**סוג המשתנה יכול להיות:**

* DB - בית (8 ביט)
* DW - מילה (16 ביט)
* DD - מילה כפולה (32 ביט)
* DQ - מילה מרובעת (64 ביט)
* DT - עשרה בתים (80 ביט)

**ערך המשתנה חייב להיות מתאים לגודל שהוקצה לו:**

* DB - מ 0 עד 255 (28) או מ 127- עד 128
* DW - מ 0 עד 65535 (216) או מ 32767- עד 32768
* DD - מ 0 עד 4294967295 (232) או מ 2147483647- עד 2147483648
* DQ - מ 0 עד 18446744073709551615 (264) או מ 9223372036854775807- עד 9223372036854775808
* DT - מ 0 עד 1208925819614629174706175 (280) או מ 604462909807314587353087- עד 604462909807314587353088

**פעולות חיבור**

ADD operand1, operand2

ADC operand1, operand2

Inc operand1

**ADD**

ADD ax, 100

בפקודה זו אנחנו יכולים לחבר רק מספרים שהם בית אחד או בגודל 2 בתים

**ADC**

ADC operand1, operand2

בפקודה זו נשתמש כאשר נצטרך לחבר מספרים בגודל יותר מ2 בתים ופקודה זו בעצם מחשבת גם כולל דגל - CF (carry flag – לכן גם קוראים לפקודה ADC כי זה כולל CF).

**INC**

INC operand1 🡪 operand1++

מגדיל את האופרנד ב1.

**פעולות חיסור**

SUB operand1, operand2

SBB operand1, operand2

DEC operand1

**SUB**

SUB ax, 100

בפקודה זו אנחנו יכולים לחסר רק מספרים שהם בית אחד או בגודל 2 בתים

**SBB**

SBB operand1, operand2

בפקודה זו נשתמש כאשר נצטרך לחסר מספרים בגודל יותר מ2 בתים ופקודה זו בעצם יודעת להלוות אם צריך מהמספר לפניו ואם הלווינו אז דגל - CF נדלק (carry flag \ borrow flag – לכן גם קוראים לפקודה SBB כי זה כולל BF\CF).

**DEC**

DEC operand1 🡪 operand1--

מקטין את האופרנד ב1.

**פעולת כפל**

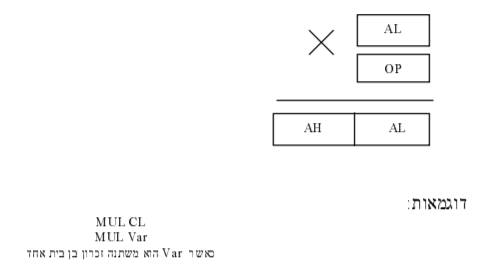
**MUL** operand1

**פקודה זו טובה לשימוש במספרים לא מסומנים בלבד(כלומר שאינם שלילים). \***

**\*חובה על האופרנד להיות מותאם בגודלו לגודל של האוגר שבו מכפילים.**

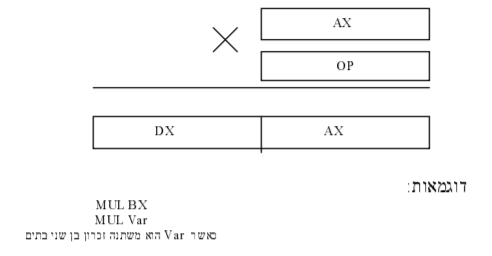
**אם operand1 הוא בן 8 ביטים**

AL מוכפל בתוכן של הoperand והתוצאה נזרקת ל-AX.



**אם operand1 הוא בן 16 ביטים**

AX מוכפל בתוכן של operand1 והתוצאה נזרקת לאוגרים DX,AX באופן הבא:

16 הסיביות המשמעותיות יותר של התוצאה יכנסו ל – DX ו-16 הסיביות הפחות משמעותיות של התוצאה יכנסו ל-AX.

**אופרנדים מותרים:**

אוגר או משתנה.

**דוגמא לתכנית עם הפקודה MUL**

.MODEL SMALL

.STACK 100H

.DATA

num DW 345

hundred DW 100 🡪

result DD ?

