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.

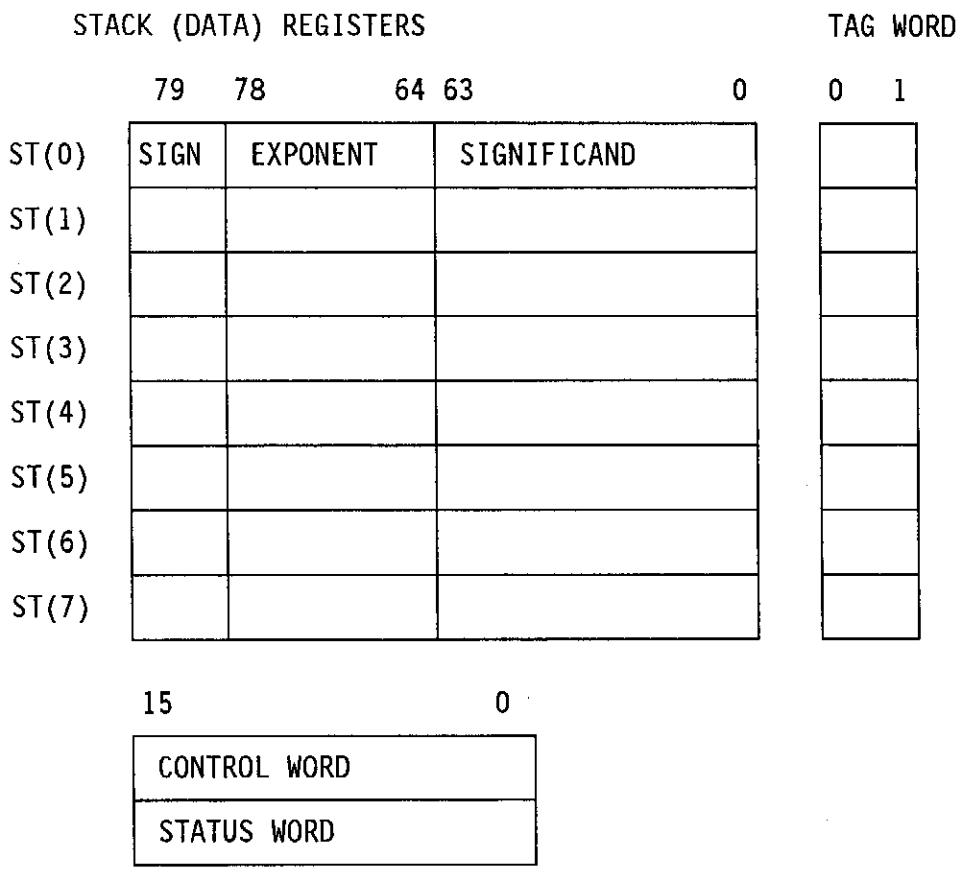
אוגר 16 ביט ה- Control Word מכיל דגלים המשפיעים על תפקוד ה-CPU. למשל ניתן להציב לתוכו ערך שמוריד את מידת הדיוק של החישובים שלו מ-80 ביט לפחות (64, 32 ביט).

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 32, 16 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.

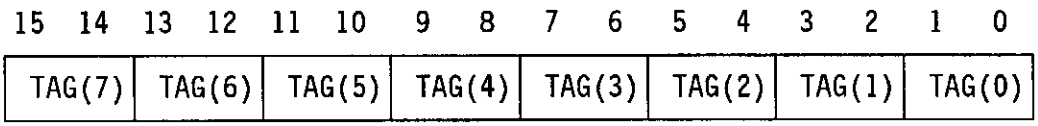
האוגרים נקראים Stack Registers מפני שהם מתפקדים חלק גדול מהזמן כמחסנית של עד שמונה מספרים. במימוש ביטויים אריתמטיים קיים מצב נפוץ שבו אנחנו מבצעים פעולה אריתמטית על שני מספרים (נניח כפל), ומאותו רגע ואילך אין לנו עינין במספרים עצמם אלא רק בתוצאה. פקודות המכונה של המעבד

