PDP-11

Structure, Design, Assembler

By Nir Adar

PDP11 - Struct, Design & Assembler

מהדורה ראשונה

חוברת זו נכתבה בהתאם לתוכנית הלימוד של הקורס "ארגון ותכנות המחשב" בטכניון. זו איננה חוברת רשמית של הטכניון אלא חוברת פרטית שנכתבה על ידי **ניר אדר**.

המקורות לחוברת זו הן הרצאות הוידאו של **שולי וינטנר** וכן עבודה עצמאית שבוצעה על ידי **ניר אדר**.

אנא שלחו תיקונים והערות אל המחבר.

תודות

איל רוטמן, הדס זמיר, רותם גרוסמן - על שתרמו מזמנם להגהת החוברת ובדיקתה.

ניר אדר
דצמבר 2002

הקדמה

יצוג מספרים בבסיסים שונים

יצוג מספרים

ייצוג מספרים בבסיס m נעשה ע"י הספרות 0..(m-1), כשהבסיס גדול מ-10 משתמשים באותיות ABC...
הערך המספרי הוא:

$$\sum_{i=0}^{n-1} d_i b^i = (d_{n-1}...d_0)_b$$

מעבר מבסיס לבסיס

שיטה 1: סכום וכפל
השיטה מומלצת כשבסיס המטרה הוא 10.

$$(\sum_{i=0}^{n-1} d_i b^i)_c$$

החישוב הוא בבסיס c.
דוגמא:

$$100 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 4$$

שיטה 2: שיטת החלוקה
השיטה מומלצת כשבסיס המקור הוא 10.
נתונה $(d_{n-1}...d_0) \leftarrow (...)_c$.

מחלקים את מחרוזת הייצוג בערך של בסיס המטרה המיוצג לפי בסיס המקור.
השארית תהווה את ספרותיה של התוצאה.
דוגמא:
נעביר את המספר העשרוני 167 לבסיס אוקטלי:

167_{10}	=	247_8
20	7	7
2	4	47
0	2	247
0 הוא השלם אחרי חילוק ב8		247 היא המספר בבסיס 8

מעבר מבסיס לבסיס כאשר אחד הוא חזקה של השני:

$$726_8 = 111\ 010\ 110_2$$

כל סיפרה בבסיס 8 הפכה ל3 ספרות בבסיס 2. אם נרצה לעבור מבסיס 2 ל8 פשוט נקבץ את הספרות.

מספרים בעלי סימן

שיטת המשלים ל-2:

שיטה זו היא השיטה איתה עובדים בPDP11.
השיטה: הופכים את הביטים ומחברים 1

```
2Comp(x)
        return (1Comp(x)+1)
```

טווח:

$$-(2^{n-1})...(2^{n-1}-1)$$

דוגמא:

4=0100
-4=1011+1=1100

חיבור וחיסור:

0111 +	7 +
1010 =	-6 =
~~1~~0001	1

התעלמנו מהcarry בחיבור.

1001 +	-7 +
1001 =	-7 =
~~1~~0010	2

גלישה – כאשר מצפים לתוצאה מסימן כלשהו ומקבלים תוצאה מסימן הפוך.

אלגוריתם הבנת מספר:
1. אם הביט השמאלי הוא 0, זהו מספר חיובי.
2. אם הביט השמאלי הוא 1, חסר 1 מהמספר, הפוך את הביטים וזכור שהוא שלילי.

הזיכרון

מבנה הזכרון ב-PDP11

יחידת בית (byte). את הבתים ניתן לקבץ לקבוצות של מילים.
בPDP11 גודל מילה הוא 16 ביטים. הביטים ממוספרים מימין לשמאל מ0 עד 15.
ביטים 0 עד 7 נקראים הבית הנמוך וביטים 8 עד 15 נקראים הבית הגבוה.
מבנה מילה:

high byte								low byte							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

הכתובת הראשונה בזיכרון היא כתובת 000000. הכתובת האחרונה היא 077777.
אנו ממספרים את הכתובות בבסיס אוקטלי.
צורת ההסתכלות על הזיכרון:

	מילה	
	בית	בית
776	High	Low
1000		
1002		
1004		

777
1001
1003
1005

אכסון מחרוזות בזיכרון

נניח שנירצה לאכסן את המחרוזת "ba" בזיכרון.
ערך הASCII של b הוא 142 ושל a 141.
כל אחד מהם תופס בית אחד.

1000	142
1001	141

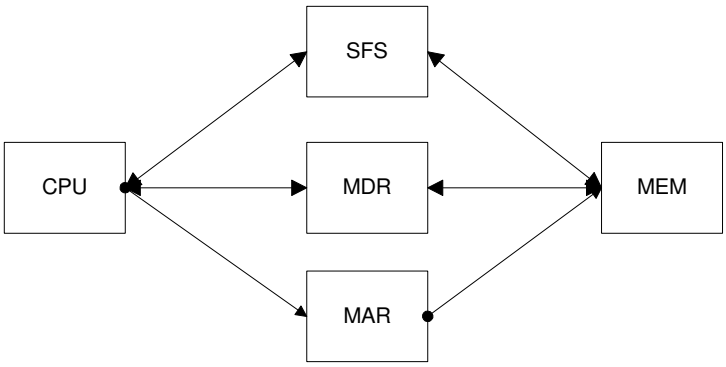
$$142 = 01100010$$
$$141 = 01100001$$

נשים אותם אחד ליד השני:
0110001001100001
נעביר לבסיס אוקטלי:
061141
נשים לב שזהו לא 142141 בבסיס אוקטלי, כי הביט השלישי של הסיפרה האוקטלית השלישית המייצגת את 141 הוא למעשה הביט הראשון של 142.

זיכרון בגישה אקראית – RAM

משמעות המושג זיכרון בגישה אקראית הוא שניתן לגשת לכל כתובת בזיכרון בזמן קבוע.
סוג אחר של זיכרון הוא למשל, זיכרון בגישה סדרתית, שם עלינו לעבור על כל התאים הקודמים לתא מסוים, על מנת לקרוא את תוכנו.

מודל התקשורת עם הזיכרון בPDP11:



Store Fetch Register – SFS
משתנה האומר האם הפעולה המבוקשת היא קריאה מהזיכרון או כתיבה לזיכרון.

Memory Data Register – MDR
פה שם הCPU את התוכן של הכתובת אותו הוא רוצה לשמור ופה שם הזיכרון את התא אותו הוא מחזיר לCPU.

Memory Register Address – MAR
פה נשמרת הכתובת איתה אנחנו עובדים.

ערכי SFS		
ערך	משמעות	שולח/מבצע
00	כתיבה STORE	CPU שולח
01	קריאה FETCH	CPU שולח
10	פקודה בוצעה	MEM מבצע
11	תקלה	MEM מבצע

גודל SFS – 2 ביטים
גודל MAR – גודל כתובת
גודל MDR – רוחב הזיכרון

```
value FETCH(address a) {
        while(SFS!=2);
        MAR=a;
        SFS=1;
        while(SFS!=2);
        return MDR;
}

void STORE(value d, address a) {
        while(SFS!=2);
        MAR=a;
        MDR=d;
        SFS=0;
}
```

הזכרון:

```
while(TRUE) {
        while(SFS==2);
        if (SFS==0) {
                a = MAR;
                MEM[a]=MDR;
                SFS=2;
        }
        if (SFS==1) {
                a = MAR;
                MDR=MEM[a];
                SFS=2;
        }
}
```

מבנה פקודה

מבנה של 1+3 כתובות:

OpCode	Operand1	Operand2	Result	Next Instruction

OpCode – מה לעשות
Operand1, Operand2 – אופרנדים – ערכים, כתובות זיכרון בהן הערכים
Result – תוצאה (כתובת)
Next Instruction – כתובת הפקודה הבאה

יתרון: כל פקודה מכילה את כל המידע הדרוש.
חסרון: הפקודה נהיית מאוד רחבה

מבנה פקודה של 3 כתובות (ללא שדה הפקודה הבאה):

OpCode	Operand1	Operand2	Result

יתרון:
1. חיסכון מקום בזיכרון

חסרונות:
1. לא ניתן לפזר את הפקודות בזכרון
2. צריך לכתוב מנגנון שקובע מה הפקודה הבאה לביצוע
   פתרון: רגיסטר Program Counter (PC).

מבנה פקודה של 2 כתובות:

OpCode	Operand1	Operand2

אחד האופרנדים ישמש גם לקלט וגם לפלט.

מבנה פקודה של כתובת אחת:

שימוש בצובר – Accumulator
הצובר הוא רגיסטר, שהוא משתנה של המכונה.
הרעיון של מבני פקודה של כתובת אחת הוא שישנו OpCode, כתובת אחת, והכתובת השניה נאגרת בצובר.

שיטות מיעון בPDP-11

ב-PDP11 קיימות פקודות המקבלות 0 אופרנדים, אופרנד יחיד או שני אופרנדים.
האופרנד לרוב מקודד בתור 6 ביטים במילת הפקודה.
למשל, פקודות של שני אופרנדים ב-PDP11 יראו לרוב כך:

0-3	4-9	10-15
OpCode	Operand 1	Operand 2

ב6 הביטים המייצגים אופרנד:
השלושה השמאליים מציינים מצב (mode) והשלושה הימניים מציינים רגיסטר.
רגיסטר הוא משתנה פנימי של המעבד.

האופרנד עליו מתבצעת הפעולה נקרא אופרנד האקטואלי.
הכתובת של האופרנד האקטואלי נקראת כתובת אפקטיבית.

Mode 0 – רגיסטר ישיר

0	Register

בRegister נשמר על איזה רגיסטר תתבצע הפעולה.
בשיטה זו, הארגומנט נמצא ברגיסטר.

דוגמא:

```
mov r1, r6
```

הסבר: העתק תוכנו של r1 לr6.
ובשפת מכונה:

```
010106
```

בכתיבה זו, הספרה האוקטלית השמאלית היא תמיד ביט אחד.

בשיטת מיעון זו אין כתובת אפקטיבית. האופרנד האקטואלי נמצא ברגיסטר.

Mode 1 – רגיסטר עקיף

תוכן הרגיסטר הוא הכתובת בה מאוכסן האופרנד.
צורת כתיבה:

```
mov (r1), r6
mov @r1, r6
```

ובשפת מכונה:

```
011106
```

בשיטת מיעון 1, הכתובת האפקטיבית היא הרגיסטר.

```
EA=Rn
```

הערה – רק בשיטת מיעון Mode 1 ניתן להשתמש ב@ במקום בסוגריים.

Mode 2 – הגדלה אוטומטית ישיר

שיטה זו זהה לMode 1 למעט side effect – לאחר חישוב הכתובת האפקטיבית, גדל הרגיסטר ב-2.
צורת כתיבה:

```
mov (r1)+, r5
```

ובשפת מכונה:

```
012105

EA=r1
r1+=2
```

שימוש לדוגמא: מעבר על מערך

Mode 4 – הקטנה אוטומטית ישיר

השיטה דומה לMode 1 אלא שהרגיסטר מוקטן ב2 לפני חישוב הכתובת האפקטיבית.

```
mov –(r1), r5
014105

Rn-=2
EA=Rn
```

בשיטת מיעון 2 ו-4 האוגר יקודם/יוסג ב1 ולא ב2 אם:
하나. הרגיסטר אינו 6 או 7
둘. הפקודה פועלת על בתים.

Mode 3 – הגדלה אוטומטית עקיף

כמו שיטה 2, עם רמה נוספת של עקיפות.
צורת כתיבה:

```
mov @(r1)+, r5
013105

EA=fetch(Rn)
Rn+=2
```

בכתובת המאוכסנת ברגיסטר מאוכסנת הכתובת בה נמצא האיבר המבוקש.
במילים: קח את התוכן של Rn, למשל 1002, לך לכתובת 1002, מצא את הערך, למשל 3006, שם נמצא האופרנד.
שימוש – לעבור על מערך המכיל כתובות של משתנים.

Mode 5 – הקטנה אוטומטית עקיף

```
mov @-(r1), r5
015105
```

Rn-=2
EA=fetch(Rn)
בModes 3, 5 הקידום/ההקטנה הם תמיד ב2! (כי אלו מצביעים לכתובות וכתובות הן זוגות של בתים).

Mode 2, Register 7 – מיעון מיידי

```
mov #1000, r1
012705
1000
```

הPC בתחילת מחזור פקודה מצביע אל הפקודה הבאה לביצוע. מיד לאחר שליפת מילת הפקודה גדל הPC ב-2. רק לאחר מכן ממשיך מחזור ביצוע הפקודה: חישוב האופרנד, פענוח האופרנדים, ביצוע פקודה.

Mode 3, Register 7 – מיעון מוחלט

```
mov @#1000, r1
013701
1000
```

בסוף הפעולה ישב ברגיסטר r1 התוכן של כתובת 1000.

