BP,DI}

או אף אוגר. לפיכך אפשר לראות פקודות למשל

```
MOV  AX,[BX+DI+7]
MOV  WORD PTR [BP+SI-1],7
MOV  [9],CX
MOV  WORD PTR [SI],9
```

בין הסוגריים חייב להיות משהו, למשל, אם לא בוחרים אוגר כלשהוא חייבים לבחור בקבוע.

<u>שימוש ב-label-ים.</u>

קבוע ההזזה (ה-Displacement) לא חייב להיות קבוע מספרי, הוא יכול להיות גם label של התוכנית, בדרך כלל label ל-מידע (Data). כאן חשוב להבין מהוא label. label, הנחיה של התוכניתן לאסמבלר בצורה של "שם:" (למשל Aname:) הוא למעשה נתינת שם סימלי לנקודה בתוכנית, או יותר נכון

לנקודה כזכרון. ה"ערך" של השם הזה, הוא רק ההיסט (Offset) בסגמנט שבו הוא מוגדר. כיוון שההיסט אינו תלוי באיזה מספר סגמנט מדובר, ניתן למעשה להתיחס לערך שלו כקבוע. לפיכך הפקודות הבאות היא חוקיות:

```
.DATA
Var1 DW 5,10,15

.....
.CODE

....

MOV AX,[Var1+2]
MOV BX,4
MOV CX,[Var1+BX]
```

MOV AX,[Var1+2] תכתוב לתוך AX את הערך 10, ואילו MOV CX,[Var1+BX] תכתוב לתוך CX את הערך 15. יש כמה צורות לכתוב את השימוש הזה. למשל הפקודות:

```
MOV [Var1+BX+3],AX
MOV Var1[BX+3],AX
MOV Var1[BX][3],AX
```

הן כולם שקולות.

למרות ש-label הוא קבוע, האסמבלר מגביל את השימוש בו. אי אפשר "לסכם" או "להכפיל" label-ים. באופן עקרוני אפשר להשתמש רק ב-label אחד בפקודה, החריג היחיד הוא שניתן להחסיר label-ים, ויש בזה הגיון, זה מאפשר לחשב את כמות הזכרון שיש בין נקודת זכרון אחת לנקודת זכרון שניה.

היתרון בשימוש ב-label-ים הוא שהאסמבלר מחשב עבורינו את ערכם (יחסית מתחילת הסגמנט בו הם מוגדרים) ופוטר אותנו מלעשות זאת. אם התוכניתן עושה שינוי בתוכנית, ערכם של ה-label-ים מתעדכנים אוטומטית בקומפלציה הבאה.

בחירת אוגר סגמנט

כל התיחסות לזכרון הוא יחסית לאוגר סגמנט. חשוב לזכור זאת גם כאשר מרבית פקודות האסמבלר לא מציין אוגר סגמנט. פקודות מהסוג

```
MOV  AX,[BX+DI+7]
MOV  BYTE PTR [BP+SI+1],7
```

כאילו לא מתיחסים לעינין אוגר הסגמנט.

הכללים של בחירת אוגר סגמנט הם כלהלן:

במידה שבו אוגר הסגמנט מצוין במפורש (Segment Override) אז זה קובע את אוגר הסגמנט. אם ההתיחסות לזכרון היא, למשל, "ES:[BX]" אז ברור שמדובר באוגר הסגמנט ES. פקודות מהסוג הזה הם למשל

```
MOV AX,CS:[SI+7]
ADD DS:[BP-9],DX
```

- במידה ואוגר הסגמנט מצוין בפקודה, זהו אוגר הסגמנט הנבחר, בלי תלות במה שכתוב בתוך הסוגריים המרובעים.

במידה ואין ציון אוגר סגמנט, יש ברירות מחדל. מבדיקות שערכתי הכללים של האסמבלר הם כלהלן:

- אם האוגר BP מופיע בהתיחסות לזכרון, אוגר הסגמנט הנבחר הוא SS. אם ההתיחסות לזכרון היא כמו בפקודה:

```
MOV AX,[BP+SI]
```

אז ההתיחסות היא יחסית ל-SS. רק Segment Override יכול להפר את הכלל הזה. האוגר BP נועד להצבעה יחסית ל-SS. להשתמש בו למטרה אחרת כמעט תמיד חסר טעם.

