תקציר מספר 11

יצוג מספרים ממשיים

יצוג מספרים ממשיים - תאוריה

היצוג של מספרים ממשיים במחשב מזכיר את מה שקרוי כתיבה "מדעית" של מספרים:
בשימושים פיזיקליים ומדעיים אחרים אנחנו נוטים לכתוב מספר בצורה כמו

$$-5.71325 * 10^{13}$$

שמשמעותו למעשה

$$(-1)*(5*10^{0} + 7*10^{-1} + 1*10^{-2} + 3*10^{-3} + 2*10^{-4}) * 10^{13}$$

כלומר הוא מורכב למעשה משלושה חלקים: סימן, ערך המיוצג בצורה מנורמלת (בין 1.0 ל- 9.9999...) וסדר גודל (חזקה של 10).

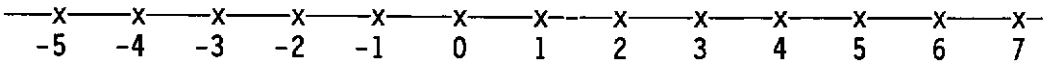
בחומרת מחשב המספרים הממשיים מיוצגים באופן עקרוני באותה צורה, אבל בבינארי במקום בדצימל. היצוגים של מספרים בחומרת המחשב (שלמים וממשיים) הם בגדלים קבועים מראש. במידה ורוצים ליצג מספרים בגודל רצוי או שרירותי יש לעשות זאת בתוכנה.

לפיכך "מספר ממשי" במחשב הוא למעשה שבר עם מספר סופי של ספרות משמעותיות + הכפלה בגורם הקובע סדר גודל + סימן.

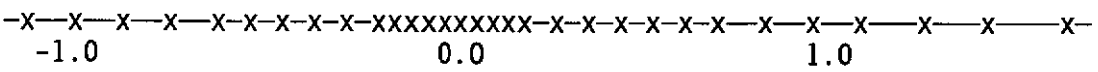
כאן אנחנו כבר יכולים לעמוד על הבדל בין היצוג של מספרים שלמים לבין היצוג של מספרים ממשיים:

בעוד מספרים שלמים המיוצגים במחשב נמצאים במרווחים שווים על הציר הממשי בין מקסימום למינימום (+32767 -- -32768) במקרה של מספרים 16 ביט עם סימן), זה אינו נכון עבור מספרים ממשיים. קבוצת המספרים הממשיים המיוצגים בחומרת המחשב נעשים צפופים יותר ככל שמתקרבים ל-0.0. בערך מחצית מכל המספרים נמצאים בין +1.0 ל- -1.0. ככל שמתרחקים מ-0.0 המרווחים ביניהם הולכים וגדלים.

שלמים



ממשיים



<u>יצוג מספרים בפועל</u>

מספר ממשי מיוצג בחומרת המחשב בצורה:

א. יצוג מיוחד לאפס: כל הביטים אפס (0000000...000B).

ב. כל מספר <u>שונה</u> מאפס בצורה:

SIGN	EXPONENT	SIGNIFICAND

כאשר

SIGN - ביט סימן (0 חיובי, 1 שלילי)

EXPONENT - מספר שלם שהוא חזקה של 2

SIGNIFICAND - סידרה של ספרות בינאריות המיצג מספר ממשי מנורמל (בין 1.0 ל- 1.9999...99). לפעמים עם ה-1.0 המוביל ולפעמים לא.

הערך של המספר הוא

$$(-1)^{SIGN} * (1.0 + SIGNIFICAND[k] * 2^{-k}) * 2^{EXPONENT}$$

<u>דוגמא</u>:

את המספר 1.625 ניתן ליצג:

$$1.625 = 1.0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} = 1.0 + 0.5 + 0 + 0.125$$

לפיכך

SIGN = 0
EXPONENT = 0
SIGNIFICAND (1.0 -ללא יצוג ל) = 1010000...0

היצוג הזה הוא היצוג התאורטי.

