תקציר מספר 8

המעבד המתמטי

פקודות המכונה שהכרנו עד כה - פקודות ה-CPU הרגילות - תומכות בפעולות אריתמטיות רק עבור מספרים שלמים. ב-PC הראשונים יצוג ופעולות על מספרים ממשיים מומשו רק ברמת התוכנה.

תפקידו העיקרי של המעבד המתמטי הוא לתמוך במימוש מספרים ממשיים (כפי שתוארו קודם) ברמת פקודות מכונה. עקרונית, כל שהוא עושה הוא לממש פעולות על מספרים (ממשיים ושלמים), ורשימת הפקודות שלו מזכירה במידה רבה מה שקרוי "מחשב כיס מדעי". השם "מעבד מתמטי" קצת מטעה במובן הזה. שם יותר מוצלח הוא FPU - Floating Point Unit.

התרומה של המעבד המתמטי לתוכניתן האסמבלי היא:

1. אוגרים נוספים.

2. פקודות מכונה נוספות.

לכל אחד מהמעבדים הראשונים - 8086, 286, 386 היה מעבד מתמטי משלו (8087, 287, 387) שהיה chip בפני עצמו. מי שרצה בהם היה צריך לשלם תשלום נוסף כדי שיותקנו במחשב שלו. מאז ה-486 נכלל המעבד המתמטי בתוך ה-CPU עצמו באופן אוטומטי. אומנם היה 487 אבל היה רק מעין תוספת (enhancement) של מעבד קיים. מנקודת ראותו של התוכניתן אין הרבה הבדלים בין הגרסאות של המעבד המתמטי.
הארכיטקטורה הנגישה לתוכנה (האוגרים) משותפת לכולם. מלבד העובדה שלכל גירסה של ה-CPU היה צורך להתאים מעבד מתמטי שיתאים לה (מנקודת ראות המהירות למשל) הגרסאות המתקדמות יותר מכילות מספר פקודות מכונה שלא היו קיימים קודם.

הארכיטקטורה (הנגישה לתוכנה) של המעבד המתמטי.

כללי

הארכיטקטורה הנגישה לתכנות של המעבד המתמטי היא קבוצת האוגרים הבאה:

- שמונה אוגרי מספרים המכונים Stack Registers (הנקראים ST(0) עד

.(ST(7)

- אוגר 16 ביט Status Word

- אוגר 16 ביט Control Word

- אוגר 16 ביט Tag Word

למעבד המתמטי שמונה אוגרים בני 80 ביט שתפקידם לאגור מספרים ממשיים 80 ביט.
הם מכונים Stack Registers ונקראים
.ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7)
לאוגר ST(0) קוראים גם ST.

לכל אחד מהאוגרים הללו יש 2 ביטים באוגר מיוחד הנקרא Tag Word (16 ביט סה"כ) המכיל אינפורמציה על הסטטוס של כל אחד מהאוגרים (ערך תקין, אפס, מיוחד, ריק).

אוגר 16 ביט Status Word מכיל דגלים המעידים על הסטטוס של המעבד בעיקר בהשפעת הפעולה האחרונה (משהוא דומה לדגלי הבקרה של ה-CPU). למשל כאשר המעבד מבצע השוואות, תוצאות ההשוואה מוצבות ב-Status Word.

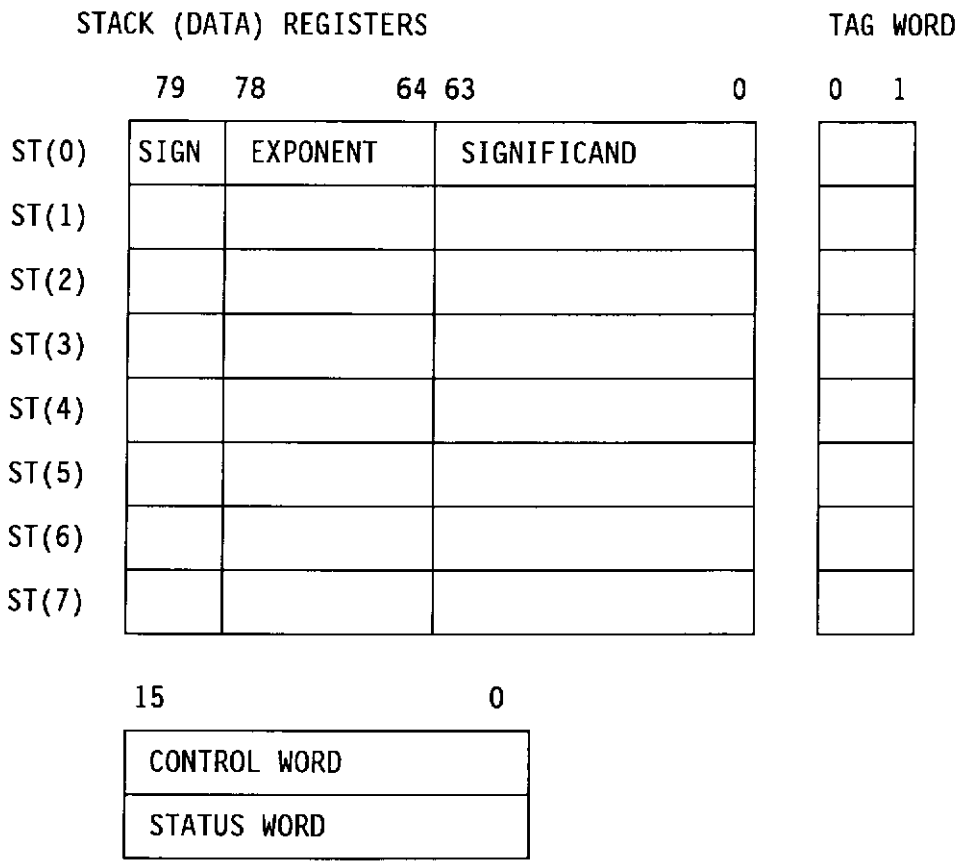
אוגר 16 ביט ה- Control Word מכיל דגלים המשפיעים על תפקוד ה-CPU. למשל ניתן להציב לתוכו ערך שמוריד את מידת הדיוק של החישובים שלו מ-80 ביט לפחות (32, 64 ביט).

