МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Государственное образовательное учреждение
высшего профессионального образования
ОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Ф.М. ДОСТОЕВСКОГО

Н.Ф. Богаченко, Д.Н. Лавров, Ю.С. Ракицкий

АССЕМБЛЕР В ПРИМЕРАХ И ЗАДАЧАХ

Учебно-методическое-пособие

Часть 1

Омск

2011

УДК 519.682
ББК 32.973.26–018.1я73
Б733

Рекомендовано к изданию редакционно-издательским советом ОмГУ

Рецензент
канд. физ.-мат. наук, доцент *В.В. Коробицын*

Богаченко, Н.Ф.

Б733 Ассемблер в примерах и задачах: учебно-методическое пособие. Ч. 1 / Н.Ф. Богаченко, Д.Н. Лавров, Ю.С. Ракицкий. – Омск: Изд-во Ом. гос. ун-та, 2011. – 72 с.

ISBN 978-5-7779-1257-2

В пособии представлены практические задания по курсу «Ассемблер». Большинство заданий сопровождено примерами реализации с подробными комментариями и пояснениями.

Для студентов, обучающихся по специальностям 090102.65 «Компьютерная безопасность» и 230101.65 «Вычислительные машины, комплексы, системы и сети».

УДК 519.682
ББК 32.973.26–018.1я73

ISBN 978-5-7779-1257-2

ПРЕДИСЛОВИЕ

Ассемблер – это программа, которая переводит **язык ассемблера** в машинный код. Вместе с тем, слово «ассемблер» стало также и названием самого языка программирования. Ассемблер тесно связан с архитектурой (структурной, схемотехнической и логической организацией) самого микропроцессора. В представленном пособии – это микропроцессоры **Intel**.

В пособии для удобства приведены краткие сведения, касающиеся вопросов представления данных, регистров процессора (CPU) и сопроцессора (FPU), способов адресации. Подробное описание команд языка ассемблера можно найти в работах [2; 4].

Примеры, аналогичные разобранным в разделе 8, приведены в книгах [2, с. 442; 3, с. 53, с. 250].

Большинство представленных в пособии примеров и заданий для самостоятельной работы выполняются как **ассемблерные вставки на языке C/C++**. Рекомендуемая среда разработки – **Microsoft Visual Studio 2008**.

При работе с программой в режиме отладки полезно анализировать состояние регистров, открыв окно **Registers** (Меню **Debug → Windows → Registers**). Вызвав в этом окне контекстное меню, можно добавить регистр флагов, регистры сопроцессора и т.д. Следует обратить внимание, что имена флагов в среде **Microsoft Visual Studio** отличаются от общепринятых:

Номер бита	11	10	09	07	06	04	02	00
Флаги	**of**	**df**	**if**	**sf**	**zf**	**af**	**pf**	**cf**
Флаги в **Visual Studio**	**OV**	**UP**	**EI**	**PL**	**ZR**	**AC**	**PE**	**CY**

Работая с ассемблерными вставками в **Microsoft Visual Studio**, вызов функций языка C/C++ осуществляется следующим образом: в стек надо поместить адреса аргументов функции, перечисляя их справа налево, а затем вызвать саму функцию, например **call scanf**. После выполнения функции, удаление параметров из стека выполняет вызывающая процедура.

Если с вызовом функции возникают проблемы, попробуйте осуществить «дальний» вызов: **call dword ptr scanf** или внесите поправку в свойства проекта: через меню **Project → Properties** откройте окно **Property Pages**, в разделе **Configuration Properties → C/C++ → Code Generation** для **Runtime Library** выберите спецификацию **/MTd**.

1. ПРЕДСТАВЛЕНИЕ ДАННЫХ

1.1. Двоичная система счисления

$10010110\mathbf{b} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 =$
$= 128 + 16 + 4 + 2 = 150.$

	Остаток	Разряд
150/2 = 75	0	0
75/2 = 37	1	1
37/2 = 18	1	2
18/2 = 9	0	3
9/2 = 4	1	4
4/2 = 2	0	5
2/2 = 1	0	6
½ = 0	1	7
Результат: 10010110**b**		

Чтобы отличать двоичные числа от десятичных, в ассемблерных программах в конце каждого двоичного числа ставят букву «**b**».

1.2. Биты, байты, слова

1	**0**	**0**	**1**	**0**	**1**	**1**	**0**
7	6	5	4	3	2	1	0
байт							

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**
15							8	7							0
слово															
старший байт								младший байт							
адрес (x + 1)								адрес (x)							

1.3. Шестнадцатеричная система счисления

Перевод из шестнадцатеричной системы счисления в двоичную и обратно осуществляется следующим образом: вместо каждой шестнадцатеричной цифры подставляют соответствующее четырехзначное двоичное число:

1001 0110 **b** = 96 **h** 0AD **h** = 1010 1101 **b**

В ассемблерных программах при записи чисел, начинающихся с A, B, C, D, E, F, в начале приписывается цифра 0, чтобы нельзя было спутать такое число с названием переменной или другим идентификатором. После шестнадцатеричных чисел ставится буква «**h**».

Десятичное	Двоичное	Шестнадцатиричное
0	0000b	00h
1	0001b	01h
2	0010b	02h
3	0011b	03h
4	0100b	04h
5	0101b	05h
6	0110b	06h
7	0111b	07h
8	1000b	08h
9	1001b	09h
10	1010b	0Ah
11	1011b	0Bh
12	1100b	0Ch
13	1101b	0Dh
14	1110b	0Eh
15	1111b	0Fh
16	10000b	10h

1.4. Числа со знаком

Для представления отрицательных чисел существует специальная операция, известная как **дополнение** до двух. Для изменения знака числа выполняют инверсию, то есть заменяют в двоичном представлении числа все единицы нулями и нули единицами, а затем прибавляют 1.

150 = 0096 **h** = **0**000 0000 1001 0110 **b**
инверсия дает: 1111 1111 0110 1001 **b**
+1 = **1**111 1111 0110 1010 **b** = 0FF6A **h**

Проверим, что полученное число на самом деле –150: сумма с +150 должна, быть равна нулю:

+150 + (–150) = 0096 **h** + FF6A **h** = 10000 **h**

Единица в l6-м разряде не помещается в слово, и значит, мы действительно получили 0.

1.5. Организация памяти

Процессор **Intel** после включения питания оказывается в так называемом режиме реальной адресации памяти, или просто **реальном режиме**. Большинство операционных систем сразу же переводят его в **защищенный режим**, позволяющий им обеспечивать многозадачность, распределение памяти и др. Пользовательские программы в таких операционных системах часто работают еще в одном режиме, **режиме V86** (V – virtual), из которого им доступно все тоже, что и из реального, кроме команд, относящихся к управлению защищенным режимом.

С точки зрения процессора, **память** – это последовательность байт, каждому присвоен уникальный номер (**физический адрес**).

Плоская (flat) модель памяти **защищенного режима** – это непрерывный массив; объем равен 2^{32} байт (от 0 до 2^{32}–1).

Если используется **сегментированная модель памяти**, то программы могут работать с памятью как с несколькими непрерывными последовательностями байт – **сегментами**. Размер сегмента для **сегментированной** модели памяти **защищенного режима**: 2^{32} байт (от 0 до 2^{32}–1). Размер сегмента для **сегментированной** модели памяти **реального режима**: 2^{20} байт (от 0 до 2^{20}–1). Для задания **физического адреса** любого байта требуется два числа:

– адрес начала сегмента – **база** (**селектор**),

– адрес искомого байта внутри сегмента – **смещение** (**эффективный адрес**) (см. также раздел 2.2).

физический адрес =
= база (селектор) : смещение (эффективный адрес)

Кроме основной памяти программы могут использовать **регистры** – специальные ячейки памяти, расположенные физически внутри процессора, доступ к которым осуществляется не по адресам, а по именам.

2. РЕГИСТРЫ

Начиная с i386, процессоры **Intel** содержат:

– 16 **основных регистров** для пользовательских программ:

– регистры общего назначения (8);

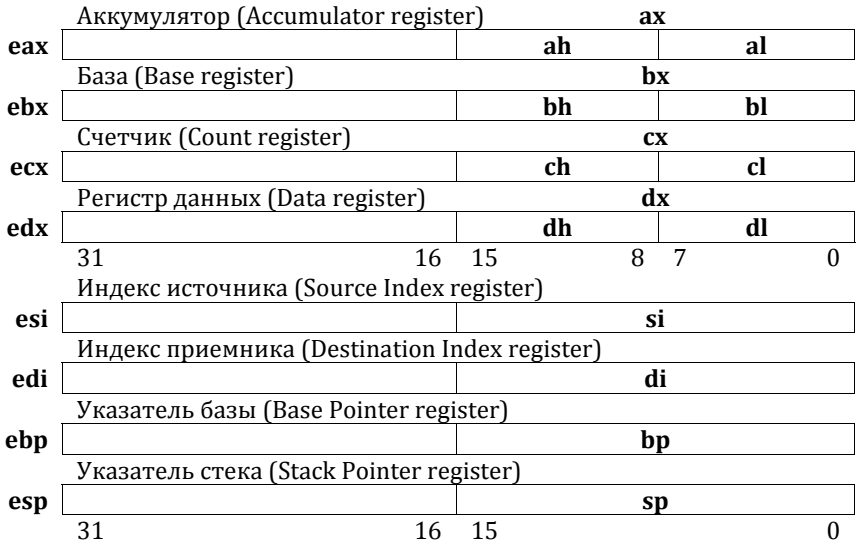
– сегментные регистры (6);

– регистры состояния и управления (2);

– регистры для работы с числами с плавающей точкой (**регистры сопроцессора**) и мультимедийными приложениями (**регистры MMX-расширения**);

– **системные регистры**.

2.1. Регистры общего назначения



Регистры **eax**, **ebx**, **ecx**, **edx** могут использоваться без ограничений для любых целей. Названия этих регистров происходят от того, что некоторые команды применяют их специальным образом (либо использование какого-то регистра обязательно, либо происходит неявно).

Следующие 4 регистра имеют более конкретное назначение и могут применяться для хранения всевозможных временных переменных, только когда они не используются по назначению:

1) регистры **esi** и **edi** используются в паре в строковых операциях (текущий адрес элемента в цепочке-источнике (**esi**) и в цепочке-приемнике (**edi**));

2) регистры **ebp** и **esp** используются при работе со стеком (**esp** – указатель вершины стека).

Младшие 16 бит каждого из этих регистров могут использоваться как самостоятельные регистры, и имеют имена (в процессорах i86 – i286 все регистры имели размер 16 бит и назывались именно так, а 32-битные появились с введением 32-битной архитектуры в i386 («e» – extention)).

Отдельные байты в первых четырех 16-битных регистрах тоже имеют свои имена и могут использоваться как 8-битные регистры.

2.2. Сегментные регистры

Процессор аппаратно поддерживает организацию программы в виде 3-х частей: **сегмент данных** – содержит обрабатываемые программой данные; **сегмент кода** – содержит команды исполняемой программы; **сегмент стека** – содержит временные данные (параметры вызываемых подпрограмм, локальные переменные, адреса возврата).