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



ערכים אפשריים לכל TAG(i):

00 - Valid - ערך ממשי כלשהוא, מלבד אפס

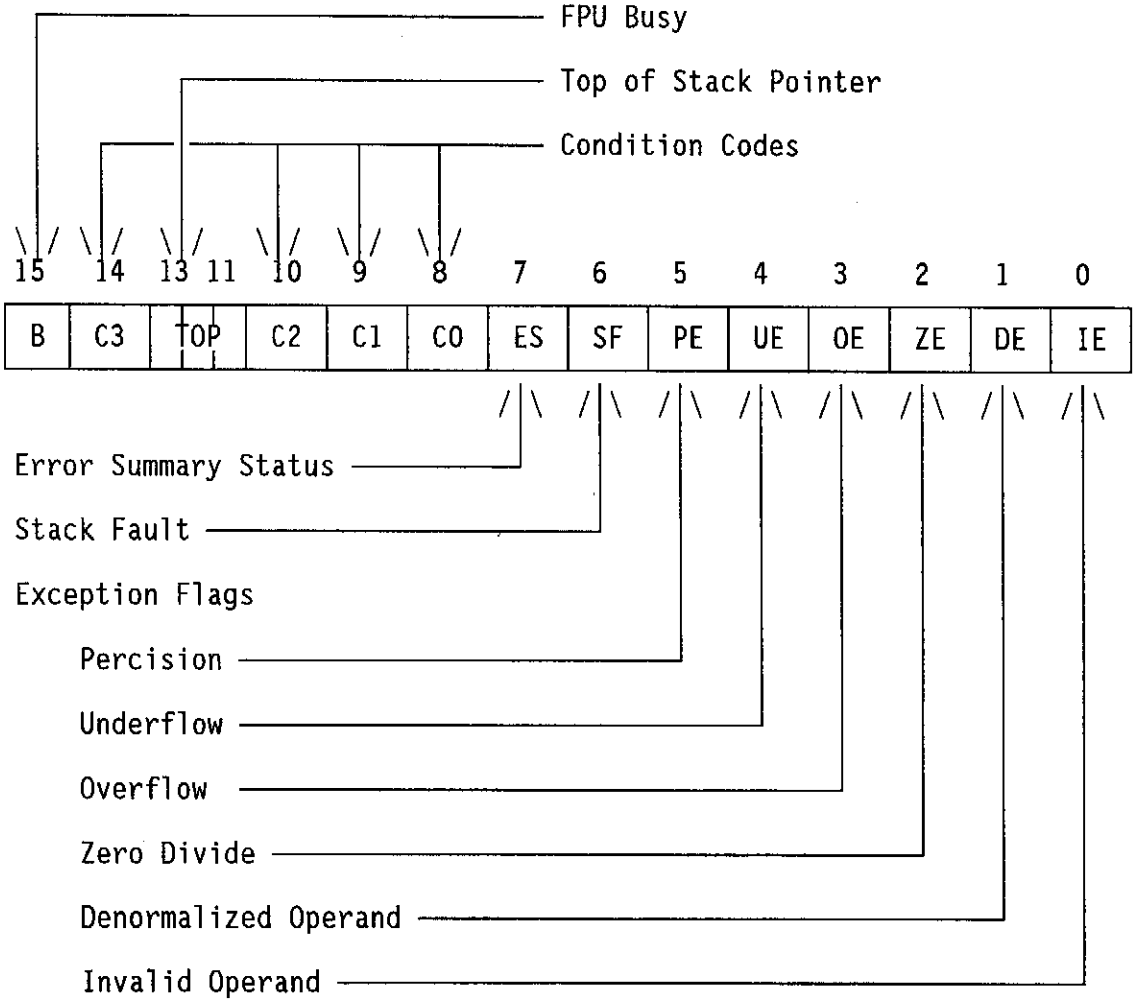
01 - Zero - ערך אפס,

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

11 - Empty - ריק.

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

מבנה ה- STATUS WORD



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.

