**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**

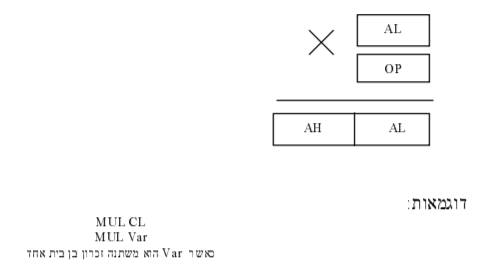
**MUL** operand1

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

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

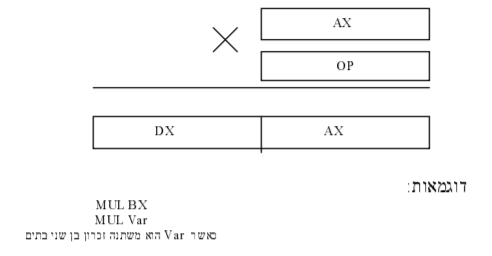
**אם operand1 הוא בן 8 ביטים**

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.

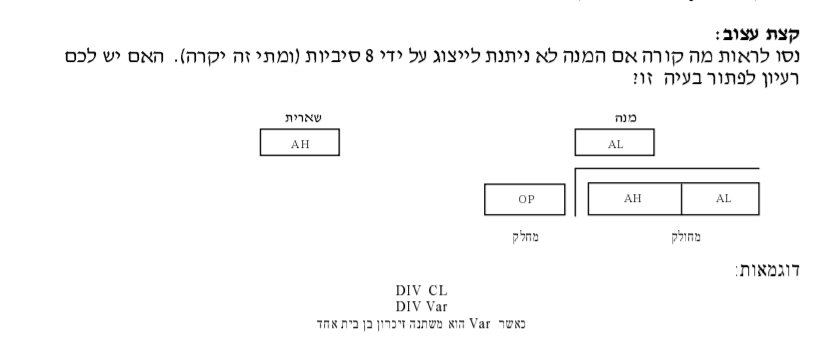
לבסוף נקבל את התוצאה שלנו בשלמות במשתנה result.

**פעולת חילוק**

**DIV** operand1

**אם operand1 הוא בן 8 ביטים**

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

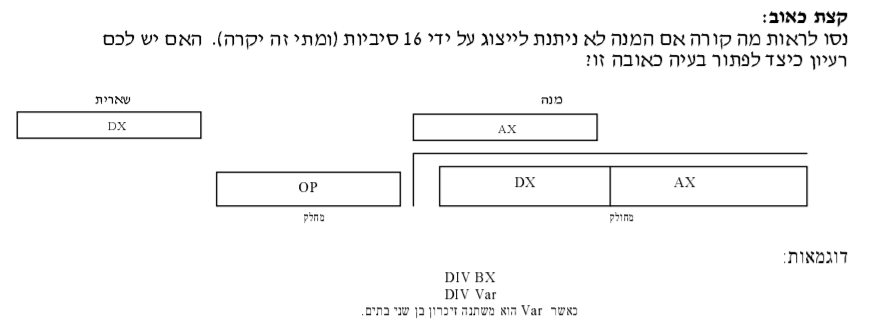


תשובה לבעיה:

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

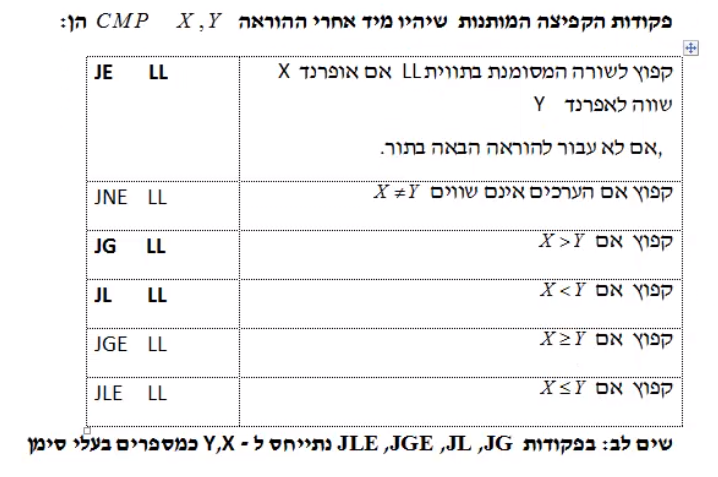
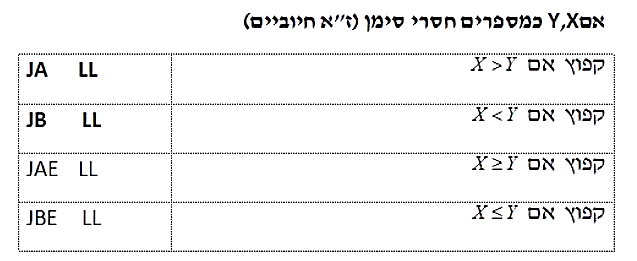
**JMP** FAR label

פקודות אלו תמיד יקפצו אל ה-label.