בפועל עדיין חסרים מספר פרטים: כמה ביטים מוקצה ל-EXPONENT ול-SIGNIFICAND, האם ה-1.0 מיוצג ואיך מיוצג ה-EXPONENT.

2. <u>יצוג מספרים ממשיים ב-PC</u>.

ב-PC המספרים הממשיים מיוצגים לפי הסטנדרד של ה-IEEE משנת 80:

א. גודל המספרים הממשיים הינם 32 ביט, 64 ביט, ו-80 ביט.
יצוג 32 ביט הוא ה-float של C, ובאופן כללי יותר נקרא דיוק רגיל או דיוק יחיד Single Percision.
יצוג 64 ביט הוא ה-double של C, ובאופן כללי יותר נקרא דיוק כפול double persion.
יצוג 80 ביט הוא ה-long double או באופן כללי יותר נקרא דיוק מורחב Extended Percision.

ב. חלוקת הביטים לכל האורך הוא כלהלן:

32 ביט: 1 ביט ל-SIGN, 8 ביט ל- EXPONENT, 23 ביט ל-SIGNIFICAND.

64 ביט: 1 ביט ל-SIGN, 11 ביט ל- EXPONENT, 52 ביט ל-SIGNIFICAND.

80 ביט: 1 ביט ל-SIGN, 15 ביט ל- EXPONENT, 64 ביט ל-SIGNIFICAND.

מנקודת ראות של <u>דיוק</u> בספרות <u>עשרוניות</u>, משתנים בדיוק רגיל (32 ביט) מישמים דיוק של 7 - 6 ספרות עשרוניות, דיוק כפול מישמים דיוק של 16 - 15 ספרות עשרוניות, דיוק מורחב (80 ביט) מישמים דיוק של 19 - 18 ספרות עשרוניות. איך מחשבים את הדיוק הזה נראה בהמשך.

ג. ה- 1.0 <u>אינו</u> מיוצג ב-32 ביט וה-64 ביט, אבל <u>מיוצג</u> ב-80 ביט.

ד. היצוג של ה-EXPONENT ע"י מספר שלם חיובי עם הטיה (BIAS).
ערך ה-BIAS תלוי באורך המספר:
127 עבור 32 ביט.
1023 עבור 64 ביט.
16383 עבור 80 ביט.

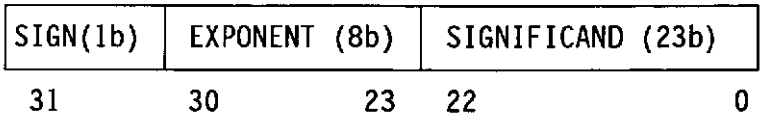
אגב, הערכים הקיצוניים של ה-EXPONENT אינם בשימוש על מנת לאפשר

יצוג של אינסוף (infinity), לא מספר (Not a Number - NaN), ושדה ללא ערך (Empty).

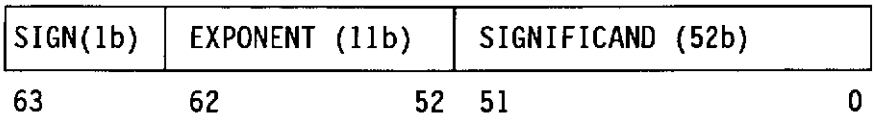
ה-EXPONENT עבור 32 ביט הוא בין 127 ל- 126- ולא בין 128 ל- 127- כפי שאפשר היה לעשות. כמו כן ה-EXPONENT של 64 ביט הוא בין 1023 ל- 1022- (ולא בין 1024 ל- 1023-), ה-EXPONENT של 80 ביט הוא בין 16383 ל- 16382- (ולא בין 16384 ל- 16383-).

