ЭВМ и Периферийные устройства

лекция 9

Вещественные числа

Существует 2 способа хранения вещественных чисел.

1. С фиксированной точкой.
Основная идея- мы договариваемся где в регистре (или регистрах) проходит граница между целой и дробной частью числа.
Например пусть в 8 битном регистре старшие байта отвечают за целую часть, младшие – за дробную. Тогда число 15.937 можно представить как:
$2^3+2^2+2^1+2^0+2^{-1}+2^{-2}+2^{-3}+2^{-4}=8+4+2+1+1/2+1/4+1/8+1/16=15.9375$

У данного способа есть преимущество – проще контролировать округление.

Основной недостаток- число всегда занимает один и тот же размер. Этот способ очень неэффективен для хранения чисел.

IIII.FFFF
От 0000.0001 до 9999.9999.

Вещественные числа

2. С плавающей точкой.

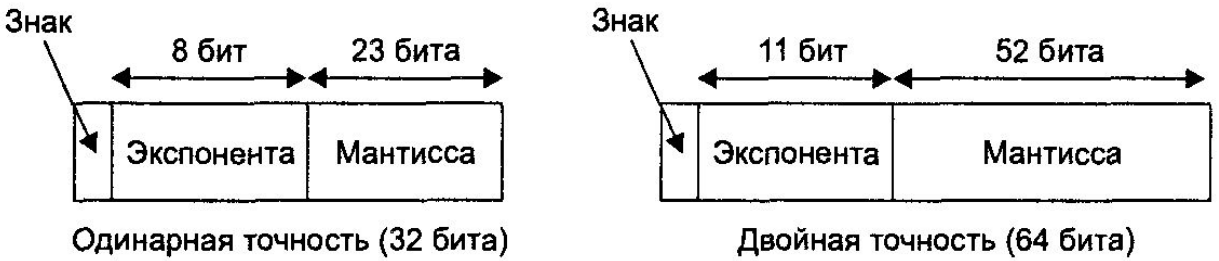
Основная идея число представляется в виде набора компонентов:

-118.625

$$-\ 1.18625\ *\ 10^{+\ 2}$$

Знак мантиссы — Мантисса — Знак порядка — Порядок (экспонента)

-1.18625E+2



Такой способ хорош тем, что позволяет представлять огромные диапазоны чисел не требую при этом больших затрат памяти:

6.63E-34

Недостаток: сложнее контролировать округление.

Вещественные числа

2. С плавающей точкой.

То же самое, но в двоичном представлении:

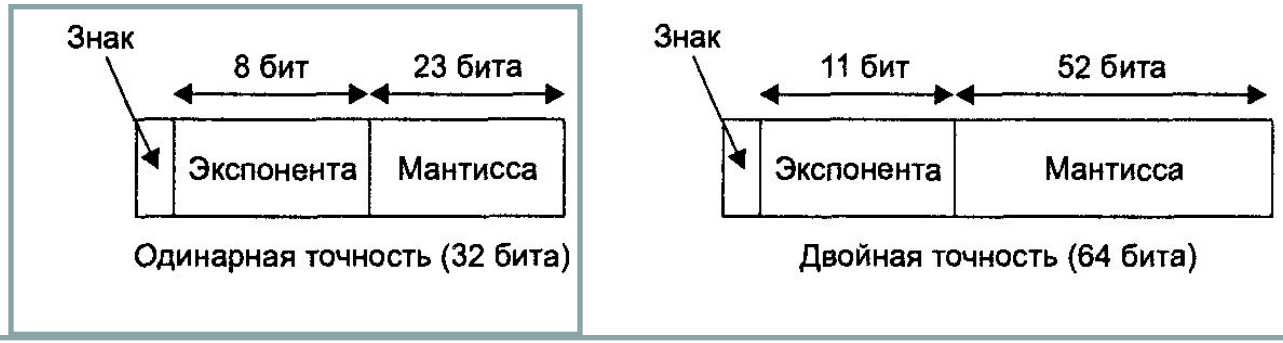
-118.625 118d = 1110110b

$0.625d = 1 \cdot 2^{0} + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.101b$