.code

MOV AX,@DATA

MOV DS,AX

MOV AX, num

MUL hundred

MOV WORD PTR result,AX

MOV WORD PTR result+2,DX

**הסבר התכנית:**

**.DATA – חלק הגדרת המשתנים**

**מדוע המשתנים num ו- hundred מוגדרים בתור DW?**

Num מוגדר כ- DW כיוון שהוא גדול מ-255 ובנוסף לכך הוא חייב לעבור אחר כך ל-AX ועל מנת שנוכל להעבירו הוא גם צריך להיות בגודל 16 בתים.(הוא צריך לעבור ל- AX כיוון שהפקודה MUL מכפילה את מה שיש ב-AX באופרנד המוצג).

Hundred צריך להיות מוגדר כ-DW כיוון שתוצאת ההכפלה של המספר שלנו במאה כמעט בטוח תעבור את הBYTE- ותצטרך להיות מיוצגת ב2 בתים. ולא תוכן להיות משוכנת ב-AX כמו הכפלה באופרנד של BYTE אחד.

**.CODE – חלק הקוד**

נעביר ל-AX את num(חייב ל-AX הסברנו בפירוט הפקודה מדוע).

נכפיל את מה שיש ב-AX ב-100 והתוצאה תיזרק אל האוגרים DX:AX כמפורט לעיל.

נפנה ל-WORD הראשון של result כלומר ל-2 הבתים הראשונים ונשים שם את החצי התחתון של הלולאה

כלומר את AX(כי התוצאה נזרקה ל-DX:AX).

לאחר מכן נפנה ל-WORD השני של result כלומר ל-2 הבתים האחרונים ונשים שם את החצי העליון של התוצאה שלנו שהיא משוכנת ב-DX.

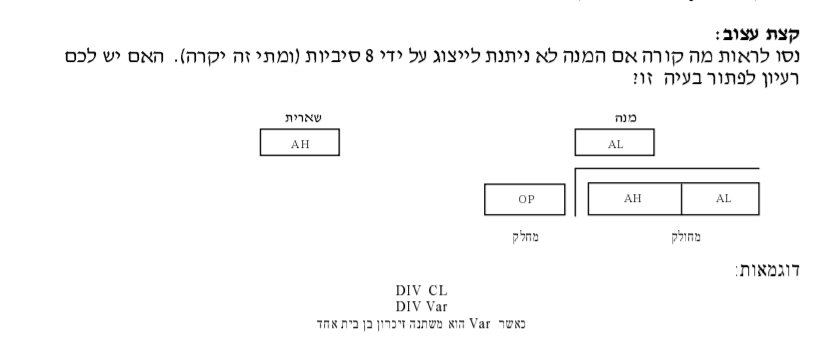
לבסוף נקבל את התוצאה שלנו בשלמות במשתנה result.

**פעולת חילוק**

**DIV** operand1

**אם operand1 הוא בן 8 ביטים**

חילוק AX ב- operand1. המנה נזרקת ל – AL והשארית ל – AH.

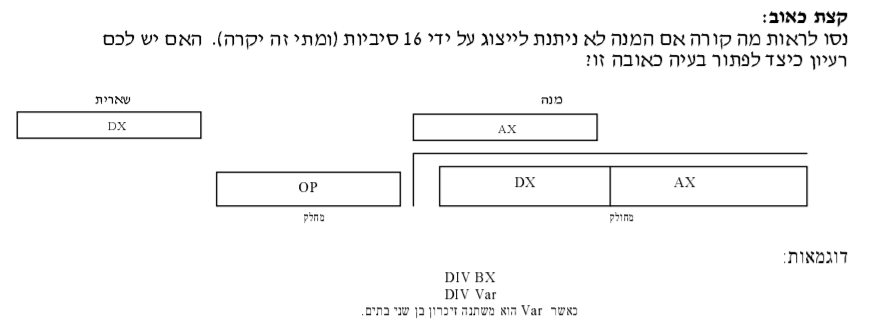


תשובה לבעיה:

נגדיר אותו מההתחלה ב DW ולא ב- DB

**אם operand1 הוא בן 16 ביטים**

חילוק DX:AX ב – operand1. המנה נזרקת ל – AX והשארית ל – DX.



תשובה לבעיה:

המשתנה שיכיל את התוצאה יהיה בגודל .(result)DD  
בחישוב, קודם נחלק ב2 את המספר שנרצה לחלק כלומר את DX:AX.  
לאחר מכן נחלק את מה שיצא לנו ב- operand1 ולבסוף נכפיל ב- 2 ונקבל את התשובה הרצויה במשתנה שהגדרנו בגודל DD(result).