לפיכך שלושת האפשרויות של יצוג מספרים ממשיים הם:

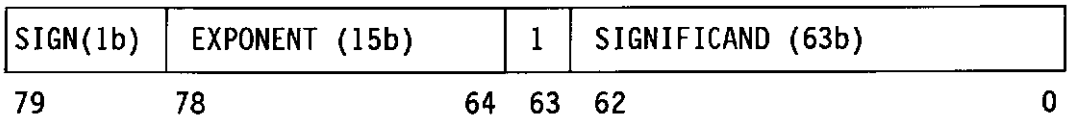
Single Percision (float)



Double Percision (double)



Extended Percision (long double)



<u>תחומים</u>:

טווח הערכים החיוביים (ערך מוחלט) שניתן ליצג ביצוג המספרים במחשב הינו:

32 ביט:

החזקות של 2 הם בין 126- ל- 127+.
לפיכך הערכים הם בתחום:

בין $1.0 * 2^{-126} = 1.175 * 10^{-38}$

לבין $1.9999... * 2^{127} = 3.4 * 10^{38}$

בצורה דומה עבור 64 ביט החזקות של 2 הם בין 1022- לבין 1023+ והערכים הם:

בין $1.0 * 2^{-1022} = 2.225 * 10^{-308}$

לבין $1.9999... * 2^{1023} = 1.798 * 10^{308}$

ובצורה דומה עבור 80 ביט החזקות של 2 הם בין 16382- לבין 16383+ והערכים הם:

בין $1.0 * 2^{-16382} = 3.362 * 10^{-4932}$

לבין $1.9999... * 2^{16383} = 1.189 * 10^{4932}$

באופן כללי, אם ביצוג מספר ממשי יש ב-SIGNIFICAND M ביטים משמעותיים, אזי הדיוק של ה-SIGNIFICAND הוא פשוט 2^{-M}.

בכדי להבין מדוע, אולי קל יותר לחשוב במושגים עשרוניים. לכל מספר ממשי בישר הממשי R יש יצוג עשרוני מנורמל (שעשוי להיות אינסופי). אם נניח שאנחנו קוטעים את היצוג של ה-SIGNIFICAND של מספר אחרי 3 ספרות אחרי הנקודה העשרונית, אובדן הדיוק של ה-SIGNIFICAND חסום ע"י 0.00099999... או בעצם כמעט 0.001 או 10^{-3}.

המקרה הבינארי באופן עקרוני אותו דבר. ההבדלים הם ש-2 מחליף את 10 בחזקה, וכל הביטים מיצגים דיוק שמאחורי הנקודה הבינארית (המקדם הוא תמיד 1.0 ולא מיוצג במפורש).

לפיכך הדיוק של ה-SIGNIFICAND בדיוק רגיל (23 מתוך 32) הוא $2^{-23} = 1.19*10^{-7}$. משמעות הדבר שעם אנחנו מבצעים פעולה אחת על שני מספרים המיוצגים באופן מדויק (נניח חילוק של שני שלמים), אפשר יהיה לסמוך על 6 הספרות העשרוניות הראשונות של התוצאה, אבל לא על השביעי.

בצורה דומה הדיוק של ה-SIGNIFICAND בדיוק כפול הוא $2^{-52} = 2.22*10^{-16}$.

הדיוק של ה-SIGNIFICAND בדיוק מורחב הוא $2^{-63} = 1.08*10^{-19}$.

דוגמאות למספרים ממשיים ויצוגם:

הערך 1.625 מיוצג כ-

32 ביט: 3FD00000h או

SIGNIFICAND = 101b

0011 1111 1101 0000 0000 0000 0000 0000b

127 = EXPONENT

64 ביט: 3FFA000000000000h או

SIGNIFICAND = 101b

0011 1111 1111 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000h

1023 = EXPONENT

80 ביט: 3FFFD0000000000000000h או

SIGNIFICAND = 1101b

0011 1111 1111 1111 1101 0000 0000 ..... 0000 0000

16383 = EXPONENT

דוגמא:

הערך - (1.0 + 0*0.5 + 1*0.25)- = 1.25- מיוצג כ-

SIGNIFICAND = 01b ללא יצוג ה- 1.0.