Сегментные регистры содержат (с небольшой поправкой) адреса памяти, с которых начинаются соответствующие сегменты.

Регистр сегмента кода (Code Segment register)

cs

Регистры сегмента данных (Data Segment register)

ds
es
fs
gs

(**fs**, **gs** появились в i286)

Регистр сегмента стека (Stack Segment register)

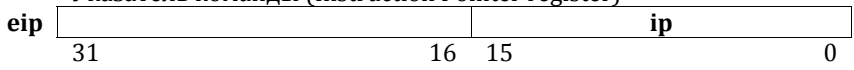
ss

15 0

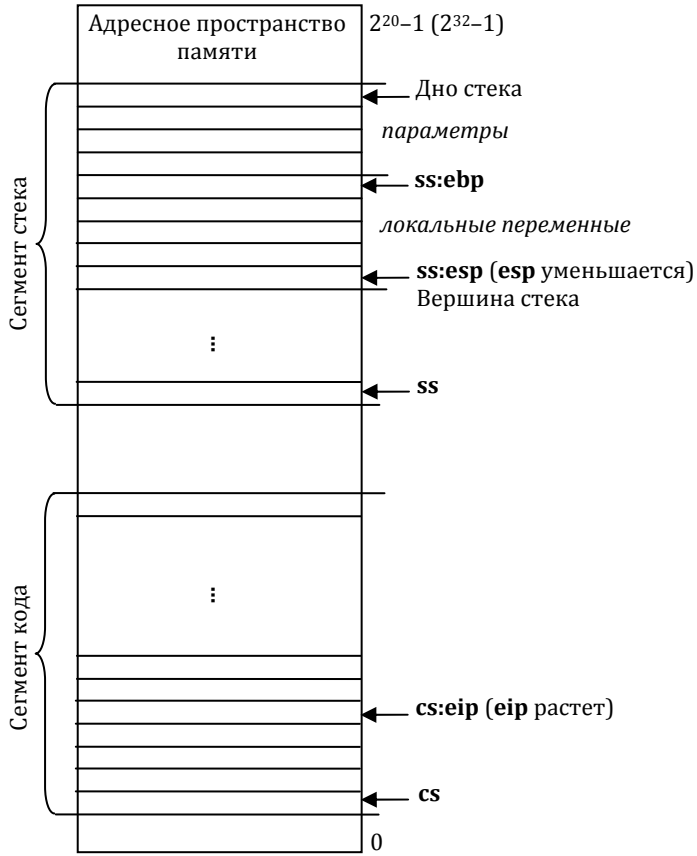
В **реальном режиме** селектор любого сегмента равен адресу его начала, деленному на 16 (сдвинут на 4 бита вправо). Чтобы получить физический адрес в памяти, 16-битное смещение складывают с этим селектором, сдвинутым предварительно влево на 4 бита (эта операция сдвига выполняется на аппаратном уровне). Таким образом, оказывается, что максимальный доступный адрес в реальном режиме 2^{20}–1 (всего 2^{20} раз-

личных адресов). Для сравнения, в **защищенном режиме** адрес начала для каждого сегмента хранится отдельно, так что возможно 2^{32} (для 32-разрядной адресной шины) различных адресов.

Смещение следующей выполняемой команды всегда хранится в специальном регистре **eip** (он относится к регистрам состояния и управления).



cs : eip – физический адрес следующей выполняемой команды
ss : esp – физический адрес вершины стека

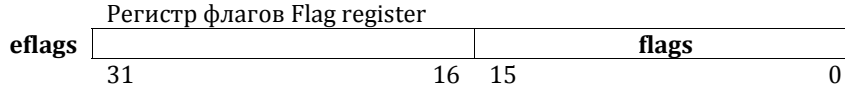


При записи в стек значение смещения уменьшается, то есть **стек «растет вниз»** от максимально возможного адреса.

При вызове подпрограммы параметры в большинстве случаев помещают в стек, а в **ebp** записывают текущее значение **esp**. Тогда, если подпрограмма использует стек для хранения локальных переменных, **esp** изменится, но **ebp** можно будет использовать для того, чтобы считывать значения параметров напрямую из стека (их смещения будут записываться как **ebp** + номер параметра). Более подробно см. в разделе 8.8.

2.3. Регистр флагов

Еще один важный регистр, использующийся при выполнении большинства команд, – регистр флагов **eflags** (относится к регистрам состояния и управления).



В этом регистре каждый бит является **флагом**, то есть устанавливается в 1 при определенных условиях или установка его в 1 изменяет поведение процессора.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	**nt**	**iopl**		**of**	**df**	**if**	**tf**	**sf**	**zf**	**0**	**af**	**0**	**pf**	**1**	**cf**

cf – флаг переноса. Устанавливается в 1, если результат выполненной операции не уместился в приемнике и произошел перенос из старшего бита или если требуется заем (при вычитании), иначе устанавливается в 0. Например, после сложения слова 0FFFF**h** и 1, если регистр, в который надо поместить результат, – слово, в него будет записано 0000**h** и флаг **cf** = 1.

pf – флаг четности (паритета). Устанавливается в 1, если младший байт результата выполненной команды содержит четное число бит, равных 1; устанавливается в 0, если число единичных бит нечетное.

af – флаг полупереноса или вспомогательного переноса. Устанавливается в 1, если в результате выполнения команды произошел перенос (или заем) из третьего бита в четвертый. Этот флаг используется автоматически командами двоично-десятичной коррекции.

zf – флаг нуля. Устанавливается в 1, если результат выполненной команды – ноль.

sf – флаг знака. Этот флаг всегда равен старшему биту результата.

tf – флаг трассировки. Этот флаг был предусмотрен для работы отладчиков, не использующих защищенный режим. Установка его в 1 приводит к тому, что после выполнения каждой команды программы управление временно передается отладчику (вызывается прерывание № 1).

if – флаг прерываний. Установка этого флага в 1 приводит к тому, что процессор перестает обрабатывать прерывания от внешних устройств. Обычно его устанавливают на короткое время для выполнения критических участков кода.

df – флаг направления. Этот флаг контролирует поведение команд обработки строк – когда он установлен в 1, строки обрабатываются в сторону уменьшения адресов, а когда = 0 – наоборот.

of – флаг переполнения. Этот флаг устанавливается в 1, если результат выполненной арифметической операции над числами со знаком выходит за допустимые для них пределы. Например, если при сложении двух положительных чисел получается число со старшим битом, равным единице (то есть отрицательное) и наоборот.

Флаги **iopl** (уровень привилегий ввода-вывода) и **nt** (вложенная задача) применяются в защищенном режиме.

Все флаги, расположенные в старшем слове регистра **eflags**, имеют отношение к управлению защищенным режимом.

3. СПОСОБЫ АДРЕСАЦИИ

Рассмотрим способы задания адреса операнда для команд процессора – способы адресации.

1. Регистровая адресация (по регистру). Операнды могут располагаться в любых регистрах общего назначения и сегментных регистрах.

```
mov   eax,ebx
;eax = ebx
;комментарий в ассемблере начинается с «;»
```

2. Непосредственная адресация. Некоторые команды (все арифметические команды, кроме деления) позволяют указывать один из операндов непосредственно в тексте программы.

```
mov   eax,2
;eax = 2
```

3. Прямая адресация (по смещению). Если известен адрес операнда, располагающегося в памяти, можно использовать этот адрес.

```
mov   ax,es:0001
;в es - начало сегмента, 0001 – смещение
mov   ax,es:i
;переменная i должна быть описана
;в сегменте es – ассемблер сам заменит
;слово «i» на соответствующий адрес.
```

Адресация отличается для реального и защищенного режимов: в реальном режиме смещение всегда 16-битное (слово); в защищенном режиме (например, ассемблерная вставка в C++), смещение не может превышать 32 бита (двойное слово).

Здесь и далее по умолчанию используется сегментный регистр **ds**, но если смещение берут из регистров **esp** или **ebp**, то в качестве сегментного регистра по умолчанию используется **ss**.

4. Косвенная адресация (по базе). Адрес операнда в памяти можно не указывать непосредственно, а хранить в регистре общего назначения: **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **ebp**, **esp**.

```
mov   ax,[ebx]
;ax = *(ds:ebx) – операция разадресации:
;помещает в ax слово из ячейки памяти,
;селектор сегмента которой находится в ds,
;а смещение — в ebx
```

В реальном режиме можно свободно пользоваться всеми 32-битными регистрами, надо только следить, чтобы их содержимое не превышало границ 16-битного слова.

5. Косвенная адресация со сдвигом. Следующие три команды эквивалентны.

```
mov   ax,[ebx+2]
mov   ax,[ebx]+2
mov   ax,2[ebx]
;помещает в ax слово, находящееся в
;сегменте, указанном в ds, со смещением на
;2 большим, чем число, находящееся в ebx
```

6. Косвенная адресация с масштабированием.

```
mov   ax,[ebx·МАСШТАБ]+2
;МАСШТАБ – размер элемента массива
;в байтах, может быть равным 1, 2, 4, 8
```

7. Косвенная адресация с индексированием. Следующие пять команд эквивалентны.

```
mov   ax,[ebx+esi+2]
mov   ax,[ebx][esi]+2
mov   ax,[ebx+2][esi]
mov   ax,[ebx][esi+2]
mov   ax,2[ebx][esi]
;в регистр ax помещается слово из ячейки
;памяти со смещением, равным сумме чисел
;из ebx и esi плюс 2
```

8. Косвенная адресация с индексированием и масштабированием. Это самая полная возможная схема адресации, в которую входят все случаи, рассмотренные ранее, как частные.

```
mov   ax,es:[ebx+esi·МАСШТАБ+СДВИГ]
```

4. ПОЛЕЗНЫЕ ЗАМЕЧАНИЯ

1. Обнуление регистра **ax**.

```
xor   ax,ax
```

Обмен местами содержимого регистров **ax** и **bx**.

```
xor   ax,bx
xor   bx,ax
xor   ax,bx
```

2. Нельзя использовать переменную одновременно для источника и для приемника.

```
int i = 10;
int j = 20;
_asm{
 mov  eax,i
 add  eax,j
 mov  i,eax
 //i = i + j
}
```

3. Умножение может быть знаковым и беззнаковым.

```
int i;
_asm{
 mov  eax,-10      //eax == FFFFFFF6
 imul eax,10       //eax == FFFFFF9C
 mov  i,eax
}
cout << i << endl;
//на консоли: -100; если unsigned int i, то 65535
```

4. Получение абсолютного значения числа, используя всего две команды – изменение знака и переход на первую команду еще раз, если знак отрицательный.

```
char i = -1;
_asm{
 mov  al,i
 //al == 11111111b == FFh == -1
```

```
l:
 neg  al
 //al == 000000001b == 01h == 1
 js   l
 //если sf == 1 (число отрицательное),
 //то переход на метку l
}
cout << i << endl;
```

