ראינו את צורת מימוש הפרמטרים ב-C, כאן נשלים את התמונה בכל הקשור למימוש משתנים. אנחנו נמשיך להתרכז במודל SMALL, אבל התמונה אינה שונה באופן מהותי במודלים האחרים.

יש עוד שני סוגים עיקריים של משתנים מלבד פרמטרים: משתנים סטטיים ואוטומטיים. ב-C משתנים גלובליים מממומשים באותה צורה כמו משתנים אוטומטיים וסטטיים בהתאם להכרזה על המשתנה, ועל משתני אוגר נדבר בהמשך.

כאשר מלמדים את שפות העילית בדרך כלל נותנים את התאור הבא:

"משתנים אוטומטיים של פונקציה הם משתנים שמקבלים הקצאה עם הקריאה לפונקציה ומשתחררים עם החזרה ממנה. במידה והמשתנה הוא משתנה לוקלי מאותחל, האתחול מתבצע בכל פעם מחדש.

משתנים סטטיים הם משתנים שמוקצים פעם אחת לכל אורך התוכנית והם קיימים לכל זמן הריצה של כל התוכנית, גם אם מדובר במשתנה לוקלי (להבדיל מגלובלי). יחד עם זאת, עבור משתנה סטטי לוקלי, רק הפונקציה שהגדירה את המשתנה יכולה לגשת אליו לפי השם שלו."

המשתנים שהגדרנו עד עכשיו תחת ה- DATA. הם משתנים סטטיים. במידה והמשתנים מאותחלים האתחול מתבצע עם העתקת קובץ ה-EXE מהדיסק לזיכרון. לפיכך כאשר אנחנו מגדירים בתוכנית

```
.DATA
Var1   DW  3201
```

אזי המילה Var1 הוא במושגים של C משתנה סטטי. הערך 3201 מופיע איפוא שהוא בקובץ ה-EXE. הערך 3201 מועתק לזיכרון עם העלאת התוכנית לזיכרון. זה יהיה האתחול היחיד של המשתנה Var1. כל הצבה לתוך Var1 לא יתבטל אלא ע"י הצבה אחרת.

אשר למשתנים האוטומטיים, הם מיושמים במחסנית בצורה דומה מאד לפרמטרים. כל פונקצית TURBO C במודל SMALL מישמת את הסכמה שתתואר להלן. בשלב זה נניח שהקומפילר מיצר קוד שאינו משתמש במשתני אוגר (למשל tcc -r-). נתאר את ההשלכות של ישום משתני אוגר מאוחר יותר.

מימוש הסכמה היא כלהלן:

הפקודות הראשונות שמבצעת כל פונקציה של C הם:

```
_myfunc PROC NEAR
PUSH BP     "ישן"  BP  שימור
MOV  BP,SP  "ישן"  BP  מצביע על  BP
SUB  SP,k   הקצאת שטח משתנים לוקליים
            k הוא גודל שטח המשתנים הלוקליים
```

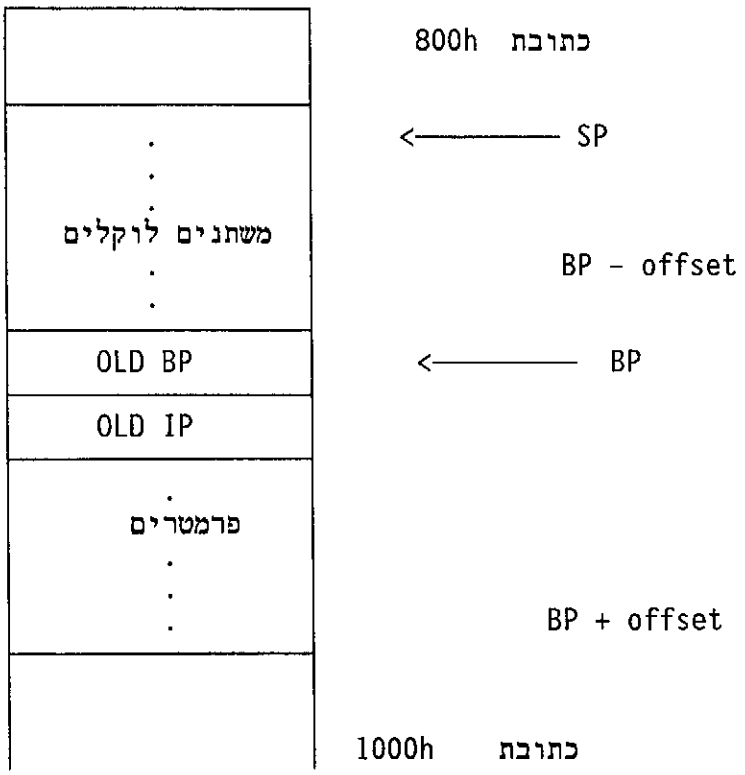
מרגע זה המשתנים הלוקליים קיימים וניתן לגשת אליהם באמצעות BP עם תוספת שלילית. במידה ויש פקודת אתחול למשתנים האוטומטיים הם יופיעו כאן. לדוגמא, אתחול משתנה לוקלי - אוטמטי בגודל 16 ביט ב-7 עשוי להראות כמו

```
MOV  WORD PTR [BP-6],7
```