דוגמא:

```
mov #1000, @#2000
```

```
012737
1000
2000
```

הערך 1000 יושם בכתובת 2000.

Mode 6, index

צורת כתיבה:

```
mov 4(r1), r5
```

בשיטה זו תמיד יש צורך במילה נוספת עבור האופרנד.

```
016105
4
```

הכתובת האפקטיבית היא תוכן הרגיסטר הבסיס + אינדקס (זהו המספר לפני האוגר).

X ← fetch(PC)
PC+=2
EA=Rn + X

באמצעות שיטה זו אפשר לגשת למקום ה i במערך. או לאיבר במבנה.

Mode 7

זהה ל6, מלבד שצריך לעשות fetch לכתובת EA של Mode 6.

```
mov @x(Rn),
x=fetch(PC)
PC+=2
EA=fetch(Rn+x)
```

מיעון יחסי, Mode 6

```
l:      mov A, r1

.
.
.

A:      .word 5
```

נרצה שיוצב ב-R1 הערך 5

l: 01 67 01
X
...
...
...
A: 5

EA= Rn+X

במקרה שלנו:

EA=R7+X

כאשר מתבצעת הפקודה, R7 הוא L+4

EA=L+4+X
EA=A=L+4+X
X=A-L-4

במיעון יחסי מקודד בתא אחרי L, המרחק "היחסי" בין L לבין התא המבוקש.

תרגום של:

```
mov @#A, whatever
```

העבר את תוכן (@) הכתובת (#) A לwhatever.

```
mov @#1000, r1
```

זהה ל

```
mov 1000, r1
```

אבל לא ל

```
mov #1000, r1
```

ההבדל בין שיטה אבסולוטית ליחסית הוא ששיטת מיעון יחסית מאפשרת לעבוד בלי לדעת איפה ממוקמת התוכנית בזיכרון.

מיעון יחסי עקיף, Mode 7

l: mov @a, r1

נלך לכתובת של a, נסתכל על תוכן הכתובת a, שם נמצאת הכתובת האפקטיבית.

<u>הערה</u>

mov @#L, r1

העבר את <u>תוכן L</u> אל R1.

mov r1, @#L

העבר את תוכן R1 אל כתובת L.

יש הבדל בין חישוב SRC וDEST.

דוגמא:

```
. = torg + 2000

main:
        mov #66, r1
        mov r1, #L
        mov #M, r2
        halt

. = torg + 3000
L: .word 4
M: .word 7
```

השורה המודגשת אומרת דבר שלא התכוונו אליו.
L# זוהי שיטת מיעון 2, רגיסטר 7. הפקודה mov שומרת את הsource בכתובת האפקטיבית של הdestination.
הכתובת האפקטיבית במקרה זה היא 2006 (כתובת הפקודה mov r1, #l הוא 2004). במקרה זה הפקודה למעשה משנה את עצמה. בתא 2006 יהיה בסוף הפקודה הערך 66, ולא ב-L.
על מנת לשים ערך ב-L, היינו צריכים לשים את השורה

mov r1, L

במקום השורה המודגשת.

ברגיסטר r2 תשב בסוף התוכנית הכתובת של M. (3002)

מחזור פקודה בPDP11

מחזור הפקודה הוא התהליך המתבצע על מנת לבצע פקודה אחת. תהליך זה ב-PDP11 הוא אטומי. אנו מניחים שהוא מתרחש מיידית כאשר מתבצעת פקודה, וכן שפקודה אחרת לא יכולה להתחיל להתבצע כל עוד לא הסתיים מחזור הפקודה של הפקודה הקודמת.

נציג את מחזור הפקודה בצורה דמויות C.

```
Fetch
        IR = FETCH(PC)                  /* Fetch first instruction word */
        mode ≡ IR[11:9];
        i ≡ IR[8:6];
        PC += 2;                        /* Increment PC */

Decode Source Operand
        switch (mode) {
        case 0, 1:                      /* Register Addressing */
                Rs = Ri;
                break;
        case 2, 3:                      /* Auto Increment Addressing */
                Rs = FETCH(Ri);         /* Memory fetch */
                if ((Word Instruction) ||   /* Case of Increment by 2 */
                  (i ∈ [6,7]) || (mode == 3))
                        Ri += 2;
                else Ri += 1;
                break;
        case 4, 5:                      /* Auto Decrement Addressing */
                if ((Word Instruction) ||   /* Case of Decrement by 2 */
                  (i ∈ [6,7]) || (mode == 5))
                        Ri -= 2;
                else Ri -= 1;
                break;
        case 6, 7:                      /* Indexed addressing */
                SCRATCH = FETCH(PC);
                PC += 2;                /* Increment PC */
                Rs = FETCH(SCRATCH+ Ri);    /* Memory Fetch */
                break;
}

if (mode ∈ [1, 3, 5, 7])                /* Deferred addressing */
        Rs = FETCH(Rs);                 /* Memory fetch */
```

```
Decode Destination Operand
        mode ≡ IR[11:9];
        i ≡ IR[8:6];
        switch (mode) {
                case 0:                         /*Register Direct Addressing*/
                        SCRATCH = Rj;
                        break;
                case 1:                         /* Register deferred addressing */
                        Rd = Rj;
                        break;
                case 2, 3:                      /* Autoincrement addressing */
                        Rd = Rj;                        /* Memory fetch */
                        if ((Word Instruction) ||       /* Case of Increment by 2 */
                  (i ∈ [6,7]) || (mode == 3))
                        Rj += 2;
                        else Rj += 1;
                        break;
                case 4, 5:
                        if ((Word Instruction) ||       /* Case of decrement by 2 */
                          (i ∈ [6,7]) || (mode == 5))
                                Rj -= 2;
                        else Rj -= 1;
                        Rd = Rj;
                        break;
                case 6, 7:                              /* Indexed addressing */
                        SCRATCH = FETCH(PC);
                        PC += 2;                        /* Increment PC */
                        Rd = SCRATCH + Rj;      /* Memory fetch */
                        break;
        }
        if (mode ∈ [3, 5, 7])                   /* Deferred addressing */
                Rd = FETCH(Rd);                 /* Memory fetch */

        If (mode ≠ 0)                           /* addressing ≠ register direct */
                SCARTCH = FETCH(Rd)

/* IR[15:12] determines OP */
Execute
        SCARTCH = OP(Rs, SCARTCH);
Store
        if (mode ≠ 0)
                STORE(SCRATCH, Rd);  /* Store result in memory */
        else
                Rj = SCRATCH;                   /* Destination is register */
```

עבודה עם הסימולטור והאסמבלר

בקורס אתי"מ לא נעבוד מול מחשבי PDP-11, אלא נעבוד מול תוכנה, סימולטור, המדמה את מחשב הPDP-11.

הסימולטור איתו נעבוד נכתב על ידי איתן אביאור, מהפקולטה למדעי המחשב בטכניון.

נתמקד בעבודה מול גירסה 2.2 של הסימולטור. למרות שישנה גירסה חדשה יותר, עקב באגים שונים היא איננה מתאימה לכתיבת תוכניות גדולות. על מנת לחסוך את הצורך ללמוד לעבוד עם שני סימולטורים, נעבוד מול הסימולטור הישן יותר.

לאחר שנכתוב קוד באסמבלר, נצטרך להעביר אותו קומפילציה, ואז להריצו על הסימולטור.
ניתן לכתוב את קוד המקור של הקבצים בפנקס הרשימות (NotePad) או בכל עורך טקסט אחר. סיומת קבצי המקור שלנו היא s11. נזכור לשמור את קבצי המקור שלנו עם סיומת זו.

את הקומפילציה וההרצה נעשה מתוך Dos Shell בWindows.

על מנת לקמפל את הקוד, נשתמש באסמלבר שמסופק בחבילה התוכנה. שם קובץ ההפעלה של האסמבלר הוא as11.exe.

<u>דוגמא</u>
נניח שבספרייה של האסמבלר קיים קובץ בשם hello.s11 אותו נרצה לקמפל, אזיי נכתוב את השורה הבאה:

```
as11 hello.s11
```

בהנחה שתהליך ההידור יעבוד בלי בעיות, נקבל שני קבצי פלט:

```
hello.lst
hello.o11
```

hello.lst מכיל את קוד התוכנית שכתבנו, ולצידו מופיעה ההמרה לשפת מכונה של הקוד, וכן הערות ושגיאות שהאסמבלר מצא בזמן עבודתו.
hello.o11 הוא קובץ הפלט להרצה.

לאחר שיש בידינו תוכנית בשפת מכונה, נפעיל את הסימולטור על מנת לראות את פעולתה.
קובץ ההפעלה של הסימולטור הוא sim11.exe.

בהמשך לדוגמא שהתחלנו, אם נרצה כעת להפעיל את הסימולטור עם קובץ הקלט hello.o11, נכתוב בשורת ההפעלה

```
sim11 hello.o11
```

הסימולטור יעלה ונוכל להריץ את התוכנית שכתבנו.

על מנת לבצע פקודות שונות בסימולטור, נכתוב את הפקודה הרצויה בשורת הפקודה, ונלחץ Enter.

נביט בטבלה המרכזת פקודות חשובות של הסימולטור:

הפעולה	הפקודה
הצגת תוכן הרגיסטרים על המסך.	r
הצגת תוכן הזיכרון על המסך. הטווח שיוצג הוא כל תחום הזיכרון בו התוכנית עושה שימוש	e
הצגת תוכן מילת הזיכרון המצויה בכתובת number	e <number>
הצגת תוכן הזיכרון בין הכתובת number1 לכתובת number2	e <number1, number2>
התחל לרוץ מכתובת התווית main עד סיום התוכנית	g
התחל לרוץ מהכתובת number עד סיום התוכנית	g number
הרץ פקודה אחת ועצור	s
הכנס breakpoint בכתובת number	b number
הכנס breakpoint בכתובת עם התווית label	b label
המשך לרוץ (לאחר breakpoint)	c
ריצה במצב "מעקב". הפקודות יבוצעו אחת אחרי השנייה, ומצב הרגיסטרים יודפס לאחר ביצוע כל פקודה	t
ביטול breakpoint בכתובת number	d number

אם נרצה לשמור חלק מהפקודות שהקשנו ואת הפלט שלהן לתוך קובץ, נשתמש בפקודה +L על מנת להתחיל לשמור את הנתונים לקובץ. כאשר נרצה להפסיק את רישום הפעולות, נשתמש בפקודה -L.
על מנת להציג גם את גישות לזיכרון שמתבצעות במהלך ביצוע כל פקודה, נשתמש בפקודה +M. על מנת להפסיק להציג את הגישות לזיכרון, נשתמש בפקודה -M.

הסתעפויות וחוגים

מבני ההסתעפויות והחוגים המוכרים משפות עיליות, כגון לולאות, if ,do while וכו' אינם קיימים בPDP11 אולם הם ניתנים למימוש באמצעות אבני הבניין שמספקת לנו המכונה.

הסתעפויות מותנות

קפיצות מותנות מורכבת משני שלבים:
1. חשב את התנאי של הקפיצה.
2. קפוץ על סמך תוצאת הפעולה הקודמת.

תוצאת חישוב התנאי נשמרת באוגר מיוחד. (PSW).
בPDP11 באוגר PSW ארבעה דגלי בקרה:

NZVC

N – Negative – שווה 1 אמ"מ תוצאת החישוב הקודם היא שלילית.
Z – Zero – שווה 1 אמ"מ תוצאת החישוב היא 0.
V – Overflow.
C – Carry

גלישה

מקור הבעיה הוא שהחשבונות שכל מכונה עושה נכונים בדיוק מסוים.
אם כתוצאה מחישוב נשאר נשא מהMSB (הביט השמאלי ביותר) פירושו – רוחב המילה אינו מספיק לתוצאה.

חיבור
במספרים UNSIGNED כאשר יש CARRY התוצאה אינה נכונה.
במספרים SIGNED יתכן מצב שאין CARRY והתוצאה אינה נכונה.
דגל V נדלק אמ"מ שני המחוברים בעלי MSB זהה והתוצאה בעלת MSB הפוך.

נסכם: דגל C נדלק אמ"מ חלה טעות בחישוב כאשר האופרנדים ללא סימן. דגל V נדלק אמ"מ חלה טעות בחישוב כאשר המספרים בעלי סימן.

חיסור
החוק שנתנו לחיבור תקף: דגל C נדלק אמ"מ חלה טעות בחישוב כאשר האופרנדים ללא סימן. דגל V נדלק אמ"מ חלה טעות בחישוב כאשר המספרים בעלי סימן, כלומר, הsource והdestination בעלי סימנים שונים, והתוצאה בעלת אותו סימן כמו אופרנד המקור.

השוני:
דגל C נדלק אמ"מ אין נשא. אין שינוי לגבי דגל V.
הסתעפויות

cmp A, B

cmp מבצעת (A-B). ההפרש בין A לB לא נשמר בשום מקום, אולם הדגלים משתנים. הפקודה beq L היא קפיצה אמ"מ Z=1.