5. РАБОТА СО СТРОКАМИ

Пусть необходимо вычислить длину введенной с консоли строки. Представленная программа реализована как ассемблерная вставка на языке C/C++.

```
#include <stdio.h>
void main(){

 char s[256];
 char f[] = "%s";
 char fd[] = "%d\n";
 //в ассемблерных вставках на C/C++ не требуется
 //явно определять сегменты данных, стека и кода

 _asm {
  //=====Запрос строки для анализа=====
  //в C/C++ мы бы вызвали scanf("%s",s);
  //в ассемблерных вставках на C/C++ в стек надо
  //поместить адреса аргументов функции,
  //перечисляя их справа налево
  lea       ebx,s
  //поместили адрес s в ebx
  push      ebx
  //поместили ebx (адрес s) в стек, автоматически
  //сместился (уменьшился) указатель стека esp на
  //4 байта
  lea       ecx,f
  //поместили адрес f в ecx
  push      ecx
  //поместили ecx (адрес f) в стек, автоматически
  //сместился (уменьшился) указатель стека esp
  //еще на 4 байта
  call      scanf
  //вызвали функцию scanf(f,s)
  add       esp,8
  //вернули указатель стека esp в исходное
  //состояние (увеличили на 8),
  //так как в ассемблерных вставках на C/C++
  //удаление параметров из стека выполняет
  //вызывающая процедура
```

```
  //=====Вычисление длины строки=====
  mov       ecx,-1
  //положили в ecx -1
  //это будет счетчик символов в строке
  dec       ebx
  //уменьшили ebx на 1
  //теперь в ebx адрес байта, предшествующий
  //первому байту строки s
b1:
//метка
  inc       ecx
  //увеличили ecx на 1
  inc       ebx
  //увеличили ebx на 1
  mov       al,[ebx]
  //скопировали в al байт, находящийся по адресу
  //из регистра ebx
  cmp       al,0
  //сравнили al с 0 (с символом '\0')
  //если не равны, то zf (флаг нуля)
  //автоматически устанавливается в 0
  jnz       b1
  //если zf == 0 (нет '\0'),
  //то перешли на метку b1

  //=====Вывод результата на консоль=====
  //в C/C++ мы бы вызвали printf("%i\n",ecx);
  //в стек надо поместить адреса аргументов
  //функции, перечисляя их справа налево
  push      ecx
  //поместили ecx (число символов) в стек
  lea       ebx, fd
  //поместили адрес fd в ebx
  push      ebx
  //поместили ebx (адрес fd) в стек
  call      printf
  //вызвали функцию printf(fd,s)
  add       esp,8
  //вернули указатель стека esp в исходное
  //состояние (увеличили на 8)
 }
}
```

Блок «Вычисление длины строки» можно реализовать с использованием строковых (цепочечных) команд.

```
mov       ecx,0FFFFFFFFh
//ecx = FFFFFFFF
lea       edi,s
//поместили в edi адрес строки-приемника
mov       al,0
//al = '\0'
cld
//флаг направления df = 0 (строки
//обрабатываются в сторону увеличения адресов)
repne     scasb
//scasb – сканирование байта из edi
//(сравнивает содержимое al с байтом из edi)
//repne повторяет команду scasb столько раз,
//сколько указано в ecx (уменьшая его
//содержимое на каждом шаге)
//или пока не равно (пока s[i] != '\0')

//по окончанию цикла ecx ==
//(FFFFFFFF - ("длина строки" + 1))
//тогда "длина строки" == FFFFFFFF - ecx - 1
//или "длина строки" == FFFFFFFE - ecx
mov       ebx,0FFFFFFFEh
//ebx = FFFFFFFE
sub       ebx,ecx
//ebx = ebx - ecx
//теперь ebx == "длина строки"
```

6. ЦЕЛОЧИСЛЕННЫЕ ВЫЧИСЛЕНИЯ

6.1. Обработка переполнений

Команды **add** и **sub** не различают знак числа. Но использовать их можно как для чисел со знаком, так и для беззнаковых. При этом надо анализировать значения флагов: **cf** – перенос при сложении чисел без знака или займ при вычитании чисел без знака; **of** – переполнение при сложении чисел со знаком.

```
//=====Пример, приводящий к переносу=====
unsigned char i = 255, j = 1;
_asm{
 mov  al,i  //al == 255 ==  11111111b == 0FFh
 add  al,j  //al == 256 == 100000000b == 100h,
 //of = 0, cf = 1, sf = 0
 //произошел перенос при сложении чисел без знака
 jc   l1
 //если был перенос (cf == 1), переходим на l1
 . . .
 //что-то делаем, если не было переноса
l1:
 . . .
 //что-то делаем, если был перенос
}

//=====Пример, приводящий к переполнению=====
char i = 127, j = 1;
_asm{
 mov  al,i  //al == 127 == 01111111b == 7Fh
 add  al,j  //al == 128 == 10000000b == 80h,
 //of = 1, cf =0, sf = 1 произошло переполнение
 //при сложении чисел со знаком
 jo   l1
 //если было переполнение (of == 1),
 //переходим на l1
 . . .
 //что-то делаем, если не было переполнения
```

```
l1:
 . . .
 //что-то делаем, если было переполнение
}
```

6.2. Вычисление значения выражения $x \cdot \Sigma a_{ij}$

```
#include<iostream>
using namespace std;
void main(){
 char a[3][3] = {1,1,1,1,1,1,1,1,1};
 char x = -1, rezult, key = 0;
 _asm{
  mov       ecx,9
  //ecx = 9 (счетчик для команды loop)
  xor       eax,eax
  //теперь eax == 0
  xor       ebx,ebx
  //теперь ebx == 0
l1:
  add       al,a[ebx]
  //al = al + a[ebx]
  jo        l2
  //если было переполнение (of == 1),
  //то переходим на метку l2
  add       ebx,1
  //ebx = ebx + 1
  //увеличиваем на 1, так как тип массива char
  loop      l1
  //уменьшаем ecx на 1 и переходим на метку l1
  imul      al,x
  //al = al * x
  //флаги of == 0, cf == 0, если результат
  //умножения поместился целиком в приемник,
  //иначе  of == 1, cf == 1.
  jo        l2
  //если было переполнение (of == 1),
  //переходим на l2
  mov       rezult,al
  //rezult = al
```

```
  jmp       l3
  //переходим на метку l3
l2:
  mov       key,1
  //key = 1
l3:
 }
 if (!key)
  cout << rezult << endl;
 else
  cout << "Overflow" << endl;
}
```

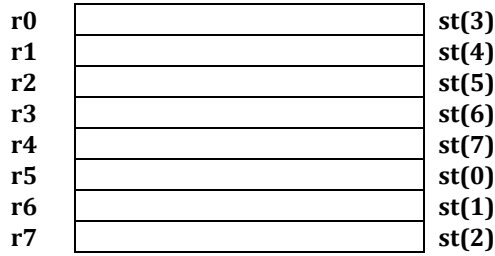
Следующий фрагмент кода демонстрирует изменения, необходимые при переходе к типу данных **int** (диапазон значений: от –2147483647 до 2147483647).

```
 int a[3][3] = {1,1,1,1,1,1,1,1,1};
 int x = -1, rezult, key = 0;
 _asm{
  mov       ecx,9
  xor       eax,eax
  xor       ebx,ebx     //можно lea ebx,a
l1:
  add       eax,a[ebx]  //тогда add eax,[ebx]
  jo        l2
  add       ebx,4
  //int занимает 4 байта
  loop      l1
  imul      eax,x
  jo  l2
  mov       rezult,eax
  jmp       l3
l2:
  mov       key,1
l3:
 }
```

7. ВЫЧИСЛЕНИЯ С ПЛАВАЮЩЕЙ ТОЧКОЙ

7.1. Регистры FPU

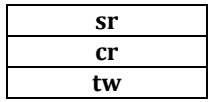
1. Регистры данных – 8 основных регистров **r0**, **r1**, **...**, **r7** (физические номера). Эти регистры рассматриваются как стек (кольцо). Вершина стека – **st(0)**, а более глубокие элементы – **st(1)**, **st(2)**, ..., **st(7)** (логические номера).



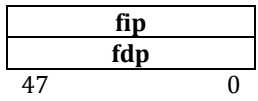
top (3-битовое поле) **= 5**

Если, регистр **r5** называется **st(0)**, то при записи в стек числа, оно будет записано в регистр **r4**, который станет называться **st(0)**, **r5** станет называться **st(1)** и т. д. К регистрам **r0**, **r1**, **...**, **r7 нельзя обращаться напрямую** (по именам).

2. Вспомогательные регистры sr, cr, tw, fip, fdp.



Регистр состояний
Регистр управления
Регистр тегов



Адрес последней выполненной команды
Адрес ее операнда

Регистр состояний sr – отражает текущее состояние FPU.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
b	**c3**	**top**			**c2**	**c1**	**c0**	**es**	**sf**	**pe**	**ue**	**oe**	**ze**	**de**	**ie**

c3, c2, c1, c0 – условные флаги (используются для условных переходов, аналог **eflags**);

top – номер регистра, являющегося вершиной стека;

es – общий флаг ошибки (равен 1, если произошло хотя бы одно из 6 исключений);

sf – ошибка работы стека FPU (попытка писать в непустую позицию в стеке или считать число из пустой позиции в стеке).

6 флагов исключительных ситуаций:

pe – флаг неточного результата - результат не может быть представлен точно;

ue – флаг антипереполнения – результат слишком маленький;

oe – флаг переполнения – результат слишком большой;

ze – флаг деления на ноль – выполнено деление на ноль;

de – флаг денормализованного операнда – выполнена операция над денормализованным числом;

ie – флаг недопустимой (недействительной) операции ($-\infty - \infty$, $\infty + \infty$ и т. п.).

Регистр управления cr – определяет особенности обработки численных данных.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
			ic	**rc**		**pc**				**pm**	**um**	**om**	**zm**	**dm**	**im**

6 масок исключений:

pm – маска неточного результата;

um – маска антипереполнения;

om – маска переполнения;

zm – маска деления на ноль;

dm – маска денормализованного операнда;

im – маска недопустимой операции.

Если маскирующий бит установлен, исключения не происходит.

rc – управление округлением

m – значение в **st**, a < m < b	
00	к ближайшему числу
01	к $-\infty$ (m = a)
10	к $+\infty$ (m = b)
11	к 0 (отбрасывается дробная часть)

pc – управление точностью

00	одинарная точность (32-битные числа)
10	двойная точность (64-битные числа)
11	расширенная точность (80-битные числа)

Регистр тегов tw содержит восемь пар бит, описывающих содержание каждого регистра данных.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
r7		**r6**		**r5**		**r4**		**r3**		**r2**		**r1**		**r0**	

00	регистр содержит число
01	регистр содержит 0
10	регистр содержит нечисло, бесконечность, денормализованное число, NAN
11	регистр пуст

Регистры fip и fdp содержат адрес последней выполненной команды (за некоторым исключением) и адрес ее операнда. Используются в обработчиках исключений для анализа вызвавшей его команды.