- במידה ואין Segment Override ולא מדובר באוגר BP אז השאלה הבאה האם יש label. במידה ויש, הוא קובע את אוגר הסגמנט. אם משתמשים בהנחיות סגמנטים מקוצרים, כפי שאנחנו עובדים לפי שעה, אז label המוגדר תחת DATA. קובע את DS כאוגר הסגמנט, label תחת CODE. קובע את CS כאוגר הסגמנט. כאשר נלמד על תוכניות המשתמשות בהנחיות סגמנטים סטנדרטיות, ונלמד על ההנחיה ASSUME, נראה איך הכלל הזה ממומש, וכן אפשריות נוספות לכללים הללו.

בכל מקרה שנותר אוגר הסגמנט הוא DS. כלומר אם אין Segment Override, אין BP, אין label, אז מה שנותר היא התיחסות שכוללת את BX או שלא, אחד מהאוגרים SI או DI או שלא, או רק קבוע. למשל:

```
MOV CX,[BX+SI-4]
MOV BYTE PTR [SI+6],8
```

בכל המצבים מהסוג הזה אוגר הסגמנט הוא DS.

מעבדים 386 ואילך

לגבי שימוש באוגרי סגמנט, ההבדל העיקרי בין ה-386 ו-8086 (מעבר לנושא של Protected Mode שלא נתיחס כאן) הוא שיש שני אוגרי סגמנט נוספים: GS ו-FS. האוגרים הללו דומים, באופן עקרוני, לאוגר ES. רק של-ES יש שימוש אוטומטי בפקודות מחרוזת (שנראה יותר מאוחר בקורס) דבר שלא קיים עבור

.FS-ו GS

בכל הקשור להיסטים, ל-386 יש אפשרות שימוש בשני סוגי היסטים:
- היסט 16 ביט.
- היסט 32 ביט.

עבור היסטים 16 ביט, הכללים ב-386 זהים ל-8086 (BX או SI, BP או DI, קבוע וכו').
עבור היסטים 32 ביט, ה-386 תומך בכל היסט מהסוג:

[קבוע32 + r32 + n*r32p]

כאשר קבוע32 הוא קבוע 32 ביט,
r32 אוגר 32 ביט כלשהוא מלבד EIP (כלומר ... EAX, EBX, ECX),
r32p הוא אוגר 32 ביט כלשהוא שונה מ-r32 אבל לא ESP,
ו-n הוא אחד המספרים 1, 2, 4, 8 או לא מציון (1 כברירת מחדל).

במילים אחרות: כעוד שבהיסטים 16 אין כל שינוי, בהיסטים 32 ביט יש גמישות רבה הרבה יותר בבחירת האוגרים, וכן אופציה של הכפלת ערך אחד מהם בחזקה של שניים עד 8. האופציה האחרונה מקלה על מימוש מערכים של מספרי שלמים 16, 32 ואף 64 ביט. לפיכך הפקודות הבאות הן חוקיות:

```
MOV AX,[EBX + 2*EAX + 7]
ADD DWORD PTR [ECX],9
MOV [4*EDX],EAX
MOV BL,[EBX + 8*EBP]
```

כללי בחירת הסגמנט דומים לאילה של ה-16 ביט:

האם יש Segment Override,
האם EBP נמצא (ולא מוכפל בקבוע),
האם יש label, וכו'.

<u>חשוב לדעת</u>

כאשר משתמשים בהיסטים 32 ביט ([EAX] וכו') יש גרסאות של tlink (ביחוד החדשות יותר) הדורשות שימוש באופציה 3/. למשל שורת הפקודה תהיה

```
tlink /3 myprog.obj
```

למרות שאתם יכולים בתוכנית DOS להשתמש בהיסט 32 ביט, הערך של ההיסט בדרך כלל לא יכול לעלות על 655535 (או 64k - 1). הסיבה לכך ש-DOS בדרך כלל עובד תחת EMM386.EXE (או אמולציה שלו), ו- EMM386.EXE חוסם היסטים בעלי ערך גבוה יותר. כלומר: אתם יכולים להשתמש באוגרים ה-32 ביט בכדי לבנות היסטים 32 ביט, אך ערך ההיסט עצמו יכול להיות לכל היותר 65535. במידה וההיסט גדול יותר, ואתם רצים ב-DOS אורגינל (עד 6.23) תחת EMM386.EXE המחשב שלכם יתקע. הדבר האחרון אפשרי רק אם אתם רצים תחת DOS שבו EMM386.EXE לא מופיע ב-CONFIG.SYS.

תוכנית דוגמא memory4.asm

תוכנית זו נוערה למחיש את האפשריות של חישוב כתובות. מה שהיא עושה בפועל זה להעתיק תוים מהמחרוזות 'ABCDEFGHI' ו-'0123456789' למחרוזת יעד (מאותחלת כרצף של התו '$') המודפסת בסוף. כל תו מועתק תוך שימוש בגירסה שונה של חישוב כתובת.

