גירסה 1.00 – 7.11.2002

מבנה ה-PC, האסמבלר של 8086

Nir Adar
Email: underwar@hotmail.com
Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

מבנה מכונת ה-PC

מעבדים

ניתן להגיד כי מעבד מגדיר מהו סוג המחשב.
המעבד 8086 היה מעבד בו מילה היתה בת 16 סיביות, ותמך בעד 1MB של זיכרון.
אחריו בא ה-8186 – שתמך ביותר פקודות.
80286 – מילה בגודל 24 ביט.
80386 – מילה בגודל 32 ביט.
וכך הלאה...
כל מעבד חדש מסוגל לבצע תוכניות שנכתבו עבור מעבדים ישנים יותר.
במסמך זה נתרכז ב-8086.

מרחב הכתובות

המעבד מסוגל לגשת למרחב כתובות של 1MB.
כל כתובת ניתנת לייצוג על ידי 20 ביטים, אולם מילת המחשב היא בגודל 16 ביטים בלבד, ולכן נעזר בשתי מילים כדי לגשת לכתובות בזיכרון.
כתובת בזיכרון תראה מהצורה :<Offset>.
 ו-<Offset> הן מילים, שהצירוף שלהן לפי הנוסחה הבאה נותן ערך אבסולוטי של כתובת בזיכרון:

<Address> = *16 + <Offset>

נשים לב שלכתובת אחת יכולים להיות ייצוגים רבים.
כל Segment וכל Offset הם מילים, ולכן עבור אות הסגמנט ניתן לגשת אל $2^{16}-1$ כתובות (64K).

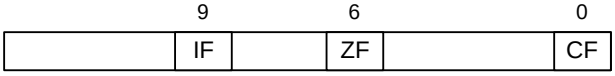
ישנם שני סוגים של כתובות: far ו-near.
כאשר אנחנו מדברים על כתובת שהיא near, אנו מתכוונים שהיא נמצאת בסגמנט הנוכחי בו אנו נמצאים. כאשר נרצה לשמור כתובת near בזיכרון, נשמור רק את ה-offset שלה, והסגמנט יהיה זהה לנוכחי.
כאשר אנו מדברים על כתובת שהיא far, היא יכולה להיות גם בסגמנטים אחרים, ולכן כשאנו באים לשמור כתובת far בזיכרון, אנו שומרים את הסגמנט וגם את ה-offset.

רגיסטרים

כל הרגיסטרים הם בגודל מילה (16 ביט). הרגיסטרים מתחלקים לקבוצות הבאות:

שימושי הרגיסטרים	הרגיסטרים הקיימים
רגיסטרים לשימוש כללי	AX, BX, CX, DX
רגיסטרי אינדקס	SI (Source Index) DI (Destination Index)
רגיסטרי Segment	SS (Stack Seg.) CS (Code Seg.) DS (Data Seg.)
מצביע לראש המחסנית	SP (Stack Pointer)
מצביע לפקודה הבאה לביצוע בקוד התוכנית	IP (Instruction Pointer)
מצביע על הפרמטרים של הפונקציה הנקראת	BP (Base Pointer)
דגלים	Flags

- AX, BX, CD, DE הם רגיסטרים בגודל מילה אחת, אך ניתן להשתמש בכל אחד מהם כשני רגיסטרים בגודל בית אחד. כדי לפנות לבית הגבוה של AX, נרשום AH, וכדי לפנות אל הבית הנמוך נרשום AL. בצורה דומה אפשר להשתמש גם ב-BX, CX, DX.
- SS מחזיק את סגמנט המחסנית. כדי לגשת לבסיס במחסנית נשתמש ב-SS:0. כדי לגשת לראש המחסנית נשתמש ב-SS:SP. המחסנית תמיד גדלה מלמעלה כלפי מטה (הוספת איבר מקטינה את SP).
- בכל רגע נתון, IP מצביע על הפקודה הבאה לביצוע בסגמנט הקוד. CS:IP מכיל את הכתובת האבסולוטית של הפקודה הבאה לביצוע.
- BP מצביע על הפרמטרים שנשלחו על המחסנית לפונקציה המתבצעת.
  על ידי שימוש במרחק יחסי מהBP ניתן להגיע לפרמטרים השונים, לכתובת החזרה מהפונקציה ולמשתנים המקומיים. לדוגמא, הפרמטר הראשון של הפונקציה נמצא ב-[BP+4].
  כדי להגיע לכתובת האבסולוטית יש להשתמש בסגמנט השמור ב-SS.
- הרגיסטר Flags כולל את ביט אישור הפסיקות וכן בית Condition Codes.