**קפיצה מותנת**

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

הפקודות הן:

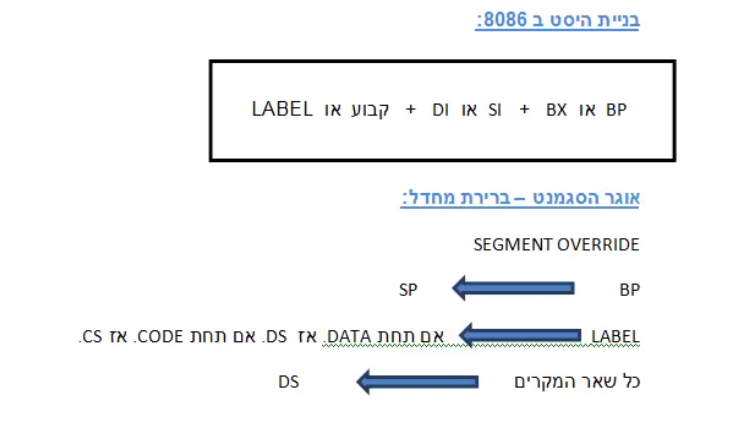


שקול לפקודה JZ כיוון ששניהם בודקים פשוט את ה-ZF

⬄ JNZ

**אלו לא כל פקודות הקפיצה אך אלו העיקריות. השאר יובאו בסוף הסיכום.**

**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).

**חשוב לזכור: 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,OFFSET 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 CWD.  
הפקודה CWD 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**

**תנאים בשפת אסמבלי יבוצעו באמצעות CMP\TEST\פקודה המשפיעה על אוגר הדגלים ולאחר מכן איזשהו JUMP.**

**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

**בשפת C:**

If((AX == BX) && (cx < dx)){

פקודות 1

{

פקודות 2

**באסמבלי:**

**Cmp AX,BX**

JNE label2

**Cmp CX,DX**

JNL label2

פקודות1

Label2:

פקודות 2

**בשפת C:**

If((AX == BX) || (cx < dx)){

פקודות 1

{

פקודות 2

**באסמבלי:**

**Cmp AX,BX**

JE label1

**Cmp CX,DX**

JL label1

JMP label2

Label1:

פקודות1

Label2:

פקודות 2

**לולאות**

בבחינה שלנו ירדו כל הלולאות חוץ מלולאת LOOP .

נסביר קצת על לולאה זו וכיצד היא עובדת.

מבנה הפקודה:

**LOOP** label

**הפקודה עושה 2 פעולות:**

1. DEC CX
2. JNZ label

כלומר הפעולה LOOP שקולה בשפת C למבנה:

**For(CX = N; CX != 0; CX--){**

**פקודות**

**{**

כיוון ש-LOOP עושה את הלולאה CX פעמים אנחנו נשתמש ב-LOOP רק כשאר אנחנו יודעים כמה פעמים אנחנו אמורים לבצע את הפעולה.

אם אנחנו רוצים לבצע לולאה אך איננו יודעים כמה פעמים היא אמורה לרוץ כלומר כמו לולאת WHILE בשפת C אנחנו נצטרך להשתמש ב-CMP וב-JUMP כלשהו.

נראה שתי אופציות לכתיבת לולאת WHILE:

1.

loop1:

CMP AX,BX

**JNE label2**

פקודות1

**JMP loop1**

label2:

פקודות2

2. 🡪 הדרך הזו נראת יותר מובנת אך שתי הדרכים טובות

**JMP loop1con**

loop1:

פקודות1

loop1con:

CMP AX,BX

**JE loop1**

פקודות2

נוכל להבין בקלות אם כך איך ליישם גם לולאת DO-WHILE.