למרות שה-Stack Registers מכילים מספרים ביצוג ממשי 80 ביט, המעבד המתמטי תומך במימוש אריתמטיקה של מספרים ממשיים 64, 32, ו-80 ביט, אריתמטיקה של מספרים שלמים 16, 32 ו-64 ביט באופן הבא: פקודות הקריאה והכתיבה של המעבד המתמטי לאוגרים שלו מ/אל הזכרון תומכים בכל ההמרות הנחוצות. לדוגמא, קיימת פקודה FILD המאפשרת לקרוא מספר שלם 32 ביט לתוך אחד האוגרים ST(i) (הערך השלם מותמר לממשי 80 ביט), לבצע עליו פעולות אריתמטיות ולהחזיר את התוצאה חזרה למשתנה השלם (תוך התמרה חזרה לשלם 32 ביט) ע"י הפקודה FIST.

האוגרים נקראים Stack Registers מפני שהם מתפקדים חלק גדול מהזמן כמחסנית של עד שמונה מספרים. במימוש ביטויים אריתמטיים קיים מצב נפוץ שבו אנחנו מבצעים פעולה אריתמטית על שני מספרים (נניח כפל), ומאותו רגע ואילך אין לנו עינין במספרים עצמם אלא רק בתוצאה. פקודות המכונה של המעבד

המתמטי תומכות במימוש נוח של מצבים כאלו. מעבר לזה, האלגוריתם הסטנדרטי של מימוש ביטויים אריתמטיים משתמש, בין השאר, במחסנית של מספרים. מחסנית של שמונה מספרים מקבילה למעשה למימוש ביטויים בשיטה הזו של עד שמונה רמות של סוגריים. לעיתים, קומפילרים מגבילים ביטויים אריתמטיים לעד שמונה רמות של סוגריים, ויתכן שהמגבלה הזו נובעת מכאן.

האוגרים של המעבד המתמטי



מבנה ה- TAG WORD

15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
TAG(7)	TAG(6)	TAG(5)	TAG(4)	TAG(3)	TAG(2)	TAG(1)	TAG(0)

ערכים אפשריים לכל TAG(i):