כאשר הפונקציה רוצה <u>לחזור</u> (תמיד בסוף הפונקציה) לתוכנית הקוראת הוא מבצעת את סדרת הפקודות הבאה:

```
   MOV SP,BP  שחרור שטח משתנים לוקליים
   POP BP     BP שחזור
   RET        חזרה לתוכנית קוראת
_myfunc ENDP
```

לפיכך סכמת המשתנים של פונקצית C במודל SMALL נראית באופן כללי כך:



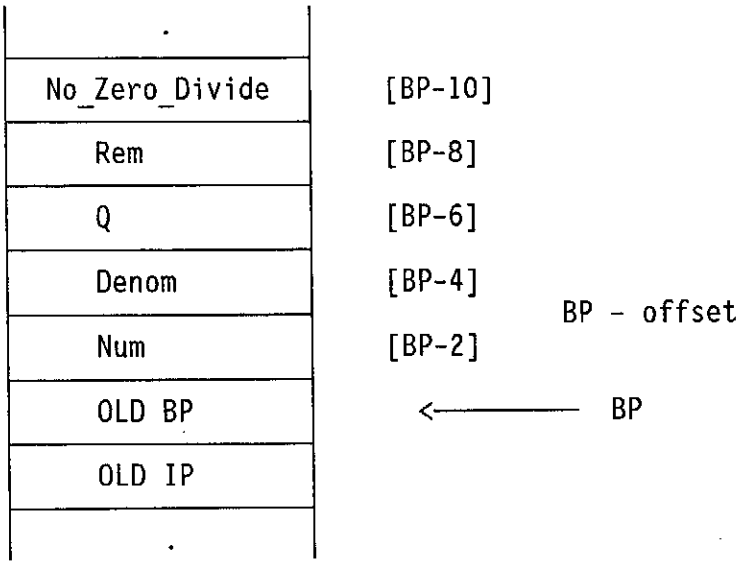
לדוגמא, בתוכנית call_idl.c, הפונקציה main (שלצורך ניהול משתנים היא כמו כל פונקציה אחרת) המשתנים הלוקליים מוגדרים בשורה

```
int Num, Denom, Q, Rem, No_Zero_Divide;
```