פשוט לא נעשה את התנאי בפעם הראשונה וניתן לו לרוץ ישר על הפקודות.

loop1:

פקודות1

CMP AX,BX

**JE loop1**

פקודות2

**JMP loop1con**

loop1:

פקודות1

loop1con:

CMP AX,BX

**JE loop1**

פקודות2

**אם נרצה לכתוב לולאת FOR נעשה זאת כך:**

**For(ax = 0; a ax < bx; ax++){**

**פקודות1**

**{**

**פקודות 2**

2.

MOV AX,0

**JMP loop1con**

loop1:

פקודות1

loop1con:

CMP AX,BX

**JE loop1**

פקודות2

1.

MOV AX,0

loop1:

CMP AX,BX

**JAE label2**

פקודות1

INC AX

**JMP loop1**

label2:

פקודות2

**המחסנית**

עבור המחסנית יש לנו 3 אוגרי הצבעה SS,SP,BP.

כאשר אנחנו בתחילת התוכנית מגדירים:

.STACK 100H

מה שקורה בעצם זה התוכנית מקצה סגמנט עבור המחסנית.

SS – הוא יצביע על תחילת אותו סיגמנט של המחסנית.

SP – יצביע לכתובת שלאחר סוף המחסנית. כלומר אם הסגמנט שלי הוא הסגמנט הראשון ואני התחלתי מכתובת 0 והגדרתי אותו בתור H100 (256) אז סוף המחסנית שלי היא בכתובת 255 ולכן SP יצביע על כתובת 256.

BP – לא יצביע על כלום בהתחלה אך נזכור שאם נפנה לכתובת כלשהי באמצעותו זה יהיה ב-SS אלא אם כן עשינו segment override כמו שאמרנו כשדיברנו על אוגרי הסגמנט.

על מנת לעבוד עם המחסנית יש לנו 2 פקודות מכונה:

1. PUSH
2. POP

**הפקודה PUSH**

הפקודה PUSH היא פקודת אופרנד 1 שיכול להיות או אוגר או זכרון אבל הוא חייב להיות בגודל 16 ביט כלומר WORD.

PUSH AX

פקודת PUSH עושה 2 פעולות:

1. SP = SP -2
2. העתק לכתובת ש-SP מצביע עליה את האופרנד(AX).

**מצב התחלתי:**

כאן מתחיל SS למשל בכתובת 0

256

0

|  |
| --- |
| SS |
|  |
|  |
|  |
|  |
|  |
|  |
| SP |

254

255

כאן נגמרת המחסנית

כאן נמצא בהתחלה ה-SP בכתובת 256

נניח ש AX = 100 -

לאחר הפקודה PUSH AX

יקרה הדבר הבא:

**מצב לאחר PUSH AX:**

כאן מתחיל SS למשל בכתובת 0

256

0

|  |
| --- |
| SS |
|  |
|  |
|  |
|  |
| 100 |
| 0 |
|  |

כאן נמצא ה-SP בכתובת 254

254

255

כאן נגמרת המחסנית

**הפקודה POP**

הפקודה POP היא פקודת אופרנד 1 שיכול להיות או אוגר או זכרון אבל הוא חייב להיות בגודל 16 ביט כלומר WORD.

POP AX

פקודת POP עושה 2 פעולות:

1. השמה לאופרנד(AX) את הערך שבכתובת ש-SP מצביע עליה.
2. SP = SP + 2

**מצב לאחר PUSH AX:**

כאן מתחיל SS למשל בכתובת 0

256

0

|  |
| --- |
| SS |
|  |
|  |
|  |
|  |
| 100 |
| 0 |
|  |

כאן נמצא ה-SP בכתובת 254

254

255

כאן נגמרת המחסנית

לאחר הפקודה POP NUM (כאשר NUM מוגדר בתור DW)

יקרה הדבר הבא:

**מצב לאחר PUSH AX:**

כאן מתחיל SS למשל בכתובת 0

256

0

|  |
| --- |
| SS |
|  |
|  |
|  |
|  |
| 100 |
| 0 |
| SP |

254

255

כאן נגמרת המחסנית

כאן נמצא ה-SP בכתובת 256

NUM = 100

SP = 256 (לכתובת כמובן)

* **אנחנו רואים שגם לאחר POP אנחנו איננו מוחקים את הערכים אך הם ידרסו בפעם הבאה שנעשה PUSH כך שזה לא משנה.**

**פרוצדורות**

**ראשית נכיר 2 פקודות:**

* + 1. CALL
    2. RET

**הפקודה CALL**

CALL func1

1. דחיפת הכתובת של הפקודה הבאה לביצוע במחסנית.
2. קפיצה ל – func1

**הפקודה RET**

RET

מבצעת POP אל תוך IP.