-1110110.101b

$$-1.110110101 * 2^{+6}$$

Знак мантиссы — Мантисса — Порядок (экспонента)



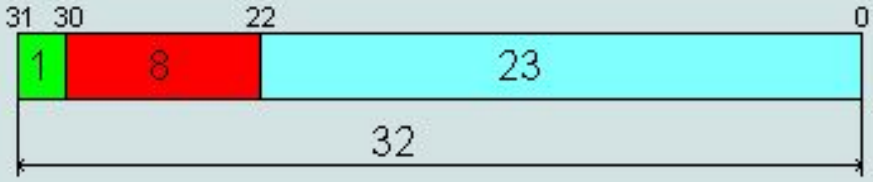
Стандарт IEEE 754

$$F = (-1)^S \; 2^{(E-2^{(b-1)}+1)} (1+M/2^n)$$

- S - бит знака, если S=0 - положительное число; S=1 - отрицательное число;
- E - смещенная экспонента двоичного числа;
  exp_2 = E - ($2^{(b-1)}$ - 1) - экспонента двоичного нормализованного числа с плавающей точкой;
  ($2^{(b-1)}$ -1) - заданное смещение экспоненты (в 32-битном ieee754 оно равно +127) . b- число бит экспоненты;
- M - остаток мантиссы двоичного нормализованного числа с плавающей точкой;
- n- число байтов мантиссы;
- F- десятичное число.

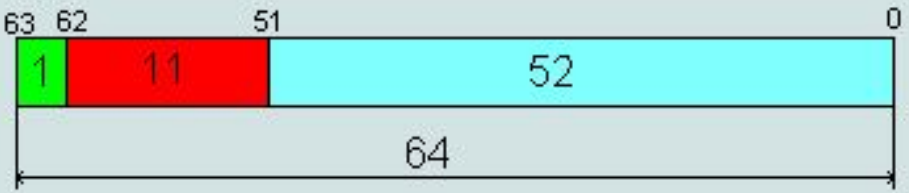
Стандарт IEEE 754

Для float (32 бит):



$$F = (-1)^S 2^{(E-127)} (1+M/2^{23})$$

Для double (64 бит):



$$F = (-1)^S \ 2^{(E-1023)} (1+M/2^{52})$$

Типа данных для больших чисел

dd или dword для хранения 4-байтов (float)
dq или qword для хранения 8 байтов (double)
dt или tbyte для хранения 10 байтов.

Обратите внимание, на то, что например, процедура MASM32 FpuFLtoA (мы рассмотрим её ниже) ждет в качестве первого параметра именно адрес 10 байтовой переменной

Проблема

Мы и можем хранить вещественные числа с плавающей точкой в памяти.

Но при работе с вещественными числами нельзя «просто» применять обычные команды, типа ADD, SUB, MUL. Они не дадут нужный результат.

Чтобы их применить, нужно будет выделить из числа мантиссу и экспоненты и произвести прочие вспомогательные действия.

К счастью всё это автоматизировано на аппаратном уровне благодаря математическому сопроцессору.

Математический сопроцессор

Математический сопроцессор - сопроцессор для расширения командного множества центрального процессора и обеспечивающий его функциональностью модуля операций с плавающей запятой, для процессоров, не имеющих интегрированного модуля.

x87 — это специальный набор инструкций для работы с математическими вычислениями, являющийся подмножеством архитектуры процессоров **x86**.

Все процессоры Intel и AMD, начиная с 486DX, имеют встроенный математический сопроцессор, и в отдельном сопроцессоре не нуждаются (за исключением Intel486SX)

Регистры математического сопроцессора

Математический сопроцессор имеет свои собственные реестры. У них есть согбенность – они связаны друг с другом и образую стек.