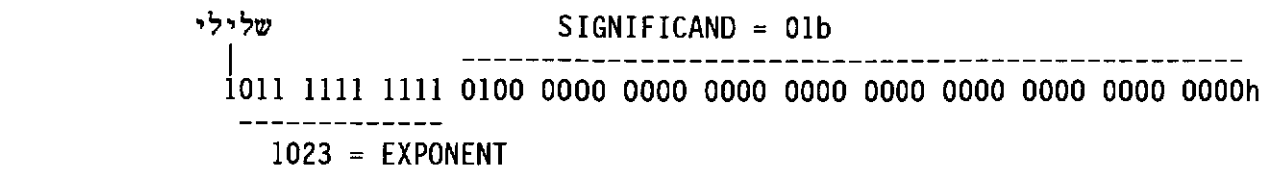
היצוג 32 ביט יהיה BFA00000h

שלילי SIGNIFICAND = 01b

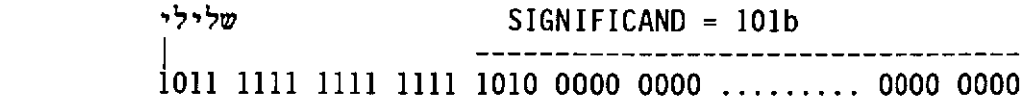
1011 1111 1010 0000 0000 0000 0000 0000b

127 = EXPONENT

64 ביט: BFF4000000000000h או



80 ביט: BFFFA0000000000000000h או



<u>דוגמא</u>:

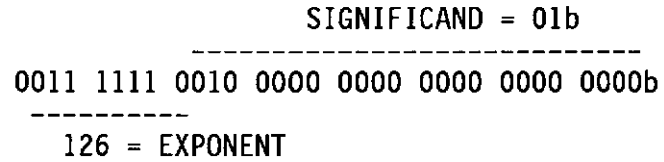
יצוג הערך 0.625:

$$0.625 = 1.25 * 2^{-1} = (1.0 + 0*0.5 + 1*0.25) * 2^{-1}$$

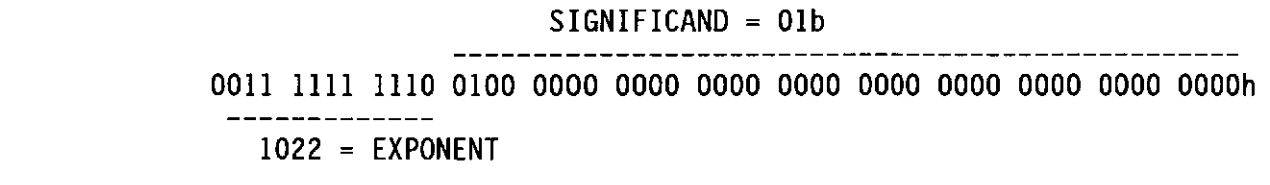
לפיכך ה-EXPONENT (ללא ה-BIAS) הוא 1-. עם ה-BIAS ה-EXPONENT יהיה:
126 = 127 + 1- עבור 32 ביט,
1022 = 1023 + 1- עבור 64 ביט,
16382 = 16383 + 1- עבור 80 ביט.

ה-SIGNIFICAND התאורטי (ללא יצוג 1.0) שוה (כמו בדוגמא הקודמת) ל-01b.

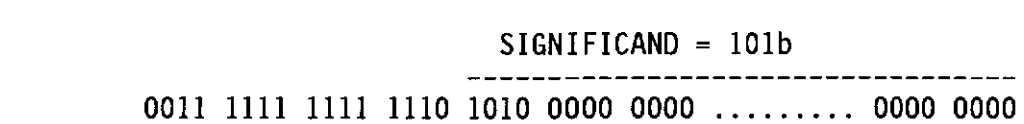
היצוג 32 ביט יהיה 3F200000h



64 ביט: 3FE4000000000000h או



80 ביט: 3FFEA0000000000000000h או



דוגמא:

יצוג הערך 576.0:

$$576.0 = 1.125 * 512 = 1.125 * 2^{9}$$

כאשר $1.125 = 1.0 + 0*0.5 + 0*0.25 + 1*0.125$.