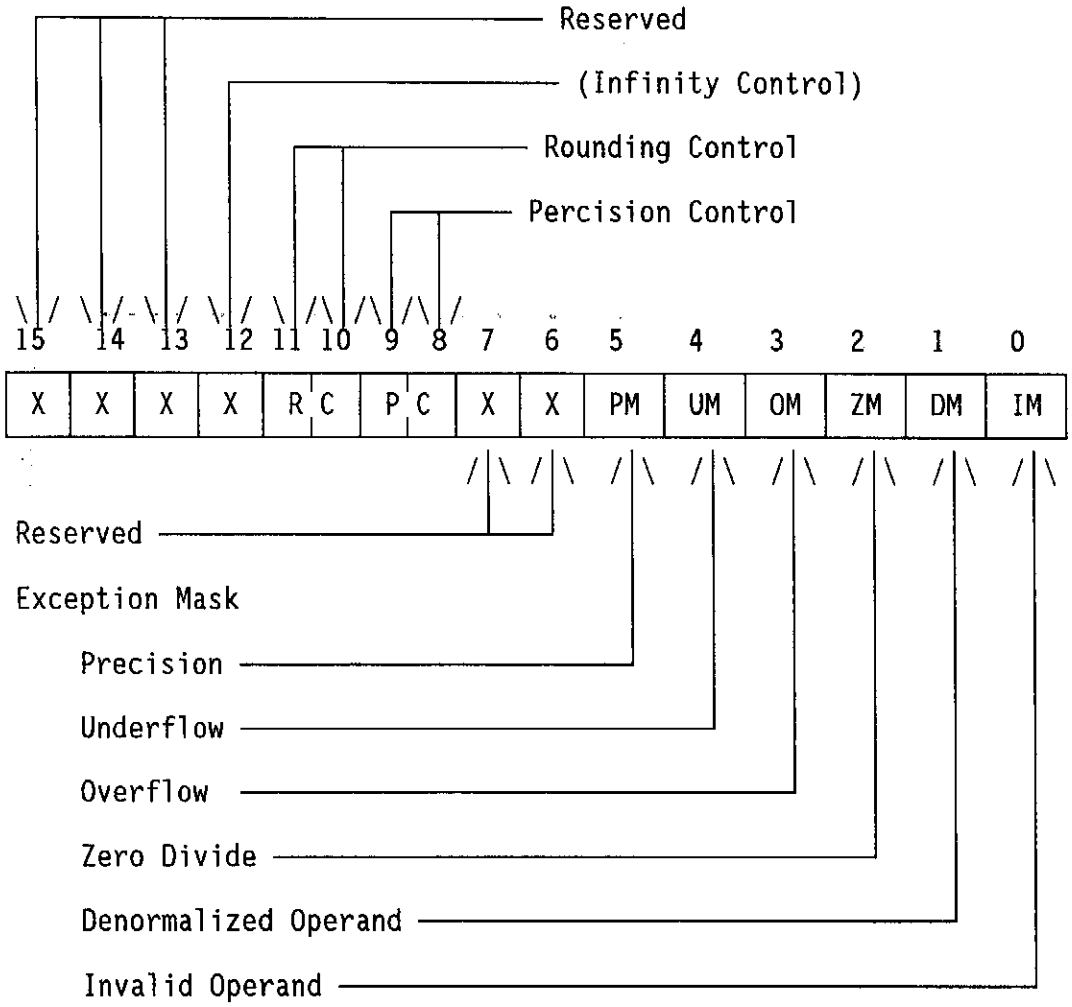
הערך של ה-Status Word מתעדכן בעקבות כל פעולה.

ה-Condition Codes - C3, C2, C1, C0 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.

מבנה ה- CONTROL WORD



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

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(0) * (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(1) is Y; ST(0) is X; pop Stack;
             This replaces ST(0) with:  ST(1) * log2( ST(0))
FYL2XP1    Compute Y * log2(X+1); ST(1) is Y; ST(0) is X;
             This replaces ST(0) and ST(1) with: ST(1) * log2( ST(0) + 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
```