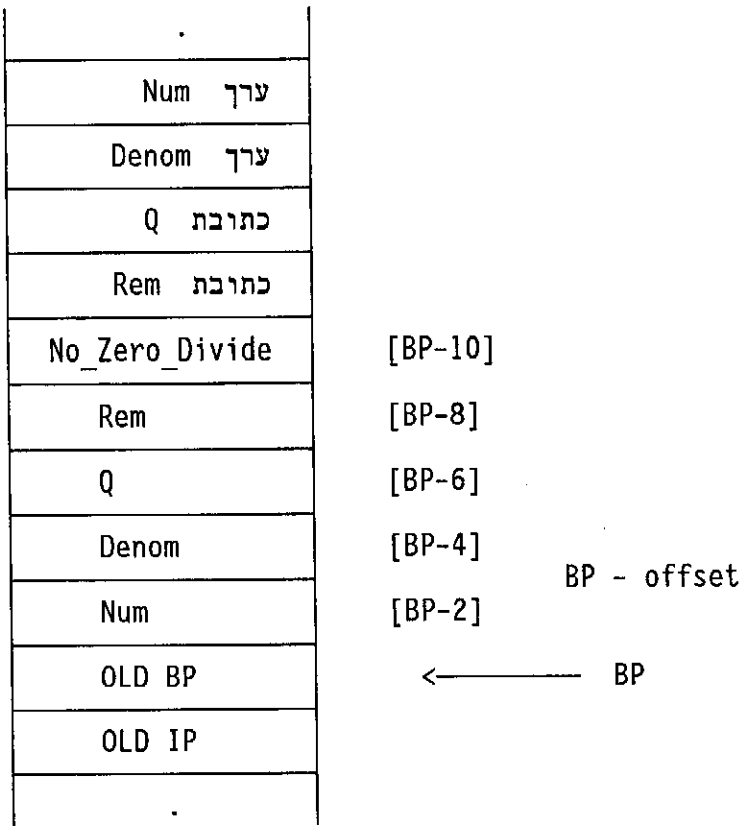
המתרגם ל-

```
SUB SP,10
```

כאשר בפועל



לפיכך אם נסתכל על המחסנית ברגע ההסתעפות לרוטינה _idiv_mod, כלומר לאחר שמירת הפרמטרים במחסנית, לפני ביצוע הפקודה call near ptr _idiv_mod, המחסנית נראית כך:



כאן אנחנו רואים צד נוסף במימוש המושג by value parameters של C:
הפרמטרים הם למעשה שטח מיוחד (במחסנית) המוקצה לרוטינה המכילים עותקים וכתובות של המשתנים של הרוטינה המקורית. הכנסת שינוי ישירות בתוכנם אינו משנה את ערכי המשתנים של התוכנית הקוראת, הללו נמצאים בעומק רב יותר במחסנית (בכתובות גבוהות יותר). לשון אחר: הכנסת שינוי במה שמצוין לעיל

כ-"ערך Num" לא יגרום לשום שינוי בשטח הזיכרון המצוין לעיל "Num".

משתני אוגר

ב-C ישנו מושג של משתני אוגר. משתני אוגר הם מצב שבו חלק מהמשתנים האוטומטיים הנוכחיים מיושמים לא בזיכרון אלא באוגרים. ב- TURBO C הדבר בא לידי ביטוי רק במימוש 2 משתנים מסוג int או פוינטרים ע"י האוגרים SI ו-DI בלבד. צריך לזכור שמדובר בקומפילר מעידן ה-8086. במידה ותכנת ה-C ציין איזה משתנים הוא מעונין שימומשו כמשתני אוגר ע"י המילה השמורה register, הם ימומשו כמשתני אוגר (במידה והדבר אפשרי). אחרת, שני המשתנים הראשונים שמתאימים (int או פוינטר) ימומשו כמשתני אוגר.

ההבדלים בסכמה יהיו שלא כל המשתנים יוקצו ע"י פקודת ה-SUB ולסכמה יתווספו פקודות שימור / שיחזור אוגרי ה- SI וה- DI. לפיכך הסכמה תיראה כך:

הפקודות הראשונות שמבצעת כל פונקציה של C הם:

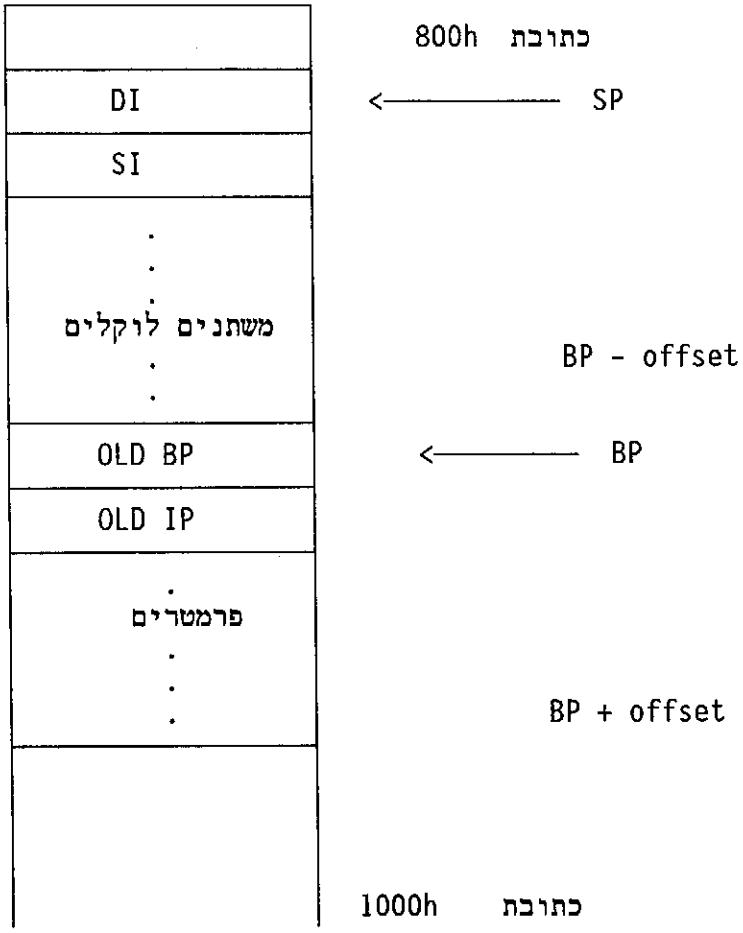
```
_myfunc PROC NEAR
  PUSH BP     "ישן"  BP  שימור
  MOV  BP,SP  "ישן"  BP  מצביע על  BP
  SUB  SP,k   הקצאת שטח משתנים לוקליים
              k הוא גודל שטח המשתנים הלוקליים
                          לא כולל משתני אוגר
  PUSH SI
  PUSH DI
```

כאשר הפונקציה רוצה <u>לחזור</u> לתוכנית הקוראת היא מבצעת את סדרת הפקודות הבאה:

```
  POP DI        שיחזור אוגרים
  POP SI

  MOV SP,BP   שחרור שטח משתנים לוקליים
  POP BP      BP שחזור
  RET         חזרה לתוכנית קוראת
_myfunc ENDP
```

לפיכך סכמת המשתנים של פונקצית C במודל SMALL נראית באופן כללי כך:



מימוש המשתנים אוטומטיים בתוכניות אסמבלי

אין שום מניעה, כמובן, לממש משתנים במשתנים אוטומטיים גם בתוכניות הנכתבות (ישירות) באסמבלי. אולם זה מפחית את הקריאות של התוכניות ובאופן כללי, לרוב זה פשוט לא טבעי ולא נוח לתכנת כך. בדרך כלל נעשה זאת רק כאשר יש בכך חסכון משמעותי, כמו מימוש מערכים זמניים.

קומפילציה tcc -S ותוכנית הדוגמא call_id1.asm

call_id1.asm הינו קובץ האסמבלי שמתקבל מקימפול התוכנית call_id1.c ע"י האופציות

```
tcc -S -r- call_id1.c
```

כאשר האופציה S- מנחה את הקומפילר לתרגם את התוכנית לקובץ אסמבלי (call_id1.asm) במקום לקבצי obj ו-exe שהקומפילר בדרך כלל מיצר. האופציה r- מנחה את הקומפילר לא להשתמש במשתני אוגר. האופציה הזו נבחרה בשביל להקל על הבנת התוכנית.

חלקי התוכן של call_id1.asm החשובים לנו נסקרו קודם לכן תחת הכותרת "סכמת ניהול המשתנים של TURBO C".