**כשאנו מוסיפים מספר ליד פקודת ה־ret ,לאחר ביצוע ה-pop ,נוסף ל-sp הערך שרשמנו ליד ה ־ret .**

**הפקודות השקולות ל־6 ret הן:**

pop bx ; pop increments sp by 2

add sp, 6 ; sp is incremented by a total of 8

jmp bx

**הגדרת פרוצדורה תתבצע באופן הבא:**

שם הפרוצדורה PROC סוג הפרוצדורה

קוד הפרוצדורה

RET

שם הפרוצדורה ENDP

\*סוג הפרוצדורה = NEAR\FAR

**לדוגמא:**

.CODE

getChar PROC NEAR

MOV AH,1

INT 21H

RET

getChar ENDP

call getChar ;כך נזמן אותה בעצם

**העברה by value**

בדוגמה הבאה נראה איך פרוצדורה משתמשת בפרמטרים שהועברו אליה בשיטת Value by Pass נניח שיש פרוצדורה בשם SimpleProc שמקבלת שלושה פרמטרים i ,j ו k - ומחשבת בתוך ax את k-j+I .

SimpleProc proc NEAR

pop ReturnAddress ;בגלל שעשינו קריאה לפרוצדורה אז נכנס למחסנית גם הכתובת של הפקודה הבאה לכן נשמור אותה רגע בצד.

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

SimpleProc endp

התכנית הראשית:

push [i]

push [j]

push [k]

call SimpleProc ;התוצאה תשמר באוגר AX

שיטה זו של שמירה את כתובת החזרה אינה מקובלת ואין עושים אותה.

אז איך כן עושים?

**נשתמש באוגר – BP**

הרגיסטר bp ,קיצור של Pointer Base ,מסייע לנו לגשת לפרמטרים שהתוכנית הראשית הכניסה למחסנית מבלי

להתעסק עם הרגיסטר ip .שימו לב לשורות הקוד שאנחנו מוסיפים לפרוצדורה (מודגשות):

SimpleProc proc NEAR

push bp

mov bp, sp

… ;Code of the stuff the procedure does

pop bp

ret 6

SimpleProc endp

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 .

|  |
| --- |
| SS |
|  |
| bp |
| Ip(from call) |
| K |
| J |
| I |
| SP |

כאן מתחיל SS למשל בכתובת 0

0

כאן נמצא ה-SP בכתובת 246

247+246

BP = SP

249+248

251+250

256

כאן נגמרת המחסנית

253+252

255+254

מה זה נתן לנו שעשינו את זה?

עכשיו גם אם נכניס עוד דברים למחסנית ונוציא אחר כך תמיד נוכל לפנות אל המשתנים שלנו (i\j\k) באמצעות BP פלוס מרחק הכתובות.(חשוב לזכור שכל איבר הוא 2 כתובת).

נדגים זאת שוב על הפרוצדורה SimpleProc ,שמבצעת את הפעולה k-j+i=ax .

הקוד הבא מבצע זאת:

SimpleProc proc NEAR

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

SimpleProc endp

נשים לב שאנחנו שמנו העתקים של המשתנים וככה אנחנו בעצם לא נשנה את המשתנים העיקריים(אם "נשנה").

**העברה by pointer**

SimpleAdd proc NEAR

;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 "

ADD [bx], 2 ; This actually changes the value of "parameter "

PUSH ReturnAddress

RET

SimpleAdd ENDP

התכנית הראשית:

PUSH offset parameter ; Copy the OFFSET of "parameter" into the stack

CALL SimpleAdd

אם נרצה לעשות את התוכנית אבל באמצעות BP כמו שאמרנו שנהוג נעשה זאת כך:

SimpleAdd proc NEAR

;Takes as input the address of a parameter, adds 2 to the parameter

PUSH bp

MOV BP,SP ;that BS will save the address of the current SP

MOV BX,BP+4;go to the first thing we insert wich is the address of the parameter and put it in bx

ADD [bx], 2 ; This actually changes the value of "parameter "

PUSH ReturnAddress

RET

SimpleAdd ENDP  
**\*the main will stay as it was**

**רקורסיה**

כמו שאנחנו מכירים משפת C יש לנו:

1. תנאי עצירה.
2. מה שמתבצע לפני הקריאה לרקורסיה.
3. הקריאה לרקורסיה.
4. מה שמתבצע בחזרה מהרקורסיה.

