**Практическая работа**

**Структуры данных**

Цель: научиться работать с структурами данных в ассемблере

Структура в ассемблере.

Структура в ассемблере (structure) — это совокупность переменных, объединенных одним именем. Переменные называются полями и могут быть разными по размеру. Очень удобно обращаться к данным по именам полей. Структура — основа абстракции, «блочности» кода. Понятие КЛАСС в языках высокого уровня есть не что иное, как разновидность структуры. Только в качестве полей в классе кроме данных присутствуют ещё и функции. В качестве поля в структуру может входить структура (пример — в коде).

По определению **структура** — это тип данных, состоящий из фиксированного числа элементов разного типа.  
  
Для использования структур в программе необходимо выполнить три действия:

1. *Задать шаблон структуры.*  
   По смыслу это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.
2. *Определить экземпляр структуры.*  
   Этот этап подразумевает инициализацию конкретной переменной заранее определенной (с помощью шаблона) структурой.
3. *Организовать обращение к элементам структуры.*

Очень важно, чтобы вы с самого начала уяснили, в чем разница между описанием структуры в программе и ее определением.  
***Описать структуру*** в программе означает лишь указать ее схему или шаблон; память при этом не выделяется. Этот шаблон можно рассматривать лишь как информацию для транслятора о расположении полей и их значении по умолчанию.  
***Определить структуру*** — значит, дать указание транслятору выделить память и присвоить этой области памяти символическое имя.  
Описать структуру в программе можно только один раз, а определить — любое количество раз.  
  
**Описание шаблона структуры**  
  
Описание шаблона структуры имеет следующий синтаксис:  
  
**имя\_структуры STRUC**  
**<описание полей>**  
**имя\_структуры ENDS**  
  
Здесь *<описание полей>* представляет собой последовательность директив описания данных **db, dw, dd, dq и dt**.  
Их операнды определяют размер полей и, при необходимости, начальные значения. Этими значениями будут, возможно, инициализироваться соответствующие поля при определении структуры. Как мы уже отметили при описании шаблона, память не выделяется, так как это всего лишь информация для транслятора.  
*Местоположение описания шаблона* в программе может быть произвольным, но, следуя логике работы однопроходного транслятора, он должен быть расположен до того места, где определяется переменная с типом данной структуры. То есть при описании в сегменте данных переменной с типом некоторой структуры ее шаблон необходимо поместить в начале сегмента данных либо перед ним.

* MY\_STRUCT\_1 STRUC; структура в ассемблере объявляется словом **STRUC**  
  member\_1 dw ?;  
  member\_2 db ?;  
  MY\_STRUCT\_1 ENDS;

….

* my\_struct MY\_STRUCT\_1 <?>; структура в ассемблере, созданная на основе объявления.
* mov my\_struct.member\_1 ,33h;используем конкретный экземпляр в коде.

Константы, массив и структура в ассемблере — наиболее часто используемые организованные виды данных, однако есть и другие. Если посмотреть на структуру повнимательнее, со стороны нашего подхода к программированию, как к совокупности кода и данных, то остальные виды сгруппированных данных : массив, перечисление, объединение, битовые поля — фактически являются разновидностями структуры, реализованные в целях экономии процессорного времени и объёма памяти.

Идея введения структурного типа в любой язык программирования состоит в объединении разнотипных переменных в один объект.

В языке должны быть средства доступа к этим переменным внутри конкретного экземпляра структуры. Для того чтобы сослаться в команде на поле некоторой структуры, используется специальный оператор — символ "**.**" (точка). Он используется в следующей синтаксической конструкции:

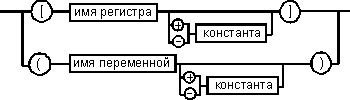
|  |
| --- |
| адресное\_выражение.имя\_поля\_структуры |

Здесь:

* адресное\_выражение — идентификатор переменной некоторого структурного типа или выражение в скобках в соответствии с указанными ниже синтаксическими правилами (рис. 1);
* имя\_поля\_структуры — имя поля из шаблона структуры.
* Это, на самом деле, тоже адрес, а точнее, смещение поля от начала структуры.