האופציה S- נתמכת ברוב הקומפילרים של C. יש לו מספר שימושים חשובים. תוכניתן אסמבלי שיש לו משימה מורכבת יכול בהחלט להסתיע בידיעת איך הקומפילר ממש אותו. בנוסף, תוכניתן בשפה C המפתח תוכנה שבו הוא מכיר את שפת האסמבלי יכול ע"י עיון בקובץ האסמבלי הנוצר לאתר בעיות הכרוכות בקימפול שונה מהצפוי של תוכנית המקור. זה יכול להוביל לאיתור שגיעות הנובעות מפרשנות שונה מהצפוי של הקוד ע"י הקומפילר או איתור סיבות לאיטיות של קוד איטי מהצפוי בקובץ ה-exe. אפשר אפילו "לשפר" את ה-exe הנוצר ע"י הכנסת שינויים בקובץ ה-asm והכללותו בקימפול במקום קובץ המקור ב-C.

call_id1.asm

tcc -S -r- call_id1.c

```
        ifndef  ??version
?debug  macro
        endm
$comm   macro   name,dist,size,count
        comm    dist name:BYTE:count*size
        endm
        else
$comm   macro   name,dist,size,count
        comm    dist name[size]:BYTE:count
        endm
        endif
        ?debug  S "call_id1.c"
        ?debug  C E9857A13270A63616C6C5F6964312E63
        ?debug  C E90018521815433A5C54435C494E434C5544455C737464696F2E68
        ?debug  C E90018521815433A5C54435C494E434C5544455C5F646566732E68
        ?debug  C E90018521815433A5C54435C494E434C5544455C5F6E756C6C2E68
_TEXT   segment byte public 'CODE'
_TEXT   ends
DGROUP  group   _DATA,_BSS
        assume  cs:_TEXT,ds:DGROUP
_DATA   segment word public 'DATA'
d@      label   byte
d@w     label   word
_DATA   ends
_BSS    segment word public 'BSS'
b@      label   byte
b@w     label   word
_BSS    ends
_TEXT   segment byte public 'CODE'
   ;
   ;     void main()
   ;
        assume  cs:_TEXT
_main   proc    near
        push    bp
        mov     bp,sp
        sub     sp,10
   ;
   ;        {
   ;          int Num, Denom, Q, Rem, No_Zero_Divide;
   ;
   ;          printf("\Enter Numerator, Denominator\n:");
   ;
        mov     ax,offset DGROUP:s@
        push    ax
        call    near ptr _printf
        pop     cx
```

```
;
;           scanf("%d %d",&Num, &Denom);
;
        lea     ax,word ptr [bp-4]
        push    ax
        lea     ax,word ptr [bp-2]
        push    ax
        mov     ax,offset DGROUP:s@+31
        push    ax
        call    near ptr _scanf
        add     sp,6
;
;           No_Zero_Divide = idiv_mod(Num,Denom,&Q,&Rem);
;
        lea     ax,word ptr [bp-8]
        push    ax
        lea     ax,word ptr [bp-6]
        push    ax
        push    word ptr [bp-4]
        push    word ptr [bp-2]
        call    near ptr _idiv_mod
        add     sp,8
        mov     word ptr [bp-10],ax
;
;           if  (No_Zero_Divide)
;
        cmp     word ptr [bp-10],0
        je      short @1@86
;
;             printf("\n %d div %d = %d, mod(%d,%d) = %d\n",
;
;
;                         Num, Denom, Q, Num, Denom, Rem);
;
        push    word ptr [bp-8]
        push    word ptr [bp-4]
        push    word ptr [bp-2]
        push    word ptr [bp-6]
        push    word ptr [bp-4]
        push    word ptr [bp-2]
        mov     ax,offset DGROUP:s@+37
        push    ax
        call    near ptr _printf
        add     sp,14
        jmp     short @1@114
```

```
@1@86:
   ;
   ;           else
   ;              printf("\nError: Zero Divide.\n");
   ;
        mov     ax,offset DGROUP:s@+72
        push    ax
        call    near ptr _printf
        pop     cx
@1@114:
   ;
   ;         } /* main */
   ;
        mov     sp,bp
        pop     bp
        ret
_main   endp
        ?debug  C E9
_TEXT   ends
_DATA   segment word public 'DATA'
s@      label   byte
        db      'Enter Numerator, Denominator'
        db      10
        db      ':'
        db      0
        db      '%d %d'
        db      0
        db      10
        db      ' %d div %d = %d, mod(%d,%d) = %d'
        db      10
        db      0
        db      10
        db      'Error: Zero Divide.'
        db      10
        db      0
_DATA   ends
_TEXT   segment byte public 'CODE'
_TEXT   ends
        public  _main
        extrn   _idiv_mod:near
        extrn   _scanf:near
        extrn   _printf:near
_s@     equ     s@
        end
```

