ביצוע השוואות במעבד המתמטי

אחד התפקידים של המעבד המתימטי הוא לתמוך בהשוואות בין מספרים ממשיים: כלומר לדעת בין שני מספרים (שלפחות אחד מהם, מן הסתם, ממשי) האם הם שווים, או האם הראשון גדול מהשני וכו'. התהליך אינו לחלוטין פשוט.

לפני שנכנס לפרטים של איך זה עובד ולמה זה נכון, להלן מרשם איך לבצע את השוואה בין שני מספרים.

הדרך המקובלת לבצע השוואה בין שני מספרים הוא כלהלן:

1. הבא לפחות את אחד המספרים לתוך אחד מאוגרי ה-ST(i).
2. השתמש באחד מפקודות ההשוואה (FCOMx, FICOMx)
3. בצע את הפקודת המכונה FSTSW AX
4. בצע את פקודת המכונה SAHF
5. במידה וקיימת אפשרות שהאופרנדים אינם ניתנים להשוואה בדוק זאת על ידי בדיקת ה-PF (על ידי הפקודות JP או JNP).
6. המשך כאילו בצעת את פקודת המכונה CMP על אופרנדים שלמים, אותם אתה מעונין לפרש כמספרים שלמים חסרי סימן.

לדוגמא, הקוד הבא משווה בין שני משתנים ממשיים בזיכרון Var1 ו-Var2 (32 או 64 ביט, נניח), כאשר הוא מציב לתוך CX את הערך 1 אם Var1 > Var2. לצידו ישנו קוד מקביל למצב שבו Var1 ו-Var2 היו משתנים שלמים חסרי סימן 32 ביט:

```
MOV CX,1

FLD Var1
FCOMP Var2
FSTSW AX
SAHF

JA  Skip1
  MOV CX,0
Skip1:
```

```
MOV CX,1

MOV EDX,Var1
CMP EDX,Var2

JA  Skip1
  MOV CX,0
Skip1:
```

שים לב בשימוש ב-JA ולא ב-JG.

מנגנון ההשוואה.

בתוך ה-Status Word קיימים ארבע ביטים הקרויים C0, C1, C2, C3.

כאשר מתבצע השוואה, הביטים C0, C2, C3 הם שמכילים את תוצאת ההשוואה.

C2 משקף מצב שבו האיברים שנעשו בהם השוואה אינם ניתנים להשוואה (ערכים מיוחדים של NaN, אינסוף)

C0 ו-C3 מכילים את תוצאות ההשוואה (בהנחה שהאופרנדים היו תקינים).

C3 הוא למעשה ה-Zero Flag של המעבד המתימטי: C3 = 1 פירושו שהאופרנים היו שווים.

C0 = 1 אם האופרנד השני (Var2 בדוגמא) גדול מהראשון. האופרנד הראשון הוא בדרך כלל ST(0).

לפיכך אפשר לסכם את משמעות הביטים C0, C2, C3 בטבלה הבאה:

C3	C2	C0	משמעות
0	0	0	אופרנד ראשון > אופרנד שני
0	0	1	אופרנד ראשון < אופרנד שני
1	0	0	אופרנד ראשון == אופרנד שני
1	1	1	אופרנדים בלתי ניתנים להשוואה

בפועל, בדרך כלל נתעלם מהנושא של אופרנדים "בלתי ניתנים להשוואה". בדרך כלל באחריות המתכנת לדאוג לכך שפקודות ההשוואה יופעלו רק על אופרנדים תקינים. כשמצמצמים את הטבלה האחרונה תוך ויתור על ה-PF והאפשרות של בלתי ניתנים להשוואה, מקבלים

C3	C0	משמעות
0	0	אופרנד ראשון > אופרנד שני
0	1	אופרנד ראשון < אופרנד שני
1	0	אופרנד ראשון == אופרנד שני

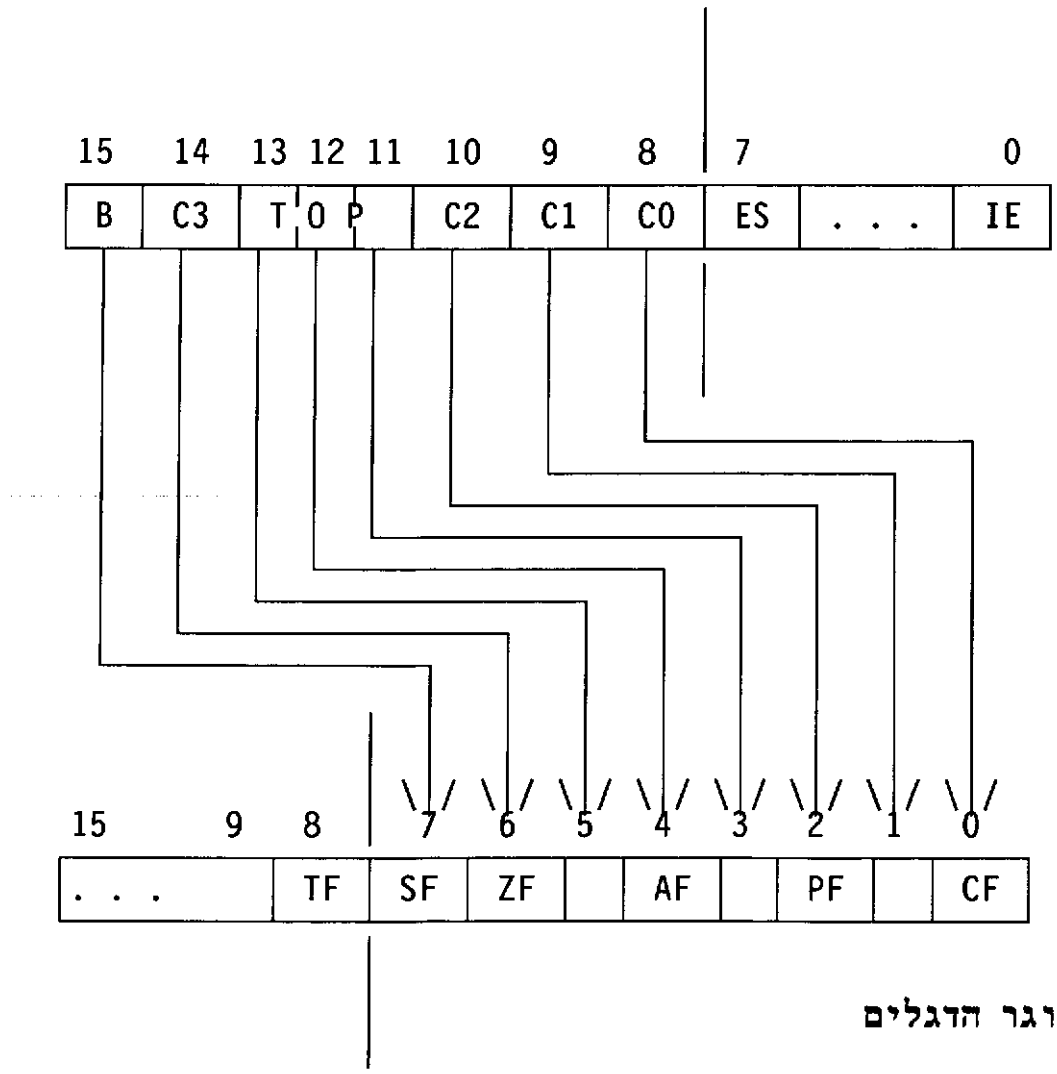
מה שממחיש את תסריט ש-C3 מתנהג כמו ZF ו-C0 מתנהג כמו carry (או יותר נכון borrow) בכל הקשור להשוואה בין מספרים ממשיים, כאילו היו מספרים חסרי סימן.