7.2. Правила образования команд FPU

1. Все названия команд начинаются с **f** (float).
2. Вторая буква определяет тип операнда в памяти:
   i – целое двоичное число,
   b – целое десятичное число,
   отсутствие буквы – вещественное число.
3. Последняя буква **p** означает, что последним действием команды является извлечение (выталкивание) из стека: **st(0)** помечается как пустой, **top** = **top** + 1.
4. Последняя или предпоследняя буква **r** (reversed) означает реверсионное следование операндов.

```
int i = 10; double y = 10;
```

```
fld y
fadd y
//st(0)== 20
```

```
fld i
fadd i
//в st(0)
//"мусор"
```

```
fild i
fiadd i
//st(0)==20
```

7.3. Примеры вычислений с плавающей точкой

1. Вычислить $z = \left(\sqrt{|x|} - y\right)^2$.

```
//здесь и далее содержимое стека регистров данных
//(или просто стека) будем заключать в квадратные
//скобки
double x = -0.25, y = 1, z;
_asm{
 finit
 //инициализация сопроцессора
 fld        x
 //st(0) = x
 fabs
 //st(0) = |st(0)|
 //[st(0) == |x|]
 fsqrt
 //st(0) = sqrt(st(0))
 //[st(0) == sqrt(|x|)]
 fsub       y
 //st(0) = st(0)-y
 //[st(0) == sqrt(|x|)-y]
 fst        st(1)
 //st(1) = st(0)
 //[st(0) == st(1) == sqrt(|x|)-y)]
 fmul
 //st(1) = st(1) * st(0) и вытолкнуть
 //[st(0) == (sqrt(|x|) - y)*(sqrt(|x|) - y)]
 fstp       z
 //z = st(0) и вытолкнуть
 //[стек пуст]
}
cout << z << endl;
```

2. При вычислениях с плавающей точкой следите, чтобы не происходило заполнение всех 8 регистров данных. В случае переполнения стека регистров данных результат вычислений не определен. Следует, где это возможно, использовать команды с выталкиванием из стека (заканчивающиеся символом **p**).

```
int i = 10;
double z = 0;
_asm{
 finit
 //инициализация сопроцессора
 mov        ecx,i
 //ecx = i (счетчик для команды loop)
l1:
 fld1
 //st(0) = 1
 fadd       z
 //st(0) = st(0) + z
 fstp       z
 //z = st(0) и вытолкнуть
 //в случае fst z результат после 10
 //итераций не определен!
 loop l1
}
```

3. Вычислить $y = 1/1! + 1/2! + \ldots + 1/i!$.

```
#include<iostream>
using namespace std;
void main(){
 int i = 10;
 double y;
 _asm{
  finit
  //инициализация сопроцессора
  fld1
  //st(0) = 1
  fld1
  //st(0) = 1
  //[st(1) == 1  st(0) == 1]
  fst       y
  //y = st(0)
  //(y == 1 == 1!/1 (первое слагаемое))
  //[st(1) == 1!  st(0) == 1]
  mov       ecx,i
  //ecx = i
  dec       ecx
  //ecx = ecx - 1 (счетчик для команды loop)
```

```
l1:
  //сейчас st(1) == (i-1)!  st(0) == i-1
  fld1
  //st(0) = 1
  //[st(2)==(i-1)!  st(1) == i-1  st(0) == 1]
  faddp     st(1),st(0)
  //st(1) = st(1)+st(0) и вытолкнуть
  //[st(1)==(i-1)!  st(0) == i]
  fmul      st(1),st(0)
  //st(1) = st(1)*st(0)
  //[st(1) == i!  st(0) == i]
  fld1
  //st(0) = 1
  //[st(2) == i!  st(1) == i  st(0) == 1]
  fdiv      st(0),st(2)
  //st(0) = st(0)/st(2)
  //[st(2) == i!  st(1) == i  st(0) == 1/i!]
  fadd      y
  //st(0) = st(0)+y
  //[st(2) == i!  st(1) == i  st(0) == y + 1/i!]
  fstp      y
  //y = st(0) и вытолкнуть
  //(y == 1/1! + 1/2! + ... + 1/i!)
  //[st(1) == i!  st(0) == i]
  loop      l1
 }
 cout << y << endl;
}
```

4. Команды сравнения вещественных чисел, например **fcom y**, устанавливают флаги **c0**, **c2** и **c3** регистра управления **sr**. Содержимое этого регистра можно скопировать в регистр **ax**: **fstsw ax**, а затем командой **sahf** – в регистр **flags** (в регистр **flags** попадет содержимое регистра **ah**, при этом флаги продублируются следующим образом: **zf** = **c3**, **pf** = **c2**, **cf** = **c0**). Теперь можно использовать условные команды (например, $\mathbf{j}_{\mathbf{xx}}$) как после команды **cmp**.

Условие	**c3 → zf**	**c2 → pf**	**c0 → cf**
st(0) > y	0	0	0
st(0) < y	0	0	1
st(0) = y	1	0	0
не сравнимы	1	1	1

```
fldz
//st(0) = 0
fcom        y
//st(0) сравнить с y
fstsw       ax
//ax = sr
sahf
//flags = ah: zf = c3, pf = c2, cf =c0
je          error
//если операнды сравнения равны,
//то переход на метку error
```

Начиная с процессоров P6 (Pentium Pro и Pentium II), для сравнения вещественных чисел появилась команда fcomi. В отличие от предыдущего способа, эта команда не требует изменения регистра ax и выполняется быстрее.

```
fldz
//st(0) = 0
fld         y
//st(0) = y
//[st(1) == 0  st(0) == y]
fcomi       st(0),st(1)
//st(0) сравнить с st(1)
je error
//если операнды сравнения равны,
//то переход на метку error
```

5. Ввод/вывод вещественного числа практически не отличается от ввода/вывода, рассмотренного в разделе 5.

```
double x = 0;
char s[] = "%lf", char s1[] = "%lf\n";
_asm{
 finit //инициализация сопроцессора
 //=====Ввод=====
 lea        eax,x
 //поместить адрес переменной x в eax
 push       eax
 //поместить eax (адрес x) в сегмент стека,
 //автоматически уменьшится указатель сегмента
 //стека esp на 4 байта
```

```
lea         eax,s
//поместить адрес строки s в eax
push        eax
//поместить eax (адрес s) в сегмент стека,
//автоматически уменьшится указатель сегмента
//стека esp еще на 4 байта
call        scanf
//вызвать функцию scanf(s,&x)
add         esp,8
//вернуть указатель сегмента стека esp в
//исходное состояние (увеличить на 8)
//=====Вывод=====
fld         x
//st(0) = x
sub         esp,8
//указатель сегмента стека esp уменьшить на 8
fstp        qword ptr [esp]
//st(0) (то есть x) загрузить в сегмент стека
//(в соответствии с указателем сегмента стека
//esp) и вытолкнуть
lea         eax,s1
//поместить адрес строки s1 в eax
push        eax
//поместить eax (адрес s1) в сегмент стека,
//автоматически уменьшится указатель сегмента
//стека esp еще на 4 байта
call        printf
//вызвать функцию printf(s1,x)
add         esp,12
//вернуть указатель сегмента стека esp в
//исходное состояние (увеличить на 12)
}
```

7.4. Вычисление значений функции на заданном отрезке

Необходимо вывести на консоль таблицу значений функции

$$y=\begin{cases} ax^2+b, & x<0, b\neq 0 \\ (x-a)/(x-c), & x>0, b=0 \\ x/c, & \text{иначе} \end{cases}$$

на отрезке [–10, 10.4] с шагом 0.5. Реализацию программы разобьем на несколько этапов.

Этап I

```
#include<stdio.h>
int main()
{
 double a=0.5, b=0, c=1, x, y;
 double x1=-10;   //левая граница отрезка
 double x2=10.4;  //правая граница отрезка
 double t=0.5;    //шаг
 int key;
 char m1[]="----------------------------\n";
 char m2[]="|      x      |      y      |\n";
 char m3[]="----------------------------\n";
 char s0[]="| %11.3lf | %11.3lf |\n";
 char s1[]="| %11.3lf | ----------- |\n";
 _asm{
  //=====Заголовок таблицы=====
  . . .
  //=====Таблица значений функции y=====
  . . .
 }
}
```

Этап II

```
 _asm{
  //=====Заголовок таблицы=====
  lea       ebx,m1
  push      ebx
  call      printf
  add       esp,4
  lea       ebx,m2
  push      ebx
  call      printf
  add       esp,4
  lea       ebx,m3
  push      ebx
  call      printf
  add       esp,4
  //=====Таблица значений функции y=====
  . . .
 }
```

Этап III

```
  //=====Таблица значений функции y=====
  finit
  //инициализация сопроцессора
  fld       x1
  //st(0) = x1
  fstp      x
  //x = st(0) и вытолкнуть (x == x1)
  //[стек пуст]
  fldz
  //st(0) = 0
begin:
  fld       x2
  //st(0) = x2
  //[st(1) == 0  st(0) == x2]
  fsub      x
  //st(0) = st(0)-x (st(0) == (x2-x))
  fcomip    st(0),st(1)
  //st(0) сравнить с st(1) и вытолкнуть
  //[st(0) == 0]
  jb        end
  //если (x2-x)<0 (выход за правую границу),
  //то переход на метку end

  //=====Расчет y(x)=====
  . . .

  //=====Вывод строки таблицы=====
  . . .

  fld       x
  //st(0) = x
  //[st(1) == 0  st(0) == x]
  fadd      t
  //st(0) = st(0)+t (st(0) == (x+t))
  fstp      x
  //x = st(0) и вытолкнуть
  //(x увеличился на шаг t) [st(0) == 0]
  jmp       begin
  //переход на метку begin
end:
```