לפיכך ה-EXPONENT (ללא ה-BIAS) הוא 9. עם ה-BIAS ה-EXPONENT יהיה:
136 = 127 + 9 עבור 32 ביט,
1032 = 1023 + 9 עבור 64 ביט,
16392 = 16383 + 9 עבור 80 ביט.

ה-SIGNIFICAND התאורטי (ללא יצוג 1.0) הוא 001b.

היצוג 32 ביט יהיה 44100000h

SIGNIFICAND = 001b

0100 0100 0001 0000 0000 0000 0000 0000b

136 = EXPONENT

64 ביט: 4082000000000000h או

SIGNIFICAND = 001b

0100 0000 1000 0010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000h

1032 = EXPONENT

80 ביט: 400890000000000000h או

SIGNIFICAND = 1001b

0100 0000 0000 1000 1001 0000 0000 ..... 0000 0000

16392 = EXPONENT

דוגמא:

דוגמא ליצוג בינארי לא סופי

נבדוק את היצוג של המספר התמים 1.4:

לפיכך $0.4 < 0.5$

$1.375 = 1.0 + 0*0.5 + 1*0.25 + 1*0.125$ זה עדיין < 1.4.

$0.0625 = .2^{-4}$ $1.4375 = 1.375 + 0.0625$ זה יותר מדי.

$0.03125 = .2^{-5}$ $1.40625 = 1.375 + 0.03125$ זה עדיין יותר מדי.

2^{-6} = 0.015625 . 1.375 + 0.015625 = 1.390625. קרוב יותר אבל עדיין לא מספיק. אבל התקרבנו ל-1.4 עד כדי 1.390625 - 1.4 = 0.009375.

אפשר להגיד שה-SIGNIFICAND 01100lb המקביל ל-

1.0 + 0*0.5 + 1*0.25 + 1*0.125 + 0*0.0625 + 0*0.03125 + 1*0.015625 = 1.390625

מיצג את 1.4 בדיוק של 6 ספרות בינאריות.

אפשר להמשיך כך ... לעולמים. למרות של- 1.4 יש יצוג עשרוני סופי, אין לו יצוג בינארי סופי! לא לכל מספר שיש לו יצוג עשרוני סופי, יש יצוג בינארי סופי. למעשה רוב המספרים שיש להם יצוג עשרוני סופי אינם כאלו!

זה למעשה אינו חדש. גם ביצוג העשרוני אין דיוק סופי למספרים כמו 1 / 3 או 2 / 7. שלא לדבר על מספרים כמו e או ד. תמיד פעולות על מספרים ממשיים הם עד דיוק כזה או אחר.

לסיכום הנושא הזה היצוג של 1.4 הוא:

32 ביט: 3FB33333h או

SIGNIFICAND = 01100l.....b

0011 1111 1011 0011 0011 0011 0011 0011b

127 = EXPONENT

64 ביט: 3FF6666666666666h או

SIGNIFICAND = 01100l.....b

0011 1111 1111 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110h

1023 = EXPONENT

80 ביט: 3FFFB333333333333333h או

SIGNIFICAND = 1011001.....b

0011 1111 1111 1111 1011 0011 0011 ..... 0011 0011

16383 = EXPONENT

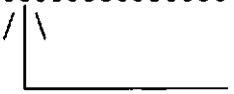
יצירת דוגמאות נוספות:

אם ברצונך לדעת איך מספר ממשי כלשהוא מיוצג (מבלי לחשב זאת בעצמך) דרך פשוטה לעשות זאת הוא להגדיר בתוכנית אסמבלר כלשהוא (נניח myprog.asm):

```
A1 DD 1.625
AA1 DQ 1.625
AAA1 DT 1.625
```

ובצע tasm /la myprog.asm של הקובץ.
יווצר לכך קובץ myprog.lst, שבתוכו תראה (דרך תוכנית עריכה):

```
 6 0000                                .DATA
 7 0000  3FD00000                A1    DD  1.625
 8 0004  3FFA000000000000        AA1   DQ  1.625
 9 000C  3FFFD000000000000000    AAA1  DT  1.625
```



היצוג הבינארי (בהקסה) נמצא כאן

תקציר מספר 12

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.