רקורסיה ותוכנית הדוגמא fibo4.asm

fibo4.asm הינו קובץ האסמבלי שמתקבל מקימפול התוכנית fibo4.c ע"י האופציות

```
tcc -S -r- fibo4.c
```

כאשר האופציה S- מנחה את הקומפילר לתרגם את התוכנית לקובץ אסמבלי (fibo4.asm) במקום לקבצי obj ו-exe שהקומפילר בדרך כלל מיצר. האופציה r- מנחה את הקומפילר לא להשתמש במשתני אוגר. האופציה הזו נבחרה בשביל להקל על הבנת התוכנית.

תפקידה של הדוגמא למחיש את מימוש מושג הרקורסיה ברמת הקומפילציה.

החשיבות של הנושא הזה בפרק זה הוא להמחיש שכאשר ממשים משתנים / פרמטרים וכתובות חזרה דרך המחסנית כפי ש-Turbo C ורוב הקומפילרים של C עושים, מימוש קוד רקורסיבי הוא אופציה המתקבלת **בחינם** או שהצורך של התחשבות נוספת במימוש קוד רקורסיבי **מינימלי ביותר**. אם נעיין במימוש של הקוד שנפרש על מנת לממש את הקריאה

```
fibo(n-2)
```

המימוש שלו באסמבלי יהיה

```
mov ax,word ptr [bp+4]
sub ax,2
push ax
call near ptr _fibo
```

לפיכך הקריאה הרקורסיבית אינה שונה במאומה מקריאה לפונקציה חיצונית. יש הבדל לוגי בכך שההסתעפות היא לראשות הפונקציה עצמה ולא לכתובת שמחוץ לרוטינה אבל זה לא בא לידי ביטוי בטקסט של הקוד שנוצר. לשון אחר: מתכנת שהיה צריך לממש את הרקורסיה ידנית באסמבלי לא צריך לדעת יותר מהמוסכמות הרגילות של מימוש פונקציות ב-C.

העובדה שמימוש רקורסיה מתקבלת כתוספת חינם למוסכמות מימוש פונקציות ב-C לא היה, כנראה, הסיבה העיקרית להגדרת בצורתן מבוססת המחסנית. סביר להניח שהמניע העיקרי היה לקבל אפקט זהה (או כמעט זהה, תלוי בהשקפה) של הרצת קוד במקביל במימוש מושג התהליכים, נושא מתקדם שלא נכנס לו כאן. למעשה מושג "מחסנית מערכת" הקיימת בכל הארכיטקטורות של מחשבים מודרניים הומצאה על מנת לקבל את האפקט הזה, הנקרא Re-entrancy. אנחנו נומר שהסיבה שהאפקט מתקבל היא מאותה סיבה שהמוסכמות משרתות גם מימוש תהליכים.

הסיבה שמוסכמות הללו תומכות ברקורסיה (או ב-Re-entrancy באופן כללי) היא בכך שהם ממשות את מנגנוני המשתנים, הפרמטרים ושימור כתובות הסתעפות עתידיות בתוך שטחי זיכרון דינמיים שמשתחררים בסדר של "אחרון נכנס ראשון יוצא".

על מנת לממש קריאה רקורסבית צריך לממש רובד חדש של פרמטרים ומשתנים לוקליים תוך שימור הערכים הקודמים והסתעפות לראשית הרוטינה תוך שימור כתובת החזרה. על מנת לממש חזרה מרקוסיה צריך לחזור חזרה לפקודה שמעבר לקריאה הרקורסיבית האחרונה (או מעבר לקריאה המקורית) ולשחרר את רובד המשתנים הלוקליים והפרמטרים האחרון תוך שחזור הקודם לו. את כל אלה המוסכמות עושות ממילא מעצמם.

```
/* fibo4.c - implement Fibonacci numbers - naive recursion  */

#include <stdio.h>

unsigned long int fibo(unsigned int n)
{
  unsigned long int x, y, z;

  if  ( n <= 2 )
     return 1L;
  else
    x = fibo(n-1);
    y = fibo(n-2);
    z = x + y;
    return z;
}

 void main()
  {
    unsigned int n;
    printf("Enter an integer:\n");
    scanf("%d",&n);

    printf("Fibonacci(%u) = %lu\n", n, fibo(n));

  }
```

```
E:\>fibo4.exe
Enter an integer:
10
Fibonacci(10) = 55

E:\>
```