Этап IV

```
  //=====Расчет y(x)=====
  //если x<0, b!=0, то y=a*x*x+b
  //если x>0, b==0, то y=(x-a)/(x-c)
  //        но если (x-c)==0 - деление на 0
  //иначе y=x/c
  //        но если c==0 - деление на 0
  //Стратегия: если значение функции в точке x
  //определено, то результат помещаем в y
  //и key присваиваем 0, иначе key присваиваем 1.
  //В стеке [st(0) == 0]

//случай x<0, b!=0, y=a*x*x+b
  fld       x
  //st(0) = x
  //[st(1) == 0  st(0) == x]
  fcomip    st(0),st(1)
  //st(0) сравнить с st(1) и вытолкнуть
  /[st(0) == 0]
  jnb       l2
  //если x >= 0 (x не меньше 0),
  //то переход на метку l2
  fld       b
  //st(0) = b
  //[st(1) == 0  st(0) == b]
  fcomip    st(0),st(1)
  //st(0) сравнить с st(1) и вытолкнуть
  //[st(0) == 0]
  je        l2
  //если b == 0, то переход на метку l2
  fld       a
  //st(0) = a
  //[st(1) == 0  st(0) == a]
  fmul      x
  //st(0) = st(0)*x (st(0) == a*x)
  fmul      x
  //st(0) = st(0)*x (st(0) == a*x*x)
  fadd      b
  //st(0) = st(0)+b (st(0) == a*x*x+b)
  fstp      y
  //y = st(0) и вытолкнуть (y == a*x*x+b)
  //[st(0) == 0]
```

```
  mov       key,0
  //key = 0 (значение y определено)
  jmp       l1
  //переход на метку l1

//случай x>0, b==0, y=(x-a)/(x-c)
l2:
  fld       x
  //st(0) = x
  //[st(1) == 0  st(0) == x]
  fcomip    st(0),st(1)
  //st(0) сравнить с st(1) и вытолкнуть
  //[st(0) == 0]
  jbe       l3
  //если x <= 0 (x меньше или равно 0),
  //то переход на метку l3
  fld       b
  //st(0) = b
  //[st(1) == 0  st(0) == b]
  fcomip    st(0),st(1)
  //st(0) сравнить с st(1) и вытолкнуть
  //[st(0) == 0]
  jne       l3
  //если b != 0 (b не равно 0),
  //то переход на метку l3
  fld       x
  //st(0) = x
  //[st(1) == 0  st(0) == x]
  fsub      c
  //st(0) = st(0)-c (st(0) == x-c)
  fcomip    st(0),st(1)
  //st(0) сравнить с st(1) и вытолкнуть
  //[st(0) == 0]
  je        l4
  //если (x-c) == 0,
  //то переход на метку l4
  fld       x
  //st(0) = x
  //[st(1) == 0  st(0) == x]
  fsub      a
  //st(0) = st(0)-a (st(0) == x-a)
```

```
  fld       x
  //st(0) = x
  //[st(2) == 0  st(1) == x-a  st(0) == x]
  fsub      c
  //st(0) = st(0)-c (st(0) == x-c)
  fdivp     st(1),st(0)
  //st(1) = st(1)/st(0) и вытолкнуть
  //[st(1) == 0  st(0) == (x-a)/(x-c)]
  fstp      y
  //y = st(0) и вытолкнуть (y == (x-a)/(x-c))
  //[st(0) == 0]
  mov       key,0
  //key = 0 (значение y определено)
  jmp       l1
  //переход на метку l1

//случай y=x/c
l3:
  fld       c
  //st(0) = c
  //[st(1) == 0  st(0) == c]
  fcomip    st(0),st(1)
  //st(0) сравнить с st(1) и вытолкнуть
  //[st(0) == 0]
  je        l4
  //если c == 0,
  //то переход на метку l4
  fld       x
  //st(0) = x
  //[st(1) == 0  st(0) == x]
  fdiv       c
  //st(0) = st(0)/c (st(0) == x/c)
  fstp      y
  //y = st(0) и вытолкнуть (y == (x/c))
  //[st(0) == 0]
  mov       key,0
  //key = 0 (значение y определено)
  jmp       l1
  //переход на метку l1
```

```
//деление на 0
l4:
  mov       key,1
  //key = 1 (значение y не определено)
l1:
```

Этап V

```
  //=====Вывод строки таблицы=====
  //Напомним, что если значение функции
  //в точке x определено,
  //то результат лежит в y и key == 0,
  //иначе key == 1
  //В стеке [st(0) == 0]

  cmp       key,0
  //сравнить key с 0
  je        l5
  //если равны, переход на метку l5

//~~~случай, когда y не определен (key==1)~~~
  sub       esp,8
  //указатель сегмента стека esp уменьшить на 8
  fld       x
  //st(0) = x [st(1) == 0  st(0) == x]
  fstp      qword ptr [esp]
  //st(0) (то есть x) загрузить в сегмент стека
  //(в соответствии с указателем сегмента стека
  //esp) и вытолкнуть [st(0) == 0]
  lea       eax,s1
  //поместить адрес строки s1 в eax
  push      eax
  //поместить eax (адрес s1) в сегмент стека,
  //автоматически уменьшится указатель сегмента
  //стека esp еще на 4 байта
  call      printf
  //вызвать функцию printf(s1,x)
  add       esp,12
  //вернуть указатель сегмента стека esp в
  //исходное состояние (увеличить на 12)
  jmp       l6
  //переход на метку l6
```

```
//~~~случай, когда y определен (key==0)~~~
15:
  sub       esp,8
  //указатель сегмента стека esp уменьшить на 8
  fld       y
  //st(0) = y [st(1) == 0  st(0) == y]
  fstp      qword ptr [esp]
  //st(0) (то есть y) загрузить в сегмент стека
  //(в соответствии с указателем сегмента стека
  //esp) и вытолкнуть [st(0) == 0]
  sub       esp,8
  //указатель сегмента стека esp уменьшить на 8
  fld       x
  //st(0) = x [st(1) == 0  st(0) == x]
  fstp      qword ptr [esp]
  //st(0) (то есть x) загрузили в сегмент стека
  //(в соответствии с указателем сегмента стека
  //esp) и вытолкнуть [st(0) == 0]
  lea       eax,s0
  //поместить адрес строки s0 в eax
  push      eax
  //поместить eax (адрес s0) в сегмент стека,
  //автоматически уменьшится указатель сегмента
  //стека esp еще на 4 байта
  call      printf
  //вызвать функцию printf(s0,x,y)
  add       esp,20
  //вернуть указатель сегмента стека esp в
  //исходное состояние (увеличть на 20)
16:
```

8. РЕАЛИЗАЦИЯ 16- И 32-РАЗРЯДНЫХ ПРОГРАММ НА АССЕМБЛЕРЕ

8.1. Прерывания

Прерывание – инициируемый определенным образом процесс, временно переключающий процессор на выполнение другой программы с последующим возобновлением выполнения прерванной программы. В реальном и защищенном режимах обработка прерываний осуществляется принципиально разными методами.

В реальном режиме адреса обработчиков прерываний начинаются с 0000h:0000h (по 4 байта на адрес каждого обработчика). Команда вызова прерывания **int** помещает в стек содержимое регистров **eflags**, **cs**, **eip**, далее действует аналогично команде **call**, вызывая прерывание с указанным номером.

Пример вызова прерывания:

```
mov     ax,4C00h        ;номер функции
int     21h             ;номер прерывания
;завершение выполнения программы
```

В защищенном режиме программа должна обращаться к операционной системе (так как адреса прерываний не доступны для прямого чтения).

8.2. Определение данных

Формат определения переменной:

имя d_x значение

При этом **d_x** может иметь следующий вид:

db определить байт;
dw определить слово (2 байта);
dd определить двойное слово (4 байта);
df определить 6 байт;
dq определить учетверенное слово (8 байт).

Примеры определения переменных:

```
text_string       db 'Hello world!$'
message           db "Hi, people!!!$"
number            dw 7
table             db 1,2,3,4,5,6,7,8,9,0Ah
float_number      dd 3.5e7
```

Имя переменной соответствует адресу первого из указанных значений.

```
mov     al,text_string      ;al == 48h (код 'H')
```

Переменная считается неинициализированной и ее значение на момент запуска программы может оказаться любым, если ее определить следующим образом:

```
i      db     ?
```

Если нужно заполнить участок памяти повторяющимися данными, используется специальный оператор **dup**:

```
massiv  dw 512 dup(?)
;создается массив из 512 неинициализированных
;слов, на первое из которых указывает переменная
;massiv
```

8.3. Модели памяти и объявление сегментов

1. Модель памяти задается директивой

.model модель, язык, модификатор

Модель:

1) **tiny** – код, данные и стек размещаются в одном сегменте размером до 64 Кб;

2) **small** – код размещается в одном сегменте, а данные и стек – в другом;

3) **medium** – код размещается в нескольких сегментах, а все данные – в одном, поэтому для доступа к данным используется только смещение, а для вызова подпрограмм – полные адреса;

4) **lage, huge** – и код, и данные могут занимать несколько сегментов;

5) **flat** – то же, что и **tiny**, но регистры 32-битные.

Язык (необязательный операнд): **c**, **pascal**, **basic**, **fortran**. Если язык указан, подразумевается, что процедуры рассчитаны на вызов из программ на соответствующем языке высокого уровня.

Модификатор (необязательный операнд):

1) **nearstack** – по умолчанию;

2) **farstack** – сегмент стека не объединяется в одну группу с сегментами данных.

2. Сегмент кода описывается директивой

.code

3. Сегмент стека описывается директивой

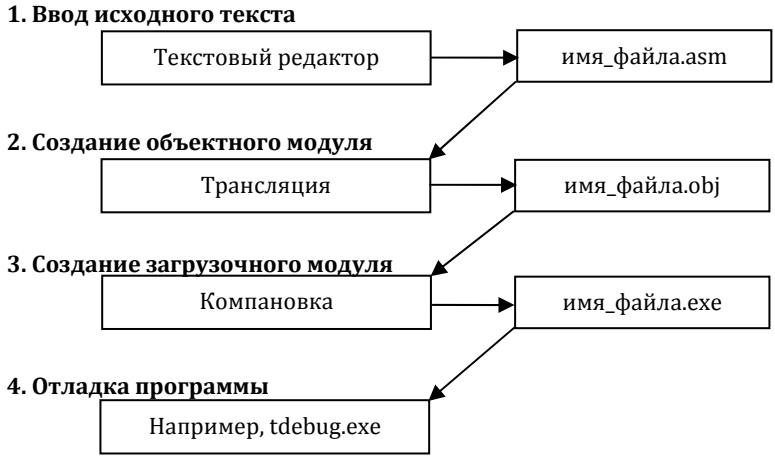
.stack размер

Необязательный параметр указывает размер стека.

4. Сегмент данных описывается директивой

.data

8.4. Процесс разработки программы



Создание exe-файла на ассемблере TASM:

Трансляция: `tasm.exe имя_файла.asm`

Компоновка: `link16.exe имя_файла.obj ,,,,,`

Создание exe-файла на ассемблере MASM:

Трансляция: `ml.exe /c имя_файла.asm`

Компоновка: `link16.exe имя_файла.obj ,,,,,`

8.5. Структура программы

Общая структура 16-ти разрядной программы на ассемблере TASM/MASM имеет нижеследующий вид (для 32-разрядных приложений – другая модель памяти (например, **flat**)).

```
.model small        ;модель памяти
.stack 256          ;объем стека 256 байт
.data               ;сегмент данных
      ;данные
.code               ;сегмент кода
start:
      ;команды
end start
```

```
;16-разрядная программа на ассемблере TASM/MASM
;вывод на экран заданной строки
.model small        ;модель памяти
.stack 256          ;объем стека 256 байт
.data               ;начало сегмента данных
      message       db    "Hi, people!!!$"
.code               ;начало сегмента кода
start:
      mov           ax,@data
      mov           ds,ax
      ;теперь в ds адрес сегмента данных
      lea           dx,message
      ;поместить адрес message в dx
      mov           ah,9h ;9-я функция
      int           21h   ;21-го прерывания
      ;вывод на экран информации по адресу из dx
      mov           ax,4C00h
      int           21h
      ;завершение выполнения программы
end start
```

8.6. Несколько решений одной задачи

Пусть заданы две строки **s1** и **s2**. Необходимо скопировать содержимое **s1** в **s2** и вывести **s2** на экран.