פקודה	אפקט
BEQ L	קפוץ אם Z=1
BNE L	קפוץ אם Z=0
BMI L	קפוץ אם N=1
BPL L	קפוץ אם N=0
פקודות פחות שימושיות	
BCC	קפוץ אם C=0
BCS	קפוץ אם C=1
BVC	קפוץ אם V=0
BVS	קפוץ אם V=1

פקודות הסתעפות מותנית למספרים בעלי סימן:

פקודה	אפקט
BLT L	קפוץ אם B גדול מA (N && !V) || (V && !N)
BLE L	קפוץ אם B גדול שווה A ((N && !V) || (V && !N)) || Z
BGT L	קפוץ אם A גדול מB !BLE
BGE L	קפוץ אם A גדול שווה מB (V && N) || (!V && !N)

פקודות הסתעפות מותנית למספרים חסרי סימן:

פקודה	אפקט
BHI L	A>B !(C||Z)
BLOS L	A<=B C||Z
BLO L	A<B C
BHIS L	A>=B !C

קפיצה לא מותנית מתבצעת על ידי

br L

<u>מיעונים בפקודות הסתעפות</u>

לפקודות הBranch שיטת מיעון משלהן:

OPCODE	WO
8-15	0-7

WO – Word Offset – מספר **המילים** שיש לקפוץ קדימה מערכו הנוכחי של הPC בזמן ביצוע הפקודה.
הנחה 1: המרחק אליו קופצים קצר יחסית.
הנחה 2: קופצים מספר זוגי של בתים.

WO הוא Signed.

"לולאת FOR"

SOB Rn, L

הפחת 1 וקפוץ
הארגומנט הראשון של SOB הוא רגיסטר בלבד. בארגומנט השני ניתן להשתמש בכל אחת משיטות המיעון הרגילות.

077	Rn	Offset
9-15	6-8	0-5

פעולת SOB:
הפחת את Rn ב1
אם Rn!=0 קפוץ לL.

את SOB כותבים בסוף הלולאה. OFFSET הוא מספר חיובי UNSIGNED בין 0 ל63 המציין את מספר **המילים** שיש לקפוץ **אחורה**.
SOB משמשת לקפיצה אחורה בלבד!!!

פקודה לקפיצות רחוקות

jmp D
0001DD

- ניתן להשתמש בכמעט כל שיטות המיעון (מלבד Mode 0)
- טווח הקפיצה לא מוגבל.
- בד"כ דרושה מילה נוספת (אם משתמשים בשיטת המיעון 6).

הקפיצה היא לכתובת האפקטיבית ולא לאופרנד האקטואלי.

jmp L

האם קופצים לL או לכתובת הרשומה בL? אנו נרצה לקפוץ לכתובת L.

כנ"ל אם נכתוב:

```
jmp @#L
```

הפקודה הנ"ל היא שגיאה, יש להימנע מצורת הכתיבה הבאה:

```
jmp #L
```

נדגים למה בד"כ הפקודה היא "שגיאה" בעזרת התוכנית הבאה:

```
. = torg + 2000

main:
        mov #4, r1
        jmp #010102
        halt


; r2 = r4
```

התוכנית מתפרשת בזכרון כ:

```
012701  000004
000127  010102
000000
```

הפקודה המודגשת היא ה-jmp. בשיטת מיעון 2,7 הכתובת האפקטיבית היא ה-PC, כלומר הכתובת של המילה הבאה (010102). לכתובת זו נקפוץ.
לאחר הפקודה jmp התבצעה המילה 010102# כפקודה, כלומר:

```
mov r1, r2
```

בסופו של דבר, r2 יכיל 4.

דוגמא להסתעפויות ולשיטות מיעון:
התוכנית הבאה מסכמת את אברי המערך array לתוך r1. אנו משתמשים בr0 כמשתנה עזר המצביע על הoffset מתחילת המערך במהלך הלולאה.

```
. = torg + 2000

main:
        mov pc, sp                      ;
        sub #2, sp                      ; Init stack

        clr r0
        clr r1
```

```
loop:
        add array(r0), r1     ; add each element of array to r1
        add #2, r0
        cmp r0, #20.
        blt loop

        halt
array:
        .word 1, 2, 3, 4, 5, 6, 7, 10, 11, 12
```

בסוף התוכנית,

r0 = 24 (20_{10})

r1 = 67 (55_{10})

הנחיות לאסמבלר

הקצאת n מילים בזיכרון (ללא אתחול):

.blkw n

הקצאת n מילים בזיכרון ואיתחולן:

.word W1, W2, … , Wn

הקצאת n בתים בזיכרון ואיתחולן:

.byte B1, B2, … , Bn

מספר יחשב אוקטלי כברירת מחדל. מספר ייחשב עשרוני אם הוא מסתיים בנקודה עשרונית.
תו שלפניו גרש מסמל את ערך הASCII של התו.
הקצאת מחרוזת:

.ascii <str>

השמת כתובת a אל LC (Location Counter):

. = torg + a

קידום LC לכתובת זוגית:

.even

asl / asr

Arithmetic Shift Left / Arithmetic Shift Right

Syntax:
asl operand

הפקודה מזיזה את הביטים של האופרנד שמאלה בביט 1. הביט השמאלי ביותר נכנס לcarry ובביט הימני ביותר נכנס 0.

asr operand

ההזזה מתבצעת ימינה. הביט הימני יוצא לcarry, הביט השמאלי משוכפל.

mul / div

mul

אם שני מספרים בגודל n, תוצאת מכפלתם היא עד 2n.
אם n הוא 16bits, התוצאה היא 32bits
התוצאה אם כך תכנס ל2 רגיסטרים.

mul src, Ri

אם i זוגי,

src * Ri → (Ri : R(i+1))

אם i אי זוגי,

src * Ri → Ri

הLSB הוא התא השמאלי.

דוגמא:

```
mov #2, r1
mov #4, r3
mul r1, r3     ; r3 is now 10
```

דוגמא נוספת:

```
mov #10., r1
mov #40., r3
mul r1, r2     ; r3 = 400          r2 = 0
```

div

div src, Ri

אם i זוגי, מתבצע:

$(R_i : R_{i+1}) / src = R_i$

$(R_i : R_{i+1}) \% src = R_{i+1}$

אם i אי זוגי, הפקודה אינה מבצעת דבר.
דוגמא:

```
mov #2., r1
mov #16., r3
div r1, r2      ; r2 = r3 / r1
```

בנספח ג' ניתן למצוא דוגמא לבעיה העלולה להתעורר כאשר מנסים לחלק מספר גדול.
הערה חשובה: אופרנד היעד חייב להיות רגיסטר!

חיבור ב32 ביט

ניקח 4 מילים –

A	A1	A0
B	B1	B0

נניח שA וB מספרים חיוביים.
מה שנרצה:

B ← A+B

```
add a0, b0
adc b1
add a1, b1
```

Add Carry – adc
3 פקודות אלו מבצעות את המטלה המבוקשת.

אם המספר היא signed יש טעות בחישוב.

חיסור ב32 ביט

B ← A – B

```
sub b1, a1
sub b0, a0
sbc b1
```

Subtract Carry – sbc

גם במקרה זה, אם המספר הוא signed יש טעות בחישוב.

שגרות

על מנת שמכונה כלשהי תתמוך בשגרות, היא צריכה לתמוך בשני מנגנונים:
1. מנגנון להעברת בקרה:
   - קריאה לשגרות.
   - חזרה משגרות.
2. מנגנון להעברת פרמטרים.

העברת בקרה

הפתרון הממומש צריך לכלול פקודות call וreturn עבור השגרות.
פתרון נאיבי יכול להיות שימוש ברגיסטר מיוחד שישמור את כתובת החזרה.
פתרון כזה לא טוב כי:
1. השיגרה עלולה לשנות את מצב המכונה. מנגנון הקריאה והחזרה צריך לשמור את ערכי הרגיסטרים ולשחזר אותם.
2. לא ניתן לקנן שגרות (עד אין סוף), מכיוון שיש מספר מוגבל של רגיסטרים.

תמיכה במנגנון תמיכה בשגרות ללא חסם עליון ניתנת על ידי מחסנית.
בPDP11 אין מחסנית מתוכננת. על המשתמש לבנות מחסנית. לעומת זאת, יש מספר הנחות, איך מחסנית צריכה להיות ממומשת.
1. SP=6
2. המחסנית גדלה אל עבר הכתובות הנמוכות.

בעזרת ההנחות האלו ניתן לממש מחסנית בצורה הבאה:
את הפקודה push נממש כך:

```
mov x, -(sp)
```

את הפקודה pop נממש בצורה הבאה:

```
mov (sp)+, x
```

נהוג למקם את המחסנית מתחת התוכנית.
שתי שיטות לאתחול:
1.

```
main: mov #main, sp
```

2.

```
main: mov pc, sp
      tst –(sp)
```

השיטה השנייה עדיפה כי אין בה שימוש בשיטת מיעון אבסולוטית.

נדגים תוכנית קטנה, אשר מחליפה בין ערכי רגיסטרים בעזרת מחסנית:

```
. = torg + 2000
```

```
main:
        mov pc, sp                      ;
        tst -(sp)                       ; Init stack

        mov #10., r0
        mov #7, r1

        mov r0, -(sp)                   ; Push r0
        mov r1, r0
        mov (sp)+, r1                   ; Pop value to r1
        halt
```

בתחילה בr0 הצבנו 10 וב7. r1. לאחר מכן דחפנו את r0 למחסנית, שמנו ברגיסטר r0 את תוכן r1 ואז שמנו ברגיסטר r1 את האיבר ששלפנו מהמחסנית (העותק המקורי של r0).

מנגנון העברת בקרה לשגרות

קפיצה לשיגרה

jsr Rn, SubR

שיטת המיעון של jsr זהה לזו של jmp.
ל-Rn נקרא רגיסטר הקישור.

פעולת jsr:

push Rn
Rn ← PC
PC ← SubR

הערך של PC נשמר ברגיסטר הקישור ולא במחסנית.
jsr מניחה שR6 הוא SP. R6 אינו יכול להיות מצביע הקישור מכיוון שהוא מצביע המחסנית, והמכונה משנה אותו במהלך jsr.
באופן מעשי, הpush מתבצע בסוף מחזור הפקודה. הצגנו אותו פה בהתחלה מכיוון שכך קל יותר להבין את משמעות פעולת jsr.
הפעולה push X היא: הזז את sp צעד אחד אחורה ואז הכנס את האיבר X.

שימוש בPC כרגיסטר קישור זהו השימוש הטיבעי בjsr. נקבל את האפקט הבא:

push PC
PC ← SubR

חזרה משיגרה

rts Rn

Rn הוא רגיסטר הקישור.

PC ← Rn
Rn ← pop

אם Rn = PC אזי מתבצע:

R7 ← pop

ראש המחסנית מכילה במקרה זה את כתובת החזרה.
הפעולה pop X היא: שולפים נתון, ואז מזיזים את sp קדימה.
ניתן להשתמש באותו רגיסטר קישור לקריאות שונות מכוננות, אולם רגיסטר הקריאה ורגיסטר החזרה חייבים להיות זהים.

העברת פרמטרים לשגרות

- לפי ערך by value
- לפי כתובת by address

בשיטה הראשונה הפרמטר שמועבר לשיגרה הוא הפרמטר עצמו.
העברת פרמטר לפי כתובת, כלומר, מעבירים את הפרמטר בעזרת הכתובת שלו, וכאשר הפונקציה רוצה לפנות לפרמטר היא פונה לכתובת.

פרמטר אקטואלי

```
x = sin(y);
```

y הוא הפרמטר האקטואלי.

```
y = sin(0.8);
```

0.8 הוא הפרמטר האקטואלי.

```
float sin(float x)
{
```

x בתוך השיגרה נקרא הפרמטר הפורמלי.

```
}
```

מיקום הפרמטר בזכרון
ב-pdp11 יש 4 שיטות שונות למקם את הפרמטרים האקטואלים כך שהשיגרה תוכל להשתמש בהם.
לא חובה לקבל ולהחזיר את הפרמטרים באותה שיטה.

רגיסטרים

הפרמטרים יוצבו ברגיסטרים.
השיגרה מקבלת פרמטר ברגיסטר מסוים ומחזירה ברגיסטר כלשהו.
שיגרה טובה היא שיגרה שלא משנה את ערך הרגיסטרים (מלבד רגיסטר החזרה אם התוצאה מוחזרת ברגיסטר). אולם, קיים צורך לשנות רגיסטרים, לכן שיגרה תעשה push לרגיסטרים ו-pop רגע לפני החזרה לשיגרה הקוראת.

יתרונות השיטה:

- פשטות
- יעילות

חסרונות השיטה:

- לא ניתן להעביר הרבה פרמטרים
- תופס רגיסטרים

Inline

הפרמטרים יושבים בגוף התוכנית הקוראת מיד לאחר הקריאה.