**אופרנדים מותרים:** אוגר או משתנה.

**פקודת השוואה**

**CMP op1, op2**

הפקודה מבצעת חיסור op2 מ- op1 אך לא מציבה את התוצאה לתוך op1.

**אם כן מה הטעם? הפקודה משפיעה על אוגר הדגלים(אשר מושפע כהרגלו מהפעולה האריתמטית\לוגית האחרונה שהתבצעה) ולפי השינו בדגלים ניתן לבדוק איזה אופרנד גדול יותר או אם הם בכל שווים(אם הם שווים קל להבין שה – zero flag ידלוק לאחר פעולת CMP).**

* יש לשמור על כלל התאמת האופרנדים כלומר ש – op1 ו – op2 יהיו אופרנדים בגודל זהה.

לדוגמא:

**CMP** AX, BX

* אם באוגר הדגלים ה – ZF ידלק זאת אומרת שיצא לנו 0 מהפעולה האחרונה ולכן AX ו – BX שווים.
* אם באוגר הדגלים ה – SIGN FLAG ידלק זאת אומרת שתוצאת החיסור הזו גרמה למספר שלילי ולכן BX יותר גדול מ – AX.
* לעומת זאת אם ZF וגם SIGN FLAG לא דולקים זה אומר שלא יצא מספר שלילי והתוצאה היא גם לא 0 משמע AX יותר גדול מ – BX.

אופרנדים מותרים:

אוגר או זיכרון או קבוע, אך לא יכול להיות זיכרון לזיכרון.

**פקודות קפיצה**

**קפיצה בלתי מותנת**

**JMP** label

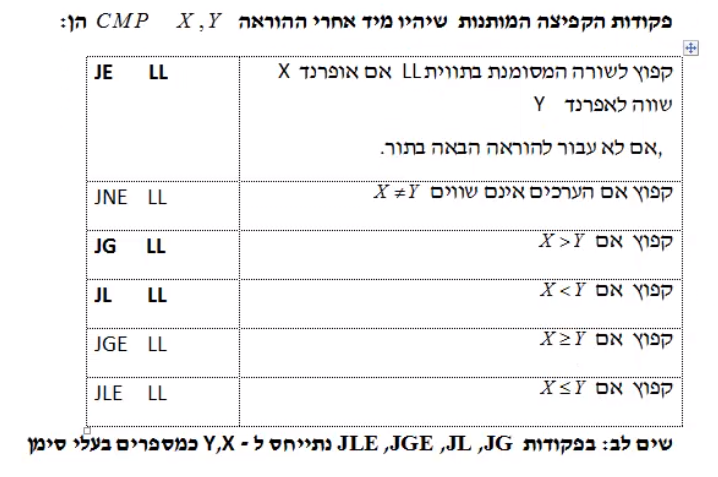
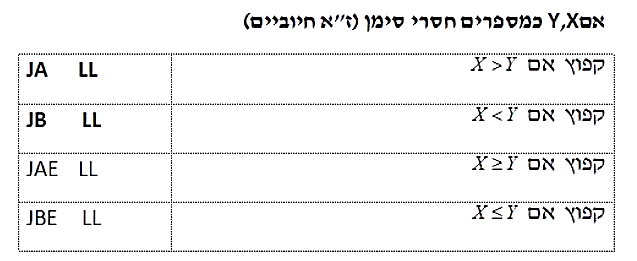
**JMP** FAR label

פקודות אלו תמיד יקפצו אל ה-label.

**קפיצה מותנת**

**פקודות הקפיצה המותנת יהיו תמיד לאחר ההוראה CMP X,Y.**

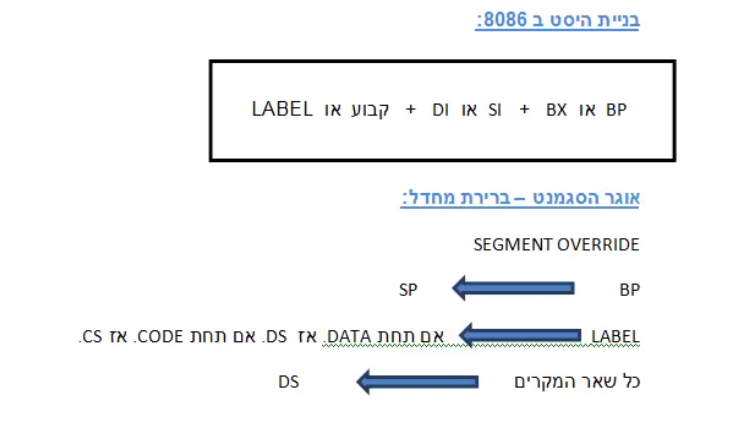
הפקודות הן:



שקול לפקודה JZ כיוון ששניהם בודקים פשוט את ה-ZF

⬄ JNZ

**אוגרי הסגמנט – (ברירת מחדל כאשר משתמשים במצביעים)**



* בתוך סוגריים מרובעים לא יכול להיות גם BX וגם BP.
* כמו כן DI לא יכול להיות ביחד עם SI.
* קבוע לא יכול להיות מיוצג עם LABEL כשלהו.
* כל השאר יכולים להיות אחד עם השני לדוגמא: [1000 + DI + BX](יהיה ב-DS כי אין BP ואין LABLE ולכן מדובר בסגמנט DS).

**כלל: מצביע חייב להיות בגודל 16 ביט כיוון שכל סגמנט הוא 64k ועל מנת שנהיה מסוגלים לעשות היסט של 64k נצטרך מצביע בגודל WORD.**

אם בפקודה השתמשנו ב-BP אז הסגמנט הוא SP.

אם בפקודה השתמשנו ב-label כלשהו אז תלוי, אם הlbl הזה הוגדר תחת .DATA אז הסגמנט הוא DS אם ה – label הוגדר תחת .CODE אז הסגמנט CS.

לכל שאר המקרים הסגמנט הוא DS.

**שימוש ב-OFFSET**

.DATA

Num DW ?

.CODE

MOV DI, OFFSET Num

MOV WORD PTR [DI], 1234 ; 🡪 MOV Num,1234

איך המחשב יודע מה ההיסט של- Num ומהיכן הוא מתחיל את ההיסט?

המחשב ממתחיל את ההיסט מהDS כיוון שהסגמנט הDEFFULT של DI הוא DS וזה מתאים לי כי Num אכן מוגדר ב-DS.

לאחר מכן הוא לוקח את הכתובת של Num ביחס לכתובת ההתחלתית של DS וזהו ההיסט.

נראה זאת כך:

כאן מתחיל DS למשל בכתובת 1000

|  |
| --- |
|  |
|  |
|  |
|  |
| Num |
|  |
|  |
|  |

1007

1005

1002

1004

1003

1000

1001

1006

עכשיו Num למשל הוא בכתובת 1004 והוא ב-DS לכן אנחנו יודעים להתחיל מ1000.

אחרי שאנחנו יודעים מאיזה סגמנט להתחיל אנחנו מחשבים את ההיסט על פי הכתובת של המשתנה פחות הכתובת של הסגמנט.

כלומר: 1004-1000 = 4.

לכן, ההיסט של Num כלומר ה-OFFSET שלו יהיה 4 ביחס לסגמנט DS אך אנחנו לא מציינים זאת שזה אוגר DS כיוון שעבדנו עם אוגר DI והאוגר ה-Deffult שלו הוא DS.

\*\* OFFSET עושים רק למשתנים!

**הפקודה LEA**

בנוסף לפקודה OFFSET ישנה את הפקודה LEA . מה היא עושה?

הפקודה LEA מקבלת שני פרמטרים **LEA** operand1, operand2

לדוגמא: **LEA DI,NUM** פעולה זו שקולה לפקודה **MOV DI,OFFSET NUM** כלומר, הפונקציה LEA מקבלת אוגר מצביע ומצביעה איתו על הכתובת של מה שהיא קיבלה ב-operand2.

אז מדוע יש את הפקודה הזו אם יש לנו את הפקודה OFFSET?

כיוון שהחישוב של ההיסט ב LEA מחושבת בזמן ריצה ולא כמו בOFFSET שהקדם מעבד מחשב את ההיסט לפני שהתכנית ריצה ומחליף שם במקום את הערך (כמו #define בשפת C).

לדוגמא:

אם נרצה לחשב את ההיסט של NUM + AX אבל לקלוט ל-AX מספר ורק אז לבצע את החישוב?

לא נוכל לעשות זאת באמצעות הפקודה OFFSET ולכן יש לנו את הפקודה LEA.

כאן מתחיל DS למשל בכתובת 1000

1007

1006

1005

1002

1004

1003

1000

1001

|  |
| --- |
| Num |
|  |
| 1 |
|  |
| 2 |
|  |
| 3 |
|  |

נגיד כאן אני יודע שNum- משוכן בכתובת 1000. ואני יודע שלאחריו באופן רציף יש עוד מספרים עד 3 ואני שואל את המשתמש איזה מספר אתה רוצה את הכתובת שלו? אז נוכל לחשב זאת באמצעות Num ובאמצעות הפקודה LEA.