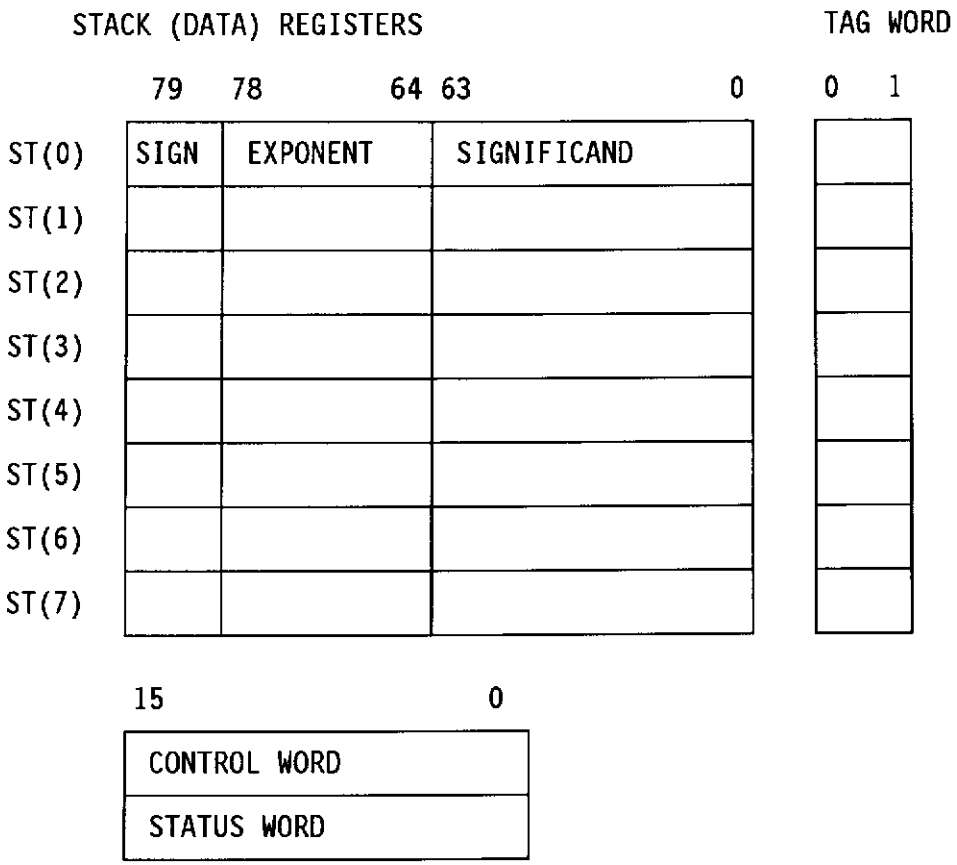
אוגר 16 ביט ה- Control Word מכיל דגלים המשפיעים על תפקוד ה-CPU. למשל ניתן להציב לתוכו ערך שמוריד את מידת הדיוק של החישובים שלו מ-80 ביט לפחות (64, 32 ביט).

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 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.