הפקודה FSTSW AX מעביר את תוכן ה-Status Word לתוך AX.

הפקודה SAHF מעתיקה את AH לתוך החלק התחתון של אוגר הדגלים. במידה ו-AX מכיל את תוכן ה-Status Word, משמעות הדבר יהיה ש- PF יקבל את הערך של C2, ZF יקבל את הערך של C3, ו-CF יקבל את הערך של C0.

המשמעות של פעולת ה- FSTSW AX + SAHF

ה-Status Word של המעבד המתמטי



אוגר הדגלים

כהערות אגב נוסיף את הפרטים הבאים על C0 - C3:

ל-C1 אין תפקיד בהשוואה בין מספרים.

ל-C1 יש מספר תפקידים. במקרים של ביצוע פעולה חוקית שמחייב עיגול (למשל חישוב 1/3) הערך שלו משקף את סוג העיגול שהתבצע בפועל (עיגול כלפי מעלה C1 = 1, כלפי מטה C1 = 0).

ל-C1 יש גם תפקיד בפענוח מקרה של Stack Fault (0 במקרה של Underflow, 1 במקרה של Overflow). כלומר תפקידו לאפשר הבדלה בין מצב שניסינו להכניס איבר תשיעי למחסנית (Stack Overflow) נניח ע"י FLD לבין מצב שניסינו לדוקן מהמחסנית יותר איברים משהיו בה (Stack Underflow) נניח ע"י FCOMPP כאשר היה רק ערך אחד במחסנית.

ל-C0, C1, C3 יש גם תפקיד מיוחד בפקודות FPREM ו-FPREM1: הם מכילים את הביטים הכי פחות משמעתיים של התוצאה (C2 מקבל 0 בפקודות הללו). לא נכנס לזה כאן.

סיכום

המנגנון מתנהל כך: ביצוע אחת מפקודות ה-FCOM קובע את ערכי ה- C0, C2, C3, הפקודה FSTSW AX מעבירה את הערכים ל-AX והפקודה SAHF מעבירה אותם משם לדגלים ZF = C3, CF = C0, PF = C2. בדיקת PF מאפשר לנו לגלות שהבדיקה היתה בין אופרנדים שאינם ניתנים להשוואה. במידה שזה לא היה המקרה (או שמראש אין אפשרות כזו) ZF = 1 משקף מצב של שיוון ובמידה ו-ZF = 0 אזי CF משקף מי מהאופרנדים גדול יותר: CF = 0 הראשון, CF = 1 השני. קצת אנליזה תראה שזה בדיוק המצב של השואות בין מספרים חסרי סימן, כאשר CF משקף אפשרות של Borrow של חיסור חסרי סימן.

כמובן שאפשר גם לבדוק את ערכי הביטים ישירות לאחר הפקודה FSTSW AX ע"י הפקודות TEST או AND. אחת מתוכניות הדוגמא שבהמשך מממשת את האפשרות הזו.

תוכניות דוגמא ccomp1.c, mc1.asm, ccomp2.c mc2.asm, ccomp3.c ו-mc3.asm

מטרת התוכניותת המשולבות הזו להמחיש את נושא השוואת 2 מספרים ממשיים במעבד המתמטי.

התוכניות הראשית הכתובה ב-C מקבלת 2 מספרים מהמשתמש כפרמטרים לתוכנית ראשית וקוראת לפונקציית האסמבלי comp להשוות ביניהם. comp מחזירה תוצאה שלמה 0 אם שניהם שווים, 1 אם הפרמטר הראשון גדול מהשני ו-1- אם הפרמטר השני גדול מהראשון. ההנחה היא ששני המספרים הממשיים תקינים, לא מביאים בחשבון את האפשרות של "לא ניתנים להשוואה". הזוג ccomp1.c ו-mc1.asm מישם את החישוב ב-float, הזוג ccomp2.c ו-mc2.asm מישם את החישוב ב-double והזוג ccomp3.c ו-mc3.asm מישם את החישוב ב-long double.