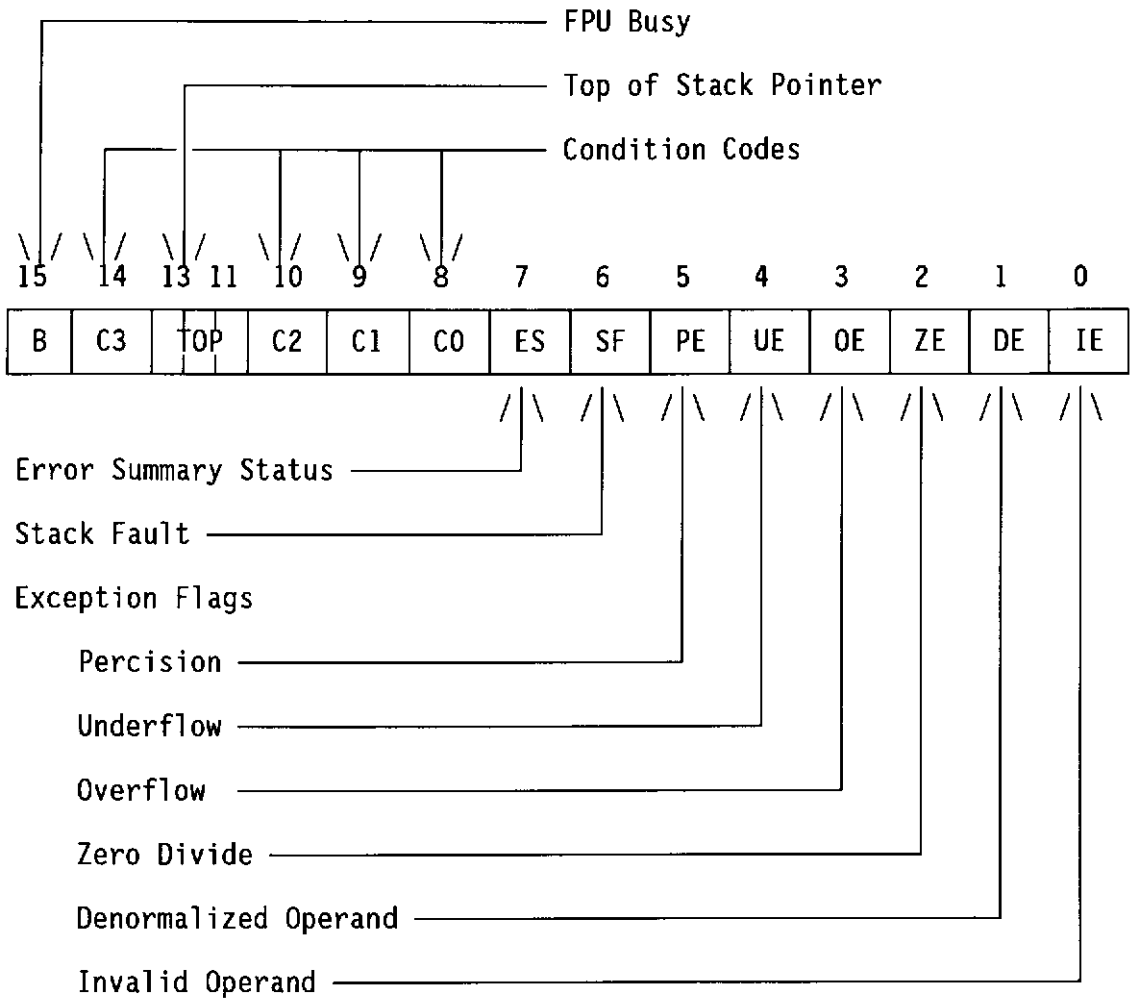
00 - Valid - ערך ממשי כלשהוא, מלבד אפס

01 - Zero - ערך אפס,

10 - Special:Invalid - ערך מיוחד או בלתי חוקי (INFINITY, DENORMAL, UNSUPPORTED, NAN),

11 - Empty - ריק.

הערכים הללו נקבעים ע"י המעבד בעקבות כל שינוי שהאוגרים ST(i).



ES = 1 במידה אם אחד מהדגלים בביטים 6 - 0 דלוק. אחרת ES = 0.

TOP = 111 אם ST(0) הוא ראש המחסנית,

000 אם ST(1) הוא ראש המחסנית,

.....

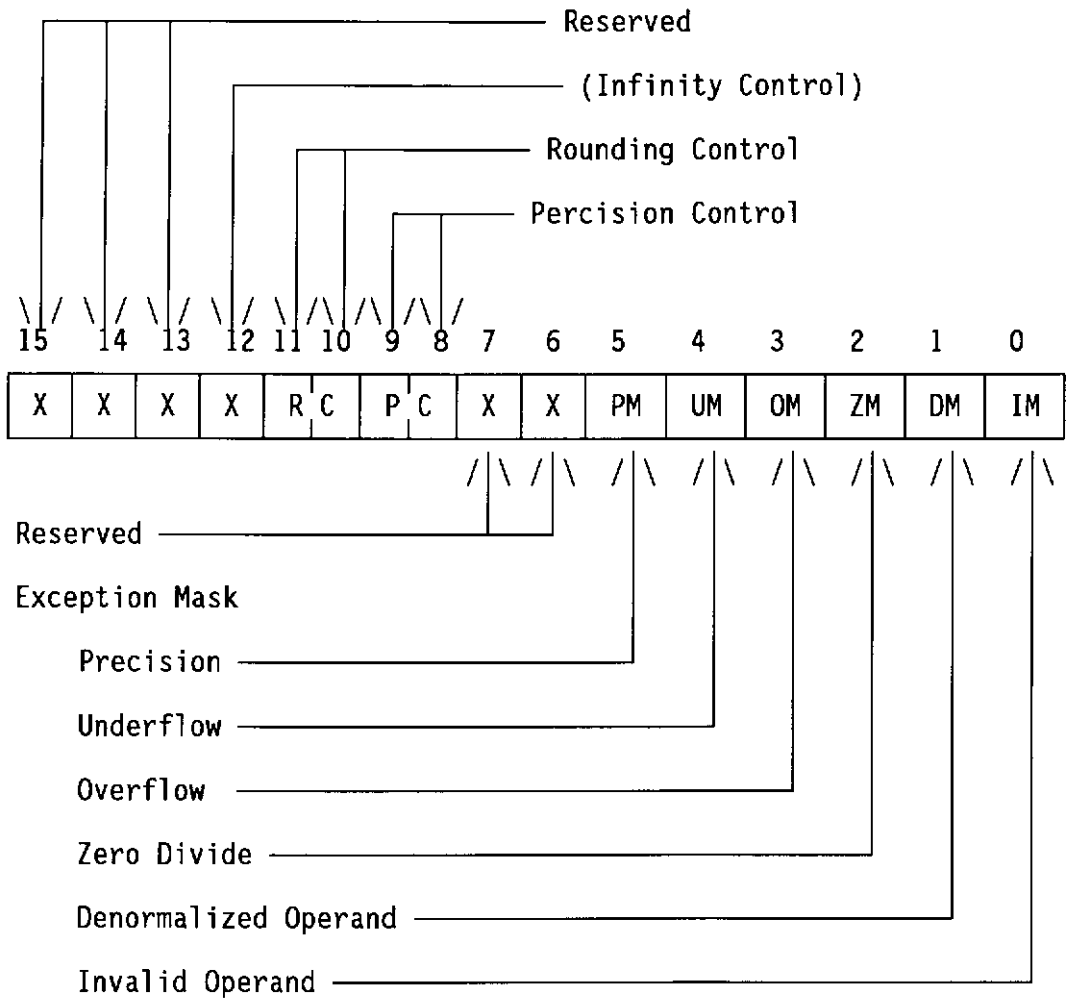
110 אם ST(7) הוא ראש המחסנית.