8.6.1. Ассемблерная вставка на C/C++

```
#include<iostream>
#include<string>
using namespace std;
void main(){
 char s1[] = "Hi, people!!!\n";
 int i = strlen(s1) + 1;
 char s2[] = "123456789123456789\n";
 _asm{
  cld
  //флаг направления df = 0 (строки
  //обрабатываются в сторону увеличения адресов)
  mov ecx,i
  //ecx = i
  lea esi,s1
  //поместить адрес строки s1 в esi (в источник)
  lea edi,s2
  //поместить адрес строки s2 в edi (в приемник)
  rep movsb
  //rep повторяет команду movsb столько раз,
  //сколько указано в ecx
  //movsb – копирование байта из esi в edi
 }
 cout << s2;
}
```

8.6.2. 16-разрядная программа на ассемблере TASM/MASM

```
.model small
.stack 256
.data
      s1    db 'Hi, people!!!',0Dh,0Ah,'$'
      len   equ $-s1
      ;equ – это аналог #define
      ;len == адрес '$' минус адрес 'H'
      ;в итоге len == длина строки s1
      s2    db '123456789123456789',0Dh,0Ah,'$'
      adr_s1      dd s1
      ;в adr_s1 адрес s1
      adr_s2      dd s2
      ;в adr_s1 адрес s2
```

```
.code
start:
      mov         ax,@data
      mov         ds,ax
      ;теперь в ds адрес сегмента данных
      cld
      ;флаг направления df = 0
      mov         cx,len
      ;cx = len
      lds         si,adr_s1
      ;поместить адрес из adr_s1 в ds:si
      les         di,adr_s2
      ;поместить адрес из adr_s2 в es:di
      rep         movsb
      ;rep повторяет команду movsb столько раз,
      ;сколько указано в cx
      ;movsb – копирование байта из si в di
      lea         dx,s2
      ;поместить адрес s2 в dx
      mov         ah,9
      int         21h
      ;вывод на экран информации по адресу из dx
      mov         ax,4C00h
      int         21h
      ;завершение выполнения программы
end start
```

8.6.3. Графическое win32-приложение на ассемблере MASM

```
include def32.inc
include kernel32.inc
include user32.inc
.386
.model flat
.data
      s     db    "My message",0
      s1    db    "Hi, people!!!",0
      len   equ   $-s1
      ;в len длина строки s1
      s2    db    "123456789123456789",0
.code
_start:
```

```
;метка точки входа должна начинаться
;с подчеркивания
      cld
      mov         ecx,len
      mov         esi,offset s1
      ;поместить адрес строки s1 в esi
      mov         edi,offset s2
      ;поместить адрес строки s2 в edi
      rep         movsb

      push        MB_ICONINFORMATION
      ;стиль окна
      push        offset s
      ;адрес строки с заголовком окна
      push        offset s2
      ;адрес строки с сообщением
      push        0
      ;идентификатор предка
      call        MessageBox
      ;вызов системной функции
      ;«окно с указанным сообщением»

      push        0
      ;код выхода
      call        ExitProcess
      ;вызов системной функции
      ;«завершение программы»
end _start
```

1. Чтобы вызвать системную функцию Windows, программа должна поместить в стек все параметры от последнего к первому и передать управление командой **call**. Все эти функции сами освобождают стек. Такая договоренность о передаче параметров называется **stdcall**[1].

2. Прежде чем скомпилировать **asm**-файл, нужно создать **inc**-файлы, в которые необходимо поместить директивы, описывающие вызываемые системные функции.

[1] В отличие от **stdcall**, соглашение **cdecl**, которое принято по умолчанию в C/C++, таково: параметры также перечисляются справа налево, но стек освобождает вызывающая процедура.

Имена всех системных функций Win32 модифицируются так, что перед именем функции ставится подчеркивание, а после – знак «@» и число байт, которое занимают параметры, передаваемые ей в стеке; затем «процедура-заглушка» добавляет к имени префикс «_imp_»[2]: **ExitProcess() ⇒ _ExitProcess@4() ⇒ __imp__ExitProcess@4**. В примерах мы будем обращаться напрямую к **__imp__ExitProcess@4.**

```
;======Файл kernel32.inc======
includelib        kernel32.lib
;включаемый файл с определениями функций
;из kernel32.dll
extrn             __imp__ExitProcess@4:dword
;истинные имена используемых функций
ExitProcess       equ   __imp__ExitProcess@4
;присваивания для облегчения читаемости кода

;======Файл user32.inc======
includelib        user32.lib
;включаемый файл с определениями функций
;из user32.dll
;это библиотека, в которую входят основные
;функции, отвечающие за оконный интерфейс
extrn             __imp__MessageBoxA@16:dword
;истинные имена используемых функций
MessageBox        equ   __imp__MessageBoxA@16
;присваивания для облегчения читаемости кода

;======Файл def32.inc======
MB_ICONINFORMATION      equ   40h
;включаемый файл с определениями констант и типов
;для программ под win32 из winuser.h
```

Для компиляции потребуются файлы **kernel32.lib** и **user32.lib**, а также файл **mspdb50.dll**.

3. Создание exe-файла на ассемблере MASM:

Трансляция: `ml/c/coff/Cp имя_файла.asm`

Компоновка: `link имя_файла.obj /subsystem:windows`

[2] Для соглашение **cdecl**: **ExitProcess() ⇒ _ExitProcess()**

8.6.4. Консольное win32-приложение на ассемблере MASM

```
include def32.inc
include kernel32.inc
.386
.model      flat
.data
      mes   dd    ?
      ;переменная для функции WriteConsole
      s1    db    "Hi, people!!!" ,0Dh,0Ah,0
      len   equ   $-s1
      //в len длина строки s1
      s2    db    "123456789123456789",0Dh,0Ah,0
.code
_start:
;метка точки входа должна начинаться
;с подчеркивания
      ;. . . код в рамке из раздела 8.6.3

      push        STD_OUTPUT_HANDLE
      call        GetStdHandle
      ;вызов системной функции
      ;«возврат идентификатора stdout в eax»

      mov         ebx,len
      ;в ebx длина строки для вывода
      push        0
      push        offset mes
      ;адрес переменной, в которую будет занесено
      ;число байт, действительно выведенных
      ;на консоль
      push        ebx
      ;сколько байт надо вывести на консоль
      push        offset s2
      ;адрес строки для вывода на консоль
      push        eax
      ;идентификатор буфера вывода
      call        WriteConsole
      ;вызов системной функции
      ;«вывод строки на консоль»

      push        0
      ;код выхода
```

```
      call        ExitProcess
      ;вызов системной функции
      ;«завершение программы»
end _start
```

1. Все функции, работающие со строками (как, например, **WriteConsole()**), существуют в двух вариантах. Если строка рассматривается в обычном смысле, как набор символов ASCII, к имени функции добавляется «A» (**WriteConsoleA()**). Другой вариант функции, использующий строки в формате UNICODE (два байта на символ), заканчивается буквой «U». В примерах будем использовать обычные ASCII-функции.

2. Как и в предыдущем примере, прежде чем скомпилировать asm-файл, нужно создать inc-файлы, в которые необходимо поместить директивы, описывающие вызываемые системные функции.

```
;======Файл kernel32.inc======
includelib        kernel32.lib
;включаемый файл с определениями функций
;из kernel32.dll
extrn             __imp__ExitProcess@4:dword
extrn             __imp__GetStdHandle@4:dword
extrn             __imp__WriteConsoleA@20:dword
;истинные имена используемых функций
ExitProcess       equ   __imp__ExitProcess@4
GetStdHandle      equ   __imp__GetStdHandle@4
WriteConsole      equ   __imp__WriteConsoleA@20
;присваивания для облегчения читаемости кода

;======Файл def32.inc======
STD_OUTPUT_HANDLE       equ   -11
;включаемый файл с определениями констант и типов
;для программ под win32 из winuser.h
```

Для компиляции потребуется файл **kernel32.lib**, а также файл **mspdb50.dll**.

3. Создание exe-файла на ассемблере MASM:

Трансляция: `ml/c/coff/Cp имя_файла.asm`

Компоновка: `link имя_файла.obj /subsystem:console`

8.6.5. Отладка win32-приложений в Microsoft Visual Studio

32-битные приложения на ассемблере **MASM** можно компилировать и отлаживать в **Microsoft Visual Studio**:

1. Создайте новый консольный проект.

2. Добавьте в него уже существующий **asm**-файл.

3. Заголовочные **inc**-файлы должны находиться в одной папке с **asm**-файлом.

8.7. Процедуры

Структура процедуры на ассемблере **TASM / MASM** имеет следующий вид:

```
имя_процедуры     proc
;точка входа в процедуру
;имя_процедуры считается меткой
      ;. . .
      ret
      ;возврат в вызывающую процедуру
имя_процедуры     endp
```

Если процедура должна возвращать результат, он помещается в регистр **eax**:

```
имя_процедуры     proc
      ;. . .
      mov         eax,результат
      ret
имя_процедуры     endp
```

Имеется 2 способа передачи параметров процедуре: передача параметров процедуре через стек (см. раздел 8.8) и передача параметров процедуре через регистры (см. нижеследующий пример).

```
;консольное win32-приложение, демонстрирует
;передачу параметров процедуре через регистры
;. . . (пропущенные строки кода из раздела 8.6.4)

      push        STD_OUTPUT_HANDLE
      call        GetStdHandle
      ;вызов системной функции
      ;«возврат идентификатора stdout в eax»
```

```
        mov         ebx,len
        ;в ebx длина строки для вывода
        lea         esi,s2
        ;поместить адрес строки s2 в esi
        call        output_string
        ;вызов нашей процедуры
        push        0
        ;код выхода
        call        ExitProcess
        ;вызов системной функции
        ;«завершение программы»

;процедура вывода строки на экран
;в eax - идентификатор буфера вывода
;в esi - адрес строки
;в ebx - длина строки
output_string proc
        push        0
        push        offset mes
        ;адрес переменной, в которую будет занесено
        ;число байт, действительно выведенных
        ;на консоль
        push  ebx
        ;сколько байт надо вывести на консоль
        push  esi
        ;адрес строки для вывода на консоль
        push  eax
        ;идентификатор буфера вывода
        call  WriteConsole
        ;вызов системной функции
        ;«вывод строки на консоль»
        ret
output_string endp

end _start
```

8.8. Интерфейс с C/C++

В файле **str.asm** содержится код процедуры, копирующей содержимое одной строки в другую (см. разделы 8.6, 8.7). Имя процедуры соответствует требованиям соглашения **stdcall**, напомним, что «0» в конце имени означает, что процедуре параметры не передаются.

```
;======Файл str.asm======
.386
.model   flat
.data
      s1    db    "Hi, people!!!",0Dh,0Ah,0
      len   equ   $-s1
      s2    db    "123456789123456789",0Dh,0Ah,0
.code
_str2str@0 proc
      cld
      mov         ecx,len
      mov         esi,offset s1
      mov         edi,offset s2
      rep         movsb
      mov         eax,offset s2
      ;результат работы процедуры помещается
      ;в eax (в eax адрес строки s2)
      ret
_str2str@0 endp
end
```

В заголовочном файле **str.h** содержится объявление функции, написанной на ассемблере. При этом ключевое слово **extern** означает, что функция является внешней; оператор **"C"** необходим для компилятора C++, директива **__stdcall** устанавливает соглашение об именовании.

```
//=====Файл str.h=====
extern "C" char* __stdcall str2str();
```

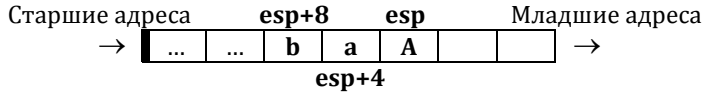
В файле **main.cpp** вызывается функция, реализованная на ассемблере.

```
//=====Файл main.cpp=====
#include<stdio.h>
#include "str.h"
void main(){
 printf("%s",str2str());
}
```

Реализуем теперь функцию **intMint(int a, int b),** которая имеет два целочисленных параметра и в качестве результата возвращает их разность (a – b).