אשר למימוש של עצמם, הם נעשים לפי המרשם של השוואות מספרים במעבד המתמטי שתואר קודם. התוכניות mc1.asm ו-mc2.asm מנצלות את העובדה שניתן בפקודה FCOMP להשוות מספרים ממשיים 32 ו-64 ביט ישירות מהזכרון ואילו mc3.asm, מאחר והיא משווה 2 מספרים 80 ביט, חייבת לטעון את שניהם לאוגרי המעבד ומבצעת שם את הפקודה FCOMPP. שימו לב לסדר הטעינה של המספרים ב-mc3.asm. על מנת לשמר ב-mc3.asm את אותה פקודת הסתעפות (JBE) כמו ב-mc1.asm ו-mc2.asm, התוכנית טוענת למעבד קודם את המספר השני ורק אחריו את המספר הראשון. בתוכנית הדבר בא לידי ביטוי בשורות

```
FLD  TBYTE PTR [BP+14]  ;  ST(0) = p1
FLD  TBYTE  PTR [BP+4]
FCOMPP                  ; Compare p1 with p2, pop twice
```

זאת משום שהפקודה FCOMPP משווה את ST(0) לעומת ST(1).

מענין לעשות אנלוגיה למצב שבו נניח היינו רוצים לממש רוטינה דומה המחזירה אותה תשובה נניח אם היינו משוים 2 מספרים אי שליליים 32 ביט.

השוואה במעבד המתימטי		אנלוגיה לשלמים
FLD DWORD PTR [BP+4]	┬	MOV EAX,[BP+4]
FCOMP DWORD PTR [BP+8]	│	CMP EAX,[BP+8]
FSTSW AX	│	
SAHF	┴	

```
JBE Not_Above                    JBE Not_Above
   MOV AX,1                         MOV AX,1
   JMP SHORT Finish                 JMP SHORT Finish
Not_Above:                       Not_Above:
   JE Equal                         JE Equal
    MOV AX,-1                       MOV AX,-1
    JMP SHORT Finish                JMP SHORT Finish
Equal:                           Equal:
    MOV AX,0                        MOV AX,0
Finish:                          Finish:
    POP BP                          POP BP
    RET                             RET
```