fib04.asm

tcc -S -r- fib4.asm

```
_TEXT   segment byte public 'CODE'
_TEXT   ends
DGROUP  group   _DATA,_BSS
        assume  cs:_TEXT,ds:DGROUP
_DATA   segment word public 'DATA'
d@      label   byte
d@w     label   word
_DATA   ends
_BSS    segment word public 'BSS'
b@      label   byte
b@w     label   word
_BSS    ends
_TEXT   segment byte public 'CODE'
   ;
   ;    unsigned long int fibo(unsigned int n)
   ;
        assume  cs:_TEXT
_fibo   proc    near
        push    bp
        mov     bp,sp
        sub     sp,12
   ;
   ;    {
   ;      unsigned long int x, y, z;
   ;
   ;      if  ( n <= 2 )
   ;
        cmp     word ptr [bp+4],2
        ja      short @1@142
   ;
   ;          return 1L;
   ;
        xor     dx,dx
        mov     ax,1
@1@86:
        jmp     short @1@198
        jmp     short @1@170
@1@142:
   ;
   ;      else
   ;        x = fibo(n-1);
   ;
        mov     ax,word ptr [bp+4]
        dec     ax
        push    ax
        call    near ptr _fibo
        pop     cx
        mov     word ptr [bp-2],dx
        mov     word ptr [bp-4],ax
@1@170:
   ;
   ;        y = fibo(n-2);
   ;
        mov     ax,word ptr [bp+4]
        sub     ax,2
        push    ax
        call    near ptr _fibo
        pop     cx
        mov     word ptr [bp-6],dx
        mov     word ptr [bp-8],ax
```

```
   ;
   ;         z = x + y;
   ;
        mov     ax,word ptr [bp-2]
        mov     dx,word ptr [bp-4]
        add     dx,word ptr [bp-8]
        adc     ax,word ptr [bp-6]
        mov     word ptr [bp-10],ax
        mov     word ptr [bp-12],dx
   ;
   ;         return z;
   ;
        mov     dx,word ptr [bp-10]
        mov     ax,word ptr [bp-12]
        jmp     short @1@86
@1@198:
   ;
   ;     }
   ;
        mov     sp,bp
        pop     bp
        ret
_fibo   endp
   ;
   ;      void main()
   ;
        assume  cs:_TEXT
_main   proc    near
        push    bp
        mov     bp,sp
        sub     sp,2
   ;
   ;       {
   ;         unsigned int n;
   ;         printf("Enter an integer:\n");
   ;
        mov     ax,offset DGROUP:s@
        push    ax
        call    near ptr _printf
        pop     cx
   ;
   ;         scanf("%d",&n);
   ;
        lea     ax,word ptr [bp-2]
        push    ax
        mov     ax,offset DGROUP:s@+19
        push    ax
        call    near ptr _scanf
        pop     cx
        pop     cx
```

```
   ;
   ;
   ;        printf("Fibonacci(%u) = %lu\n", n, fibo(n));
   ;
        push    word ptr [bp-2]
        call    near ptr _fibo
        pop     cx
        push    dx
        push    ax
        push    word ptr [bp-2]
        mov     ax,offset DGROUP:s@+22
        push    ax
        call    near ptr _printf
        add     sp,8
   ;
   ;
   ;      }
   ;
        mov     sp,bp
        pop     bp
        ret
_main   endp
        ?debug  C E9
_TEXT   ends
_DATA   segment word public 'DATA'
s@      label   byte
        db      'Enter an integer:'
        db      10
        db      0
        db      '%d'
        db      0
        db      'Fibonacci(%u) = %lu'
        db      10
        db      0
_DATA   ends
_TEXT   segment byte public 'CODE'
_TEXT   ends
        public  _main
        public  _fibo
        extrn   _scanf:near
        extrn   _printf:near
_s@     equ     s@
        end
```