נגיד המשתמש אמר אני רוצה את המספר 1 והתשובה משוכנת ב-AX אז נוכל לעשות בזמן ריצה

LEA DI,NUM+AX\*2 למה AX\*2 ? כי בדוגמא הזו כל מספר פה הוא מסוג WORD ותופס 2 בתים.

מה שהפקודה הזאת תעשה היא תגרום ל-DI להצביע על כתובת 1002.

כך נדע כתובות של משתנים תוך כדי זמן ריצה .

**שיטת המשלים ל-2**

אם נרצה לייצג מספר שלילי(קטן מ-0) עלינו לעשות 2 דברים.

* + 1. להפוך את כל הסיביות של המספר החיובי שלו
    2. להוסיף לו 1

למשל ניקח את המספר 100. על מנת לייצג -100 בבינארי ניקח את המספר 100 ונהפוך את הסיביות.

0 1 1 0 0 1 0 0 🡪100

1 1

1 0 0 1 1 0 1 1

0 0 0 0 0 0 0 1

---------------------------

1 0 0 1 1 1 0 0 🡪-100

**הרחבת מספרים**

אם נרצה להרחיב מספר כלומר למשל יש לי מספר ב – AL אבל אני רוצה לשים אותו ב-AX אז אם מדובר במספר שלם פשוט מאפסים את AH אבל אם מדובר במספר שלילי נשתמש בפקודה CBW.

**הפקודה CBW** – convert byte to word כביכול מורחת את סיבית הסימן שב-AL לאורך כל AH.

**.DATA**

**byte\_val DB -100**

**.CODE**

**mov al, byte\_val ; AL = 9Ch = -100**

**cbw ; AX = FF 9Ch**

**המספר -100 הסימן '-' מרוח על כל – AH**

|  |  |
| --- | --- |
| **1 1 1 1 1 1 1 1** | **1 0 0 1 1 1 0 0** |

**אוגר AL**  **אוגר AH**

על ידי כך נוכל לגרום ל -100 להיות מיוצג מעכשיו בAX- ולא ב- AL.

בדומה לכך אם נרצה להפוך מספר המיוצג ב16 סיביות שיהיה מיוצג ב32 סיביות נשתמש בפקודה CBW.  
הפקודה CBW מורחת את סיבית הסימן שב-AX על כל אורך DX.

**.DATA**

**word\_val DW -100 ; FF9Ch**

**.CODE**

**mov ax, word\_val ; AX = FF9Ch**

**cwd ; DX:AX = FFFFh:FF9Ch**

**המספר 100- הסימן '-' מרוח על כל – AH הסימן '-' מרוח על כל – DX**

|  |  |  |  |
| --- | --- | --- | --- |
| **1 1 1 1 1 1 1 1** | **1 1 1 1 1 1 1 1** | **1 1 1 1 1 1 1 1** | **1 0 0 1 1 1 0 0** |

**אוגר AX** **אוגר DX**

על ידי כך נוכל לגרום ל -100 להיות מיוצג מעכשיו בDX- ולא ב- AX.

**נשים לב פעולות אלו אינן מקבלות אופרנדים כלל.**

**נשתמש בפקודות אלו רק למספרים שלילים! אם המספר לא שלילי פשוט נאפס את החלק השמאלי.**

**הפקודה NEG**

הפעולה NEG הופכת מספר שלילי לחיובי ומספר חיובי לשלילי.

כלומר עושה כפל במינוס 1.  
פעולה זו היא פעולת אופרנד 1.

נניח שב-AL יש את המספר 8(00001000).

לאחר ביצוע הפקודה:

NEG AL

ב-AL יהיה הערך 8- (11111000).

אם נעשה שוב:

NEG AL

AL יחזור להיות הערך 8.

**תנאים ולולאות**

**תנאי בשפת C:**

If(AX == BX)

{

פקודות 1

}

פקודות 2

**תנאי באסמבלי:**

Cmp AX,BX

JNE label2

פקודות 1

label2:

פקודות 2

**If-else in c**

If(AX == BX)

{

פקודות 1

{

Else

{

פקודות 2

{

פקודות 3

**If-else in assembly**

Cmp AX,BX

JNE label22

פקודות 1

JMP label3

label2:

פקודות 2

label 3:

פקודות 3