Первым помещается в стек параметр **b**, вторым – параметр **a**. **A** – адрес команды, следующей после вызова функции, – точка возврата.

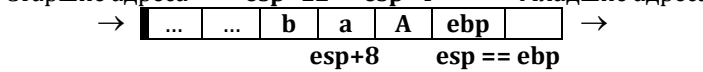
1. Вызов функции **intMint(int a, int b)**:



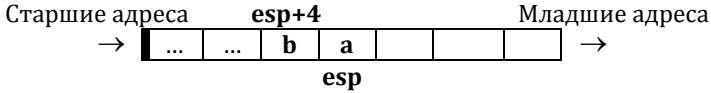
Так как указатель стека **esp** в процессе выполнения функции может изменяться, его значение сохраняется в регистре **ebp** (предварительно сохранив **ebp** в стеке), чтобы иметь возможность работать с параметрами функции:

```
push        ebp
mov         ebp,esp
```

2. Содержимое стека после **push ebp** и **mov ebp,esp**:

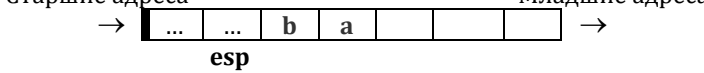


3. Содержимое стека после завершения функции:



4. Надо сместить **esp** на 8 байт в сторону старших адресов:

либо **ret 8** – в вызываемой процедуре (**stdcall**)
либо **add esp 8** – в вызывающей процедуре (**cdecl**)




```
;======Файл functions.asm======
.386
.model      flat
.data
.code
_intMint@8 proc
;8 байт приходится на два целочисленных параметра
```

```
      push        ebp
      mov         ebp,esp
      ;если esp изменится, через ebp можно
      ;добраться до параметров
      mov         eax,[ebp+8]
      ;eax = a
      sub         eax,[ebp+12]
      ;eax = eax - b (eax == (a - b))
      ;теперь в eax результат работы процедуры
      pop         ebp
      ;вернули ebp в исходное состояние
      ret         8
      ;очистка стека
_intMint@8 endp

//=====Файл functions.h=====
extern "C" int __stdcall intMint(int a, int b);

//=====Файл main.cpp=====
#include<stdio.h>
#include "functions.h"
int a = 10, b = 2;
void main(){
      printf("%i\n",intMint(a,b));
}
```

В заключении реализуем аналогичную функцию, также вычисляющую разность, но уже вещественных чисел. Основное отличие заключается в том, что результат вычислений не входит в регистр eax (вещественное число занимает 8 байт), а значит функция должна вернуть указатель на вещественное число.

```
;======Файл functions.asm======
.386
.model     flat
.data
      rezult      dq    ?
.code
_doubleMdouble@16 proc
;16 байт приходится на два вещественных параметра
```

```
      push        ebp
      mov         ebp,esp
      ;если esp изменится, через ebp можно
      ;добраться до параметров
      finit
      //инициализация сопроцессора
      fld         qword ptr [ebp+8]
      ;st(0) = a
      fsub        qword ptr [ebp+16]
      ;st(0) = st(0) – b (st(0) == (a - b))
      fstp        rezult
      ;rezult = st(0) и вытолкнуть
      ;(rezult == (a - b))
      lea         eax,rezult
      ;поместить адрес rezult в eax
      ;теперь в eax результат работы процедуры
      pop         ebp
      ;вернули ebp в исходное состояние
      ret         16
      ;очистка стека
_doubleMdouble@16 endp


//=====Файл functions.h=====
extern "C" double* __stdcall doubleMdouble
                        (double a, double b);
//функция возвращает указатель
//на вещественное число


//=====Файл main.cpp=====
#include<stdio.h>
#include "functions.h"
double x = 10.5, y = 2.5;
void main(){
      printf("%lf\n",*doubleMdouble(x,y));
      //* - операция разадресации
}
```

9. СВОБОДНЫЙ КОМПИЛЯТОР GAS И СИНТАКСИС AT&T

Программисты на ассемблере в **DOS** и **Windows** используют синтаксис **Intel**, но в **Unix**-подобных системах принято использовать синтаксис **AT&T**. Именно синтаксисом **AT&T** написаны ассемблерные части ядра **Linux**, в синтаксисе **AT&T** компилятор **GCC** выводит ассемблерные листинги [5].

9.1. Подготовка рабочего места

Под ОС **Linux** компилятор **GCC**, как правило, уже установлен или легко может быть получен из репозитория **Linux**. В системах использующих репозиторий **Debian** этого можно добиться командой: **`bash$ sudo apt-get install gcc`**

Если Вы работаете на компьютере под управлением **Windows**, то необходимо вначале установить набор свободных утилит разработчика. Например, таким удачным набором утилит обладает инструментарий **MinGW** (**http://www.mingw.org**), который не только содержит компилятор **GCC** для языка C, но также **GAS** для Ассемблера. Рекомендуется сделать полную установку, в частности, установить модули **MSYS**.

Если установка производилась в папку **C:\MinGW**, то после установки настоятельно рекомендуется добавить в переменную окружения **PATH** следующие пути: **C:\MinGW\bin\; C:\MinGW\msys\1.0\bin\;**

В качестве альтернативы **MinGW** можно использовать пакет **CygWin**, который также имеет подходящие наборы утилит.

В качестве оболочки будем использовать многоплатформенную среду разработки **NetBeans** (**http://www.netbeans.org**). Необходимо в настройках **NetBeans** активировать **plugin** поддержки разработки на C/C++, а если его нет, то установить его (рис. 1). Примеры и снимки экрана были подготовлены на ОС **Windows 7 x64** с **NetBeans 6.9.1**.

Затем необходимо создать набор компиляторов **MinGW**, указав пути до соответствующих утилит (рис. 2).

Аналогично устанавливается и настраивается пакет **CygWin**.

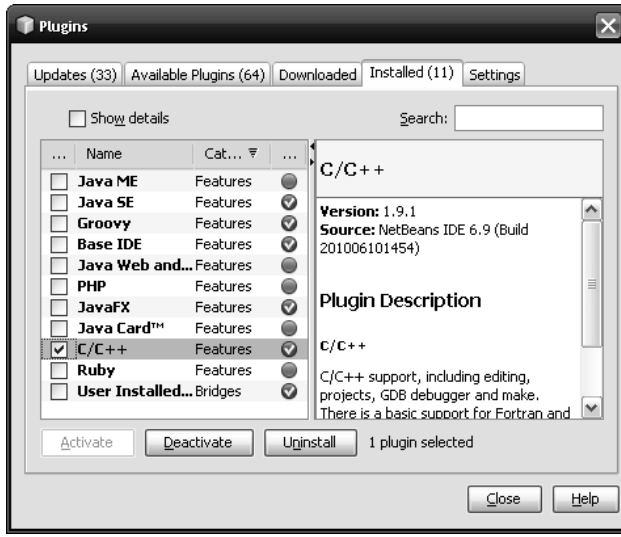


Рис. 1. *Активация плагина поддержки C/C++*

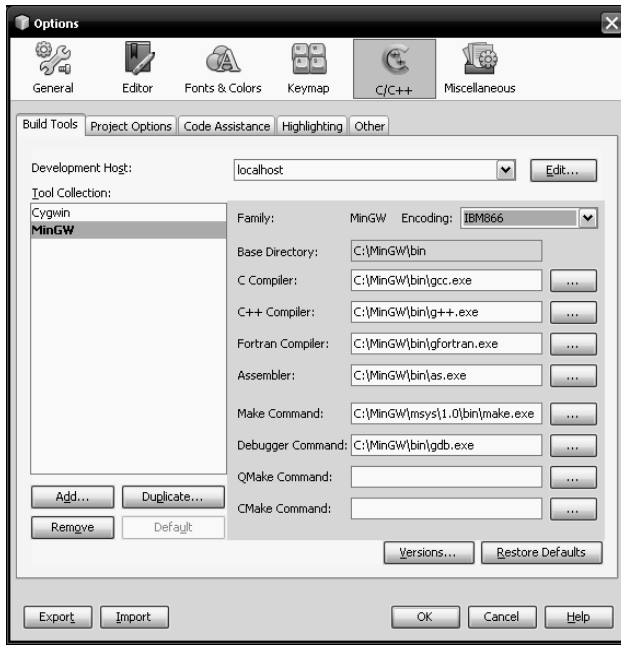


Рис. 2. *Настройки набора* ***MinGW***

9.2. Отличия синтаксиса AT&T от синтаксиса Intel

Перечислим главные отличия синтаксиса **AT&T** от синтаксиса **Intel** [6].

Именование регистров. Имена регистров имеют префикс **%**. Поэтому, если будет использован регистр **eax**, то он должен обозначаться как **%eax**.

Порядок следования операндов источника и приемника. В любой инструкции сначала указан операнд источника, за ним следует операнд приемника. В этом состоит отличие от синтаксиса фирмы Intel, где операнд источника следует после операнда приемника.

```
mov   %eax, %ebx
;пересылает содержимое регистра eax в регистр ebx
```

Размер операнда*.* Инструкции содержат суффиксы **b**, **w** или l, если операнд имеет размер 1 байт, 2 байта (слово) или 4 байта (длинное слово). Использование суффиксов не является обязательным – компилятор GCC ставит подходящий суффикс путем чтения операндов. Однако явное указание суффиксов улучшает читаемость кода и исключает возможность неправильных решений компиляторов.

```
movb  %al, %bl
;пересылка байта из al в bl
movw  %ax, %bx
;пересылка слова из ax в bx
movl  %eax, %ebx
;пересылка длинного слова из eax в ebx
```

Непосредственный операнд. Непосредственный операнд указывается с помощью символа $.

```
movl $0xffff, %eax
;пересылает значение 0xffff в регистр eax
```

Косвенная ссылка в память. Для задания косвенных ссылок в память используются скобки ( ).

```
movb (%esi), %al
;пересылает байт памяти по адресу,
;указанному в регистре esi, в регистр al
```

Встроенный ассемблер. Если нет необходимости делать большой кусок работы на ассемблере, то лучше сделать вставку ассемблерного кода в программу на языке более высокого уровня [1]. Компилятор **GCC** позволяет вставлять ассемблерный код прямо в текст программы на языке С. Используемые ассемблерные инструкции зависят от архитектуры.

Для вставки ассемблерного кода используется инструкция **asm**:

```
asm (ассемблерный шаблон
: выходные операнды //необязательные
: входные операнды //необязательные
: список используемых регистров //необязательный
);
```

Входные и выходные параметры задаются в следующем виде:

```
asm("код"
:"=признак_ограничения" //результат
:"признак_ограничения"  //исходные данные
);
```

Если что-нибудь из этих двух вариантов нас не интересует, то соответствующая секция просто пропускается. Например:

```
asm ("fsin"
: "=t" (answer)
: "0" (angle)
);
```

1) **=** означает, что операнд используется только для записи, предыдущее значение заменяется на выходные данные.