Таким образом оператор "**.**" (точка) вычисляет выражение

|  |
| --- |
| (адресное\_выражение) + (имя\_поля\_структуры) |



**Рис.** Синтаксис адресного выражения в операторе обращения к полю структуры

**Массив в ассемблере.**

Массив — структура данных, хранящих значения, которые идентифицируются по индексам, начиная с нулевого индекса. Рассмотрим работу с одномерным массивом — с учётом нашего начального уровня.

* my\_mass\_1 db 10 dup(8); создать байтовый массив, состоящий из 10 байт и заполнить его цифрами 8. Реально : дублировать (DUPlicate) 10 раз число 8.
* mov my\_mass\_1[0],1 ; поместить в первый байт число 1 (поля массива считаются с нуля, а не с единицы).

**Перечисление в ассемблере.**

Перечисления (enum) представляет собой структуру, состоящую из именованных констант. Создана для удобства программирования в Си. При этом можно просто перечислять константы, компилятор будет присваивать им целые значения в порядке возрастания, начиная с нуля. Если присвоить полю конкретное значение, которое не совпадает с простой последовательностью, отсчёт следующих, не определённых полей будет происходить по алгоритму «+1».  
enum eDirection  
{  
RIGHT, // по умолчанию = 0  
LEFT, // = 1  
DOWN=5, // = 5 — присвоили, если бы не присвоить, то DOWN==2 (предыдущее поле +1)  
UP // = 6 (+1)  
};

В ассемблере перечисление (enum) как отдельный, самостоятельный тип данных не существует. При необходимости используют структуру с целочисленными полями либо просто отдельные целочисленные константы.

* \_ENUM STRUC;В ассемблере Нет ENUM, используем STRUC  
  RIGHT db ?;  
  LEFT db ?;  
  DOWN db ?;  
  UP db ? ;  
  \_ENUM ENDS
* my\_enum \_ENUM <0,1,2,3>

…

* xor ax,ax;ax==0
* mov ah,my\_enum.LEFT

**Объединения в ассемблере.**

Объединение (union) —  одна и та же область памяти, используемая как разные типы данных. Естественно, в таком случае размер объединения будет равен размеру наибольшего из значений и не равна сумме длин всех запоминаемых, как в структуре. Тип данных создавался для Си, как способ экрномии памяти компьютера (сейчас — не актуально, но ранее активно использовался в написании кода, в том числе и сетевого характера, поэтому применяется и сейчас для совместимости).

* MY\_UNION union  
  \_word dw ?  
  \_byte db ?  
  MY\_UNION ends
* \_union MY\_UNION <1234h>

…

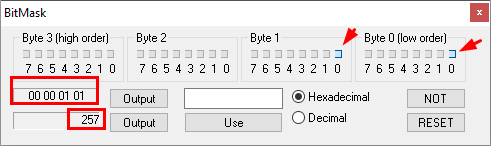
После этого \_union.\_word=1234h, а \_union.\_byte=34h.

* xor ax,ax;ax==0
* mov ah,\_union.\_byte

**Записи с битовыми полями (запись).**

Бит — единица данных, может содержать значение 1 или 0. Записи с битовыми полями (records) используют эту возможность.

Каждое битовое поле имеет заданную длину (в битах) и начальное значение. Размер данных типа записи равен сумме длин всех полей

[](http://assembler-code.com/wp-content/uploads/2020/04/record-asm.png)Запись с битовыми полями (запись) — 32 бита.

Опять таки — удобно, экономит место и вычислительное время. Например, чтобы задать цвет точки в изображении (совокупность различных оттенков красного, зелёного, синего (RGB) или свойств окна в операционке Windows.

Не будем подробно разбирать тему, приведём пример кода.

* BitMask RECORD f0:4=1,f1:4=1,f2:4=0,f3:4=0

…

* xor ax,ax;ax==0
* mov ax,BitMask;ax==257

**Задание**

На основе предыдущей работы требуется прочитать данные из файла «base.dat».

Файл представляет собой последовательно хранящиеся записи о сотрудниках, каждая запись является структурой с следующими полями

Описание структуры «информация о сотруднике»

id dw 0h; идентификатор

fio db 30 dup (' ') ; фамилия, имя, отчество

gender db 'm';пол, по умолчанию 'm' – мужской ‘w’ - женский

age db 18h;возраст

standing db 1;стаж

salary dd 1000;оклад

birthdate db 8 dup(' ') ;дата рождения

1. Необходимо посчитать количество сотрудников мужского и женского пола;
2. Вычислить средний возраст всех сотрудников, используя целочисленное деление;
3. Вычислить средний оклад всех сотрудников, используя FPU
4. Вывести ФИО сотрудника с идентификатором 7.