הערה: "JMP SHORT" מנחה את האסמבלר לבחור את הגרסה של 127+ 128- של JMP. במקרה של הסתעפות קדימה זה מיעל במשהו את הקובץ הבינארי שנוצר. זה לא קשור בהכרח למעבד המתמטי. לא נכנס לזה כאן.

```
/* ccomp1.c  - compare real numbers */

#include <stdio.h>
#include <stdlib.h>

extern int comp( float p1, float p2 );

void main(int argc, char *argv[])
{
  float x, y;
  int cmp_flag;

  if (argc < 3)
    {
     fprintf(stderr,"Usage: ccomp1 real1 real2\n");
     exit(1);
    }

  sscanf(argv[1],"%f", &x);
  sscanf(argv[2],"%f", &y);

  cmp_flag = comp(x,y);

  if ( cmp_flag == 0 )
     printf("%f == %f\n\n", x, y);
  else
    if ( cmp_flag > 0 )
       printf("%f > %f\n\n", x, y);
   else
    if ( cmp_flag < 0 )
       printf("%f < %f\n\n", x, y);

}   /* main */
```

```
E:\>tcc ccomp1.c mc1.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
ccomp1.c:
mc1.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   mc1.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  398k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4104476

E:\>CCOMP1.EXE 432.65 988.22
432.649994 < 988.219971


E:\>
```

```
;  mc1.asm  - Compare within the math coprocessor
;
;       int _comp( float p1, float p2)
;                    [BP+4]    [BP+8]
;
;  If p1 = p2,     Return  0
;  If p1 > p2,     Return  1
;  If p1 < p2,     Return -1
;
;FCOMP =  Compare and Pop
;Return    C3  C2  C1  C0  Meaning
;          0   0   ?   0    ST > Source
;          0   0   ?   1    ST < Source
;          1   0   ?   0    ST == Source
;          1   1   ?   1    ST is not comparable to source
;
; SAHF - Store AH into flags
;
;  Status Word
;  ------------
;  15  14   13 12 11  10    9    8    7    6    5    4    3    2   1   0
;  ---+----+--+--+--+----+----+----+----+----+----+----+----+----+----+----
; | B | C3 |  TOP   | C2 | C1 | C0 | ES | SF | PE | UE | OE | ZE | DE | IE |
;  ---+----+--+--+--+----+----+----+----+----+----+----+----+----+----+----
;
  PUBLIC _comp
;
  ASSUME CS:_TEXT
;
_TEXT    SEGMENT BYTE PUBLIC 'CODE'
 .386
 .387
_comp   PROC    NEAR
   PUSH   BP
   MOV    BP,SP
   FLD      DWORD PTR [BP+4]   ;  ST(0) = p1
   FCOMP    DWORD PTR [BP+8]   ; Compare ST with p2, pop
   FSTSW   AX                  ;  Store comparison result in AX
;
   SAHF     ; Store C3, C2, C0 in Flags register (ZF, PF, CF)
   ;
   ; Proceed as we did CMP on two unsigned integer operands
   ;
            ; Was  ST > ST(1) ( p1 > p2 ) ?
   JBE     Not_Above       ; No, more tests
     MOV   AX,1         ; Yes, return 1
     JMP SHORT Finish
Not_Above:    ; ST <= ST(1)
           ; Was ST < ST(1) ( p1 < p2 )?
   JE  Equal                  ; No, ST == ST(1)
     MOV   AX,-1         ; Yes, return 0
     JMP SHORT Finish
;
Equal:
   MOV  AX,0  ; ST == ST(1) or incomparible
;
Finish:
   POP   BP
   RET
ENDP _comp
;
_TEXT  ENDS
;
END
```

```
/* ccomp2.c  - compare real numbers */

#include <stdio.h>
#include <stdlib.h>

extern int comp( double p1, double p2 );

void main(int argc, char *argv[])
{
  double x, y;
  int cmp_flag;

  if (argc < 3)
    {
     fprintf(stderr,"Usage: ccomp1 real1 real2\n");
     exit(1);
    }

  sscanf(argv[1],"%lf", &x);
  sscanf(argv[2],"%lf", &y);

  cmp_flag = comp(x,y);

  if ( cmp_flag == 0 )
     printf("%lf == %lf\n\n", x, y);
  else
    if ( cmp_flag > 0 )
       printf("%lf > %lf\n\n", x, y);
   else
    if ( cmp_flag < 0 )
       printf("%lf < %lf\n\n", x, y);

}   /* main */
```

```
E:\>tcc ccomp2.c mc2.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
ccomp2.c:
mc2.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   mc2.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  398k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4103660

E:\>ccomp2.exe 321.2 -400.9
321.200000 > -400.900000


E:\>
```

```
;  mc2.asm  - Compare within the math coprocessor
;
;       int _comp( double p1, double p2)
;                    [BP+4]     [BP+12]
;
;  If p1 = p2,     Return  0
;  If p1 > p2,     Return  1
;  If p1 < p2,     Return -1
;
;FCOMP =  Compare and Pop
;Return    C3  C2  C1  C0  Meaning
;          0   0   ?   0    ST > Source
;          0   0   ?   1    ST < Source
;          1   0   ?   0    ST == Source
;          1   1   ?   1    ST is not comparable to source
;
; SAHF - Store AH into flags
;
;  Status Word
;  ------------
;  15  14   13 12 11  10    9    8    7    6    5    4    3    2   1   0
;  ---+----+--+--+--+----+----+----+----+----+----+----+----+----+----+----
; | B | C3 |  TOP   | C2 | C1 | C0 | ES | SF | PE | UE | OE | ZE | DE | IE |
;  ---+----+--+--+--+----+----+----+----+----+----+----+----+----+----+----
;
  PUBLIC _comp
;
  ASSUME CS:_TEXT
;
_TEXT    SEGMENT BYTE PUBLIC 'CODE'
 .386
 .387
_comp   PROC    NEAR
   PUSH   BP
   MOV    BP,SP
   FLD      QWORD PTR [BP+4]   ;  ST(0) = p1
   FCOMP    QWORD PTR [BP+12]  ; Compare ST with p2, pop
   FSTSW   AX                  ;  Store comparison result in AX
;
   SAHF     ; Store C3, C2, C0 in Flags register (ZF, PF, CF)
   ;
   ; Proceed as we did CMP on two unsigned integer operands
   ;
            ; Was  ST > ST(1) ( p1 > p2 ) ?
   JBE     Not_Above        ; No, more tests
     MOV   AX,1         ; Yes, return 1
     JMP SHORT Finish
Not_Above:    ; ST <= ST(1)
           ; Was ST < ST(1) ( p1 < p2 )?
   JE  Equal                 ; No, ST == ST(1)
     MOV   AX,-1         ; Yes, return 0
     JMP SHORT Finish
;
Equal:
   MOV  AX,0  ; ST == ST(1) or incomparible
;
Finish:
   POP   BP
   RET
ENDP _comp
;
_TEXT  ENDS
;
END
```

```
/* ccomp3.c  - compare real numbers */

#include <stdio.h>
#include <stdlib.h>

extern int comp( long double p1, long double p2 );

void main(int argc, char *argv[])
{
  long double x, y;
  int cmp_flag;

  if (argc < 3)
    {
     fprintf(stderr,"Usage: ccomp1 real1 real2\n");
     exit(1);
    }

  sscanf(argv[1],"%Lf", &x);
  sscanf(argv[2],"%Lf", &y);

  cmp_flag = comp(x,y);

  if ( cmp_flag == 0 )
     printf("%Lf == %Lf\n\n", x, y);
  else
    if ( cmp_flag > 0 )
       printf("%Lf > %Lf\n\n", x, y);
   else
    if ( cmp_flag < 0 )
       printf("%Lf < %Lf\n\n", x, y);

}   /* main */
```

```
E:\users\eytan\asm\braude>tcc -v ccomp3.c mc3.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
ccomp3.c:
mc3.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   mc3.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  429k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4138012

E:\>CCOMP3.EXE 654.6 32.87
654.600000 > 32.870000


E:\>
```

```
;  mc3.asm  - Compare within the math coprocessor
;
;       int _comp( long double p1, long double p2)
;                    [BP+4]          [BP+14]
;
;  If p1 = p2,     Return  0
;  If p1 > p2,     Return  1
;  If p1 < p2,     Return -1
;
;FCOMP =  Compare and Pop
;Return     C3   C2   C1   C0  Meaning
;           0    0    ?    0    ST > Source
;           0    0    ?    1    ST < Source
;           1    0    ?    0    ST == Source
;           1    1    ?    1    ST is not comparable to source
;
; SAHF - Store AH into flags
;
;  Status Word
;  ------------
;  15  14    13 12 11  10    9    8    7    6    5    4    3    2   1   0
;  ---+----+--+--+--+----+----+----+----+----+----+----+----+----+----+----
; | B | C3 |   TOP    | C2 | C1 | C0 | ES | SF | PE | UE | OE | ZE | DE | IE |
;  ---+----+--+--+--+----+----+----+----+----+----+----+----+----+----+----
;
  PUBLIC _comp
;
  ASSUME CS:_TEXT
;
_TEXT    SEGMENT BYTE PUBLIC 'CODE'
 .386
 .387
_comp   PROC    NEAR
   PUSH   BP
   MOV    BP,SP
   FLD  TBYTE PTR [BP+14]  ;  ST(0) = p1
   FLD  TBYTE  PTR [BP+4]
   FCOMPP                  ; Compare p1 with p2, pop twice
   FSTSW   AX                ;  Store comparison result in AX
;
   SAHF     ; Store C3, C2, C0 in Flags register (ZF, PF, CF)
   ;
   ; Proceed as we did CMP on two unsigned integer operands
   ;
            ; Was  ST > ST(1) ( p1 > p2 ) ?
   JBE     Not_Above       ; No, more tests
     MOV   AX,1         ; Yes, return 1
     JMP SHORT Finish
Not_Above:    ; ST <= ST(1)
          ; Was ST < ST(1) ( p1 < p2 )?
   JE  Equal                 ; No, ST == ST(1)
     MOV   AX,-1         ; Yes, return 0
     JMP SHORT Finish
;
Equal:
   MOV  AX,0  ; ST == ST(1) or incomparible
;
Finish:
   POP   BP
   RET
ENDP _comp
;
_TEXT  ENDS
;
END
```

אריתמטיקה של מספרים שלמים במעבד המתמטי

מלבד התמיכה במספרים ממשיים (64, 32 ו-80 ביט) מעבד המתמטי "תומך" באריתמטיקה של מספרים שלמים 32, 16 ו-64 ביט. מאז ה-386 התמיכה במספרים שלמים 16 ו-32 ביט אולי לא כל כך חשוב, אבל עבור מספרים 64 ביט התמיכה כאן היא משמעותית.