Их всего 8. Они имеют название ST0, ST1, ST2… ST7 . Соответственно, сопроцессор может хранить не более 8 чисел одновременно. Попытка загрузить девятое придет к потере одного из чисел и загрузке в ST0 плохого (bad) числа.

Загружаемое в сопроцессор число попадает в ST0. При этом все остальные числа сдвинуться: ST0->ST1; ST1->ST2 и т. п.

Например, если мы загрузим в стек числа 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8 , то в итоге мы получим следующее:

```
ST0 valid 8.80000000000000000000
ST1 valid 7.70000000000000000000
ST2 valid 6.60000000000000000000
ST3 valid 5.50000000000000000000
ST4 valid 4.40000000000000000000
ST5 valid 3.30000000000000000000
ST6 valid 2.20000000000000000000
ST7 valid 1.10000000000000000000
```

А если докинем туда 9.9, то уже это ->

```
ST0 bad   -NAN FFFF C0000000 00000000
ST1 valid 8.80000000000000000000
ST2 valid 7.70000000000000000000
ST3 valid 6.60000000000000000000
ST4 valid 5.50000000000000000000
ST5 valid 4.40000000000000000000
ST6 valid 3.30000000000000000000
ST7 valid 2.20000000000000000000
```

Базовые команды.

FINIT – освобождает все регистры сопроцессора.

FFREE *регистр* – освобождает указанный регистр сопроцессора. По факту, он просто помечается пустым. Обратите внимание, сам регистр из стека не исчезает (что естественно), он просто помечается пустым.

Вместо *регистра* вы можете указать ST(0) (или просто ST) , ST(1), ST(2) … ST(7)

Команды загрузки в стек (*Fpu LoaD*)

FLD *память* - загружает из памяти в вершину стека ST(0) вещественное число
FILD *память* - загружает из памяти в вершину стека ST(0) целое число
FBLD *память* - загружает из памяти в вершину стека ST(0) двоично-десятичное
число

При выполнение всех указанных команд происходит Push. То есть вы заталкиваете значение в стек регистров сопроцессора.

У **FLD** вместо *памяти* можно указать номер регистра сопроцессора, например ST(0). Это приведёт к тому, что указанный регистр сопроцессора будет помещён в стек сопроцессора.

Команды извлечения из стека (*Fpu STore and **P**op*)

FSTP *память* - извлекает из вершины стека ST(0) в память вещественное число
FISTP *память* - извлекает из вершины стека ST(0) в память целое число
FBSTP *память* - извлекает из вершины стека ST(0) в память двоично-десятичное число

Эти команды сначала сохраняют вершину стека в памяти, а потом удаляют данные из вершины стека. Обратите внимание, на окончание **P**. В данном случае, это расшифровывается, как **P**op.

У **FSTP** *памяти* можно указать регистр сопроцессора, например ST(2). Это приведёт к тому, что ST(0) сопроцессора будет скопирован в указанный регистр сопроцессора и после этого будет произведено выталкивание из стека (Pop)

Команды копирования данных (*Fpu Store без Pop*)

FST *память* - извлекает из вершины стека ST(0) в память вещественное число
FIST *память* - извлекает из вершины стека ST(0) в память целое число
FBST *память* - извлекает из вершины стека ST(0) в память двоично-десятичное число

Команда обмена *(Fpu eXCHange)*

FXCH *регистр*
обмен содержимым верхушки стека ST(0) и регистра сопроцессора, указанного в качестве операнда команды.

Если параметр не указать, то поменяются местами ST(0) и ST(1)

Арифметические команды. Шаблон.

Их много, но все они работаю по шаблону. В дальнейшем вместо xxx просто подставите нужную комманду. У этого шаблона есть несколько видов написания:

1) Fxxx

Первый операнд берется из ST(1) второй – из ST(0). Результат выполнения команды записывается в ST(1). Затем ST(0) выталкивается из стека и ST(1) занимает место ST(0). Выходит, что результат оказывается в ST(0), а старое ST(0) соответственно исчезает из стека.