אדגים בעזרת תוכנית המחלקת מספר ב10 :

```
main:
        jsr r5, rem10
        .word 35.
result:.word 0        ; here will be the result
        ...
        ...
        halt
rem10:
        mov r0, -(sp)
        mov r1, -(sp)

        mov (r5)+, r1           ; get parameter into r1
        sxt r0
        div #10., r0
        mov r1, (r5)+           ; Return value
        mov (sp)+, r1
        mov (sp)+, r0
        rts r5
```

כתובת החזרה נשמרת ברגיסטר הקישור ולכן ניתן להשתמש בה.

יתרונות השיטה :

- הפרמטרים צמודים לקריאה.
- חיסכון בגישות למחסנית.

חסרונות השיטה :

- עירוב בין תוכנית ונתונים.
- בזבוז רגיסטר.

אם נרצה להעביר את הפרמטר by address, תיראה התוכנית כך :

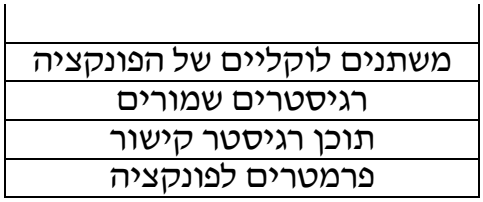
```
main:
        jsr r5, rem10
        .word x
result:.word 0        ; here will be the result
        ...
        ...
x: .word 35.

rem10:
        mov r0, -(sp)
        mov r1, -(sp)
```

```
        mov @(r5)+, r1          ; get parameter into r1
        sxt r0
        div #10., r0
        mov r1, (r5)+           ; Return value
        mov (sp)+, r1
        mov (sp)+, r0
        rts r5
```

העברת פרמטרים על ידי מחסנית

מצב המחסנית בזמן עבודה עם פונקציה שהועברו לה פרמטרים על ידי שימוש במחסנית:



דוגמא:

```
main:
        mov x, -(sp)
        jsr pc, rem10
        mov (sp)+, y
        ...
        ...
x: .word 35.
y: .word 0

rem10:
        mov r0, -(sp)
        mov r1, -(sp)
        mov 6(sp), r1           ; get parameter into r1
        sxt r0
        div #10., r0
        mov r1, 6(sp)           ; Return value
        mov (sp)+, r1
        mov (sp)+, r0
        rts r5
```

אנו יכולים לדרוס את הפרמטר כי זהו העותק שנשמר במחסנית.

הערות:

- זוהי אחריותנו כמתכנתים שהשיגרה לא תשנה את המחסנית.

- זוהי אחריות הסביבה הקוראת לעשות שחרור למשתנים (אם למשל קיבלנו 3 פרמטרים והחזרנו 1).

יתרונות השיטה:
- מודולריות.
- תמיכה ברקורסיה.

חסרונות השיטה:
- איטית ביחס לעבודה עם רגיסטרים.

העברת משתנים בשטח גלובלי

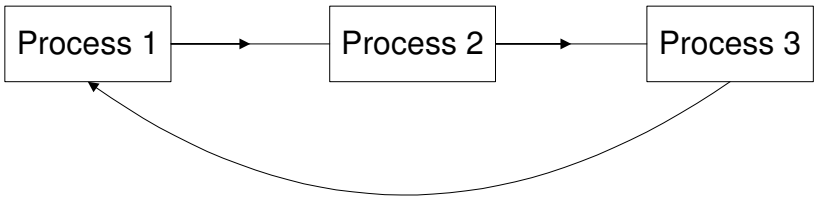
מסכמים שפרמטר השיגרה יהיה תמיד בכתובת x והתוצאה בy.
אין למעשה העברת פרמטרים.

יתרונות השיטה:
- הכי פחות פעולות.

חסרונות השיטה:
- כל החסרונות של משתנים גלובליים.

<u>דרך עבודה מומלצת</u>
פרמטרים: קלט במחסנית, פלט ברגיסטר.
אחריות הסביבה הקוראת לשלוף את הפרמטרים.
אחריות השיגרה לא לשנות את המחסנית.

נהלים מצוותים / co routines



התהליך הראשון עובד, בשלב מסויים הוא קורא לשני. וככה ממשיכה להעבור הזרימה בין תהליך לתהליך.

נרצה לממש דרך בה נוכל לקפוץ לביצוע תהליך אחר, ולחזור לאותה נקודה בקוד מייד לאחר סיומו.

נעשה זאת על ידי:

jsr pc, @(sp)+

האלגוריתם שאנו ממשים הוא: "שלוף מראש המחסנית את הכתובת לקפוץ אליה, הכנס לראש המחסנית את כתובת הפקודה הבאה, וקפוץ לפקודה שהוצאנו".

jsr מצפה לקפוץ לכתובת האפקטיבית, לכן הוספנו @.
@(sp)+ עשה pop למחסנית.
jsr דאג לעשות push.

בשיטה זו נוכל לעשות שתי תוכנות שמדלגות אחת לשניה.

Re-entrancy

דוגמא : 50 תוכנות משתמשות בprintf בו זמנית. נרצה להחזיק רק עותק אחד של השיגרה.

שגרות מערכת משותפות למשתמשים- רבים. נרצה להחזיק רק עותק אחד של שגרות כאלו. שגרות יצטרכו להיכתב באופן מסוים על מנת להיות re-entrancy.
Time-sharing למשל הוא בעייתי, כי באמצע ביצוע שיגרה, יתכן שיגיע תורו של משתמש אחר לפעול.

שגרות Re-Entrant

- אינה משנה את עצמה.
- מכילה פקודות + קבועים בלבד.
- המשתנים של השיגרה נמצאים מחוץ לשיגרה וקיים עותק שלהם לכל תהליך המשתמש בה.
- המחסנית צריכה להיות פרטית לכל משתמש (לא הגלובלית של המערכת).
- מכילות "קוד טהור" – פקודות בלי מבנה נתונים פנימיים.

מקרה פרטי של re-entrancy הוא רקורסיה.
למה מותר לשיגרה רקורסיבית כן להשתמש במחסנית? ניתן לחשוב על כך בצורה כזו שלכל עותק של הפונקציה יש מחסנית משלה, ומניחים שבמקרה כל מחסנית נמצאת מיד מעל הקודמת.

קלט / פלט

כל ההתקנים מחוברים לBUS. כל יחידה מסוגלת לתפקד באופן עצמאי. כאשר 2 יחידות רוצות לדבר אחת עם השניה הן יכולות לדבר ישירות דרך הBUS. כאשר יחידות מנסות לתקשר דרך ה BUS יש צורך בסינכרון ובפרוטוקול תקשורת.
לכל בקר יש כתובת במרחב כתובות, הכתובות דומות אך לא זהות לכתובות זיכרון.
הקריאה והכתיבה היא לאוגר ממשק. לכל אוגר ממשק כתובת משלו במרחב הזיכרון.

אפיון התקני קלט פלט

- צורת הגישה (ישרה / סדרתית)
- אופן העברת הנתונים (יחידה בודדת / בלוק)
- סוג התקן (זיכרון עיקרי, זיכרון משני)

לגורם המקשר בין ההתקן לעולם קוראים בקר.

אוגרי ממשק בPDP11

שני סוגי אוגרים:

1. אוגר בקרה.
2. אוגר נתונים.

אוגר בקרה:

12-15	11	8-10	7	6	1-5	0
Error Code	Busy	Unit Select	Done/ Ready	Interupt Enabled	Function	Device Enabled

ביט	משמעות
0	1 אם ההתקן פעיל, אחרת 0.
1-5	כאן כותבים פונקציה שרוצים שההתקן יבצע.
6	כאשר הביט דלוק, ההתקן מסוגל לייצר פסיקות.
7	כאשר הביט 0, ההתקן לא סיים את הפעולה הנוכחית. כאשר הביט 1 ההתקן מוכן לפעולה הבאה. ניתן לבדוק ביט זה בקלות.
8-10	אם הבקר מחובר ליותר מהתקן אחד, ניתן לבחור בעזרת ביטים אלו מול איזה התקן לעבוד.
11	דלוק כשההתקן עסוק.
12-15	Error Code

אוגר נתונים:

8-15	0-7
Unused	Data

התקנים

מקלדת

אוגר בקרה: 177560

12-15	11	8-10	7	6	1-5	0
	BUSY (r)		DONE (r)	INT ENABLE (w)		RE (w)

אוגר נתונים: 177562
לא ניתן לשנות בטעות ביטים שהם read only.
בזמן קריאת התו, busy הוא 1.
כאשר יש תו מוכן באוגר הנתונים, done שווה 1. כאשר קוראים מאוגר הנתונים, DONE נקבע ל0, והמכונה מוכנה לקריאת התו הבא.

הערה לגבי המקלדת – אחרי שקוראים כל אות צריך להדליק את הביט ready על מנת לאפשר קריאה בהמשך.

מסך

אוגר הבקרה: 177564

15 8	7	6	5 0
	READY	INT ENABLED	

אוגר הנתונים: 177566

כאשר פונים לאוגר הנתונים, 7 נכבה. לאחר שנגמרת ההדפסה 7 נדלק שוב.

ישנן 2 שיטות לתקשר עם ההתקנים:

1. תשאול (polling). בשיטה זו המעבד בודק פעם בכמה זמן את המצב של ההתקנים.
2. פסיקות. בשיטה זו, ברגע שההתקן מוכן, הוא ישלח אות לCPU.

דוגמא: waitPrn

```
waitPrn:
        tst @#tps
        bgl waitPrn
```

שתי שורות אלו הן busywait על המדפסת. הלולאה מתרחשת כל עוד המדפסת לא סיימה להדפיס את התו הקודם שנשלח אליה. לאחר קטע קוד זה נוכל לשלוח תווים חדשים להדפסה.

נביא דוגמא נוספת לתשאול, תוכנית echo:
תוכנית echo מקבלת תווים מהמקלדת, ומדפיסה אותם על המסך.

```
TKS = 177560
TKB = 177562
TPS = 177564
TPB = 177566

.=torg+2000

echo: bis #1, @#TKS      ; enable keyboard read

loop1: tstb @#TKS        ; test done bit
       bpl loop1         ; repeat if not done (the byte is positive)

loop2: tstb @#TPS        ; test ready bit
       bpl loop2         ; repeat if not ready

       movb @#TKB, @#TPB

       br echo
```

דוגמא לפונקציות getc וputc ולnewline:
getc קוראת תו מהמקלדת ואילו putc מדפיסה תו על המסך.
newline משמשת לצורך העברת הסמן לשורה חדשה.

```
getc:
```

```
        mov #1, @#TKS           ; RE = 1
        tstb @#TKS              ;
        bpl .-4                 ; wait till DONE = 1
        movb @#TKB, r0          ; get a char from keyboard to r0
        rts pc

putc:
        tstb @#TPS              ;
        bpl .-4                 ; busywait till the printer is ready
        movb r0, @#TPB          ; send the byte in r0 to the printer
        rts pc

cr=15
lf = 12
newline:
        movb #cr, r0
        jsr pc, putc
        movb #lf, r0
        jsr pc, putc
        rts pc
```

פסיקות

מערכת הפסיקות – מערכת משולבת של חומרה ותוכנה המאפשרת למחשב להגיב בצורה אוטומטית מחוץ למעבד עצמו.
פסיקה היא אות מהתקן חיצוני שגורמת למעבד להפסיק לעשות את מה שהוא עושה ולהתחיל לבצע קטע קוד שיטפל בהתקן. המעבד לא מפסיק באמצע מחזור פקודה. בסוף מחזור הפקודה יתרחש ניתוח האות.

כאשר מתקבל אות, על המחסנית נכתבים ה-PSW וה-PC, ואחר כך נטענים PSW ו-PC חדשים מתוך כתובת קבועה בזיכרון.

Push psw
Push pc

עבור המקלדת כתובות אלו הן 60/62. עבור המסך אלו הם 64/66.
בכתובת 60, אנו שמים למשל את כתובת שיגרת הפסיקה המטפלת במקלדת, וב62 את הערך החדש לPSW – גם כדי לא לפגוע ב-CONDITIONS של התוכנית הראשית וגם עבור מנגנון העדיפויות.
לכל התקן קלט/פלט יש שיגרת פסיקה ייחודית משלו. חלק מהארכיטקטורה של המכונה קובע איפה כתובה כתובת השיגרה שתקרא עבור כל פסיקה.
בInterrupt Vector 2 מילים: המילה הראשונה מכילה את כתובת שיגרת הטיפול בפסיקה, והשני מכילה ערך שיטען לתוך הPSW בזמן ביצוע הפסיקה.
בעזרת הפקודה rti חוזרים מפסיקה. הפקודה לוקחת שני פרמטרים מראש המחסנית ושמה בPC ובPSW.