גירסאות מיוחדות של הפקודה CALL

השימושים בפקודה CALL שראינו עד עכשיו היו מהסוג של CALL לנקודה בזיכרון הנמצא בסגמנט קוד - יותר נכון label המצביע על פקודה. בגירסה הזו אוגר ה-IP או זוג האוגרים CS ו-IP משנים את ערכם להצביע על הנקודה הזו (תוך שימור כתובת החזרה במחסנית). זו אכן הגירסה הנפוצה ביותר של הפקודה CALL. אבל ישנם גירסאות נוספות.

ב-8086 היה קיים גירסה של CALL עם אוגר 16 ביט. המשמעות שלו היא להציב את ערך האוגר לתוך IP. לדוגמא, הפקודה

```
CALL BX
```

היתה גורמת לשימור IP והצבת הערך של האוגר BX לתוך IP, מעין השמה IP = BX.

שימוש בפקודה CALL להסתעפות עקיפה - מימוש פוינטר לפונקציה

אחד הגירסאות של הפקודה CALL היא הסתעפות עקיפה דרך משתנה בזיכרון. במקרה הזה האופרנד של הפקודה CALL הוא לא כתובת של פקודה אלא כתובת של משתנה. במקרה הזה, הכתובת בזיכרון איננו מגדיר את הערך החדש של CS:IP אלא היכן נמצא הערך החדש בזה. לסוג הזה של פקודת CALL יש גירסת NEAR וגירסת FAR. לדוגמא, נניח ששטח ה-DATA של תוכנית מכילה את ההגדרות הבאות:

```
f_ptr1  DW  100h
f_ptr2  DD  20003000h
```

אזי הפקודה

```
CALL f_ptr1
```

יגרום להסתעפות מסוג NEAR כאשר מלבד שימור הערך הנוכחי של IP, IP יקבל את הערך 100h. לעומת זאת, הפקודה

```
CALL f_ptr2
```

יגרום להסתעפות מסוג FAR כאשר מלבד שימור הערכים הנוכחים של של IP ושל CS, CS יקבל את הערך 2000h ו- IP יקבל את הערך 3000h.

אפשר גם פקודות מהצורה הבאה:

```
CALL WORD PTR [BP+6]
CALL DWORD PTR [BX-4]
```

הסוג הזה של פקודות CALL משמש למימוש המושג ב-C הנקרא פוינטר לפונקציה.

אגב, במחשבים רבים יש גם גירסאות עקיפות לפקודה המקבילה MOV. אין גירסה כזו במחשב הזה, אבל יש פקודות הסתעפות עקיפות.

386 ואילך.

בנוסף לפקודות המצוינות לעיל יש ב-386 גירסות נוספות של הפקודה CALL התומכות בהיסטים 32 ביט.

קימות פקודות הסוג

```
CALL ECX
```

שהמשמעות שלה היא שימור EIP והצבת הערך של ECX לתוך EIP.

כמו כן ישנם פקודות הסתעפות עקיפה NEAR 32 ביט (שינוי EIP בלבד) וכן הסתעפיות עקיפות 48 ביט. משתנים כאילו נקראים Full Pointer או FWORD. להגדרת משתנים בשטח המידע משתמשים בהנחיה DF או DP.

לפיכך ניתן לראות ב-386 פקודות כמו

```
f_ptrl  DF 605040302010h

  ......

CALL  f_ptrl
```

יגרום לשמירת CS ו-EIP והצבת

CS = 6050h ו- EIP = 40302010h.

```
/* pf.c - Use pointer to function */


int sqr(int x)
{
  return x*x;
} /* sqr */

int neg(int x)
{
  return -x;
} /* neg */

void main()
{

  int x,y, z;

  int (*fp)(int);


  x = 5;

  fp = sqr;

  y = (*fp)(x);

  fp = neg;

  z = (*fp)(x);

  printf("\nx = %d, y = %d, z = %d\n", x, y, z);

} /* main */
```

```
E:\>pf


x = 5, y = 25, z = -5

E:\>
```