Например:

FSUB; ST(0)=ST(1) - ST(0).

2) Fxxx *память*

Первый операнд ST(0), второй берётся из памяти. Результат сохраняется в ST(0). Указатель стека не изменяется.

Например:

FSUB *var1*; ST(0)=ST(0) - var1

Арифметические команды. Шаблон.

3) Fxxx *ST, ST(i)*
Первый операнд- регистр ST(0), второй ST(i). Результат попадает в ST(0)
Указатель стека не изменяется.
Например:
FSUB ST, ST(3); ST(0)=ST(0) - ST(3).
Примечание: Запись ST аналогична ST(0)

4) Fxxx *ST(i), ST*
Первый операнд ST(i), второй ST(0). Результат сохраняется в ST(i).
Указатель стека не изменяется.
Например:
FSUB ST(3), ST; ST(3)=ST(3) – ST(0)

Арифметические команды. Шаблон.

5) FxxxP *ST(i), ST*
Первый операнд- регистр ST(i), второй ST(0). Результат попадает в ST(i).
После происходит выталкивание из стека (Pop).
Например:
FSUBP ST(3), ST(0); ST(3)=ST(3) - ST(0) и Pop

Основные арифметические команды

В вышеуказанном шаблоне xxx может заменяться на:

- ADD - Сложение
- SUB - Вычитание
- SUBR - Обратное вычитание, уменьшаемое и вычитаемое меняются местами
- MUL - Умножение
- DIV - Деление
- DIVR - Обратное деление, делимое и делитель меняются местами

Вычисление корня

Вычисление корня из введённого числа:

```
.data
val1 dt ?  ;Объявляет 10 байтовую переменную
res dt ?; результат
outbuf db 30 dup(?); Буфер для вывода текста
inbuf  db 30 dup(?) ; Буфер для ввода текста
.code
…
invoke CharToOem, chr$("Из чего берём корень:",13,10,0), ADDR outbuf
invoke StdOut,ADDR outbuf
invoke StdIn, addr inbuf, 100
;Перевести строку в вещественное число и поместить его в val1
invoke FpuAtoFL, ADDR inbuf, ADDR val1, DEST_MEM
fld val1 ;Загрузить val1 в ST(0)
fsqrt ; Взять корень из ST(0) и сохранить результат в ST(0)
fstp res ; Сохраняем содержимое ST(0) в res и выполняет pop.
;Перевести вещественное число в текст и поместить текст в res
invoke FpuFLtoA, ADDR res, 11, ADDR outbuf, SRC1_REAL or SRC2_DIMM
invoke StdOut,ADDR outbuf; Вывести число на экран
```

FpuFLtoA

Требует подключения:
includelib \masm32\lib\fpu.lib
include \masm32\include\fpu.inc

Переводит вещественное число в ASCII строку и помещает её по указанному адресу.

```
invoke FpuFLtoA, адрес_исходного_числа,
количествово_знаков_после_запятой,
адрес_текстовой_строки,
константы_определющие_тип_параметров
```

```
invoke FpuFLtoA, ADDR res, 10, ADDR outbuf, SRC1_REAL or SRC2_DIMM
```

SRC1_FPU – игнорировать первый параметр, и брать исходное число напрямую из ST(0);
SRC1_REAL – первый параметр – адрес 10-байтового числа;

SRC2_DMEM - второй параметр(кол-во знаков после запятой) – адрес 32 битного без знакового числа;
SRC2_DIMM - второй параметр ((кол-во знаков после запятой)) - значение 32 битного беззнакового числа

FpuAtoFL

Требует подключения:
includelib \masm32\lib\fpu.lib
include \masm32\include\fpu.inc

Переводит ASCII строку в вещественное число и помещает её по указанному адресу.

```
invoke FpuFLtoA, адрес_текстовой_строки,
адрес_переменной_с_результатом,
константы_определющие_тип_параметров
```

```
invoke FpuAtoFL, ADDR inbuf, ADDR val1, DEST_MEM
```

DEST_MEM – копирование результата в память.