<u>תנאים לקבלת פסיקה</u>

1. ההתקן רשאי ליזום פסיקות (ביט 6)
2. ההתקן פעיל (ביט 0)
3. פעולת קלט/פלט הסתיימה (מעבר של ביט 7 מ0 ל1)
4. עדיפות ההתקן גדולה ממש מעדיפות המעבד.
   עדיפות זהו מספר 3 ביטים בPSW. (ביטים 5-7).

PSW			
0-3	4	5-7	8-15
Condition Codes NZVC	T_Bit	עדיפות	

התקן	מילת סטטוס	מילת Buffer	עדיפות חומרה	וקטור פסיקה
לוח מקשים	TKS = 177560	TKB = 177562	4	60, 62
מדפסת	TPS = 177564	TPB = 177566	4	64,66
שעון	LCS = 177546	none	6	100,102

השעון נחשב התקן חיצוני. כל התקן חיצוני הוא איטי. כדי לעשות ייעול זמנים התקן איטי יקבל עדיפות גבוהה יותר ולכן יש עדיפות לכל התקן.

תיאור תהליך

- התקן מבקש פסיקה.
- המכונה מקבלת / לא מקבלת פסיקה
- אם התקבלה הפסיקה, המשך.
- המכונה מסתכלת על ווקטור הפסיקה של ההתקן שביקש את הפסיקה.
- הPSW והPC נשמרים בראש המחסנית.
- ערך חדש נטען לPSW ולPC.
- השיגרה מתבצעת.
- חזרה לתוכנית עם RTI, PSW וPC משוחזרים.

עדיפויות

מנגנון העדיפות נוצר על מנת לסנכרן פסיקות. כאשר ניכנס לפסיקה נוכל להגדיר לאילו פסיקות אחרות ניתן לקרוא, ואילו ידחו.
העדיפות רשומה בPSW. עדיפות היא 3 ביטים, כלומר קיימות 8 עדיפויות שונות. עדיפות 7 היא הגבוהה ביותר ו0 הנמוכה ביותר.
סוגי עדיפויות:

- עדיפות תוכנה / עדיפות מעבד – זוהי העדיפות הרשומה בשדה PRIO של הPSW בכל רגע נתון, כלומר עדיפות זו ניתנת לשינוי. למתכנת שליטה מלאה בעדיפות זו.
- עדיפות חומרה – לכל התקן עדיפות קבועה משלו. למשל, למקלדת ולמדפסת עדיפות 4, לשעון עדיפות 6. עדיפות זו אינה ניתנת לשינוי.
- עדיפות שלישית היא העדיפות העתידית שתהיה למעבד. המתכנת שולט גם בעדיפות זו.

עדיפות ברירת המחדל היא אפס. ניתן לשנות את הPSW ישירות, אולם במסגרת הקורס את"ם לא נעשה זאת.

איך משנים בכל זאת את הPSW? נגיד שהגענו לשיגרת טיפול בפסיקה, נוכל לחטט במחסנית, לשנות את הPSW השמור ולחזור לתוכנית. לאחר שנלמד פסיקות תוכנה, נדגים קטע קוד שיעשה זאת.

דוגמא לתוכנית ECHO עם INTERRUPTS:

```
TKS = 177560
TKB = 177562
TPS = 177564
TPB = 177566

.=torg+60
```

```
.word echo
.word 200
.=torg+2000
main:
        mov #main, sp
        bis #101, @#TKS          ; EN = 1, EI = 1
        …
        …
.=torg+5000
echo:
        mov @#TKB, @#TPB
        bis #1, @#TKS            ; EN = 1
        rti
```

bis – פקודה להדלקת ביטים. הארגומנט הראשון אומר איזה ביטים להדליק והשני הוא המילה להדליק בה את הביטים. bis מבצע למעשה or בין המשתנים.

<u>השעון</u>

התקן – דומה בפעולה למטרונום. כל זמן קבוע נשלח "תיק". זהו התקן קלט שפוסק בקצב קבוע.
שימושים:

- לקבוע מה השעה.
- למדידת זמנים.
- הפעלת תהליכים בעתיד.

השעון מבקש פסיקה כל 1/50 שניה. לשעון מילת בקרה אחת: PDP11ב
CLS = 177546
הביט המשמעותי היחידי במילת הבקרה הוא 6 – EI.

הערה כללית לגבי שגרות פסיקה, הוא ששגרות פסיקה צריכות להיות קצרות ככל האפשר.

<u>קביעת עדיפויות</u>

- מומלץ שהתקן לא יהיה בעדיפות תוכנה פחותה מעדיפות החומרה שלו, כי אחרת יתכן קינון פקודות.
- מומלץ לא לעבור את עדיפות 6 כי אז מפסיקים פסיקות שעון.

<u>הפקודה wait</u>

פקודה זו דומה לפקודה halt. המעבד "מנמנם" אבל מגיב לפסיקות. בסיום שיגרת הטיפול בפסיקה חוזר הביצוע לפקודה שאחרי wait.

```
l: wait
br l
```

זהו מימוש busywait בעזרת פסיקות.

<u>פסיקות תוכנה</u>

הבעיה: יצירת פקודות חדשות למכונה, למשל.
אם יוצרים פונקציות, פסיקות יכולות להפריע במהלך ביצוע הפקודה.
הפתרון: מנגנון העברת בקרה לשיגרה דרך מנגנון העדיפויות.

מנגנון זה הוא פסיקות תוכנה / מלכודות.
המנגנון מורכב מ4 פקודות:

trap, emt, bpt, iot

לכל אחת מהפקודות אפקט דומה. הפקודה מעבירה בקרה לשיגרה ובסוף השיגרה חוזרת ההרצה אל אחרי הפקודה הפוסקת. לכל אחת מפקודות אלו ווקטור פסיקה משלה.
משיגרות שנקראות בעזרת פקודות אלו, חוזרים על ידי הפקודה rti.

ווקטורי פסיקה	
trap	34/36
emt	30/32
bpt	14/16
iot	20/22

פעולת המלכודת:
מעבר לפונקציה בלי לציין את שמה במפורש, וגם לשנות את העדיפות.
הפקודות מתחלקות ל2 סוגים:
bpt, iot פשוטות יותר
trap, emt מכילות בתוך הOPCODE שלהן פרמטר.

עדיפות החומרה של פסיקות תוכנה היא 8. פסיקות תוכנה מתקבלות תמיד.

iot משמשת באופן מסורתי לתמיכה בI/O.
bpt משמשת לbreak points.
trap n / emt n משמשות לתקשורת עם מערכת ההפעלה.

מבנה הפקודה של trap / emt:

8-15	0-7
Opcode	פרמטר - n

ניתן לכתוב:

trap 3

אין צורך ב# כאשר משתמשים בפקודות אלו.

המשמעות של

trap n / emt n

לקפוץ לשיגרה הכתובה בווקטור הפסיקה שלהן. בעזרת הפרמטר נוכל לבחור מה לעשות.

קבלת הפרמטר בשיגרת הטיפול נעשית על ידי משחק עם הPC והוצאת הפרמטר.

תבנית לעבודה עם EMT:

```
        .=torg+30
        .word emtrap
        .word 340               ; PRIO = 7

        .=torg+2000

emtrap:
        mov (sp), -(sp)         ; Get caller's pc
        sub #2, (sp)            ; Calculate emt address
        mov @0(sp), (sp)        ; Get emt machine code
        bic #177400, (sp)       ; Get service number
                                ; (177400_8 = 1111111100000000_2)
        asl (sp)                ; calculate service address (we
                                ; need even values only)
        add #emtable, (sp)
        mov @0(sp), (sp)
        jmp @(sp)+              ; Jump to service code

emtable:
        .word emt00             ; address of service 1
        .word emt01             ; address of service 2
        .word emt02             ; address of service 3
;       ...
;       ...

; EMT service 1
emt00:
```

```
;       ...
;       ...
        rti

; EMT service 2
emt01:
;       ...
;       ...
        rti


        .=torg+5000
;       ...
;       ...
```

נציג כעת דוגמא נוספת. נכתוב שיגרת פסיקה עבור emt, כך שכשנכתוב
emt 4
למשל, עדיפות המעבד תהפוך להיות 4.
הדרך בה ניגש לבעיה: נקרא לפסיקת תוכנה. נשנה את העדיפות הנמצאת בPSW במחסנית, ואז נחזור לתוכנית הראשית. הPSW החדש ייטען מתוך המחסנית.

דוגמא - SetPriority

```
. = torg + 30
.word emtrap
.word 340

. = torg + 1000
emtrap:
        sub #2, (sp)
        cmpb @0(sp), #7
        bhi exite
        mov r0, -(sp)
        mov @2(sp), r0
        ash #5, r0
        bicb #340, 4(sp)
        bisb r0, 4(sp)
```

```
        mov (sp)+, r0
exite:
        add #2, (sp)
        rti

. = torg + 2000
main:

        emt 5
        halt
```

ניתן לשנות את עדיפות המעבד גם ללא פסיקות, על ידי מעין "רמאות".
נביט בקוד הבא:

```
. = torg + 1000
main:
        mov #main, sp
        mov #340, -(sp)
        jsr pc, SetPrio
        halt

SetPrio:
        rti
```

דחפנו למחסנית את המספר 340, וקראנו לשיגרה SetPrio. מהשיגרה SetPrio חזרנו עם rti. הפקודה rti קראה שני פרמטרים מהמחסנית. אחד הוא הPC, והשני הוא ערך לטעון לPSW. אולם, האיבר שהיה במחסנית מתחת כתובת החזרה הוא 340, ולכן עדיפות המעבד היא 7 לאחר החזרה מהשיגרה.

Debuging- ניפוי שגיאות

Debugger זוהי תוכנית מחשב שמאפשרת למשתמש להריץ תוכנית אחרת על מנת לבקר את הפעולה שלה. הכלים ש-debugger נותן לנו הם Breakpoints וStepping.
הצעה לכתיבת Debugger: הוספת פקודות בין פקודות התוכניות. רעיון זה נפסל כי אסור לנו להזיז בלוקים של התוכנית (כי אז פקודות קפיצה לא יעבדו).
פתרון לבעיה זו, למשל, הוא החלפת הפקודה המקורית בפקודה לdebug. הפקודה המבוקשת נשמרת בצד והיא תבוצע בנפרד.
שימוש נוסף – profiling – איסוף מידע על ריצת התוכנית, כמה פעמים כל פונקציה נקראת ועוד.
Pdp-11 נותן לנו תמיכה בחומרה לTRACE, בעזרת t-bit.
כאשר ערך t-bit 0, המכונה עובדת כפי שראינו עד היום. כאשר ערכו של t-bit הוא 1, תתבצע פסיקת תוכנה לפני ביצוע כל פקודה.
* ווקטור הפסיקה של t-bit הוא 14/16.

איך מכבים / מדליקים t-bit?
דרך פסיקות תוכנה. הפסיקה המומלצת היא emt (כי ניתן לתת לה פרמטרים).

דוגמא, בהנחה שאנו משתמשים בשלד של emt שהוצג קודם:

```
emt00:
        bic #20, 2(sp)                  ; turn off t-bit in caller's psw
        rti

emt01:
        mov #trace, @#14                ; pc in trap vector of t-bit
        mov #340, @#16                  ; psw in trap vector of t-bit
        bis #20, 2(sp)                  ; turn on t-bit in caller's psw
        ...
        rti

trace:
        ...
        ...
        ...
        rtt


trace_off = 0
trace_on = 1
...

main:
```

```
mov pc, sp
tst –(sp)
...
emt trace_on
.
.
.
emt trace_off
...
halt
```

מיד לאחר הפקודה rti של הפונקציה שהדליקה את t-bit, תקרא trace.
כאשר אנו בשיגרת trace הt-bit כבוי (על ידי קביעת t-bit כבוי בווקטור של trace).

rtt – לפקודה אותו האפקט כמו rti אלא אם הערך של t-bit נדלק לאחר שהיה 0, לא תתצבע פסיקת t-bit (אולם פקודה אחר כך תתבצע הקריאה לt-bit).
שיגרה המשתמשת ב t-bit חייבת לחזור עם rtt.

Error Traps

אלו פסיקות שיוזם הCPU (ולא חומרה או תוכנה).
הרעיון: הCPU מגלה שקראה טעות – מצב לא תקין. הCPU מיוזמתו קורא לפסיקה.

כאשר מתרחשת תקלה, התוכנית לא תיעצר אלא תיקרא השגרה שנקבע.

הערה
הכתובת הכי נמוכה שניתן לגשת אליה היא 376. כאשר sp <= 420 נקבל כבר אזהרה.

עדיפות error traps היא 8, אולם יש גם עדיפויות ביניהן.