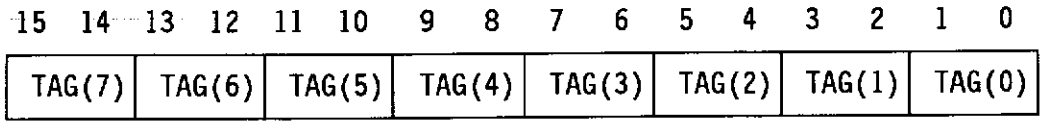
האוגרים נקראים Stack Registers מפני שהם מתפקדים חלק גדול מהזמן כמחסנית של עד שמונה מספרים. במימוש ביטויים אריתמטיים קיים מצב נפוץ שבו אנחנו מבצעים פעולה אריתמטית על שני מספרים (נניח כפל), ומאותו רגע ואילך אין לנו עינין במספרים עצמם אלא רק בתוצאה. פקודות המכונה של המעבד

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



ערכים אפשריים לכל TAG(i):

00 - Valid - ערך ממשי כלשהוא, מלבד אפס

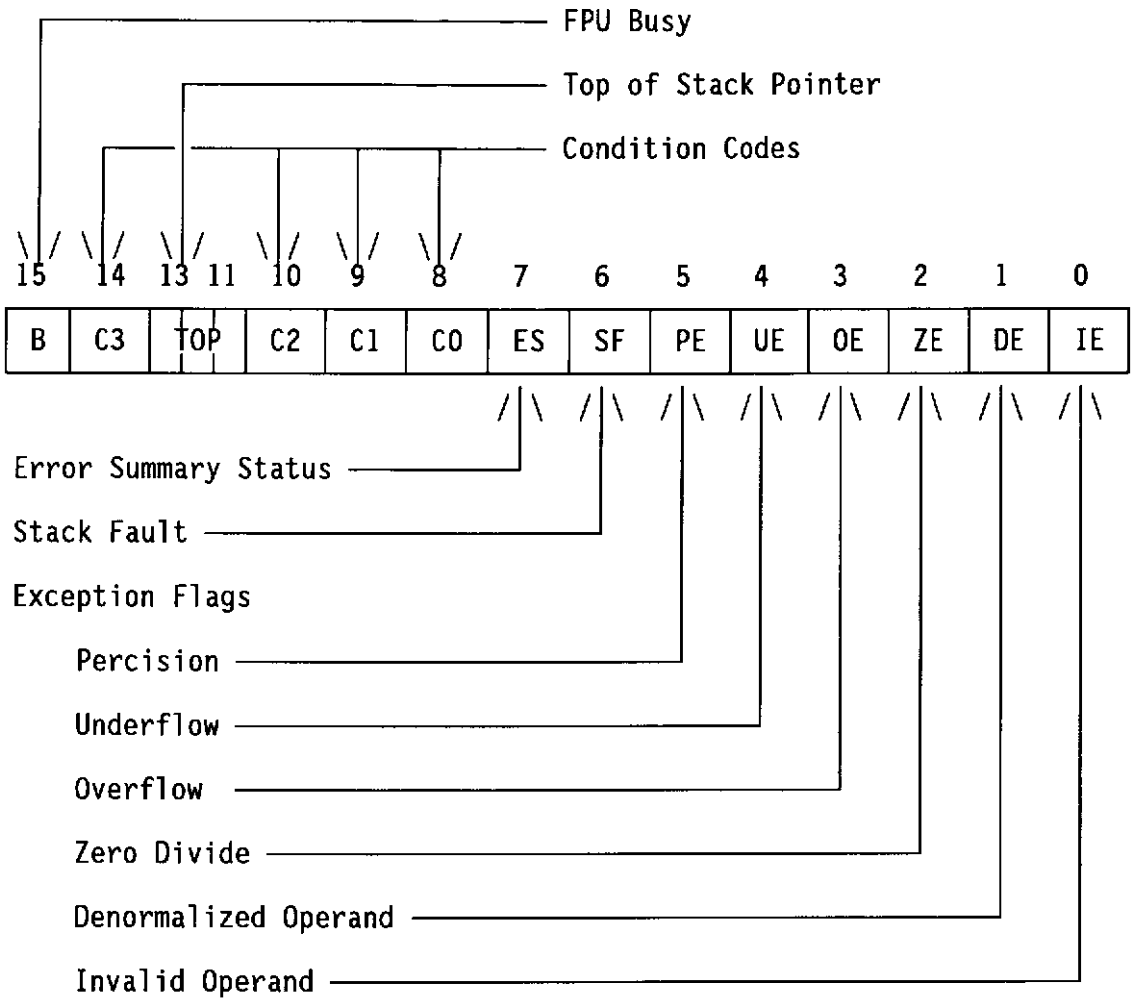
01 - Zero - ערך אפס,

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.