ה"תמיכה" במספרים שלמים במעבד המתמטי הוא במובן מסוים מאד. היצוג של מספרים באוגרי ה-ST(i) הוא תמיד ממשי - תמיד. התמיכה של המעבד מספרים שלמים בא לידי ביטוי בשתי צורות:

1. המרות: המעבד יודע לקרוא אופרנדים בזיכרון בפורמט שלם ולהמיר אותם לממשי, והוא יודע גם לכתוב אופרנדים לזיכרון בפורמט שלם תוך המרה מהפורמט הממשי שמשתמש המעבד.

2. ישנם מספר (קטן) של פקודות מכונה בתוך המעבד המדמות פעולות על מספרים שלמים בפורמט הממשי שהמעבד משתמש. דוגמאות לכך הם FRNDINT, FPREM, FPREM1.

תמיכה מסוג 1. הוא העיקר. המעבד מאפשר לכתוב קוד הקרוא מספר שלם לתוך המעבד, לפעול עליו כמספר ממשי ולכותב את התוצאה לזיכרון כמספר שלם. גם 1. וגם 2. מסתמכים למעשה על העובדה שהיצוג של מספר שלם כממשי הוא מדויק. היצוג של של המספר השלם 23 הוא 23.0, ולא 22.9999 ולא 23.0001 וכו'. בעוד שקריאת מספרים שלמים לתוך המעבד היא, איפוא, פעולה נעדרת בעיות, כתיבה של מספר ממשי לזיכרון עשויה להיות כרוכה בעיגול המספר, והמתכנת עשוי להיות במצב שעליו לדאוג לכך שיהיה עיגול מהסוג שהוא מעונין בו, אם על ידי הוספה / הפחתה של חצי או מניפולציה של ה-Control Word של המעבד המתמטי. כל הפקודות הללו פועלות על אופרנד בזיכרון, שכן בתוך המעבד - הכל ממשי.

הפקודה הקוראת לתוך המעבד מהזיכרון מספר בפורמט שלם נקרא FILD (להבדיל מ-FLD שקוראת מהזיכרון מספרים בפורמט ממשי). הפקודות הכותבות לזיכרון מספר בפורמט שלם נקראים FIST, FISTP (להבדיל מ-FST, FSTP שכותבות מספר בפורמט ממשי). עבור אופרנדים של זיכרון 32- 16 ביט ניתן גם לבצע פעולות ארתמטית שאחד האופרנדים (אופרנד מקור בלבד, לא יעד, כלומר אופרנד קריאה בלבד) הוא מספר בזיכרון בפורמט ממשי: FIADD, FISUB, FISUBR, FIMUL, FIDIV, FIDIVR. אופרנדי זיכרון שלמים בני 64 ביט נתמכים אך ורק בפקודות FILD ו-FISTP. יתר הפקודות תומכות רק באופרנדי זיכרון 16 ו-32 ביט בלבד.

לפיכך, אם יש לי 3 משתנים בשם Var1, Var2, ו-Var3 ואני רוצה, נניח, לסכם

את Var1 ו-Var2 כמספרים בפורמט שלם ולהציב את התוצאה בפורמט שלם ב-Var3, הדרך לעשות זאת הוא:

אם Var1, Var2 ו-Var3 הם בגודל 16 או 32 ביט:

```
FILD Var1
FIADD Var2
FISTP  Var3
```

ואם הם בגודל 64 ביט:

```
FILD Var1
FILD Var2
FADD
FISTP Var3
```

שימו לב ש-FADD הוא סכום ממשי.

חיסור וכפל ניתן לפתור באותה צורה משום שהפעולות הללו משמרות את אופי המספרים כמספרים שלמים ביצוג ממשי, דבר שאינו כך בחילוק למשל. במקרה של חילוק המצב יותר מורכב.

תוכניות דוגמא call_idl.c, idiv_mo2.asm, idiv_mo3.asm

מה שיש כאן הוא מימוש חדש של התוכנית call_idl.c שראינו את המימוש שלו בעזרת רוטינת אסמבלי המשתמשת באוגרי המעבד הרגילים. כאן נראה את המימוש של האריתמטיקה של השלמים בעזרת המעבד המתמטי. מטרת תוכניות הדוגמא הללו הוא להמחיש אריתמטיקה של שלמים במעבד המתמטי.

לצורך הדוגמא, נניח שאנחנו מעונינים שהעיגול של החלוקה ללא שארית תהיה תמיד למינוס אינסוף. ההבדל בין שני המימושים של idiv_mod בקבצים idiv_mo2.asm ו- idiv_mo3.asm הוא הצורה שהפונקציה מודאת עיגול כלפי מטה. ב- idiv_mo2.asm העיגול היא ע"י הפחתה של 0.499999 ואילו ב- idiv_mo3.asm זוהי הצבה של הערך 01 ל-RC שב-Control Word של המעבד.

ניהול ה-Control Word בקובץ idiv_mo3.asm

ניהול ה-Control Word ב-idiv_mo3.asm נעשה על ידי שמירת ערכו המקורי לתוך משתנה מיועד לכך Save_CW ושיחזור של הערך לפני החזרה. הצבת הערך 01 ל-RC נעשה על סמך הערך הנוכחי של ה-Control Word על ידי הצבתו למשתנה נפרד New_CW, ביצוע פעולות ביטיות עליו וטעינתו חזרה ל-Control Word. השימור של ה-Control Word נעשה בפקודה

```
FSTCW Save_CW
```

בתחילת הרוטינה והשיחזור נעשה ע"י הפקודה

```
FLDCW Save_CW
```

השינוי הזמני ב-Control Word נעשה ע"י הפקודות

```
FSTCW New_CW
AND  New_CW,1111001111111111B
OR   New_CW,0000010000000000B
FLDCW New_CW
```

הכופה על ה-RC את הערך 01 ע"י איפוס הערך הנוכחי שלו על ידי פקודת ה-AND וכתיבת 1 לביט הנמוך של ה-RC ע"י פקודת ה-OR.