הערך של ה-Status Word מתעדכן בעקבות כל פעולה.

C0, C1, C2, C3 - הם ה-Condition Codes עליהם נפרט בהמשך. הם מכילים, בין השאר, תוצאות השוואה בין מספרים.

ה- Exception, Error Flags משקפים בעיה בפעולה - למשל Overflow אם היתה גלישה כלפי מעלה, Percision - אם היה פעולה שהיה בה עיגול (דיוק לא מלא) וכו'.

הערך של TOP מושפע מפקודות מיוחדות לכך (FINCSTP, FDECSTP). הוא קובע את האוגר בראש המחסנית לאחר מאשר ST(0). השימוש בו די נדיר.



Infinity Control היה משמעותי רק ב-287. ביתר המעבדים ניתן להציב לתוכו ערך, אך הוא אינו משפיע על המעבד.

הצבת ערך 0 או 1 לכל אחד מה-Mask bit משנה את תגובת המעבד לחריגה הרלוונטית. זה די פרטני לכל חריגה, לא ניכנס לזה כאן.

ערכים אפשריים ל-Rounding Control:

00 - Round to nearest or even	עיגול לקרוב ביותר
01 - Round down	עיגול כלפי מטה (לכיוון מינוס אינסוף)
10 - Round up	עיגול כלפי מעלה (לכיוון פלוס אינסוף)
11 - Chop (Truncate towards zero)	קיצוץ

ערכים אפשריים ל-Percision Control:

00 - Single Percision (32 bits)	דיוק רגיל 32 ביט
01 - (Rerserved)	שמור
10 - Double Percision (64 bits)	דיוק כפול 64 ביט
11 - Extended Percision (80 bits)	דיוק מורחב 80 ביט

את הערך של ה-Status Word ניתן רק ליצא מהמעבד המתמטי, כלומר תוכנית יכולה רק לכתוב את התוכן שלו לזכרון או לאוגר AX. הדבר נעשה ע"י פקודת המכונה FSTSW. הפקודה FSTSW AX היא הפקודה היחידה של המעבד המתמטי שבה האופרנד מחוץ למעבד הינו אוגר רגיל של ה-CPU. בכל מקרה אחר של אופרנד יעד חיצוני וכל מקרה של אופרנד מקור חיצוני מדובר בזכרון.

ה-Control Word מקבל ערכים מהזכרון ע"י פקודה מיוחדת FLDCW. ניתן גם ליצא את הערך לזכרון ע"י פקודת המכונה FSTCW. אם תוכנית משנה לצרכיה את ה-Control Word, היא בדרך כלל תשחזר אותו לערך המקורי לפני סיום (שימור + שיחזור נוסח טבלת הפסיקות). הערך החדש של ה-Control Word יהיה בדר"כ מבוסס על הערך הקודם, או סטנדרטי.

את ה-Tag Word אפשר רק ליצא לזכרון ע"י פקודות מיוחדות (FSAVE, FSTENV) והתוכנית יכולה לפענח את הערכים שם.

רוטינה שמשתמשת במעבד המתמטי צריכה להחזיר את המעבד המתמטי ריק או עם ערך רק ב-ST(0) במידה שהיא מחזירה תשובת פונקציה ממשית. פונקציות Turbo C מחזירות תוצאה ממשית (float, double, long double) בתוך ה-ST(0) ובאחריות התוכנית הקוראת לקרוא ולרוקן אותו משם. על כך נרחיב בהמשך.