- Zero Flag – ZF – מציין האם הערך האחרון שחושב הוא 0.
- Carry Flag – CF – מציין אם בעת חישוב הערך האחרון נוצר Carry אל מעבר לביט העליון של המילה. (גלישה)
- Interrupt Flag – IF – קובע האם פסיקות חומרה מאופשרות או לא. כשכבוי הפסיקות אינן מאושרות, וכשדלוק הן מאופשרות.
  ניתן לכבות או להדליק דגל זה בעזרת פקודות המכונה sti, cli.

רגיסטרים לשימוש כללי

הרגיסטרים AX, BX, CX, DX הם רגיסטרים לשימוש כללי.
לכל אחד מהם, עם זאת, יש שימושים.
פסיקות תוכנה ופקודות שונות באסמבלי מניחים לעיתים שיש בהם מידע מסויים.

AX	BX	CX	DX
Acc – משמש לפעולות אריתמטיות.	Base – משמש לפניה לזיכרון	Counter – מונה בלולאות	Data – שומר מספר port של פעולות I/O.

ווקטור הפסיקות

טיפול בפסיקה מתבצע באמצעות הרצת שיגרה מיוחדת האחראית לטפל באותה הפסיקה. לשיגרה כזו קוראים ISR – Interrupt Service Routine.
למבני הנתונים המקשר בין כל מספר פסיקה לשגרת הטיפול בה אנו קוראים ווקטור הפסיקות. 1Kbyte בתחילת הזיכרון מוקצה עבור ווקטור הפסיקות.
כל כניסה בווקטור מכילה כתובת של שגרת טיפול בפסיקה, עבור הפסיקה המתאימה. כתובת שיגרת ISR: 4 בתים – 2 עבור הסגמנט ועוד 2 עבור ה-Offset.
לכן יש 1024/4=256=0x100 כניסות בווקטור הפסיקות.
הפסיקות מזוהות על ידי מספר 0..FF. כשמגיעה פסיקה מספר v, מריצים את שגרת הטיפול הנמאת בזיכרון בכתובת 4v.

חריגות

חריגות – חילוק באפס, נפילת מתח וכו', מטופלות כפסיקות.
דוגמאות:
כניסה 0 בווקטור הפסיקות שמורה לחריגה – חילוק ב-0.
כניסה 4 שמורה עבור גלישה (overflow).

פסיקות תוכנה

ניתן לייצר פסיקות תוכנה באמצעות פקודת האסמלבר int.
תחביר:

int <מספר פסיקה>

סביבת ריצה

ישנם מספר מודלים של סביבת ריצה. המודל אותו נציג:
התוכנה נחלקת לשני סגמנטים, שכל אחד מהם מכיל עד64K.

- סגמנט קוד – מוצבע על יד CS.
  מכיל את כל הקוד של התוכנית.
- סגמנט נתונים ומחסניות, מוצבע על ידי SS, DS.
  מכיל את כל הנתונים הסטטיים בזיכרון, את המחסנית/מחסניות ואת שטח הזיכרון המיועד להקצאות זיכרון דינמיות.

פקודות Assembly

פקודות רבות בשפת האסמבלי של ה-8086 פועלות על שני אופרנדים.
פקודות אלו הן מהצורה הבאה:

OP R1, R2

הפעולה שמתבצעת כתוצאה מהרצת פקודה מצורה זו היא:
R1 ← R1 OP R2

כלומר, הפעולה מתבצעת על הרגיסטרים, ולאחר מכן התוצאה נשמרת ברגיסטר הראשון.
נביט כעת במבחר פקודות של שפת האסמבלי.