נקודות שיש לשים לב עליהן:

- ההנחיה DUP אומר לאסמבלר לשכפל ערך באתחול מערך כך וכך פעמים, השימוש בהנחיה בפקודה

```
DisplayString DB  20 DUP ('$')
```

פירושו ליצור מערך תוים בשם DisplyString בגודל 20 תוים המאותחלים כולם בקוד אסקיי של תו ה-'$' (גם הקצעת שטח וגם אתחול). במקרה של DisplayString הכוונה היא שסימן הדולר הראשון שלא ימחק יעצור את הרוטינה INT 21h, AH = 9.

- מחרוזת הספרות Digits נמצאת תחת ההנחיה CODE. כלומר היא ממומשת בפועל בסגמנט הביצועי של התוכנית, זו המוצבעת ע"י האוגר הסגמנט CS. אפשר לעשות זאת, אם כי צריך לדאוג לכך שה-CPU לא ינסה לפרש את תוכן המחרוזת כתאור של פקודות מכונה. הדבר נעשה בתוכנית הזו ע"י כך שהפקודה הראשונה לביצוע של התוכנית (מוצבעת ע"י הליבל ProgStart) נמצאת מעבר לשטח המחרוזת. השימוש בשטח הזה לזכרון אינו שונה בהרבה מהמחרוזת האותיות Letters הנמצא בסגמנט המוצבע ע"י DS. אם משתמשי בשם המחרוזת Digits כמו בפקודה

```
MOV DL,[Digits+BX]
```

האסמבלר יודע לבד שההתיחסות לזכרון הוא יחסית ל-CS. יחד עם זאת בקטע הקוד

```
MOV BX, OFFSET Digits+3
MOV DL,CS:[BX+1]
```

כאן חייבים לציין במפורש את אוגר הסגמנט ("CS:") משום שהאסמבלר בוחר אוגרי סגמנט רק מתוך פענוח שורה הנוכחית לפי הכללים שתוארו. לפיכך, למרות שלעין האנושית אולי ברור שצריך לגשת לזכרון יחסית ל-CS, במקרה הזה הדבר יעשה רק אם CS יצוין במפורש. אחרת לפי הכללים יבחר

.DS

- הפקודה LEA היא פקודה המקבלת אופרנד מקור שהיא תאור היסט של כתובת בזכרון, וההסט הזה מוצב לאפרנד היעד. למעשה מדובר בתהליך דומה לאופרטור OFFSET אלא ש-LEA היא פקודה כללית הרבה יותר מ-OFFSET שכן OFFSET עובד רק על כתובות קבועות (ללא אוגרים), כלומר היסטים הניתנים לחישוב בזמן האסמבלי. לעומת זאת LEA מחשב את ההיסט בזמן ריצה לכן הוא יכול להכיל אוגרים. למשל הפקודות

```
LEA SI,Letters

MOV SI,OFFSET Letters
```

שקולות מבחינת התוצאה. אך אם אנחנו רוצים לחשב כתובת נוסח

```
LEA SI,Digits[BX]
```

חישוב כתובת כזו ניתנת רק ע"י LEA. הסיבה לכך היא שחישוב כתובות ע"י האופרטור OFFSET ממומש כביצוע MOV של קבוע לאוגר, קבוע שניתן לחישוב רק בזמן אסמבלי. אם תכתוב בתוכנית אסמבלי את הפקודה

```
MOV SI,OFFSET [BX+5]
```

זה עשוי לעבור אסמבלי אבל התוצאה תהיה אקראית וחסרת משמעות (האסמבלר ישתמש בערך של BX בזמן אסמבלי, שמן הסתם יהיה חסר משמעות לתוכניתן ובודאי לא קשור לערך של BX בזמן ריצה).

- בשביל שהאסמבלי יקיר אוגרים ופקודות של ה-386 יש לציין את ההנחייה 386. בגוף התוכנית. האסמבלר יכיר את האוגרים והפקודות שלח ה-386 מאותה נקודה בתוכנית ואילך (ברירת מחדל הם פקודות 8086 בלבד). אפשר לחזור למציאות הקודמת ע"י ציון 86. או 8086. הנחיות נוספות שידועות לי הם:

.8086 .86	פקודות 8086
.87	הכר פקודות מעבד מתמטי
.186	הכר פקודות 186
.286 .286C	הכר פקודות 286
.286P	הכר פקודות 286 מיוחסות
.287	הכר פקודות מעבד מתמטי 287
.386P	הכר פקודות 386 מיוחסות
.386 .386C	הכר פקודות 386
.387	הכר פקודות מעבד מתמטי 387
.486	הכר פקודות 486
.486P	הכר פקודות 486 מיוחסות
.586	הכר פקודות 586
.586P	הכר פקודות 586 מיוחסות
.686	הכר פקודות 686
.686P	הכר פקודות 686 מיוחסות

וכו'. לא כל אסמבלר מכיר את האחרונים.