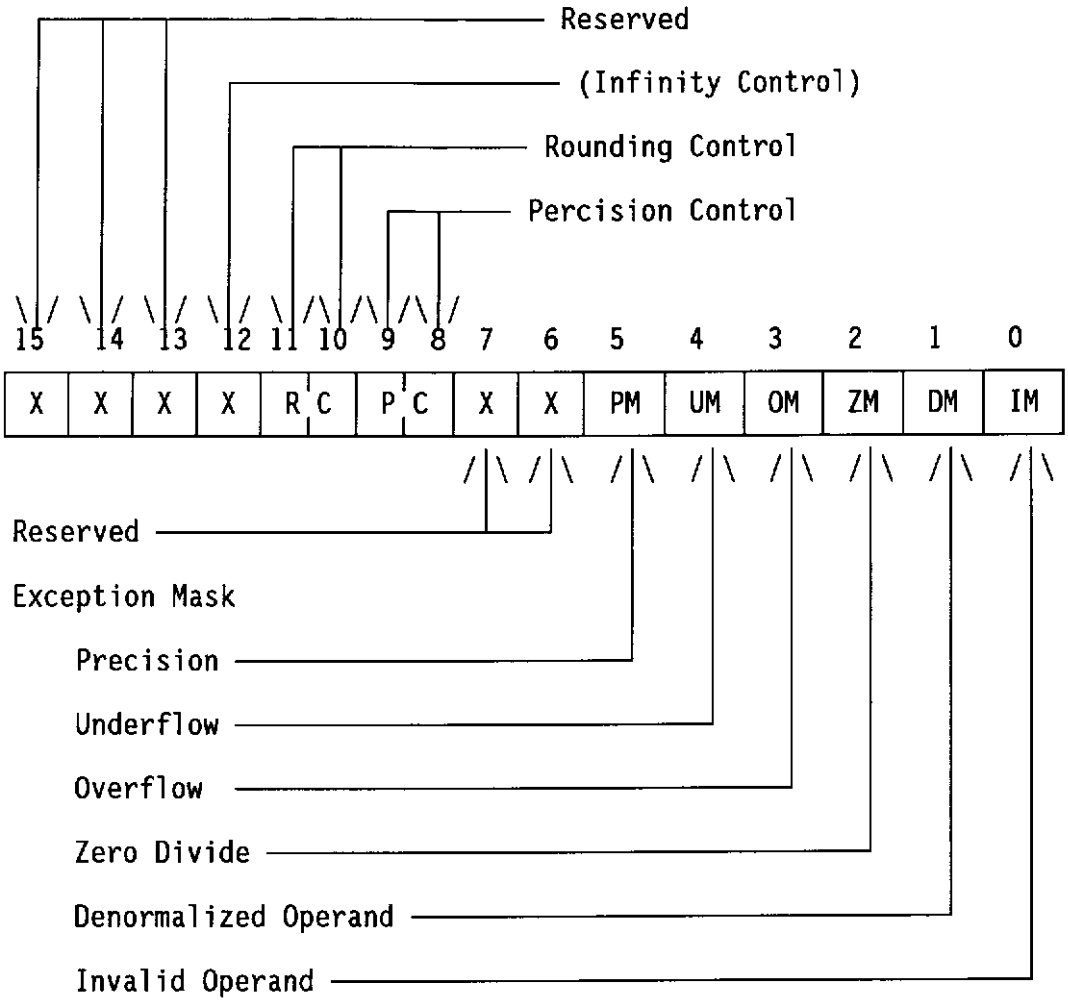
הערך של ה-Status Word מתעדכן בעקבות כל פעולה.

C3, C2, C1, C0 - 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.

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 * (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
```