1. ADD

דוגמא:

ADD AX, BX

חבר את תוכן BX ל-AX, ושמור את התוצאה ב-AX.
AX ← AX + BX.

2. MOV

דוגמא:

MOV BH, DL

העבר את תוכן DL אל BH.
BH ← DL

3. SUB

דוגמא:

SUB AX, [BX]

מתוכן AX מחסירים את תוכן הזיכרון ש-BX מצביע אליו ושמים את התוצאה ב-AX.

4. גישה לכתובת בזיכרון

MOV AX, SS:[BX]

פנייה לכתובת אבסולוטית בזיכרון, שמוגדרת על ידי SS ו-BX.

5. JMP

דוגמא:

JMP 1234H

הפקודה הבאה לביצוע תהיה : CS:1234H.

6. פקודות קפיצה מותנית

להלן רשימת פקודות הקפיצה המותנית לפי דגלים, הקיימים ב-8086 :

Instruction	Description	Condition	Aliases	Opposite
JC	Jump if carry	Carry = 1	JB, JNAE	JNC
JNC	Jump if no carry	Carry = 0	JNB, JAE	JC
JZ	Jump if zero	Zero = 1	JE	JNZ
JNZ	Jump if not zero	Zero = 0	JNE	JZ
JS	Jump if sign	Sign = 1	-	JNS
JNS	Jump if no sign	Sign = 0	-	JS
JO	Jump if overflow	Ovrflw=1	-	JNO
JNO	Jump if no Ovrflw	Ovrflw=0	-	JO
JP	Jump if parity	Parity = 1	JPE	JNP
JPE	Jump if parity even	Parity = 1	JP	JPO
JNP	Jump if no parity	Parity = 0	JPO	JP
JPO	Jump if parity odd	Parity = 0	JNP	JPE

7. DIV

פקודה זו משמשת לחלוקה מספרים.

דוגמא:

DIV BH

הפעולה שתתבצע :

AL ← AX / BL
AH ← AX % BL

8. קריאה לפונקציות של DOS

בעזרת הפקודה int ואחריה פרמטר מספרי אנו יכולים להפעיל פסיקת תוכנה.
אם זאת, הפרמטר המספרי שאחרי int מוגבל לתחום 0-255.
נרצה לתמוך במכונה המבינה יותר מ-255 פקודות – רק MS-DOS מגדירה 100 פקודות, בעוד שה-BIOS מגדיר אלפי פקודות.

לפיכך, נעבוד בצורה הבאה:
כל פרמטר מספרי יציין למעשה משפחה של פונקציות, אותן תוכל הפסיקה לבצע.
ברגיסטר כלשהו על המעבד יישמר מספר נוסף, שיציין איזו פקודה נרצה לבצע.
MS-DOS בחרה את פסיקה מספר 21H למטרה זו. כאשר נרצה לבצע קריאת ל-System Call של מערכת ההפעלה, נשים ב-AH את מספר הפונקציה המבוקשת, ונקרא לפסיקה 21H. לדוגמא:

```
mov   ah, 4ch          ;DOS terminate opcode.
int   21h              ;DOS call
```

9. RET

כאשר אנו רוצים לחזור מפונקציה אל הפונקציה שקראה לה, נשתמש בפקודה RET. הפקודה RET לוקחת מהמחסנית כתובת וקופצת אליה.

10. IRET

הפקודה IRET דומה לפקודה RET, מלבד זאת שהיא משמשת לחזרה מפונקצית פסיקה.

שיטות המיעון בשפת אסמבלי

נציג כעת חלק משיטות המיעון של שפת האסמבלי של 8086.
שיטות המיעון הן למעשה הדרך בעזרתה אנו ניגשים לזיכרון המחשב.