קריאה של ערכים בפורמט ממשי מהזכרון לתוך המעבד המתמטי נעשה ע"י הפקודה FLD ל-ST(0). הפקודה מבצעת פעולת PUSH אוטומטית - אם יש ערך ב-ST(0) הוא נדחף ל-ST(1) וכו'. ה-FLD טוען ערכים בפורמט ממשי 80, 64, 32 ביט (כלומר הוא מפרש את תוכן הזכרון כמספרים בפורמט ממשי), תוך התמרה לפורמט ממשי 80 ביט במקרה של אופרנדי זכרון 32 ו-64 ביט. לדוגמא, אם מצב מחסנית המספרים הוא

ST(0) 1.1
ST(1) 4.4
ST(2) 2.2
ST(3) 3.3
ST(4) ריק
ST(5) ריק
ST(6) ריק
ST(7) ריק

וישנו משתנה בזכרון Var1 מוגדר כניח

```
Var1 DQ 6.753
```

ומתבצעת הפקודה

```
FLD Var1
```

במידה ו-Var1 עדיין מכיל את הערך 6.753, המצב של האוגרים אחרי הפעולה תהיה:

```
ST(0)  6.753
ST(1)  1.1
ST(2)  4.4
ST(3)  2.2
ST(4)  3.3
ST(5)  ריק
ST(6)  ריק
ST(7)  ריק
```

FILD מבצע תהליך דומה ל-FLD, אך הוא קורא ערכים בפורמט שלם (16, 32, 64) מהזכרון לתוך אוגרי ה-ST(0) תוך ביצוע המרה לממשי 80 ביט. כלומר, הוא מפרש את תוכן הזכרון כיצוג של מספר שלם.

לדוגמא הקוד הבא:

```
Var2 DW ?

MOV Var2,45
FLD Var2
```

יגרום ל-ST(0) להכיל את הערך הממשי 45.0.

כתיבת ערכים מתוך המעבד המתמטי לזכרון נעשה ע"י הפקודה FST ליעדי זכרון ממשיים 32, 64 ביט או FSTP ליעדי זכרון ממשיים 32, 64, ו-80 ביט. הפקודה FIST כותבת ערכים ליעדע זכרון שלמים 16 ו-32 ביט ו- FISTP ליעדי זכרון שלמים 16, 32, ו-64 ביט. להוציא יעדי זכרון ממשיים 80 ביט, הפקודה מבצעת את ההתמרות הדרושות.

לדוגמא, אם ST(0) מכיל את הערך 7.91, ומתבצעת הפקודה

```
FIST Var2
```

לתוך Var2 ערך מעוגל של 7.91 (בדרך כלל 8) תלוי בתוכן שדה ה-RC ב-

.Control

ריקון איברים
FADDP, FISTP,
ים עם סיום
מיוצג במעבד ו
ן שמביניהם שהי
ת אוגר בשם REE
א, אם מצב המחס

שה בדרך כלל על ידי גירסאות ה-POP של הפקודות כמו
של. הפקודות הללו מבצעות POP אוטומטי של מחסנית
עולה שלהם. המשמעות של POP הוא שהערך של ST(0) חדל
ר הערכים במחסנית המספרים מקודמים לעבר ST(0), כאשר
לו ערך "משתחרר" או "מתרוקן" (אגב ישנה גם פקודה שרק
.(
ת הוא

ST(0) 1.1
ST(1) 4.4
ST(2) 2.2
ST(3) 3.3
ST(4) ריק
ST(5) ריק
ST(6) ריק
ST(7) ריק

צע הפקודה

FSTP Var1

Var יוצב הערך

1, ומצב מחסנית המספרים יהיה:

ST(0) 4.4
ST(1) 2.2
ST(2) 3.3
ST(3) ריק
ST(4) ריק
ST(5) ריק
ST(6) ריק
ST(7) ריק

ם בין 2 אופרנדים בתוך ה-ST(i). אולם ניתן גם לבצע
ר ST(0) ואופרנד בזכרון. לשם כך יש גירסאות אוגר -
טיות כמו FADD, FIADD וכו'. יחד עם זאת יש לשים לב
גירסאות הללו אינן תומכות בגדלים הארוכים ביותר.
ך בחיבור עם משתנה זכרון ממשי 80 ביט, FIADD לא תומך
שלם 64 ביט. רק חלק מהפקודות של המעבד המתמטי
יעדי זכרון ממשיים 80 ביט או שלמים 64 ביט, כנראה
ל, אם צריך לחבר שני משתנים ממשיים 80 ביט או שני
, יש צורך לקרוא את שני הערכים לתוך שני ST(i) שונים
S) ולבצע FADD או FADDP עליהם.

ן לבצע חישו
א אריתמטית עם א
של פקודות ארי
ה ל-FST ו-FIST
FADD אינו ת
ר עם משתנה זכר
ת בפעולות על
לי יעילות. ל
ם שלמים 64 ב
כלל ST(1), (0

לדוגמא הפקודה

FDIV Var1

ST(0) = Var1 / ST(0) אילו רצינו .ST(0) = ST(0) / Var1 תבצע את החישוב
היינו משתמשים בפקודה FDIVR, מה שנקרא גירסת ה-reverse, כלומר

FDIVR Var1

ל-FADD ו-FMUL אין גירסאות reverse כי בחיבור וכפל הסדר אינו משנה.