מימוש האריתמטיקה בשני המימושים

שתי המימושים מבצעים את משימתם על ידי טעינת המספרים השלמים לאוגרי המעבד ע"י הפקודה FILD וחלוקה ע"י FIDIVR. בתוך המעבד האריתמטיקה היא ממשית אך התמרת התוצאה נעשית ע"י הפקודה FISTP. חישוב שארית החלוקה נעשה ע"י טעינת 2 המספרים למעבד ושימוש בפקודה FPREM.

הפקודה

```
FILD WORD PTR [BP+6]
```

טוען את המכנה. בכדי להתגונן מפני חלוקה באפס, הפקודה

```
FTST
```

משווה את המספר עם אפס. הפקודות

```
FSTSW AX
SAHF
```

מעבירה את תוצאות ההשוואה לאוגר הדגלים. משם אנחנו ממשיכים כמו ב-idiv_mol.asm בכל הקשור לטיפול בחלוקה באפס (שכן מענין אותנו כאן רק שיוויון עם עם אפס או אי שיוויון, וכאן אין הבדל בין חסרי סימן לעם סימן).

במקרה התקין שבו לא נדרשנו לחלק באפס ההמשך הוא ביצוע החלוקה ע"י הפקודה

```
FIDVR WORD PTR [BP+4]
```

בשלב זה התוצאה ממשית, ע"י הפקודות

```
MOV BX,[BP+8]
FISTP WORD PTR [BX]
```

התוכנית מציבה את התוצאה ביעד תוך ביצוע ההמרה והעיגול הדרושים, ו-ST מתרוקן.

בדוגמת ריצה של התוכנית המהלך הזה מתבטא בכך ש-ST(0) מקבל את הערך 44.0, לאחר מכן מוחלף ב-2.38636, ליעד נכתב המספר השלם 2 ו-ST(0) מתרוקן.

חישוב שארית החלוקה נעשה ע"י הפקודות

```
FILD WORD PTR [BP+6]
FILD WORD PTR [BP+4]
FPREM
```

שים לב שהסדר של ה-FILD-ים חשוב. עכשיו ב-ST(0) נמצא השארית ואילו ב-ST(1) נמצא המכנה. עכשיו בעקבות ביצוע הפקודות

```
MOV BX,[BP+10]
FISTP WORD PTR [BX]
```

מוצב התוצאה ליעד ומתרוקן ST(1). ערך המכנה נמצא עכשיו ב-ST(0). חובה לרוקן אותו והדבר נעשה ע"י הפקודה

```
FFREE ST
```

בדוגמת הריצה יש לנו מציאות שבו האוגרים ST(0) ו-ST(1) עוברים שרשרת מצבים ממצב של שניהם ריקים ל-

שלב ראשון, אחרי FILD WORD PTR [BP+6],

ST(0) 44.0
ST(1) ריק

שלב שני, אחרי FILD WORD PTR [BP+4],

ST(0) 105.0
ST(1) 44.0

שלב שלישי, אחרי FPREM,

ST(0) 17.0
ST(1) 44.0

שלב רביעי, אחרי FISTP WORD PTR [BX],
ליעד מוצב הערך השלם 17 ואילו האוגרים של המעבד עוברים למצב