שיטת המיעון רגיסטר

כפי שראינו, על ידי שימוש בשם הרגיסטר כאופרנד, אנו יכולים לשנות או לקבל את תוכן הרגיסטר.
מספר דוגמאות לשימוש בשיטת מיעון זו:

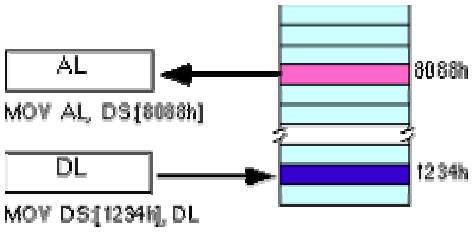
```
mov     ax, bx          ;Copies the value from BX into AX
mov     dl, al          ;Copies the value from AL into DL
mov     si, dx          ;Copies the value from DX into SI
mov     sp, bp          ;Copies the value from BP into SP
mov     dh, cl          ;Copies the value from CL into DH
mov     ax, ax          ;Yes, this is legal!
```

שיטת המיעון **Displacement Only**

בשיטת מיעון זו אנו נותנים כתובת קבועה, אליה נכתוב או ממנה ניקח נתונים.
דוגמא:

```
mov al,ds:[8088h]
```

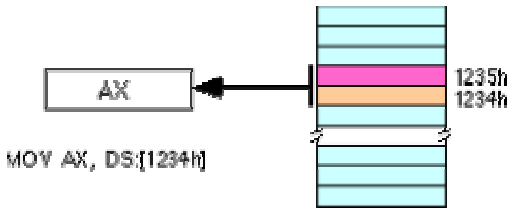
בדוגמא זו אנו שמים ב-al את הבית המצוי בכתובת 8088h.



אנו יכולים גם להעתיק מילים מהזיכרון, ולא רק בתים.
אם נכתוב לעומת זאת את השורה הבאה:

```
mov ax,es:[1234h]
```

אזי תועתק מילה מהכתובת 1234h אל ax. כמות הזיכרון המועתקת נקבעת בהתאם לגודל האופרנד שלנו.

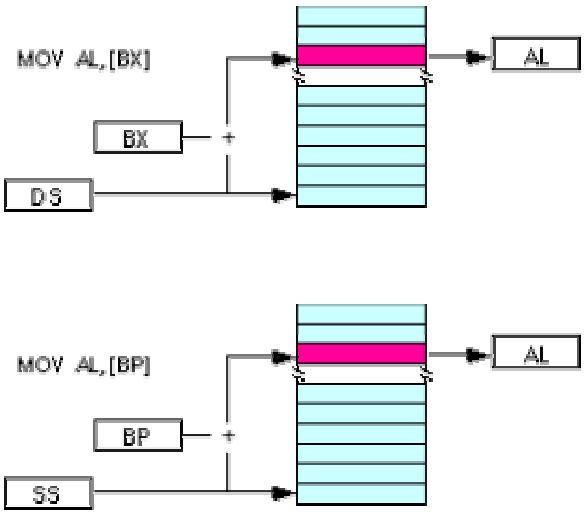


שיטת המיעון Register Indirect

בשיטת מיעון זו אנו ניגשים לרגיסטר, המכיל Offset, ואז פונים לסגמנט, מוסיפים לו את ה-Offset השמור ברגיסטר, וזהו למעשה האופרנד שלנו.
קיימים 4 ניואנסים שונים של שיטת מיעון זו – הסגמנט אליו נפנה תלוי ברגיסטר בו נשתמש.
נביט בדוגמא שתסביר זאת:

```
mov     al, [bx]
mov     al, [bp]
mov     al, [si]
mov     al, [di]
```

לכאורה ב-4 המקרים אנו לוקחים את תוכן הרגיסטר, מתייחסים אליו ככתובת, ושמים את תוכן כתובת זו ב-al, אולם בחלק מדוגמאות אלו אנו מתייחסים לסגמנט אחר.
[bx], [si] ו- [di] משתמשות בסגמנט ds כברירת המחדל, בעוד ש- [bp] משתמש בסגמנט ss כברירת המחדל.



אם נרצה, נוכל להכתיב לאסמבלר באיזה סגמנט להשתמש, ללא קשר למהו סגמנט ברירת המחדל. נעשה זאת בצורה הבאה:

```
mov     al, cs:[bx]
mov     al, ds:[bp]
mov     al, ss:[si]
mov     al, es:[di]
```

עבור כל אחד מרגיסטרים אלו, נפנה אל סגמנט אחר.