Для выполнения каждого пункта 1-4 задать свою переменную в сегменте .data

Вывод сделать как скриншот значения переменной в отладчике.

**Для выполнения потребуются дополнительные сведения**

worker struc ;информация о сотруднике

id dw 0h

fio db 30 dup (' ') ;фамилия, имя, отчество

gender db 'm';пол, по умолчанию 'm' - мужской

age db 18h;возраст

standing db 1;стаж

salary dd 1000;оклад

birthdate db 8 dup(' ') ;дата рождения

worker ends

recs worker<?>; структура в ассемблере, созданная на основе объявления

**Получение адреса переменной**   
  
Есть такая команда **LEA** (Load Effective Address) и альтернативный оператор **OFFSET**. Как **OFFSET** так и **LEA** могут быть использованы для получения смещения адреса переменной.  
  
**LEA** более мощная, т.к. она также позволяет вам получить адрес индексированных переменных. Получение адреса переменной может быть очень полезно в различных ситуациях, например, если вам необходимо поместить параметр в процедуру.

lea ebx,recs

**Получение размера**

LENGTHOF variable

SIZEOF variable

SIZEOF type

LENGTH expression

SIZE expression

Оператор LENGTHOF возвращает количество элементов данных, выделенных для <variable>. Оператор SIZEOF возвращает общее количество байтов, выделенных для <variable>, или размер <type> в байтах.

Для переменных SIZEOF равно значению LENGTHOF, умноженному на количество байтов в каждом элементе.

Операторы LENGTH и SIZE разрешены для совместимости с предыдущими версиями ассемблера. При применении к метке данных оператор LENGTH возвращает количество элементов, созданных оператором DUP; в противном случае возвращается 1. При применении к метке данных оператор SIZE возвращает количество байтов, выделенных первым инициализатором в метке <переменная>.

mov eax,SIZEOF worker

**Сложение и вычитание.**

**ADD** – команда для сложения двух чисел. Она работает как с числами со знаком, так и без знака.

**ADD *Приемник*, *Источник***

Логика работы команды:

<*Приемник*> = <*Приемник*> + <*Источник*>

Возможные сочетания операндов для этой команды аналогичны команде MOV.

По сути дела, это – команда сложения с присвоением, аналогичная принятой в языке C/C++:

*Приемник* += *Источник;*

Операнды должны иметь одинаковый размер. Результат помещается на место первого операнда.

После выполнения команды изменяются флаги, по которым можно определить характеристики результата:

1. Флаг *CF* устанавливается, если при сложении произошёл перенос из старшего разряда. Для беззнаковых чисел это будет означать, что произошло переполнение и результат получился некорректным.
2. Флаг *OF* обозначает переполнение для чисел со знаком.
3. Флаг *SF* равен знаковому биту результата (естественно, для чисел со знаком, а для беззнаковых он равен старшему биту и особо смысла не имеет).
4. Флаг *ZF* устанавливается, если результат равен 0.
5. Флаг *PF* — признак чётности, равен 1, если результат содержит нечётное число единиц.

Примеры:

add ax,5     ;AX = AX + 5

add dx,cx    ;DX = DX + CX

add dx,cl    ;Ошибка: разный размер операндов.

**SUB** - команда для вычитания одного числа из другого. Она работает как с числами со знаком, так и без знака.

**SUB *Приемник*, *Источник***

Логика работы команды:

<*Приемник*> = <*Приемник*> - <*Источник*>

Возможные сочетания операндов для этой команды аналогичны команде MOV.

По сути дела, это – команда вычитания с присвоением, аналогичная принятой в языке C/C++:

*Приемник* -= *Источник;*

Операнды должны иметь одинаковый размер. Результат помещается на место первого операнда.

На самом деле вычитание в процессоре реализовано с помощью сложения. Процессор меняет знак второго операнда на противоположный, а затем складывает два числа.