הפקודות האריתמטיות הרגילות (FADD, FSUB, FSUBR, FMUL, FDIV, FDIVR)
כאשר הן נכתבות ללא אופרנדים פועלות על ST(1) ו-ST(0) ומבצעות POP
אוטומטי (למרות העדר ה-"P" בפקודה). לדוגמא, הפקודה

FSUB

שקולה לפקודה

FSUBP ST(1),ST(0)

כלומר הרצת FSUB ללא אופרנדים תבצע ST(1) = ST(1) - ST(0) ו-POP, כלומר
תוכן ST(1) ו-ST(0) יעלמו והם יוחלפו בתוצאת ההפחתה של הערך ST(0) מההערך
של ST(1) (אילו רצינו ההפך היינו משתמשים ב-FSUBR). התוצאה של ההפחתה תהיה
בסוף הפעולה ב-ST(0). לפחות אחד מאוגרי ה-ST(i) ישתחרר, ולערכים האחרים
במחסנית המספרים, במידה וישנם, יקודמו לכיוון ה-ST(0). לדוגמא, אם לפני
ביצוע ה- FSUB ללא אופרנדים המצב באוגרים היה:

ST(0) 6.4
ST(1) 5.2
ST(2) 2.7
ST(3) 3.4
ST(4) ריק
ST(5) ריק
ST(6) ריק
ST(7) ריק

אחרי ה-FSUB המצב באוגרים יהיה:

```
ST(0)  -1.2
ST(1)  2.7
ST(2)  3.4
ST(3)  ריק
ST(4)  ריק
ST(5)  ריק
ST(6)  ריק
ST(7)  ריק
```

השימוש בפקודות האריתמטיות הללו ללא אופרנדים הוא הנוהל הסטנדרטי כאשר אנחנו מעוניינים לבצע פעולה על שני מספרים ומאותו רגע אין לנו עניין במספרים הללו אלא רק בתוצאה. שים לב שהמבנה הזה אינו תופס עבור הפקודה FCOM. FCOM ללא אופנדים אמנם משווה בין ST(0) ל-ST(1) (בסדר הזה שלא כמו ב-FSUB, כלומר כאילו ST(1) - ST(0)), אך משאיר אותם כמות שהם. אם רוצים לשחרר את תוכן ST(0) ו-ST(1) תוך פעולת ההשוואה משתמשים בפקודה FCOMPP.

לדוגמא, אם יש לנו שלושה משתנים:

```
Var3 DT 100.1
Var4 DT 200.2
Var5 DT  ?
```

ואנחנו מעוניינים לחשב Var5 = Var3 / Var4 אנחנו נכתוב:

```
FLD Var3  ;  ST(0) = 100.1
FLD Var4  ;  ST(0) = 200.2, ST(1) = 100.1
FDIV      ;  ST(0) = 0.5
FSTP Var5 ;  Var5 = 0.5
```

בנוסף למתואר לעיל, המעבד המתמטי תומך בפונקציות מתמטיות שונות (FSQRT, FSIN ברדיאנים וכו'). יש פקודות המדמות אריתמטיקה של מספרים שלמים כמו עיגול לשלם (FRNDINT) וחישוב שארית חלוקה (FPREM, FPREM1).

כמו כן יש לו פקודות בקרה שדיברנו קודם (FLDCW למשל). הם מאפשרות יצוא של תוכן האוגרים של המעבד (FSAVE, FSTENV) וכן פקודות בקרה שמשפיעות על תפקוד המעבד (FINIT, FCLEX ...).

מוכמות העברת פרמטרים ממשיים ותוצאות פונקציות ב-Turbo C

פונקציות Turbo C מקבלות פרמטרים ממשיים כמו כל הפרמטרים אחרים, כלומר הם נדחפים למחסנית בתוך כל הפרמטרים האחרים בסדר של מימין לשמאל. אולם בעוד בהעברת פרמטרים אין חידוש עקרוני יש בפירוש הבדל בכל הקשור להחזרת תוצאות פונקציה ממשיות. כאשר פונקצית Turbo C מחזירה תוצאה ממשית בכל גודל (float, double, long double) היא מוחזרת בתוך אוגר ה-ST(0) (כאשר יתר ה-ST(i) חייבים להיות ריקים). באחריות התוכנית הקוראת להשתמש בערך ולרוקן את ST(0).

אם נניח אנחנו מקמפלים ב-Turbo C קוד נוסף:

```
extern double fun(double);
double x, y;
  ....


 y = fun(x);
```

הקוד שמיצר הקומפילר (במודל SMALL) נראה באסמבלי עקרונית משהו כמו:

```
fld qword ptr [bp-8]                          טוען את x
sub sp,8                                      מקצה מקום במחסנית
fstp qword ptr [bp-324]                       מעתיק את x למחסנית
call near ptr _fun                            קורא לפונקציה
add sp,8                                      משחרר שטח הפרמטר
fstp qword ptr [bp-16]          ST(0) ומרוקן את  מציב את התוצאה ל-y
```

לממשי הקומפילר יכולים לעבוד בצורה הזו כי הם מנהלים מעקב על ניצול המחסנית. למתכנת הכותב תוכנית באסמבלי וקורא לפונקציה כנ"ל הגישה הזו בודאי אינה נוחה. המתכנת מן הסתם יממש את x ו-y כמשתנים סטטיים לפי שם, ויממש מצביע לשטח של הפרמטר המיועד. סכמה אפשרית אחת תהיה:

```
x DQ ?
y DQ ?

  ......

FLD x                                   טוען את x
SUB SP,8                                מקצה מקום במחסנית
MOV BX,SP                               גורם ל-BX להצביע לשטח הפרמטר
FSTP QWORD PTR SS:[BX]                  מעתיק את x לשטח הפרמטר
CALL _fun                               קורא ל-fun
ADD SP,8                                משחרר שטח הפרמטר
FSTP y                                  מעתיק את התוצאה ל-y ומרוקן את ST(0)
```

במודל SMALL לא צריך את הציון המפורש של אוגר הסגמנט ("SS:") אך הוא נחוץ במודלים כלליים יותר.