- אם תוכנית אסמבלי משתמשת בהיסטים 32 ביט (אוגרים EAX, EBX ... בתור פוינטרים) יש צורך ב-tlink לציין את המאפין "/3". לפיכך פקודת ה-tlink של התוכנית הזו תהיה:

```
tlink /3 memory4.obj
```

```
;
; memory4.asm - Demonstrate addressing modes
;
     .MODEL SMALL
     .STACK 100h
     .DATA
Letters  DB 'ABCDEFGHI'
DisplayString DB  20 DUP('$')
                              ;
     .CODE
Digits  DB '0123456789'
ProgStart:
     MOV AX,@DATA
     MOV DS,AX                ; Set DS to point to data segment
                              ;
     MOV DI,OFFSET DisplayString  ; Have DI point to start of ...
                                  ; ... DisplayString
     MOV AL,[Letters+2]     ; Read 'C',
     MOV [DI],AL              ; Store in DisplayString
     INC DI                   ; point to next available position
                              ;
     MOV SI,1
     MOV BX,2
     MOV DL,[Letters+BX+SI+3]  ;  Read 'G', (Letters+6)
     MOV [DI],DL              ; Store in DisplayString
     INC DI                   ; point to next available position
                              ;
     MOV SI,2
     MOV BX,1
     MOV DL,Letters[BX][SI]+3   ;  Read 'G',  (Letters+6)
     MOV [DI],DL              ; Store in DisplayString
     INC DI                   ; point to next available position
                              ;
     LEA SI,Letters         ; Equivalent to "MOV SI,OFFSET Letters"
     MOV BX,2
     MOV DL,[BX+SI+1]           ;  Read 'D',  (Letters+3)
     MOV [DI],DL              ; Store in DisplayString
     INC DI                   ; point to next available position
                              ;
     MOV BX,2
     MOV DL,[Digits+BX]           ; READ '2'
     MOV [DI],DL              ; Store in DisplayString
     INC DI                   ; point to next available position
                              ;
     MOV BX,OFFSET Digits+3
     MOV DL,CS:[BX+1]         ; READ '4'
     MOV [DI],DL              ; Store in DisplayString
     INC DI                   ; point to next available position
                              ;
     MOV BX,6                 ; The following CANNOT be performed by
                              ;   MOV ..., OFFSET ...
     LEA SI,Digits[BX]         ; SI := OFFSET Digits+6
     MOV AX,CS                ; ES can be written only through a register
     MOV ES,AX                ; Set ES := CS
     MOV DL,ES:[SI-1]         ; READ '5' (displacement can be negative)
     MOV [DI],DL              ; Store in DisplayString
     INC DI                   ; point to next available position
```

```
;
.386                        ; Enable 386 instructions
MOV  ESI,4                  ;
MOV  EDX,0                  ;
LEA  DX,Letters       ;
MOV  DL,[EDX+ESI+3]         ; READ 'H' (Any choice of 2 regs for 386)
MOV  [DI],DL                ; Store in DisplayString
INC  DI                     ; point to next available position
                            ;
MOV  EAX,1                  ;
MOV  EBX,2                  ;
MOV  DL,Digits[EBX+4*EAX+3]   ; READ '9' (Any choice of 2 regs for 386)
MOV  [DI],DL                  ; Store in DisplayString
INC  DI                       ; point to next available position
                           ;
                           ;
MOV AH,9                   ; Set print option for int 21h
MOV DX,OFFSET DisplayString  ;  Set  DS:DX to point to DisplayString
INT 21h                    ;  Print DisplayString
                           ;
MOV AH,4Ch                  ; Set terminate option for int 21h
INT 21h                     ; Return to DOS (terminate program)
                            ;
END  ProgStart      ; Set "ProgStart:" as first executable statement
```

```
E:\>tasm memory4.asm
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:    memory4.asm
Error messages:     None
Warning messages:   None
Passes:             1
Remaining memory:   444k


E:\>tlink /3 memory4.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

E:\USERS\EYTAN\ASM\NEWDOCS3>
E:\>memory4.exe
CGGD245H9

E:\>
```