תמיד בתרגיל נסמן לנו מהו כל חלק וכך יהיה לנו הרבה יותר מובן.

נקח לדוגמא תרגיל משנת 2014(השורות המודגשות הן השלמות):

נשים לב לכמה דגשים חשובים:

* אחרי פקודה שמשנה את אוגר הדגלים בדרך כלל יהיה JUMP כלשהו.
* לפני קריאה לרקורסיה נתחייב לעשות PUSH למשהו.
* כמעט תמיד נעשה את שני השורות הנל ברקורסיה:

Push bp

Mov bp,sp

* BP בעצם משמש לנו כאן כדי שנדע לאיזה IP שמבחסנית שנכנס בעקבות הקריאות לרקורסיה אנחנו נחזור עכשיו.

Sseg segment stack 'stack'

DB 100H dup(?)

Sseg ENDS

Code segment

Assume cs:code

num DW 2364

**Start:**

Push num  
 call rec  
 mov ah,4ch  
 int 21h

**Rec:**

Push bp

Mov bp,sp

Mov ax, [bp+4]

Mov BL,10

DIV BL

מה שקורה לפני הקריאה לרקורסיה

Or AL,AL

תנאי עצירה

JZ stop\_rec

Mov [bp+4],ah

Mov byte ptr [bp+5],0

Xor AH,AH

Push AX

הקריאה לרקורסיה

Call rec

**Stop\_rec:**

Cmp AL,[BP+4]

JA con

Mov AL,[BP+4]

מה שקורה בחזרה מהרקורסיה

**Con:**

Pop ax

Ret 2

Code ends

End start

**אופרטורים**

1) HIGH: returns higher byte of an expression

2)LOW: returns lower byte of an expression.

NUM EQU 1374H

MOV AL HIGH Num ; ( [AL] 13 )

3) OFFSET: returns offset address of a variable

4) SEG: returns segment address of a variable

5) PTR: used with type specifications BYTE, WORD, RWORD, DWORD, QWORD

INC BYTE PTR [BX]

6) Segment override

MOV AH, ES: [BX]

7) LENGTH: returns the size of the referred variable

8) SIZE: returns length times type

BYTE VAR DB?

WTABLE DW 10 DUP (?)

MOV AX, TYPE BYTEVAR ; AX = 0001H

MOV AX, TYPE WTABLE ; AX = 0002H

MOV CX, LENGTH WTABLE ; CX = 000AH

MOV CX, SIZE WTABLE ; CX = 0014H

**סוגי מיעון**

**מיעון אוגר**

העתקת ערך מאוגר מקור לאוגר יעד. לדוגמא:

MOV AX, BX

**מיעון מידי**

העתקת ערך מיידי לתוך אוגר או משתנה. לדוגמא:

MOV AX, 12

**מיעון ישיר**

העתקת ערך משתנה לתוך אוגר. לדוגמא:

MOV AX, Var

**מיעון עקיף**

העתקת ערך מהזכרון לתוך אוגר. לדוגמא:

MOV AX, [BX]

הדוגמאות הבאות שוות ערך:

MOV AX, [BX+8]

MOV AX, [BX]8

MOV AX, 8[BX]

הדוגמאות הבאות שוות ערך:

MOV AX, [DS]:[BX]

MOV AX, DS:[BX]

MOV AX, [BX]

MOV AX, DS+BX

**מיעון אינדקס**

העתקת איבר במערך לתוך אוגר. הדוגמאות הבאות שוות ערך:

MOV AX, Array[BX]

MOV AX, [Array+BX]

MOV AX, [BX]+Array

הדוגמא הבאה מעתיקה לתוך) AX את התוכן שנמצא בכתובת של המשתנה Var ועוד 1: (זהה ל-Var1[1]z

MOV AX, Var+1

הדוגמאות הבאות שוות ערך:

MOV DL, Array[SI+1]

MOV DL, Array+1[SI]

הדוגמאות הבאות שוות ערך:

MOV DL, Array[BX+SI]

MOV DL, Array[BX][SI]

הדוגמאות הבאות שוות ערך:

MOV DL, Array[BX+SI+4]

MOV DL, Array[BX][SI]4

הדוגמא הבאה אינה חוקית:

MOV DL, Array[SI+DI]

שימו לב: לא ניתן לבצע העתקה מזכרון לזכרון. הדוגמא הבאה אינה חוקית:

MOV [BX], Var

כמו כן, לא ניתן לבצע מיעון על מקטע בלבד. הדוגמא הבאה אינה חוקית:

DEC [DS]