Примеры:

sub ax,13     ;AX = AX - 13

sub ax,bx    ;AX = AX + BX

sub bx,cl    ;Ошибка: разный размер операндов.

**Инкремент и декремент.** Очень часто в программах используется операция прибавления или вычитания единицы. Прибавление единицы называется инкрементом, а вычитание — декрементом. Для этих операций существуют специальные команды процессора: INC и DEC. Эти команды не изменяют значение флага *CF*.

Эти команды содержит один операнд и имеет следующий синтаксис:

**INC *Операнд***

**DEC *Операнд***

Логика работы команд:

INC: <*Операнд*> = < *Операнд* > + 1

DEC: <*Операнд*> = < *Операнд* > - 1

В качестве инкремента допустимы регистры и память: reg, mem.

Примеры:

inc ax     ;AX = AX + 1

dec ax     ;AX = AX - 1

**NEG** – команда для изменения знака операнда.

Синтаксис:

**NEG *Операнд***

Логика работы команды:

<*Операнд*> = – < *Операнд* >

В качестве декремента допустимы регистры и память: reg, mem.

Примеры:

neg ax       ;AX = -AX

**Умножение и деление целочисленные.**

**MUL** – команда умножения чисел без знака. У этой команды только один операнд — второй множитель, который должен находиться в регистре или в памяти. Местоположение первого множителя и результата задаётся неявно и зависит от размера операнда:

|  |  |  |
| --- | --- | --- |
| **Размер операнда** | **Множитель** | **Результат** |
| 1 байт | AL | AX |
| 2 байта | AX | DX:AX |
| 4 байта | EAX | EDX:EAX |

Отличие умножения от сложения и вычитания в том, что разрядность результата получается в 2 раза больше, чем разрядность сомножителей.

Примеры:

mul bl    ;AX = AL \* BL

mul ax    ;DX:AX = AX \* AX

Если старшая часть результата равна нулю, то флаги CF и ОF будут иметь нулевое значение. В этом случае старшую часть результата можно отбросить.

**IMUL** – команда умножения чисел со знаком. Эта команда имеет три формы, различающиеся количеством операндов:

1.       С одним операндом — форма, аналогичная команде MUL. В качестве операнда указывается множитель. Местоположение другого множителя и результата определяется по таблице.

2.       С двумя операндами — указываются два множителя. Результат записывается на место первого множителя. Старшая часть результата в этом случае игнорируется. Кстати, эта форма команды не работает с операндами размером 1 байт.

3.       С тремя операндами — указывается положение результата, первого и второго множителя. Второй множитель должен быть непосредственным значением. Результат имеет такой же размер, как первый множитель, старшая часть результата игнорируется. Это форма тоже не работает с однобайтными множителями.

Примеры:

imul cl           ;AX = AL \* CL

imul bx,ax        ;BX = BX \* AX

imul cx,-5        ;CX = CX \* (-5)

imul dx,bx,134h   ;DX = BX \* 134h

CF = OF = 0, если произведение помещается в младшей половине результата, иначе CF = OF = 1. Для второй и третьей формы команды CF = OF = 1 означает, что произошло переполнение.

**DIV** – команда деления чисел без знака. У этой команды один операнд — делитель, который должен находиться в регистре или в памяти. Местоположение делимого, частного и остатка задаётся неявно и зависит от размера операнда:

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер операнда (делителя)** | **Делимое** | **Частное** | **Остаток** |
| 1 байт | AX | AL | AH |
| 2 байта | DX:AX | AX | DX |
| 4 байта | EDX:EAX | EAX | EDX |

При выполнении команды DIV может возникнуть *прерывание* (в данном курсе прерывания мы рассматривать не будем поэтому старайтесь избегать таких случаев):

* если делитель равен нулю;
* если частное не помещается в отведённую под него разрядную сетку (например, если при делении слова на байт частное больше 255).

Примеры:

div cl   ;AL = AX / CL, остаток в AH

div di   ;AX = DX:AX / DI, остаток в DX

**IDIV** – команда деления чисел со знаком. Единственным операндом является делитель. Местоположение делимого и частного определяется также, как для команды DIV. Эта команда тоже генерирует прерывание при делении на ноль или слишком большом частном.