80x87 Co-Processor instructions

These instructions can be executed when a 8087/80287/80387 coprocessor is available or when using a 80486 CPU

Data Transfer and Constants

```
FLD  src       Load real:       ST(0) = src (mem32 / mem64 / mem80); push Stack
FLD  ST(i)     Load real:       ST(0) = ST(i); push Stack
FILD src       Load integer:    ST(0) = src (mem16 / mem32 / mem64); push Stack
FBLD src       Load BCD:        ST(0) = src (mem32 / mem64 / mem80); push Stack

FLDZ           Load zero:         ST(0)  =  0.0; push Stack
FLD1           Load 1:            ST(0)  =  1.0; push Stack
FLDPI          Load PI:           ST(0)  =  3.14159265... ; push Stack
FLDL2T         Load log2(10):     ST(0)  =  log2(10); push Stack
FLDL2E         Load log2(e):      ST(0)  =  log2(e);  push Stack
FLDLG2         Load log10(2):     ST(0)  =  log10(2); push Stack
FLDLN2         Load loge(2):      ST(0)  =  loge(2);  push Stack

FST    ST(i)   Store real:                                  ST(i) = ST(0)
FST    dest    Store real:     (mem32 / mem64)              dest = ST(0)
FSTP   ST(i)   Store real:                                  ST(i) = ST(0); pop Stack
FSTP   dest    Store real:     (mem32 / mem64 / mem80)      dest = ST(0); pop Stack
FIST   dest    Store integer:  (mem16 / mem32)              dest = ST(0)
FISTP  dest    Store integer:  (mem16 / mem32 / mem64)      dest = ST(0); pop Stack
FBST   dest    Store BCD:      (mem80)                      dest = ST(0)
FBSTP  dest    Store BCD:      (mem80)                      dest = ST(0); pop Stack
FXCH   ST(i)   Exchange                                   Exchange ST(0) with ST(i)
```

Compare

```
FCOM           Compare real:    Set flags as for ST(0) - ST(1)
FCOM   ST(i)   Compare real:    Set flags as for ST(0) - ST(i)
FCOM   src     Compare real:    Set flags as for ST(0) - src (mem32 / mem64)
FCOMP          Compare real:    Set flags as for ST(0) - ST(1); pop Stack
FCOMP  ST(i)   Compare real:    Set flags as for ST(0) - ST(i); pop Stack
FCOMP  src     Compare real:    Set flags as for
                               ST(0) - src (mem32 / mem64); pop Stack
FCOMPP         Compare real:    Set flags as for ST(0) - ST(1); pop Stack twice

FICOM  src     Compare integer: Set flags as for ST(0) - src (mem16 / mem32)
FICOMP src     Compare integer: Set flags as for ST(0) - src (mem16 / mem32);
                                                                 pop Stack
FTST           Test for zero:   Set flags as for ST(0) - 0.0
FUCOM   ST(i)  Unordered compare: Set flags as for ST(0) - ST(i); 486 only
FUCOMP  ST(i)  Unordered compare: Set flags as for ST(0) - ST(i); pop Stack
FUCOMPP ST(i)  Unordered compare: Set flags as for ST(0) - ST(i);
                                                       pop Stack twice
FXAM           Examine: Examine ST(0) (set condition codes)
```