Error	PRIO	Order	Vector
Odd Address	8	0	4/6
Stack Violation (sp <= 376)	8	0	4/6
Stack Warning (sp <= 420)	8	1	4/6
Illegal Opcode	8	2	10/12
Power Failure	8	3	24/26

האסמבלר

האסמבלר היא תוכנה שלוקחת תוכניות בשפת אסמבלי ומייצרת מהן תוכנות בשפת המכונה של הPDP11. לתוצר פעולת האסמבלר נקרא load module או object.
מה היתרון של שימוש בשפת אסמבלי על פני כתיבה ישירות בשפת מכונה?

- שימוש ב mnemonic, (שמות לפקודות).
- תוויות (לצורך הסתעפויות, וגם למתן שמות משמעותיים לכתובות בזכרון).
- קיצורי כתיבה (בשיטות מיעון למשל).
- האסמבלר מאפשר לגלות שגיאות לפני זמן ריצה.

פעולות האסמבלר

- יצירת קוד מתוך תוכנית המקור.
- הקצאת זיכרון.
- גילוי שגיאות.
- יצירת מידע לצורך קישור וטעינה.

נעבוד על משפחה של אסמבלרים – two pass – שני מעברים.
האסמבלר לוקח את הקוד ועובר עליו פעמיים. הצורך לעבור פעמיים קיים כי בפעם הראשונה אין לנו את כל האינפורמציה. במעבר הראשון ניצור "טבלת סמלים" שתשמש במעבר בקריאה השניה.
הטבלה מורכבת מ3 עמודות. העמודה הראשונה היא המיקום בו הופיעה לראשונה התוויות. העמודה השניה היא התווית, והעמודה השלישית היא מיקומה של התווית בזיכרון. אם בסוף המעבר הראשון, תישאר תווית ללא כתובת, נדע שקראה שגיאה. במהלך המעבר הראשון, אם ננסה להכניס תווית פעמיים, זוהי שגיאה.

האסמבלר מחזיק כל הזמן מצביע אל הכתובת החל ממנה תיקרא הפקודה הנוכחית. למצביע זה קוראים LC. זהו משתנה פנימי של האסמבלר.

הבחנה חשובה בין הPC לLC:
LC קיים בזמן האסמבלי.
PC קיים בזמן הריצה.
LC הוא תמיד הכתובת שהחל ממנה תקרא הפקודה הנוכחית.
בתוכניות שאנו כותבים ניתן להשתמש בLC.
אנחנו משתמשים בתוכנה בLC כ .

```
. = torg + 100
```

משמעות הפקודה ← הצב ערך למשתנה נקודה.
ניתן לכתוב למשל

```
br .+2
```

נביט בדוגמא הבאה:

```
tstb @#tks
bpl .-4
```

כאשר אנו אומרים לקפוץ 4-. אנו אומרים למעשה לקפוץ 4 בתים אחורה מתחילת הפקודה bpl. מה שבפועל יקודד בזיכרון עבור הפקודה bpl הוא:

100375

375 מייצג את המספר 3-. נזכור שהמרחק בפקודות branch מקודד במילים. נשאלת אם כך השאלה – למה קודד 3 ולא 2 (4 בתים)? התשובה היא שהLC מתייחס למיקום הפקודה בזמן האסמבלי, הוא מצביע על תחילת bpl, אולם בזמן ריצה, כאשר נגיע לפקודה bpl, pc כבר מצביע לפקודה לאחר bpl ולכן יש צורך לקפוץ 3 מילים אחורה, גם מעל הפקודה bpl.

הנחת קורס:
במעבר ראשון לא מיוצרים אופרנדים בכלל. גם לא כאלו המבוססים על תוויות, וגם לא ללא תוויות.

טעויות

כאשר התוכנית אינה תקינה, האסמבלר מגלה את השגיאה. כאשר מכניסים תווית, האסמבלר בודק את טבלת הסמלים. ברגע שהאסמבלר מגלה תווית כפולה, למשל, הוא ידווח על שגיאה.

תווית מוגדרת פעמיים או יותר – במהלך מעבר ראשון.
שימוש בתווית שאינה מוגדרת – מעבר שני / סוף מעבר ראשון.
פקודה לא חוקית – במהלך מעבר ראשון.
קפיצה רחוקה מדיי – מעבר שני (כי כל הארגומנטים מושמים במעברים שני).
מספר אופרנדים שגוי – מעבר ראשון.
ביטוי שגוי – (mov #88, r3) – מעבר שני, כי רק אז בודקים את האופרנדים.

קישור וטעינה

קיים צורך לבצע את הפעולות הבאות:

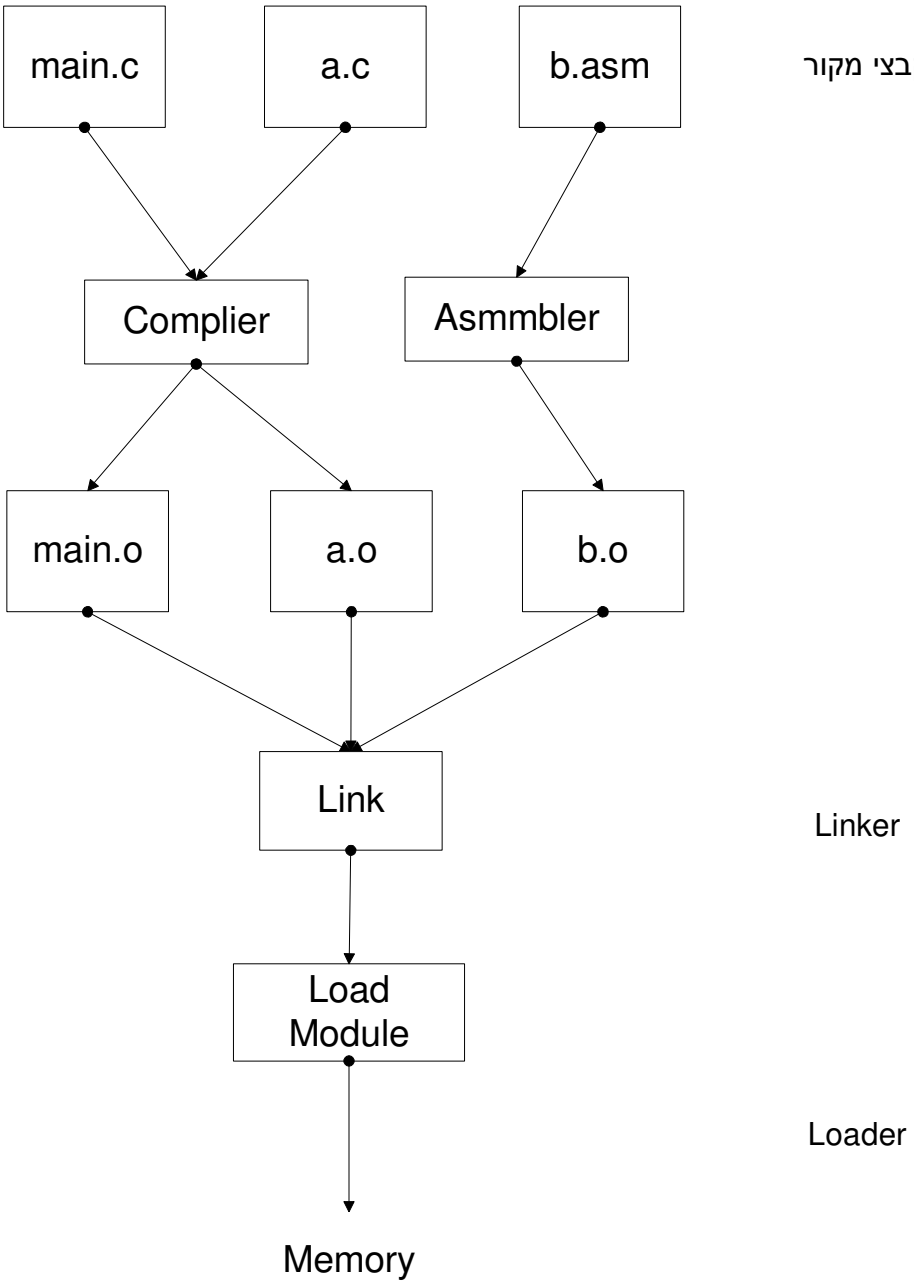
1. אסמבלר נפרד: Separate Compilation, Separate Assembly.
2. טעינה למקום לא ידוע בזיכרון, תמיכה וטיפול בתוכניות התלויות במיקומן בזכרון.

מעט מושגים:

קישור – Linkage

טען – Loader

הטען לוקח load module וטוען אותו לזכרון.



טיפול בתוכניות התלויות במיקומן בזיכרון

PIC – Position Independent Code – זוהי תוכנית שלא תלויה במיקומה בזיכרון.
כל הגישות לזיכרון שהיא מבצעת נעשות על ידי מיעונים יחסיים.

לדוגמא:
האם התוכנית הבאה היא PIC?

```
main:
        mov #A, r1
        mov #5, r2
        clr r0
loop:
        add (r1)+, r0
        sob r2, loop
        halt
A: .word 1, 5, 17, 3, 6
```

תוכנית זו לוקחת את המערך A וסוכמת את איבריו לתוך r0. תוכנית זו אינה PIC.
המערך נמצא בזיכרון, והפקודה #A שומרת בתוכה את הכתובת של A.
אפשר לתקן את התוכנית ידנית ולעשותה PIC.

במקום

```
mov #A, R1
```

נוכל, למשל, לכתוב

```
mov pc, r1
Add #A – main – 2, r1
```

אנו מורידים 2 כי r1 מכיל את כתובת הפקודה הבאה לביצוע.

עכשיו התוכנית היא PIC.

מה עושה הקשר/טען כאשר הוא מקבל תוכניות שאינן PIC?

משימות הטען-קשר

1. הקצאת מקום בזיכרון עבור התוכנית – Allocation
2. התאמת כתובות התלויות במיקום התוכנית בזיכרון – Relocation
3. פתרון התייחסות לסמלים חיצוניים – Linkage
4. טעינה פיסית של מודול הטעינה לזיכרון.

את משימה 3 עושה הקשר. את שאר המשימות עושה הטען.
הטען והקשר צריכים לקבל מידע מהאסמבלר על מנת לבצע את המשימות הנ"ל.
עבור משימה 1, נדרש גודלו של קובץ הObject.
עבור משימה 2, נדרשת רשימת כל הכתובת שיש לבצע להן Relocation.
עבור משימה 3 יש צורך ברשימת כל הסמלים המוגדרים במודול וניתן להשתמש בהן מחוץ למודול – entries. כמו כן יש צורך ברשימת כל הסמלים שאינם מוגדרים במודול אבל הוא משתמש בהם – externs.
עבור משימה 4 יש צורך בקוד המכונה עצמו.

על מנת לבצע משימות אלו, נרחיב את שפת ההנחיות של האסמבלר.

תחילת מודול:

.csect A

הנחיה זו אומרת לאסמבלר: כאן מתחיל מודול ששמו A.

.entry [list of symbols]

השמות שמופיעים ברשימה הם שמות של משתנים שמופיעים במודול הנוכחי שאנחנו מאפשרים גישה אליהם מבחוץ.

.extern [list of symbols]

רשימת סמלים חיצוניים.
הסמלים יוספו לטבלת הסמלים.

ניתן לכתוב כמה פקודות extern וentry. אין צורך להצהיר על כל המשתנים בשורת פקודה אחת ארוכה.

- האסמבלר מניח שכתובת התחלת הטעינה של כל מודול היא 0.
- האסמבלר מניח שערכם של סמלים חיצוניים הוא 0.

טבלאות קישור וטעינה

מלבד קוד, האסמבלר יוצר מידע עבור הטען קשר. הוא יוצר טבלת ESD וטבלת RLD.

ESD – External Symbol Dictonary
RLD – Relocation & Linkage Dictonary

ESD

הטבלה מכילה 3 עמודות:

Symbol	Type	RL - Relative Location
שם התוויות שהוגדרה "מיוחדת"	סוג התווית המיוחדת	כתובת יחסית – מרחק מתחילת הcsect. שדה זה רלוונטי רק לentries.

Symbol מכיל תוויות שהוגדרו כ"מיוחדות". עבור כל תוויות כזו, בשורה המתאימה בType נמצא הסוג שלה. סוגים אפשריים:

1. LD – (entry) – סמל שמוגדר לוקלית.
2. ER – (extern) – סמל שיוגדר בקובץ אחר.
3. SD – (csect) – התחלת המודול.

LD – Local Definition
ER – Extrnal Definition
SD – Segmant Deginition

דוגמא:

```
        .csect a
        .extern subr
        .entry op
main:
        mov #main, sp
        mov #2, op
        jsr pc, subr
        mov r0, res
op:
        .blk 1
res:
        .blkw 1


        .csect b
        .extern op
        .entry subr
subr:
        mov op, r0
```

```
        mov list(r0), r5
        rts pc
list:
        .blkw 5
```

לצורך בניית הטבלאות, אין לנו צורך להבין מה התוכנית עושה (וזה הרי הגיוני, כי האסמבלר שבונה את הטבלאות, לא "מבין" מה התוכנית עושה).