ST(0) 44.0
ST(1) ריק

עד לשלב החמישי (אחרי FFREE ST) ששניהם ריקים שוב.

```
/* call_id1.c - call assembler subroutine idiv_mod.asm from C program */

#include <stdio.h>

 extern  int idiv_mod(int Num, int Denom,  int *Q,  int *Rem);

 void main()
    {
      int Num, Denom, Q, Rem, No_Zero_Divide;

      printf("\Enter Numerator, Denominator\n:");
      scanf("%d %d",&Num, &Denom);
      No_Zero_Divide = idiv_mod(Num,Denom,&Q,&Rem);
      if  (No_Zero_Divide)
          printf("\n %d div %d = %d, mod(%d,%d) = %d\n",
                   Num, Denom, Q, Num, Denom, Rem);
      else
         printf("\nError: Zero Divide.\n");

     } /* main */
```

```
E:\>tcc call_id1.c idiv_mo2.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
call_id1.c:
idiv_mo2.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   idiv_mo2.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  418k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4136272

E:\>call_id1.exe
Enter Numerator, Denominator
:105 44

 105 div 44 = 2, mod(105,44) = 17

E:\>
```

```
;  idiv_mo2.asm - Assembler implementation of C-callable function idiv_mod.
;
  ; Static Variables
   ASSUME CS:_TEXT, DS:_DATA
;
_DATA SEGMENT DWORD PUBLIC 'DATA'
;
Half  DQ 0.4999999999999
_DATA ENDS
;
_TEXT SEGMENT BYTE PUBLIC 'CODE'
    .386
    .387
    ; Implementation of C callable function ...
    ;    ...  int idiv_mod(int Num, int Denom,  int *Q,  int *Rem)
    ;                       [BP+4]    [BP+6]     [BP+8]    [BP+10]
    ;  Compute Q := |_ Num / Denom _|  ,Rem := MOD(Num, Denom)
    ;  function  idiv_mod  returns 0 if Denom = 0 (illegal ..
    ;  ... division by zero), 1 otherwise
    ;
     PUBLIC _idiv_mod
_idiv_mod  PROC NEAR
     PUSH BP         ; Preserve BP
     MOV BP,SP       ; Set BP to point to Parameter area
     FILD WORD PTR [BP+6]  ; ST(0) := Denom
     FTST     ; Denom = 0 ?
     FSTSW  AX ;  AX = Status word
     SAHF     ;  Copy to flags register
     JNZ Cont        ; No, continue regular operation
          ; Yes, Denom = 0
         FFREE ST
         MOV AX,0     ; Return value := 0
         JMP Done             ; Skip following code
Cont:      ; Denom <> 0
     FIDIVR WORD PTR [BP+4]              ; ST = Num / ST, ST = Num / Denom
     FSUB  Half           ; Subtract 1/2 to ensure rounding down
     MOV  BX,[BP+8]       ; BX := Offset Q
     FISTP  WORD PTR [BX]        ; *Q := ST
     FILD  WORD PTR [BP+6]     ; ST = Denom
     FILD  WORD PTR [BP+4]     ; ST = Num, ST(1) = Denom
     FPREM                     ; ST = ST mod ST(1)
     MOV  BX,[BP+10]      ; BX := Offset Rem
     FISTP WORD PTR  [BX]         ; *Rem := ST
     FFREE ST             ;
     MOV  AX,1    ; Ensure return value = 1
Done:
     POP  BP              ; Restore BP register
     RET
_idiv_mod  ENDP
;
_TEXT ENDS
;
     END
```

```
;  idiv_mo3.asm - Assembler implementation of C-callable
;                 function idiv_mod.
;
;
;  Control Word
;  --------------
;  15  14  13  12  11 10  9  8  7   6    5    4    3   2    1    0
;  ---+---+---+---+--+--+--+--+---+---+----+----+----+----+----+----+
; | R | R | R | I |  RC | PC  | R | R | PM | UM | OM | ZM | DM | IM |
;  ---+---+---+---+--+--+--+--+---+---+----+----+----+----+----+----+
;
  ASSUME CS:_TEXT, DS:_DATA
;
  ; Static Variables
_DATA SEGMENT DWORD PUBLIC 'DATA'
;
Save_CW  DW ?
New_CW   DW ?
;
_DATA  ENDS
;
_TEXT SEGMENT BYTE PUBLIC 'CODE'
    .386
    .387
    ; Implementation of C callable function ...
    ;    ...  int idiv_mod(int Num, int Denom,  int *Q,  int *Rem)
    ;                      [BP+4]    [BP+6]     [BP+8]    [BP+10]
    ;  Compute Q := |_ Num / Denom _|  ,Rem := MOD(Num, Denom)
    ;  function  idiv_mod  returns 0 if Denom = 0 (illegal ..
    ;  ... division by zero), 1 otherwise
    ;
     PUBLIC _idiv_mod
_idiv_mod  PROC NEAR
     PUSH BP         ; Preserve BP
     MOV BP,SP       ; Set BP to point to Parameter area
;
     FSTCW   Save_CW          ;  Store status in Mem16
     FSTCW   New_CW           ;  Store status in Mem16
     AND  New_CW,1111001111111111B ; Erase existing RC
     OR   New_CW,0000010000000000B ; Set RC to 01
                                   ; (Round towards  -infinity)
     FLDCW New_CW                  ; Set New CW
;
     FILD WORD PTR [BP+6]  ; ST(0) := Denom
     FTST      ; Denom = 0 ?
     FSTSW  AX  ; Transfer SW to AX
     SAHF      ;  Copy to flags register
     JNZ Cont        ; Demon != 0
         ; Yes, Denom = 0
        FFREE ST
        MOV AX,0     ; Return value := 0
        JMP Done             ; Skip following code
```

```
Cont:       ; Denom != 0
      FIDIVR WORD PTR [BP+4]       ; ST = Num / ST, ST = Num / Denom
      MOV  BX,[BP+8]       ; BX := Offset Q
      FISTP  WORD PTR [BX]        ; *Q := ST
      FILD  WORD PTR [BP+6]     ; ST = Denom
      FILD  WORD PTR [BP+4]     ; ST = Num, ST(1) = Denom
      FPREM                     ; ST = ST mod ST(1)
      MOV  BX,[BP+10]      ; BX := Offset Rem
      FISTP WORD PTR  [BX]         ; *Rem := ST
      FFREE ST             ; Free ST(0)
      MOV  AX,1    ; Ensure return value = 1
Done:
;
      FLDCW Save_CW        ; Restore control word to original value
      POP  BP              ; Restore BP register
      RET
_idiv_mod  ENDP
;
_TEXT ENDS
;
      END
```

תוכניות דוגמא fderiv1.c, fderiv2.c, fd.asm

התפקיד העיקרי של התוכניות הללו הינו להמחיש את השימוש בפוינטר לפונקציה כפרמטר, דבר נפוץ במיוחד בחישובים נומריים.

בהנתן פונקציה נתונה נניח ע"י איזה קוד חישובי, אנחנו רוצים לעשות קירוב נומרי של הנגזרת שלה בנקודה. זהו תסריט שמתרחש כאשר יודעים לחשב פונקציה מסוימת אבל לא בהכרח יודעים איך הוא נראה אנליטית, למשל כאשר היא פתרון של משוואה שמחושבת באופן נומרי או חישוב נומרי אחר.

חישוב הקירוב הנומרי מבוסס על כך שהנגזרת הוא הגבול של

$$f'(x) = \lim_{h->0} (f(x+h) - f(x-h))/(2h)$$

כלומר אם נחשב את הקירוב לנגדזרת ע"י הנוסחה (*):

$$f'(x) = (f(x+h) - f(x-h))/(2h) \quad (*)$$

עבור h מספיק קטן, נקבל (בתאוריה לפחות) קירוב לנגזרת. השאלה היא איזה h לבחור. ככל ש-h קטן יותר החישוב יהיה יותר נכון מבחינה מתמטית אולם אם נבחר h קטן מדי יגרום לבעיות של אובדן דיוק כתוצאה משגיאות עיגול. חוץ מזה, h חייב להיות קטן יחסית ל-x ולאו דווקא קטן במושגים מוחלטים. פתרון אפשרי הוא להתחיל אם h = |x/2.0| וכל הזמן להקטין את h ע"י חלוקה ב-2.0 עד שנגיע לכך שהחישובים מתחילים להתכנס, כלומר 2 חישובים עוקבים של הנוסחה (*) ההפרש ביניהם בערכם המוחלט קטן מאפסילון נבחר. בתוכניות הללו נבחר אפסילון כערך המוחלט של f(x) חלקי 8192.0. אילו החישובים היו בדיוק יותר גבוה מ-float היינו מחלקים ביותר.