Arithmetic

```
FADD            Add real:          ST(1) = ST(1) + ST(0); pop Stack
FADD   src      Add real:          ST(0) = ST(0) + src (mem32 / mem64)
FADD   ST,ST(i) Add real:          ST(0) = ST(0) + ST(i)
FADD   ST(i),ST Add real:          ST(i) = ST(i) + ST(0)
FADDP  ST(i),ST Add real:          ST(i) = ST(i) + ST(0); pop Stack
FIADD  src      Add integer:       ST(0) = ST(0) + src (mem16 / mem32)

FSUB            Subtract real:             ST(1) = ST(1) - ST(0); pop Stack
FSUB   src      Subtract real:             ST(0) = ST(0) - src (mem32 / mem64)
FSUB   ST,ST(i) Subtract real:             ST(0) = ST(0) - ST(i)
FSUB   ST(i),ST Subtract real:             ST(i) = ST(i) - ST(0)
FSUBP  ST(i),ST Subtract real:             ST(i) = ST(i) - ST(0); pop
                                                                  Stack
FSUBR           Subtract real Reversed:    ST(1) = ST(0) - ST(1); pop Stack
FSUBR  src      Subtract real Reversed:    ST(0) = src(mem32 / mem64) - ST(0)
FSUBR  ST,ST(i) Subtract real Reversed:    ST(0) = ST(i) - ST(0)
FSUBR  ST(i),ST Subtract real Reversed:    ST(i) = ST(0) - ST(i)
FSUBRP ST(i),ST Subtract real Reversed:    ST(i) = ST(0) - ST(i); pop
                                                                  Stack

FISUB   src     Subtract integer:      ST(0) = ST(0) - src (mem16 / mem32)
FISUBR  src     Subtract integer Reversed:  ST(0) =
                                              src (mem16 / mem32) - ST(0)

FMUL            Multiply real:     ST(1) = ST(1) * ST(0); pop Stack
FMUL   src      Multiply real:     ST(0) = ST(0) * src (mem32 / mem64)
FMUL   ST,ST(i) Multiply real:     ST(0) = ST(0) * ST(i)
FMUL   ST(i),ST Multiply real:     ST(i) = ST(0) * ST(i)
FMULP  ST(i),ST Multiply real:     ST(i) = ST(0) * ST(i); pop Stack
FIMUL  src      Multiply integer:  ST(0) = ST(0) * src (mem16 / mem32)

FDIV            Divide real:       ST(1) = ST(1) / ST(0); pop Stack
FDIV   src      Divide real:       ST(0) = ST(0) / src(mem32 / mem64)
FDIV   ST,ST(i) Divide real:       ST(0) = ST(0) / ST(i)
FDIV   ST(i),ST Divide real:       ST(i) = ST(i) / ST(0)
FDIVP  ST(i),ST Divide real:       ST(i) = ST(0) / ST(i); pop Stack
FDIVR           Divide real Reversed:    ST(1) = ST(0) / ST(1);pop Stack
FDIVR  src      Divide real Reversed:    ST(0) = src(mem32 / mem64) / ST(0)
FDIVR  ST,ST(i) Divide real Reversed:    ST(0) = ST(i) / ST(0)
FDIVR  ST(i),ST Divide real Reversed:    ST(i) = ST(0) / ST(i)
FDIVRP ST(i),ST Divide real Reversed:
                                         ST(i) = ST(0) / ST(i); pop Stack
FIDIV   src     Divide integer:    ST(0) = ST(0) / src (mem16 / mem32)
FIDIVR  src     Divide integer Reversed:
                                         ST(0) = (mem16 / mem32) src / ST(0)

FSQRT           Square Root:          ST(0) = sqrt(ST(0))
FXTRACT         Extract Exponent:     push Stack; ST(0) = exponent of ST(0)
FPREM           Partial Reminder:     ST(0) = ST(0) MOD ST(1)
FPREM1          Same as FPREM, but in IEEE standard     486 only
FRNDINT         Round to nearest integer: ST(0) = INT( ST(0) );
                                          depends on RC flag
FABS            Get Absolute value:    ST(0) = ABS( ST(0) )
FCHS            Change Sign            ST(0) = 0 - ST(0)
F2XM1           Compute 2**X - 1       ST(0) = 2 ** ST(0) - 1
FSCALE          Scale                  ST(0) = ST * (2 ** ST(1))
```

```
    ----------------------
      Transcendental
    ----------------------

FCOS        Cosine:                 ST(0) = COS( ST(0) )
FPTAN       Partial Tanget:         ST(0) = TAN( ST(0) ); push Stack;
                                                          ST(0) = 1.0
FPATAN      Partial Arctanget:      ST(1) = ARCTAN( ST(1) / ST(0) );
                                                          pop Stack
FSIN        Sine:                   ST(0) = SIN( ST(0) )
FSINCOS     Sine and Cosine:        ST(1) = SIN( ST(0) )
                                    ST(0) = COS( ST(0) )
                                      other values are pushed (twice)
FYL2X     Compute Y * log2(X); ST(0) is Y; ST(1) is X;
                                                   This replaces ST(0)
FYL2XP1   Compute Y * log2(X+1); ST(0) is Y; ST(1) is X;
            This replaces ST(0) and
                          ST(1) with: ST(0) * log2( ST(1) + 1)

    ----------------------
      Processor control
    ----------------------

FINIT           Initialize FPU
FSTSW AX        Store Status Word:    AX = FPU MSW
FSTSW dest      Store Status Word:    (mem16) dest = FPU MSW
FLDCW src       Load  Control Word:   FPU CW = src (mem16)
FSTCW src       Store Control Word:   (mem16) dest = FPU CW
FCLEX           Clear Exceptions
FSTENV dest     Store Environment: Store status, control, tag words and
                                   exeption pointers at memory dest.
FLDENV src      Load  Environment: Load environment from memory src.
FSAVE  dest     Save FPU state:    Store FPU state into 94-byte memory dest.
FRSTOR src      Resore FPU state:  Restore FPU state as saved by FSAVE from
                                                              memory src.
FINCSTP         Increment FPU stack PTR
FDECSTP         Decrement FPU stack PTR
FFREE ST(i)     Mark reg ST(i) as unused
WAIT / FWAIT    Synchronize FPU & CPU: Halt CPU until FPU finishes
                                                        current opcode
FNOP            No Operation
```