נבנה ESD לשני המודולים:

ESD		
Symbol	Type	Location
a	SD	0
subr	ER	
op	LD	24

ESD		
Symbol	Type	Location
b	SD	0
op	ER	
subr	LD	0

RLD

הטבלה מכילה 3 עמודות:

Symbol	Flag	RL - Relative Location
הסמל שאת ערכו יש להוסיף או להפחית מתוכן הכתובת.	+ להוספה - להפחתה	כתובת ביחס להתחלת המודול

טבלאות RLD קוראים מימין לשמאל!! טבלת RLD מכילה אינפורמציה על כתובת שיש לתקן את תוכנן בזמן loading.

בדוגמא שלנו:

```
mov #2, op
```

את שורה זו אין צורך לתקן. op הוא יחסי, 2# הוא מספר ולכן אין צורך לעדכן אותו.

```
l: jsr pc, subr
```

למרות ש-subr הוא יחסי, מיקומו המדויק לא ידוע כי הונח ש-subr הוא 0. קיים הצורך לתקן. כמו כן, בכתובת יש ההפרש בין subr ל l ולכן יש להוסיף את subr ולהוריד את a.

במקרה זה ישנם שני תיקונים:

a	-	14
subr	+	14

בניית טבלת RLD

תווית חיצונית (מוגדרת על ידי extern)		תווית מקומית	
מיעון אבסולטי	מיעון יחסי	מיעון אבסולוטי	מיעון יחסי
.extern x tst x+2(r2)	jsr pc, sub	mov list(r0), r0	mov r0, res
תיקון אחד: ExtLbl, +, Rel LC	שני תיקונים: CrntCSect, -, Rel LC ExtLbl, +, Rel LC	תיקון אחד: CrntCSect, +, Rel LC	אין צורך בפעולה

קישור

קלט לתהליך הקישור:
- כל קבצי הObject.
- טבלאות ESD וRLD של כל המודולים.
- סדר הקישור.

לקישור שלושה שלבים:
1. בניית מפת טעינה.
2. איחוד טבלאות ESD.
3. איחוד טבלאות RLD.

בניית טבלת טעינה

מפת טעינה אומרת באיזה כתובת מתחיל כל אחד מהמודולים.
המודולים ישורשרו אחד אחרי השני בזיכרון ברצף. אם csect נגמר בכתובת אי זוגית, המודול הבא יתחיל בכתובת הזוגית הבאה.
ניתן לדעת באיזה מקום בזכרון מתחיל ונגמר כל מודול (ביחס לתחילת הבלוק הראשון).

דוגמא לטבלת טעינה:

0 26	A
30 52	B

טבלאות הRLD והESD הן לשימושו של הקשר. הקשר משנה בעזרתן את הobjects השונים.

איחוד טבלאות הESD

1. הקשר מדביק את טבלאות הESD אחת לשנייה.
2. הקשר בודק שבכל מקום שמופיע ER יש LD מתאים.
3. הקשר בודק שאין כפילויות של LD.
4. הקשר מוחק את שורות הER.
5. הכתובות הרשומות בצד ימין של הטבלה היא כתובות יחסית ביחס להתחלת הCSECT. כעת מתקנים אותן שיהיו כתובת ביחס להתחלת המודול.

איחוד טבלאות הRLD

הקשר משרשר את טבלאות הRLD אחת לשניה.
לאחר מכן הוא מתקן את הכתובות בעמודה הימנית כך שיהיו ביחס להתחלת המודול כולו.
חסרה עדיין כתובת טעינת המודול, אבל אנחנו יודעים את מיקום התוויות ביחס להתחלת המודול.

עכשיו נוכל לבצע את התיקונים הרשומים בטבלת RLD.
שורות שמציינות שני תיקונים – אחד ב(+) ואחד ב (-) יצטמצמו. זה הגיוני כי זוגות אלו מציינות שיטת מיעון יחסית, ואופרנדים המקודדים בשיטת מיעון יחסית לא אמורה להיות תלויים במיקום טעינת התוכנית.

דוגמא:

subr	+	14
A	-	14

בחישוב כתובת תחילת הטעינה תצטמצם ולכן אנו יכולים להוסיף את ההפרש subr-a.

בזמן הטעינה אין יותר צורך לתקן כתובת זו. שתי שורות אלו ימחקו מהRLD אחרי התיקון.

השורות שנותרו ללא טיפול הן שורות בודדות עם מינוס או פלוס.
נוסיף לתוכן שלהן את הערך.

a	+	2

הוסף ל2 את A.

בשורות המציינות תיקון אחד – בצע את התיקון אך אל תמחק את השורה.

מחק את עמודת symbol בטבלה המצומצמת.
האינפורמציה שנשארה היא הכתובות שערכן תלוי בתחילת כתובת הטעינה. הנעלם היחיד בטבלה הוא כתובת התחלת הטעינה.

טעינה

"השג" את כתובת התחלת הטעינה. עבור על כל שורה בטבלת הRLD המצומצמת.
הוסף או החזר את הכתובת מתוכן הכתובות המופיעות בטבלה.

הערות כלליות לנושא:

```
mov r1, x
mov r1, @#x
```

אפקט זהה, אבל הקידוד בזיכרון ופעולות הקשר שונות.

```
mov @#2000, 11
```

האם תוכן הsourcre הוא כתובת קבועה או יחסית?
כתובת קבועה, אחרת היינו משתמשים בlabel.
אין צורך במקרה כזה להוסיף שורה לRLD, מכיוון אין צורך בשינויים.
אם רוצים להתייחס לכתובת קבועה, #@ זו הדרך המומלצת.

הנה דוגמא למצב של שורה אחת עם מינוס.
המרחק בכתובת 42 יהיה המרחק הנכון מכתובת 2000.

(40) mov 2000, 2002

StartCsect	-	42

הערה נוספת – יתכן שיידרש תיקון עם RLD גם להנחיות לאסמבלר. (כגון word. וכו').

נספח א' – דוגמאות

מיעון יחסי, Mode 6,7

מצב הזיכרון לפני ריצת התוכנית:

2000	2002
4	3

```
. = torg+1000

main:
L:      mov 2000, 2002
        halt

. = torg + 2000
pos2000: .word 4
pos2002: .word 3
```

מצב הזיכרון לאחר ריצת התוכנית:

2000	2002
4	4

התוכנית העתיקה את תוכן כתובת 2000 לכתובת 2002.
שני האופרנדים של הפקודה mov 2000, 2002 מקודדים בשיטת מיעון 6 רגיסטר 7.
נרצה לדעת כיצד תקודד הפקודה בזיכרון.

עבור האופרנד הראשון:

EA= Rn+X

במקרה שלנו:

EA=R7+X

כאשר מתבצעת הפקודה, R7 הוא L+4

EA=L+4+X
EA=pos2000=L+4+X
X= pos2000-L-4 = 2000 – 1000 – 4 = 774

עבור האופרנד השני:

R7 = L + 6
EA=L+6+Y
EA=pos2002=L+6+Y
X= pos2002-L-6 = 2002 – 1000 – 4 = 774

הפקודה תיראה בזיכרון כך:

016767 000774 000774

דוגמאות – קלט פלט

דוגמא – מימוש הפקודה WriteLn

נדגים פה כיצד ניתן לממש את הפקודה WriteLn באסמבלר של PDP11.
הפקודה מקבלת מחרוזת ומדפיסה אותה על המסך.
נניח שהתו המציין סיום מחרוזת הוא ערך ASCII 0.
נשתמש בשתי פונקציות עזר – PutChar ו NewLine.
PutChar מקבלת תו אחד ומדפיסה אותו על המסך, ו NewLine מדפיסה שורה חדשה (כלומר, מקדמת אותנו לשורה הבאה).

```
;-----------------------------------------------------------------------------
; PDP-11 Consts:
;-----------------------------------------------------------------------------

TPS = 177564
TPB = 177566

CR = 15
LF = 12

;-----------------------------------------------------------------------------
; Function Name: WriteLn
;
; Gets:
;       address of string in stack by refrence
; Returns:
;       Nothing
; Operation:
;       Print string on the printer
;-----------------------------------------------------------------------------

WriteLn:
        mov r0,-(sp)
        mov 4(sp), r0

WriteLop:
        tstb (r0)                       ;
        beq EndWrite                    ; If we reach end of string, exit
                                        ; write function

WaitLoop:
        tstb @#TPS                      ; test ready bit
```

```
        bpl WaitLoop                ; repeat if not ready
        movb (r0)+, @#TPB           ; send the next char to the printer
        br WriteLop

EndWrite:
        jsr pc, NewLine             ; add new line at the end
        mov (sp)+,r0
        rts pc

;-----------------------------------------------------------------------------
; Function Name: PutChar
; Gets:
;       char in r0 by value
; Returns nothing
;       Print char on the screen
;-----------------------------------------------------------------------------

PutChar:
        tstb @#TPS
        bpl .-4
        movb r0, @#TPB
        rts pc


;-----------------------------------------------------------------------------
; Function Name: NewLine
; Gets:
;      char in r0 by value
; Returns nothing
;      Print newline on the screen
;-----------------------------------------------------------------------------

NewLine:
        mov r0, -(sp)
        movb #CR, r0
        jsr pc, PutChar
        movb #LF, r0
        jsr pc, PutChar
        mov (sp)+,r0
        rts pc
```

הסיבה שלא השתמשנו בפונקציה PutChar בפונקציה WriteLn היא שניסינו להשיג מקסימום יעילות בפונקציה זו, וקריאות מרובות לפונקציה אחרת גורמות כל הזמן ל-PC להדחף ולצאת מהמחסנית, דבר שלוקח זמן.

דוגמא – מימוש הפקודה ReadLn

בדוגמא זו נממש את הפונקציה ReadLn.
הפוקנציה מקבלת מצביע למחרוזת דרך המחסנית. הפונקציה קוראת שורה מהמקלדת ושמה אותה בתוך המחרוזת. הפונקציה אינה בודקת אם יש מספיק מקום בזיכרון עבור השורה. היא מניחה שהמחרוזת ארוכה מספיק כדי להכיל את כל השורה.
קטע ה-main מדגים שימוש בפונקציה.

```
TKS = 177560
TKB = 177562
TPS = 177564
TPB = 177566
CR = 15
LF = 12

;-------------------------------------------------------------------------------

MAX_LEN = 100

.=torg+2000

main:
        mov pc,sp
        tst -(sp)

        mov #Arr, -(sp)
        jsr pc, ReadLn
        tst (sp)+
        halt

;-------------------------------------------------------------------------------
; Function Name: ReadLn
; Gets: Allocated array by address using stack
; Returns: Nothings
; Operation: Reads data from the keyboard and put it to the array till it
reachs CR
;-------------------------------------------------------------------------------

ReadLn:
```

```
        mov r0,-(sp)        ; push r0
        mov r1,-(sp)        ; push r1
        mov 6(sp), r1               ; address for the string
echo:
        bis #1, @#TKS               ; enable keyboard read

loop1:
        tstb @#TKS          ; test done bit
        bpl loop1           ; repeat if not done (the byte is positive)

loop2:
        tstb @#TPS          ; test ready bit
        bpl loop2           ; repeat if not ready

        movb @#TKB, r0              ;

        cmpb r0, #CR                ; if (CurrentKey == '\n') goto EndRead
        beq EndRead                 ;

        movb r0, @#TPB              ; echo

        movb r0, (r1)+              ; put the CurrentKey in the array
        br echo                     ; loop

EndRead:
        movb #0, (r1)+              ; add '\0'
        mov (sp)+,r1                ; pop r1
        mov (sp)+,r0                ; pop r0
        rts pc

Arr: .blkw 100.
```

דוגמא – הדפסת מספר – פיתרון איטרטיבי

הפונקציה שאדגים כעת מדפיסה מספר עשרוני על המסך.
היא מקבלת מספר עשרוני במחסנית by value ומציגה אותו על המסך.

```
.=torg+2000

TKS = 177560
TKB = 177562
TPS = 177564
TPB = 177566
CR = 15
LF = 12

main:   mov pc,sp
        tst -(sp)
          mov #8888., -(sp)
          jsr pc, PrnASCII
          tst (sp)+
          halt

;-------------------------------------------------------------------------------

PrnASCII:
          mov r4,-(sp)
          mov r3,-(sp)
          mov r2,-(sp)
          mov r1,-(sp)
          mov r0,-(sp)

          mov #0, r0
          mov #10., r1
          mov 14(sp), r2
          mov r2, r3

ASCwhile:
          tst r2
          beq EndWhile                    ; while (r2 != 0) {

          sxt r2
          div r1, r2                ; r2 = r3 / 10 , r3 = r3 % 10;
```

```
        add #48., r3            ; convert numbers to ascii value of
numbers
        mov r3,-(sp)            ; push r3;
        add #2, r0              ; r0 += 2;
        mov r2, r3              ; r3 = r2
        br ASCwhile
EndWhile:

        mov r0,r3               ; r3 as temp, can't use stack when poping

        mov sp, r4
        add r0, r4
ASWhile2:
        cmp r4, sp
        beq EASWhile2

        mov (sp)+, r0
        jsr pc, PutChar

        br ASWhile2
EASWhile2:

        mov r3, r0

        add r0, sp              ; pop all items

EndPASCII:
        mov (sp)+, r0
        mov (sp)+, r1
        mov (sp)+, r2
        mov (sp)+, r3
        mov (sp)+, r4
        rts pc

;-------------------------------------------------------------------------------

PutChar:
        tstb @#TPS
        bpl .-4
        movb r0, @#TPB
        rts pc

;-------------------------------------------------------------------------------
```