Дополнительные арифметические команды

FSQRT – Вычислить корень ST(0). Вычисленное значение квадратного корня записывается в верхушку стека ST(0).

FSCALE - изменяет порядок числа, находящегося в ST(0). Действие этой команды можно представить следующей формулой: ST(0) = ST(0) * 2*ST(1),
где -215 <= ST(1) <= +215

FPREM - вычисляет остаток от деления делимого ST(0) на делитель ST(1). Знак результата равен знаку ST(0), а сам результат получается в вершине стека ST(0).

FRNDINT - округляет ST(0) в соответствии с содержимым поля RC управляющего регистра.

FXTRACT - Выделение порядка числа и мантиссы. Порядок экспоненты сначала попадает ST(0). Затем происходит Push и ST(0) помещается порядок числа. В итоге в ST(0) оказывается порядок числа, в ST(1) – порядок экспоненты. Старое ST(1) оказывается в ST(2).

FABS - вычисляет абсолютное значение ST(0).Результат попадает в ST(0)

FCHS - изменяет знак ST(0) на противоположный.

Трансцендентные команды. SIN. COS.

FCOS Вычисление cos(ST(0))

FSIN Вычисление sin(ST(0))

FSINCOS вычисляет одновременно значения синуса и косинуса параметра ST(0). Значение синуса записывается в ST(1), косинуса - в ST(0).

Трансцендентные команды. Частичный тангенс. FPTAN.

Команда FPTAN вычисляет частичный тангенс ST(0), размещая в стеке такие два числа x и y, что y/x = tg(ST(0)).

После выполнения команды число y располагается в ST(0), а число x включается в стек сверху (то есть записывается в ST(1)).

Аргумент команды FPTAN
должен находится в пределах:
0 <= ST(0) <= pi/4

Пользуясь полученным значением частичного тангенса, можно вычислить другие тригонометрические функции по следующим формулам:
- sin(z) = 2*(y/x) / (1 + (y/x)2)
- cos(z) = (1 - (y/x)2) / (1 + (y/x)2)
- tg(z/2) = y/x;
- ctg(z/2) = x/y;
- cosec(z) = (1 + (y/x)2) / 2*(y/x)
- sec(z) = (1 + (y/x)2) / (1 - (y/x)2)

Где z - значение, находившееся в ST(0) до выполнения команды FPTAN, x и y - значения в регистрах ST(0) и ST(1), соответственно (после вычисления FPTAN).

Трансцендентные команды. Частичный арктангенс. FPATAN.

Команда FPATAN вычисляет частичный арктангенс:
z=arctg(ST(0)/ST(1))=arctg(x/y)

Перед выполнением команды числа x и y располагаются в ST(0) и ST(1), соответственно. Аргументы команды FPATAN должен находится в пределах:
$0 < y < x$

Результат записывается в ST(0).

Трансцендентные команды. Логарифмы.

Команда **FYL2X** вычисляет выражение y*log2(x), операнды x и y размещаются, соответственно, в ST(0) и ST(1). Операнды извлекаются из стека, а результат записывается в стек. Параметр x должен быть положительным числом.