**NOP** – ничего не делающая команда.

Синтаксис:

**NOP**

Примеры:

nop

**Пример.** (5 + 8) / (2 \* 3)

#include <iostream.h>

void main()

{

   asm {

        mov bx, 5 //BL = 5

      add bx, 8 //BL = BL + 8  |  13

      sub bx, 1 //BL = BL - 1  |  12

      mov al, 2 //AL = 2

      mov cl, 3 //CL = 3

      mul cl    //AX = AL \* CL  |  6

      //AX = 6, BL = 12

      xchg bx, ax //AX = 12, BX = 6

      mov dx, 0

      div bx

   }

}

**Использование FPU для деления чисел с плавающей точкой**

Пример на паскале и его ассемблерный код

var t1,t2:single;

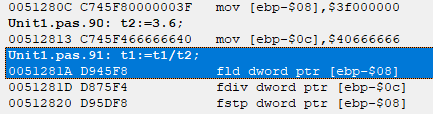
begin

t1:=0.5;

t2:=3.6;

t1:=t1/t2;

end;



В первом приближении (более подробно FPU будет рассматриваться в другой работе)

Необходимо загрузить число (делитель) в st регистр FPU

Затем дать команду делить на частное.

Затем результат выгрузить из FPU.

Все команды имеют параметр – указатель на ячейку памяти (т.е) команды оперируют только значениями находящимися в оперативной памяти (не **в регистрах общего назначения**)

Т.е необходимо выполнить

.data

t1 dd 0

t2 dd 0

.code

…

fld t1

fdiv t2

fstp t1

Но так работает, когда значение переменных в формате IEEE 754.

У нас же значения целые, поэтому поступим по другому.

Грузим целое в регистр другой командой

mov eax,10

mov t1,eax

mov eax,3

mov t2,eax

пусть t1=10 а t2 = 3

регистры FPU представляют собой стек размером 8 регистров

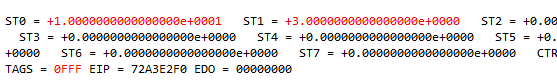
fild t2 ;– закидываем в стек (значение 3 попадет в ST0)

fild t1 ;– Затем значение 3 попадет в ST1 а в ST0 занесется 10

fdiv st,st(1) ;затем выполняется ST0=ST0/ST(1) (ST0 – вершина стека и обозначается просто ST)

fstp t1 ; - вернем результат из вершины стека

После первых двух команд в регистрах FPU увидим следующие изменения

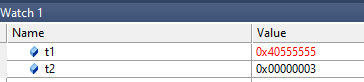


Выполнив

fdiv st(1),st

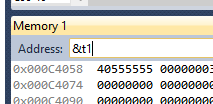


Однако в отладчике увидим следующие

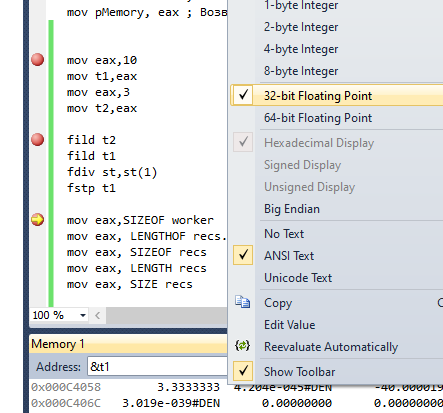


Это потому, что на самом деле отладчик интерпретирует значение как целое

Посмотрим память, где лежит значение



Значение памяти можно вывести в другом формате



Теперь видим все верно

10/3 = 3,33333333

Варианты параметров команды FDIV



Чуть более подробно про FPU

**FPU (Floating Point Unit)**

Регистры FPU (8 штук) реализованы в виде стека: st=st(0), st(1), st(2), st(3), st(4), st(5), st(6), st(7). Когда мы загружаем число в сопроцессор, оно загружается в st(0), а все другие регистры сдвигаются. Не будем допускать переполнения стека сопроцессора. Я не знаю, что при этом происходит, но описание ошибок выходит за рамки этой работы. В регистрах могут храниться данные в следующих форматах:

|  |  |
| --- | --- |
| Короткое вещественное | Бит 31 – знак мантиссы. Биты 30-23 – 8-битная\_экспонента+127  Биты 22-0 – 23-битная\_мантисса без первой цифры |
| Длинное вещественное | Бит 63 – знак мантиссы. Биты 62-52 – 11-битная экспонента+1024  Биты 51-0 – 52-битная мантисса без первой цифры |
| Супер вещественное | Бит 79 – знак мантиссы. Биты 78-64 – 15-битная экспонента+16383  Биты 63-0 – 64-битная мантисса с первой цифрой (бит 63 равен 1) |