קריאה לפונקציה בשפת המכונה

שלבי הקריאה לפונקציה:

1. באחריות הפונקציה הקוראת לשמור את ערכי הרגיסטרים AX, BX, CX, DX במידה וידרשו לאחר החזרה.
   אין צורך לשמור רגיסטרים בהם הפונקציה הקוראת אינה משתמשת.
2. הפונקציה הקוראת דוחפת את הפרמטרים על המחסנית (בסדר הפוך) – הפרמטר הראשון הוא הקרוב ביותר לכתובת החזרה (BP+4), הפרמטר השני אחריו וכו'.
3. ביצוע פקודת המכונה לקריאה לפונקציה (כתובת החזרה נשמרת על המחסנית). ה-IP שנשמר על המחסנית כבר מכיל את הכתובת של הפקודה הנמצאת אחרי פקודת הקריאה לפונקציה.
4. באחריות הפונקציה הנקראת לשמור את ערכי הרגיסטרים BP, SI, DI (ואת flags אם היא מבצעת cli או sti), ולהקצות על המחסנית מקום למשתניה המקומיים.
   SI, DI נשמרים רק אם הפונקציה הנקראת משתמש בהם.
   BP נשמר על המחסנית לפני עדכונו. בכך מבטיחים שכל ערכי BP מהווים מעין "רשימה מקושרת", כך שכאשר יוצאים מפונקציה, ניתן לשחזר את הערך הקודם.
   משתנים לוקליים "מוקצים" מתוך המחסנית (על ידי קידום SP). בכך מבטיחים שכאשר יוצאים מהפונקציה, כל המשתנים הלוקליים משתחררים באופן אוטומטי.
5. ערך הפונקציה מוחזר ברגיסטר AX.
6. אחרי החזרה, הפונקציה הקוראת אחראית להוציא את הפרמטרים מהמחסנית.

דוגמא

נביט בקוד C הבא:

```
int adder(int a,b)
{
        int z;
        z = a + b;
        return z;
}


int main()
{
        int a,b,c;
        c = adder(a,b);
        ...
        return 0;
}
```

קוד זה יהפוך לקוד האסמבלי הבא (בקירוב):

התוכנית הראשית:

```
...
PUSH b
PUSH a
CALL adder
ADD  SP,4

(result is in AX)

...
```

הפונקציה הנקראית:

```
adder: PUSH BP
       MOV  BP,SP
       SUB  SP,2
       MOV  AX, [BP+4]
       ADD  AX, [BP+6]
       MOV  [BP-2], AX
       MOV  SP,BP
       POP  BP
       RET
```

נשים לב לכך שבדוגמא זו לא נשמרו ערכי הרגיסטרים, מלבד BP.
התוכנית הראשית החליטה שאין לה צורך מיוחד לשמור את הרגיסטרים AX, BX, CX, DX, ולכן לא דחפה אותם למחסנית. יש לציין שזהו בדרך כלל המקרה, ונדיר יחסית לראות מצבים ששומרים רגיסטרים אלו.

כמו כן, הפונקציה לא שמרה את SI, DI, FLAGS, מכיוון שהיא איננה עושה בהם שימוש.
נשים לב שמייד עם הכניסה לפונקציה, אנו עושים PUSH ל-BP, ומעדכנים את ערכו של BP.

נביט בתוכן המחסנית כדי להבין את הנעשה כאן.

...	---
[SP-2]	new top of stack
[BP-2]	space for z
[BP]	old BP
[BP+2]	return address
[BP+4]	value of a
[BP+6]	value of b
...	---

BP הוא מצביע אל תחילת ה-frame של הפונקציה.
נשים לב שבכתובת [BP], נשמר ה-BP הקודם.
באופן כזה, אנו מסוגלים לשחזר את ה-BP הקודם, כאשר אנו יוצאים מהפונקציה.
כפי שכבר ציינו – נוצרת מעין רשימה מקושרת של ה-BP, כשכל אחד מצביע לקודמו.

הפרמטר הראשון של הפונקציה מצוי בכתובת [BP+4], הפרמטר השני בכתובת [BP+6] וכך הלאה.

המשתנה המקומי של הפונקציה, z, מצוי בכתובת [BP-2].

EOF