דוגמא – הדפסת מספר – פתרון רקורסיבי

פונקציה זו מבצעת את אותה משימה אשר הפונקציה בדוגמא הקודמת עושה, אך עושה זו בצורה רקורסיבית.

```
PrnNumber:
        mov 2(sp), r0
        bmi exitm
        mov r1, -(sp)
        jsr recursive
        mov (sp)+, r1
exitm:
        rts pc


; Function Nam: recursive
; Gets:
;       Number in r0 by value
recursive:
        mov r0, r1
        sxt r0
        div #10., r0
        beq put
        mov r1, -(sp)
        jsr pc, recursive
        mov (sp)+, r1
put:    jsr pc, putd
        rts


putd:
        cmp r1, #9.
        bhi exit
loop:
        tstb @#TPS
        BPL loop
        add #48., r1
        mov r1, @#TPL
exit:
        rts pc
```

דוגמאות – הבהרת נקודות לגבי פקודות

דוגמא – sob

מטרת הדוגמא: הבהרת סדר הפעולות שמבצעת sob.
נביט בקוד הבא:

```
. = torg + 2000
main:
        mov r0, #0
        clr r2

loop1:
        inc r2
        sob r0, loop1
```

האם היא תקינה?
כוונת הכותב הייתה "אם r0 גדול מ0 הקטן אותו וקפוץ. אם הוא שווה 0, המשך בפקודה אחרי sob."
אבל, זה לא מה שמתבצע.
sob ראשית מפחיתה את הרגיסטר r0. מכיוון שr0 הוא 0, יש borrow, ועכשיו r0 הוא 177777. (overflow עולה ל1). עכשיו תבוצע הלולאה 65535 פעמים, ורק אז r0 יהיה שווה ל0.

כדי להשתמש בsob כמו שצריך, בפעם הראשונה שאנחנו מגיעים לsob, על הרגיסטר להיות גדול ממש מ0. דוגמא תקינה:

```
. = torg + 2000
main:
        mov r0, #1
        clr r2

loop1:
        inc r2
        sob r0, loop1
```

(השורה המודגשת היא השורה ששונתה).

כלל לא נכנס ללולאה במקרה זה, אלא sob יקטין את r0 ב1, יראה שהוא שווה ל0, ואז הביצוע ימשיך בפקודה לאחר sob.

דוגמא – jsr

לכאורה נראה שjsr היא תמיד שתי מילים (מכיוון שכמעט תמיד אנו שמים תווית צמודה לפקודה jsr, למשל jsr L), אולם ישנם גם מקרים שjsr תתפוס רק מילה אחת. הנה דוגמא למקרה כזה:

```
.=torg+2000

main:

        mov #func, r3
        jsr pc, (r3)
        halt


func:
        mov #10, r4
        rts pc
```

בפקודה המודגשת אנו משתמשים בשיטת מיעון Mode 1 שאינה דורשת מילה נוספת.

דוגמא – jsr

מטרת הדוגמא: הצגת הבעיתיות בשימוש בsp כרגיסטר קישור.

```
. = torg + 1000
main:
        mov #main, sp                   ; init stack, sp = 1000
        jsr sp, subr
        halt
subr:   rts sp
```

jsr sp, subr

כשנריץ פקודה זו יתרחש:
sp יקטן ל776. לאחר מכן jsr ידחוף את 776 לתא 776.
לתוך pc יוכנס הערך 1012 (הכתובת של subr), ומיד אחר כך בתוך sp יוצב 1010, שהוא כתובת החזרה מהשיגרה subr.

כאן אנו רואים את הבעייתיות של השימוש בsp עם jsr, מכיוון שראש המחסנית שונה במהלך המעבר לפונקציה.

כאשר הפקודה הנ"ל תבוצע:

rts pc

הערך של sp (1010) יושם לתוך pc. לאחר מכן יתבצע pop מהמחסנית. הכתובת 1010 (ראש המחסנית) תקרא. בכתובת 1010 נמצאת הפקודה halt שהopcode שלה הוא 000000. הערך 000000 יושם בראש המחסנית, ולאחר מכן יקודם ב2.
הערך הסופי של sp יהיה 2.

חלוקת מספרים גדולים – div

נביט במצב הבא:

רגיסטר	ערך
r1	1
r2	000000
r3	177777

ונרצה להריץ את הפקודה כדי לחלק את המספר שב-r3.

```
div r1, r2
```

המספר ששמנו ב-r2|r3 הוא 65535.
לאחר החילוק התוצאה אמורה להישמר ב-r2 והשארית ב-r3, אולם זה מצב הזיכרון בתום הקריאה:

רגיסטר	ערך
r1	1
r2	000000
r3	177777

נראה כאילו לא בוצע דבר.
מה קרה? האם div לא יכלה לחלק את המספר הגדול?
div כן יכולה לחלק מספרים גדולים.
הבעיה היא הבנת הפקודה במקרה זה: div מסוגלת לקבל מספרים בין (2^31-1)..-2^31 אבל הביטוי 65535/1 אינו בתחום 32767..-32768 בו עליו להימצא כדי שניתן יהיה לבטא אותו כמספר בעל סימן ב-r2.
במקרה כזה div אינה ביצעה דבר, מלבד העלאת ערכו של דגל ה-overflow ל-1.

דוגמאות – גישות לזיכרון

בדוגמאות הבאות יובאו קטעים קצרים של קוד. בכל קטע, ננתח אילו גישות לזיכרון יתבצעו כאשר הסימולטור יריץ את הקטע.

דוגמא 1

```
                                1
000000                          2       . = torg + 1000
001000                          3       main:
001000    012700 001000         4          mov #1000, r0
001004    022747 011001         5          cmp #011001, -(pc)
001010    000000                6          halt
                                7
```

גישות לזיכרון של הקטע הנ"ל:

1000 Read	1002 Read	1004 Read	1006 Read	1006 Read	1006 Read	1000 Read	1010 Read

בתחילה תיקרא כתובת 1000, הפקודה הראשונה. שיטת המיעון של האופרנד הראשון הוא Mode 2 Register 7 ולכן תיקרא גם הכתובת 1002 על מנת לחשב את האופרנד הראשון.
לאחר מכן תיקרא הכתובת 1004, שוב האופרנד הראשון הוא בשיטת מיעון Mode 2 Register 7 ולכן תיקרא המילה 1006. הPC בשלב זה הוא 1010. שיטת המיעון של האופרנד השני היא Mode 4. PC יוקטן ב2, וכעת ערכו יהיה 1006. שוב תיקרא המילה ב1006, על מנת לחשב את האופרנד השני.
מחזור הפקודה השלישי מתחיל. הPC הוא 1006 ולכן המילה 011001 תיקרא כopcode.
011001 זהו הopcode של הפקודה mov (r0), r1. מכיוון שתוכנו של r0 הוא 1000, תיקרא הכתובת 1000. לאחר מכן תיקרא הפקודה בכתובת 1010 וריצת התוכנית תסתיים.

דוגמא 2

```
                                    1
000000                              2  . = torg + 1000
001000                              3  main:
001000  066770 000774 000004        6    add 2000, @4(r0)
001006  000000                      7    halt
```

נתונים ערכי הרגיסטרים והכתובות הבאות:

רגיסטר/כתובת	ערך
r0	1000
#2000	1000

נביט בגישות לזיכרון של הקטע הנ"ל:

1000 Read	1002 Read	2000 Read	1004 Read	1004 Read	4 Read	4 Write	1006 Read

בתחילה תיקרא הפקודה בכתובת 1000.
שיטת המיעון של האופרנד הראשון היא שיטת מיעון יחסית, ולכן תיקרא הכתובת 1002 על מנת לחשב את הכתובת האפקטיבית של האופרנד. לאחר מכן, יקרא ערך האופרנד מהכתובת 2000.
שיטת המיעון של האופרנד השני היא אינדקס עקיף. לפיכך, תיקרא המילה בכתובת 1004 על מנת לחשב את הכתובת האפקטיבית. בתא 1004 יושב הערך 4. ערך זה יוסף לערך של r0, שהוא 0 ולכן הכתובת ממנה נקרא הינה 1004. נקרא שנית את הערך המצוי ב1004. הערך הנקרא הוא 4, ולכן זוהי הכתובת האפקיטבית של האופרנד השני. נקרא את תוכנו של תא 4, ולאחר מכן נכתוב אליו את הערך החדש לאחר ביצוע פקודת add.

דוגמא 3

מה יהיה תוכן sp בסוף ריצת הקוד?

```
. = torg + 1000
main:
        mov #1000, sp
        mov (sp)+, sp
        halt
```

בסוף ריצת התוכנית בsp יהיה 012706, (תוכן התא 1000).

נשאל את אותה שאלה לגבי הקוד הבא:

```
. = torg + 1000
main:
        mov #1000, sp
        mov (sp)+, (sp)
        halt
```

כעת sp יהיה 1002. אנו מקדמים את sp רק פעם אחת, בזמן חישוב האופרנד הראשון.

נספח ב' – קבועים וערכים בPDP11

PSW			
8-15	5-7	4	0-3
	עדיפות	T_Bit	Condition Codes NZVC

התקן	מילת סטטוס	מילת Buffer	עדיפות חומרה	וקטור פסיקה
לוח מקשים	TKS = 177560	TKB = 177562	4	60, 62
מדפסת	TPS = 177564	TPB = 177566	4	64,66
שעון	LCS = 177546	none	6	100,102

וקטורי פסיקה	
trap	34/36
emt	30/32
bpt	14/16
iot	20/22

מבנה הפקודה של trap / emt:

8-15	0-7
Opcode	פרמטר - n

Error Traps			
Error	PRIO	Order	Vector
Odd Address	8	0	4/6
Stack Violation (sp <= 376)	8	0	4/6
Stack Warning (sp <= 420)	8	1	4/6
Illegal Opcode	8	2	10/12
Power Failure	8	3	24/26

טבלת RLD			
תווית חיצונית (מוגדרת על ידי extern)		תווית מקומית	
מיעון אבסולטי	מיעון יחסי	מיעון אבסולוטי	מיעון יחסי
.extern x tst x+2(r2)	jsr pc, sub	Mov list(r0), r0	mov r0, res
תיקון אחד: ExtLbl, +, Rel LC	שני תיקונים: CrntCSect, -, Rel LC ExtLbl, +, Rel LC	תיקון אחד: CrntCSect, +, Rel LC	אין צורך בפעולה

אוגר בקרה – מקלדת						
12-15	11	8-10	7	6	1-5	0
	BUSY (r)		DONE (r)	INT ENABLE (w)		RE (w)

אוגר בקרה – מסך			
15 8	7	6	5 0
	READY	INT ENABLED	

אוגר נתונים	
8-15	0-7
Unused	Data

נספח ג' – שיטות מיעון בPDP11

Mode	שם	תחביר	חישוב כתובת אפקטיבית
0	רגיסטר	Rn	-
1	רגיסטר עקיף	@Rn or (Rn)	EA = Rn
2	הגדלה אוטומטית ישיר	(Rn)+	EA = Rn Rn += 2
2,7	מיידי	#A	EA = PC PC += 2
3	הגדלה אוטומטית עקיף	@(Rn)+	EA = fetch(Rn) Rn += 2
3,7	מוחלט (אבסולוטי)	@#A	EA = A PC += 2
4	הקטנה אוטומטית ישיר	-(Rn)	Rn -= 2 EA = Rn
5	הקטנה אוטומטית עקיף	@-(Rn)	Rn -= 2 EA = fetch(Rn)
6	אינדקס (אבסולוטי)	X(Rn)	X = fetch(PC) PC += 2 EA = X + Rn
6,7	יחסי	A	EA = X + Rn
7	אינדקס עקיף (אבסולוטי)	@X(Rn)	X = fetch(PC) PC += 2 EA = X + fetch(Rn)
7,7	יחסי עקיף	@A	X = fetch(PC) PC += 2 EA = fetch(X+Rn)

ב-mode 2, עבור פקודות שמטפלות בbytes, מלבד כאשר n = 6 או 7, ההגדלה אוטומטית היא ב-1. ב-mode 4, באותם תנאים, ההקטנה האוטומטית היא ב-1.