Размер, точность и пределы различных типов вещественных чисел:

|  |  |  |  |
| --- | --- | --- | --- |
| Тип данных | Бит | значащие цифры | пределы |
| Короткое вещественное | 4 | 7 | http://osinavi.ru/asm/FPUexpansion/FPU_pictures/single.gif |
| Длинное вещественное | 8 | 15-16 | http://osinavi.ru/asm/FPUexpansion/FPU_pictures/double.gif |
| Супер вещественное | 10 | 19 | http://osinavi.ru/asm/FPUexpansion/FPU_pictures/extended.gif |

Вообще, сопроцессор выполняет все вычисления в расширенном 80-битном формате, а 32- и 64-битные числа используются для обмена с процессором и оперативной памятью   
Также в сопроцессоре есть 2 дополнительных регистра:

|  |  |  |  |
| --- | --- | --- | --- |
| 1) Регистр состояния SR: | |  | | --- | | Бит 14 – C3 (условный флаг 3) | | Бит 8 – С0 (условный флаг 0) | |

2) Регистр управления CR:   
биты 11-10 – RC (управление округлением)   
биты 9-8 – PC (управление точностью)   
бит 5: PM – маска неточного результата   
бит 4: UM – маска антипереполнения   
бит 3: OM – маска переполнения   
бит 2: ZM – маска деления на ноль   
бит 1: DM – маска денормализованного операнда   
бит 0: IM – маска недействительной операции   
Если какая-либо маска равна нулю, то при происхождении действия, соответствующего маске (например, ZM=0, и произошло деление на ноль), программа вылетает. Если исключительное действие было замаскировано (соответствующая маска равна 1), то программа не вылетает

|  |  |
| --- | --- |
| Значение RC | Способ округления |
| 0 | К ближайшему числу |
| 1 | К отрицательной бесконечности |
| 2 | К положительной бесконечности |
| 3 | К нулю |

Биты PC определяют точность результатов команд: **fadd, fsub, fsubr, fmul, fdiv, fdivr, fsqrt**

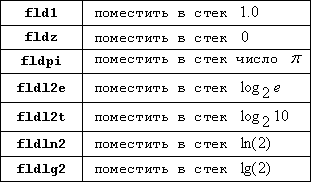
|  |  |
| --- | --- |
| Значение PC | Точность результатов |
| 0 | 32-битные числа |
| 2 | 64-битные числа |
| 3 | 80-битные числа |

Примечание: если команда оканчивается на “p” (например, **fstp faddp fmulp**) и существует такая же каманда без оканчания “p” (например, **fst, fadd, fmul**), то она после себя выталкивает из стека st(0)

**Основные команды загрузки и сохранения**

С помощью команд **fld|fild|fst|fstp** происходит обмен данными между ячейками памяти и регистрами сопроцессора:   
http://osinavi.ru/asm/doroga.gif1) **fld** источник – загрузить вещественное число в стек. Источник – m32|m64|m80. Например: **fld** qword ptr x – в Паскале загружает число типа double в сопроцессор   
http://osinavi.ru/asm/doroga.gif  
2) **fild** источник – загрузить целое число со знаком в стек. Источник – m16|m32|m64. Например, команды **fild** dword ptr x в Паскале загружает целое число типа longint (32-разрядное) в сопроцессор. При этом считается, что загружается число со знаком   
http://osinavi.ru/asm/doroga.gif  
3) **fst|fstp** приёмник – копирует st(0) в приёмник. Приёмник – m32|m64|st(n) для **fst** и m32|m64|m80|st(n) для **fstp.** **fstp** после этого выталкивает число st(0) из стека, а **fst** оставляет на месте. В случае st(n) команда **fstp** сначала загружает значение, а потом выталкивает из стека   
http://osinavi.ru/asm/doroga.gif  
4) **fist|fistp** приёмник – преобразовывает число из вершины стека в целое со знаком и записывает его в приёмник. Не грузить слишком большие числа! Если [IM](http://osinavi.ru/asm/FPUexpansion/1.html#RegistrYpravleniya)=0, то программа вылетает. Для **fist** приёмник – m16|m32; для **fistp** приёмник – m16|m32|m64. **fistp** после этого выталкивает число из стека, а **fist** оставляет на месте. При этом считается, что вещественное число преобразуется в целое со знаком   
http://osinavi.ru/asm/doroga.gif  
5) **fnop** – отсутствие операции. Занимает место и время, но не выполняет никакого действия   
http://osinavi.ru/asm/doroga.gif