Пользуясь результатом выполнения этой команды, можно вычислить следующим образом логарифмические функции:

- Логарифм по основанию два: log2(x) = FYL2(x)
- Натуральный логарифм: loge(x) = loge(2) * log2(x) = FYL2X(loge(2), x) = FYL2X(FLDLN2, x)
- Десятичный логарифм: log10(x) = log10(2) * log2(x) = FYL2X (log10(2), x) = FYL2X(FLDLG2, x)

Функция **FYL2XP1** вычисляет выражение y*log2(x+1), где x соответствует ST(0), а y - ST(1). Результат записывается в ST(0), оба операнда выталкиваются из стека и теряются.
На операнд x накладывается ограничение: 0 < x < 1 - 1/sqrt(2)

Команда **F2XM1** вычисляет выражение 2x-1, где x - ST(0). Результат записывается в ST(0), параметр должен находиться в следующих пределах:
0 <= x <=0.5

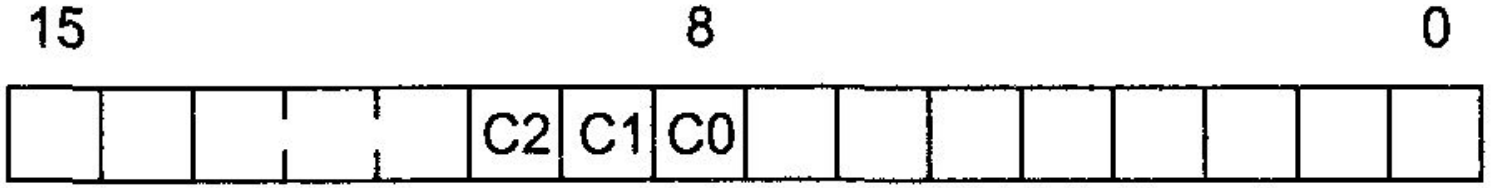
FLDLG2 заталкивает в стек значение loge(2)
FLDLG2 заталкивает в стек значение log10(2),

Вычисление exp(x)

```
.data
x dt ?  ;Объявляет 10 байтовую переменную
res dt ?; результат
outbuf db 30 dup(?); Буфер для вывода текста
inbuf  db 30 dup(?) ; Буфер для ввода текста
.code
…
invoke CharToOem, chr$("exp(x) x=",13,10,0), ADDR outbuf
invoke StdOut,ADDR outbuf
invoke StdIn, addr inbuf, 100
invoke FpuAtoFL, ADDR inbuf,ADDR x, DEST_MEM
fld x ;Загрузить x в ST(0)
;==============================
; e^x = 2^(x*log2(e))
    FLDL2E; y := x*log2e;
    FMUL
    FLD     ST(0)       ; i := round(y);
    FRNDINT
    FSUB    ST(1), ST   ; f := y - i;
    FXCH    ST(1)       ; z := 2^f
    F2XM1
    FLD1
    FADD
    FSCALE              ; result := z * 2^i
    FSTP    ST(1)
;==============================
fstp res ; Сохраняем содержимое ST(0), а это  exp(x)  в res
```

Слово состояния сопроцессора

Периодически требуется проверять значения в сопроцессоре. А значит нам нужны аналоги команд TEST и CMP для сопроцессора.
Такая команда существует и называется **FTST**.
FTST не имеет параметров. Она сравнивает ST(0) с нулём.
Результат хранится в 3 битах C2, C1, C0 слова состояния сопроцессора:



ST > 0.0	0	0	0
ST < 0.0	0	0	1
ST = 0.0	1	0	0
ST = ?	1	1	1

(FST в OllyDBG, STAT в студии)
Также существует инструкция **FSTW** переписывающая слово состояния в указанный регистр процессора.

Проверка вершины стека сопроцессора.

Вычисление корня из разности двух чисел:

```
…
fsub ST(0),ST(3) ; ST(0)=ST(0)-ST(3).
ftst ; Проверить вершину стека
fstsw ax; Прочитать слово состояния в ax
shr ah,1 ; C0 попадёт во флаг переноса CF
jc exit ; Если вST(0) <0 выход
fsqrt ; Иначе – вычисляем корень из ST(0) и сохраняем результат в ST(0).
…
```

Рассмотреть самостоятельно

Крупник А.Б. «Изучаем ассемблер»
Примеры на страницах 165-166 и 168-170