השיקולים הועמדים מאחרי האלגוריתם הזה קשורים לנושאים של דיוק חישובי שלא כאן המקום לפרטם.

התוכנית fderiv1.c היא מימוש האלגוריתם בשפת C. התוכנית המשולבת fderiv2.c ו-fd.asm ממשים את האלגוריתם הנומרי באסמבלי.

```
/* fderiv1.c - approximate derivative function */

#include <stdio.h>
#include <math.h>

float approx_fderiv(float (*f)(float), float x)
{
  float h, fd0, fd1, eps;

  h = fabs(x/2.0);
  fd1 = ((*f)(x+h) - (*f)(x-h))/(2*h);
  eps = x/8192.0;

  do {
      fd0 = fd1;
      h = h/2.0;
      fd1 = ((*f)(x+h) - (*f)(x-h))/(2*h);
     } while(fabs(fd0 - fd1) > eps );

  return fd1;

} /* approx_deriv */


float f(float x)
{
  return x*x*x - 2.0*x*x + 3.0*x - 8.0;
} /* f */

float real_fderiv(float x)
{
  return 3.0*x*x - 4.0*x + 3.0;
} /* real_fderiv */


int main()
{
  printf("approx_deriv(5.0) = %f\n", approx_fderiv(f, 5.0));
  printf("real_fderiv(5.0) = %f\n", real_fderiv(5.0));

  return 0;
} /* main */
```

```
E:\>tcc -v fderiv1.c
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
fderiv1.c:
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4103660

E:\>FDERIV1.EXE
approx_deriv(5.0) =  58.001526
real_fderiv(5.0) = 58.000000

E:\>
```

```
/* fderiv2.c - approximate derivative function */

#include <stdio.h>
#include <math.h>


extern float approx_fderiv(float (*f)(float), float x);


float f(float x)
{
  return x*x*x - 2.0*x*x + 3.0*x - 8.0;
} /* f */

float real_fderiv(float x)
{
  return 3.0*x*x - 4.0*x + 3.0;
} /* real_fderiv */


int main()
{
  printf("approx_deriv(5.0) = %f\n", approx_fderiv(f, 5.0));
  printf("real_fderiv(5.0) = %f\n", real_fderiv(5.0));

  return 0;
} /* main */
```

```
E:\>tcc -v fderiv2.c fd.asm
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
fderiv2.c:
fd.asm:
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International

Assembling file:   fd.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  394k

Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4107240

E:\>FDERIV2.EXE
approx_deriv(5.0) =  58.001526
real_fderiv(5.0) = 58.000000

E:\>
```

```
;
; fd.asm - implement numerical differentiation
;
;
ASSUME CS:_TEXT, DS:_DATA,SS:_DATA
_DATA   SEGMENT WORD public 'DATA'
h  DD ?
two DD 2.0
fd0 DD 0.0
eps_const DD 8192.0
eps  DD 0.0
temp DD 0.0
_DATA   ENDS
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
   ;
   ;    float approx_fderiv (float (*f)(float), float x)
   ;                              [BP+4]          [BP+6]
   ;
        .386
        .387
      PUBLIC _approx_fderiv
_approx_fderiv  PROC    NEAR
        PUSH    BP
        MOV     BP,SP
        FLD DWORD PTR [BP+6]
        FDIV two
        FABS
        FSTP h
   ;
   ;      compute (*f)(x+h) -(*f)(x-h) )/ (2*h);
   ;
        FLD     DWORD PTR [BP+6]
        FADD    h
        FSTP    temp
        PUSH    temp                ; Push x+h
        CALL    WORD PTR [BP+4]     ; ST(0) = f(x+h)
        ADD     SP,4    ;  Free Parameter
        FLD     DWORD PTR [BP+6]
        FSUB    h
        FSTP    temp
        PUSH    temp                ; Push x-h
        CALL    WORD PTR [BP+4]     ; ST(0) = f(x-h), ST(1) = f(x+h)
        ADD     SP,4    ;  Free Parameter
        FSUB                        ; ST(0) =  f(x+h)-  f(x-h), ST(1) = Empty
        FLD     h       ; ST(0) = h, ST(1) = f(x+h)-  f(x-h)
        FMUL    two     ; ST(0) = 2h, ST(1) = f(x+h)-  f(x-h)
        FDIV            ; ST(0) = (f(x+h)-  f(x-h))/2h
        FSTP fd0
        PUSH DWORD PTR [BP+6]
        CALL WORD PTR [BP+4]
        ADD SP,4
        FDIV eps_const
        FABS
        FSTP eps
```

```
Do1:
        FLD h
        FDIV two
        FSTP h
   ;
   ;      compute (*f)(x+h) -(*f)(x-h) )/ (2*h);
   ;
        FLD     DWORD PTR [BP+6]
        FADD    h
        FSTP    temp
        PUSH    temp                    ; Push x+h
        CALL    WORD PTR [BP+4]         ; ST(0) = f(x+h)
        ADD     SP,4    ;  Free Parameter
        FLD     DWORD PTR [BP+6]
        FSUB    h
        FSTP    temp
        PUSH    temp                    ; Push x-h
        CALL    WORD PTR [BP+4]         ; ST(0) = f(x-h), ST(1) = f(x+h)
        ADD     SP,4    ;  Free Parameter
        FSUB                    ; ST(0) =  f(x+h)-  f(x-h), ST(1) = Empty
        FLD     h       ; ST(0) = h, ST(1) = f(x+h)-  f(x-h)
        FMUL    two     ; ST(0) = 2h, ST(1) = f(x+h)-  f(x-h)
        FDIV            ; ST(0) = (f(x+h)-  f(x-h))/2h
        FLD ST  ; ST(0) = (f(x+h)-  f(x-h))/2h, ST(1) = (f(x+h)- f(x-h))/2h
        FLD fd0
        FSUB    ; ST(0) = current - fd0
        FABS    ;  ST(0) = | current - fd0 |
        FCOMP eps
        FSTSW AX
        SAHF
        FSTP  fd0
        JAE Do1
   ;
        FLD fd0
        MOV     SP,BP
        POP     BP
        RET
_approx_fderiv  ENDP
   ;
_TEXT   ENDS
        END
```