**Загрузка некоторых чисел в сопроцессор**

Вообще, целые числа загружаются в сопроцессор с помощью команды **fild**. Но некоторые числа также можно загрузить следующими командами:   
  
Начиная с сопроцессора 80387, все константы хранятся в более точном формате, чем 80-битный, и при загрузке в стек происходит округление в соответствии с полем [RC](http://osinavi.ru/asm/FPUexpansion/1.html#RegistrYpravleniya)

**Арифметические команды сопроцессора**

http://osinavi.ru/asm/doroga.gif1) **fxch** st(i) – меняет местами содержимое st(0) и st(i), i=0..7   
http://osinavi.ru/asm/doroga.gif  
2) **fadd|faddp** приёмник, источник – сложение вещественных чисел. Выполняет сложение источника и приёмника и сохраняет результат в приёмнике. Может принимать следующий вид:   
12345678901) **fadd** m32|m64, где m32|m64 – источник, а st(0) – приёмник   
12345678902) **fadd** st(0), st(i)|**fadd** st(i), st(0)|**faddp** st(n), st(0)  
http://osinavi.ru/asm/doroga.gif  
3) **fsub|fsubp** приёмник, источник – вычитание вещественных чисел (аналогично сложению). Выполняет вычитание источника из приёмника и сохраняет результат в приёмнике  
http://osinavi.ru/asm/doroga.gif  
4) **fsubr|fsubrp** приёмник, источник – обратное вычитание вещественных чисел – эквивалентны fsub|fsubp, только они выполняют вычитание приёмника из источника, а не источника из приёмника  
http://osinavi.ru/asm/doroga.gif  
5) **fmul|fmulp** приёмник, источник – выполняет умножение источника и приёмника и помещает результат в приёмник  
12345678901) **fmul** m32|m64, где m32|m64 – источник, а st(0) – приёмник  
12345678902) **fmul** st(0), st(i)|**fmul** st(i), st(0)|**fmulp** st(n), st(0)  
http://osinavi.ru/asm/doroga.gif  
6) **fdiv|fdivp** приёмник, источник – деление вещественных чисел (аналогично умножению)  
http://osinavi.ru/asm/doroga.gif  
7) **fdivr|fdivrp** приёмник, источник – обратное деление вещественных чисел – эквивалентны **fdiv|fdivp**, но при этом они выполняют деление источника на приёмник, а не приёмника на источник  
http://osinavi.ru/asm/doroga.gif  
8) **fabs** – если st(0)<0, переводит его в положительное  
http://osinavi.ru/asm/doroga.gif  
9) **fchs** – изменить знак st(0)  
http://osinavi.ru/asm/doroga.gif  
10) **frndint** – округляет значение st(0) до целого числа в соответствии с режимом округления, заданным битами [RC](http://osinavi.ru/asm/FPUexpansion/1.html#RegistrYpravleniya)  
http://osinavi.ru/asm/doroga.gif  
11) **fsqrt** – извлечь квадратный корень  
http://osinavi.ru/asm/doroga.gif

**Напоминание:**  
Чтобы указать компилятору тип данных, вы должны использовать следующие префиксы:  
  
**BYTE PTR** - для байта.  
**WORD PTR** - для слова (два байта).  
  
Например:

BYTE PTR [BX] ; доступ к байту.

или

WORD PTR [BX] ; доступ к слову.

**Emu8086** поддерживает короткие префиксы:  
  
**b.** - для **BYTE PTR**  
**w.** - для **WORD PTR**  
  
иногда компилятор может вычислить тип данных автоматически, но вы не можете и не должны полагаться на это, если один из операндов является непосредственным значением.