2) **t** означает, что ответ передается через вершину стека сопроцессора;

3) **answer** – выходной операнд, переменная языка С;

4) **0** означает, что используется то же правило для передачи (значение передается через тот же регистр), что и у операнда под номером **0**; то есть, такой же, как и у **answer**;

5) **angle** – входной операнд, переменная языка С.

Для процессоров семейства x86 представленный ассемблерный код соответствует выражению на языке С:

```
answer = sin(angle);
```

В следующей таблице указаны коды регистров, используемые для входных и выходных операндов.

Символы регистров для архитектуры x86

Коды регистра	Регистры, которые GCC может использовать
r	Общие регистры (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
q	Общие регистры для данных (EAX, EBX, ECX, EDX)
f	Регистры сопроцессора
t	Вершина стека сопроцессора
u	Второй от вершины регистр в стеке сопроцессора
a	Регистр EAX
b	Регистр EBX
c	Регистр ECX
d	Регистр EDX
x	Регистр SSE
y	Мультемедийный регистр MMX
A	8-байтовые значения регистров, формируемые из EAX и EDX
D	Указатель места назначения (адресата) для строковых операций (EDI)
S	Указатель источника для строковых операций (ESI)

В следующем примере программы на языке C для вставки фрагмента ассемблерного кода используется конструкция **asm**. В этом примере значение переменной, объявленной в коде на языке C, загружается в регистр, сдвигается на один бит в право для его деления на 2, и результат затем записывается в другую переменную C.

```
#include <stdio.h>
int main(int argc, char* argv[])
{
   int a = 40;
   int b;

   asm ("movl %1, %%eax; \
         shr %%eax; \
         movl %%eax, %0;"

   : "=r" (b)
     //Ответ возвращается через общий регистр
     //в переменную b
   : "r" (a)
     //Значение, подаваемое на вход, берем из
```

```
      //переменной a;
      //передача через общий регистр
   : "%eax"
      //Указываем, какой регистр является
      //затираемым («рабочим»)
   );
   printf ("a=%d b=%d\n",a,b);
   return 0
};
```

Шаблон ассемблерного кода состоит из одного или более операторов на языке ассемблера и представляет собой действующий код, непосредственно подставляемый компилятором в объектный модуль. Инструкции могут адресоваться: к непосредственно разрешаемым значениям (константам); к содержимому регистров; к ячейкам памяти.

Перечислим синтаксические правила, которые применяются к адресуемым величинам:

1. Перед именем регистра ставятся два знака процента **%%**. Например, **%%eax** или **%%esi**. Имена регистров процессоров **Intel** начинаются со знака **%**, а применение второго знака **%** требуют правила конструкции **asm**, так что в этом случае их должно быть два.

2. Можно определить до 10 операндов, обозначенных **%0**, **%1**,..., **%9** во входной и выходной секции. Если в какой-то секции операндов нет, – они просто опускаются. Каждый из операндов указывается порядковым индексом, соответствующим его положению в строке объявления. Первый выходной операнд адресуется в шаблоне кода сочетанием **%0**, второй выходной операнд – **%1** и т.д. Индексы входных операндов продолжают эту нумерацию. Например, при наличии двух выходных операндов первый входной операнд в шаблоне будет адресован как **%2**.

3. Расположение в памяти может быть указано его адресом, записанным в регистре процессора. В этом случае имя регистра заключается в круглые скобки. Например, нужно загрузить в регистр **%%al** один байт из ячейки памяти, аадресом которой является число, записанное в регистре **%%esi**. Инструкцию для этого действия можно записать так:

```
movb (%%esi),%%al;
```

фактически сейчас мы повторяем, общие соглашения синтаксиса **AT&T**, указанные ранее.

4. Непосредственное значение (константа) обозначается знаком доллара **$**, сразу за которым следует само число. Например: **$B6** или **$0xF12A**.

Весь фрагмент ассемблерного кода является одной символьной строкой. Поэтому каждая строка вставляемого кода должна заканчиваться терминатором – символом перехода на следующую строку. В качестве терминатора можно использовать наклонную черту «\», или «**\n**» после «**;**». Для улучшения читаемости кода допускается вставка символов табуляции.

Сформулируем основные правила для входных и выходных операндов:

1. Выражение языка С, указывающее на адрес данных программы, должно быть заключено в круглые скобки.

2. Если перед указываемыми расположениями ставится признак ограничения **"r"**, то он означает, что данные должны быть загружены из входных переменных в свободные регистры до выполнения ассемблерного кода. Применяется во входных операндах. В выходных операндах - соответственно **"=r"**, он указывает, что выходные переменные должны быть записаны после выполнения ассемблерного кода.

3. Признак ограничения может указывать назначение для переменных С определенных регистров:

"a"	**%%eax**
"b"	**%%ebx**
"c"	**%%ecx**
"d"	**%%edx**
"s"	**%%esi**
"d"	**%%edi**

4. Переменные могут быть адресованы из ассемблерного кода своим адресом памяти без их загрузки в регистры процессора, для этого ставится признак ограничения **"m"**.

5. Одна и та же переменная может использоваться и для ввода входного значения, и для вывода результата. При этом для этой переменной в выходном операнде применяется признак ограничения **"=a"**, а во входном операнде в признаке ограничения ставится его порядковый индекс. В следующем примере переменная из кода на языке С **counter** используется как для ввода, так и для вывода:

```
asm ("incw %0;"
:"=a" (counter)
:"0" (counter)
);
```

Список регистров, задействованных в ассемблерном коде, – это просто список имен регистров. Между именами ставится запятая. Например:

```
asm( . . . : "%eax" , "%esi");
```

Информация о регистрах передается компилятору. Благодаря этому исключаются ошибки, связанные с использованием этих регистров.

Рассмотрим пример: необходимо сложить две переменные **x** и **y** и поместить результат в переменную **z**.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
   int x = 2, y = 7, z = 1;
   asm ("movl %1, %%eax; \
         addl %2, %%eax; \
         movl %%eax, %0;"
   : "=r" (z)
   : "r" (x), "r" (y)
   : "%eax"
   );
   printf("x=%d y=%d  %d\n ",x,y,z);
   return 0;
};
```

Возможен вызов стандартных функций C из ассемблерной вставки согласно стандартным соглашениям языка C, что демонстрирует следующий пример.

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char** argv) {
    char st[]="Hello, GAS!!!\n";
    asm("push %0;\
         call _printf;\
         add $4, %%esp"
    :
    : "r" (st)
    );
}
```

В разных операционных системах обращение к имени стандартной функции может различаться наличием или отсутствием подчеркивания.

Во многих случаях можно разобраться, если проанализировать ассемблерный код, полученный из программы на языке C. Для того, что бы получить ассемблерный код из программы на C, необходимо запустить компилятор **GCC** с опцией **–S** и указанием исходного файла на C.

При создании данного раздела были использованы материалы из источников [1; 5–8].

10. ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

10.1. Лабораторная работа № 1

Задание выполняется на C/C++ с использованием ассемблерных вставок.

Необходимо выполнить соответствующие преобразования над строкой или строками. Как правило, в каждом задании по одной или двум входным строкам надо получить выходную строку, удовлетворяющую определенным условиям.

Программа должна иметь «дружелюбный» интерфейс (например, предлагать выполнить повторное тестирование).

Ввод/вывод с консоли выполните с помощью функций **printf** и **scanf**, вызов которых тоже должен происходить внутри ассемблерных вставок. Ввод данных из файла не требуется, хотя приветствуется.

1. Удалить из исходной строки все пробельные символы, пробегая строку справа налево.

2. Выделить из строки подстроку указанной длины, начиная с заданной позиции.

3. Определить номер позиции, с которой начинается первое слева вхождение указанной строки символов.

4. Создать строку, полученную копированием исходной строки заданное число раз.

5. Указать те символы, которые есть и в первой и во второй строке.

6. Указать те символы, которые есть в первой строке, но которых нет во второй.

7. Выделить те символы, которые встречаются в исходной строке указанное количество раз.

8. Найти в строке символы с минимальным и максимальным кодом.

9. Из исходной строки удалить все слова, встречающиеся более одного раза.

10. Из исходной строки удалить все вхождения заданной последовательности символов.

11. Найти всех соседей заданного символа в исходной строке. Первый и последний символы являются соседями.

12. Посчитать количество символов, у которых равные соседи в исходной строке. Первый и последний символы являются соседями.

13. Удалить из исходной строки все символы, встречающиеся в другой.

14. Получить строку, обратную к заданной, то есть все символы которой записаны в обратном порядке.

15. Удалить все пробелы в конце строки.

16. Удалить из строки первое вхождение заданной последовательности символов.

17. Переставить в обратном порядке все символы между первым и последним вхождением заданного символа в исходной строке, если этот символ встречается не менее двух раз.

18. Вставить после заданного символа все символы предшествующие ему. Оставшуюся часть строки оставить без изменения.

19. Вставить после заданного символа все символы предшествующие ему в обратном порядке. Оставшуюся часть строки оставить без изменения.

20. Образовать строку, повторив фрагмент исходной строки с заданной позиции данной длины требуемое число раз.

21. Образовать строку из исходной, повторив *i*-ый элемент *i* раз.

22. В строке указанное количество символов, начиная с заданной позиции, переписать в конец строки.

23. Удалить из цепочки все вхождения двух идущих подряд заглавных латинских букв (три и более оставить на месте).

24. Удалить из первой строки все символы, которых нет во второй.

25. Удалить из строки подстроку, заданную другой строкой.

26. Найти максимальное слово-палиндром в строке.

27. Найти самое длинное слово в строке.

28. Найти в строке максимальную последовательность букв.

29. Пусть каждый символ строки – цифра 0 … 9. На первом месте может быть знак. Получить строку, являющуюся результатом сложения первой и второй строки.

30. В строке заменить восемь идущих подряд пробелов символом табуляции.

31. Посчитать количество слов в строке.

32. Перевести целое число в строку символов.

33. Перевести строку символов в целое число.

34. Определить длину максимальной подстроки в первой строке, которая не содержит символов из второй строки.

10.2. Лабораторная работа № 2

Реализуйте вычисление заданного выражения как ассемблерную вставку в программе на языке C/C++.

Ввод/вывод с консоли выполните с помощью функций **printf** и **scanf**, вызов которых тоже должен происходить внутри ассемблерных вставок. Ввод данных из файла не требуется, хотя приветствуется.

Все величины являются целочисленными. A, B, C – массивы. Если у массива указан один индекс, значит он одномерный, если два – двумерный. Индекс $i = 1,...,3$.

Обработайте ситуации возможного переполнения.

1. $\Sigma (A_i \cdot X^2) + Y \cdot \Sigma B_i$

2. $X \cdot \Sigma A_i + X^2 \cdot \Sigma B_i$

3. $X \cdot \Sigma (A_i + Y \cdot \Sigma B_i) + Y \cdot \Sigma C_i$

4. $X \cdot \Sigma (A_i \cdot B_i) + Y \cdot \Sigma C_i$

5. $X \cdot \Sigma (|A_i| \cdot |B_i|) - Y \cdot \Sigma |C_i|$

6. $X \cdot \Sigma |A_i| + Y \cdot \Sigma B_i$

7. $X \cdot \Sigma |A_i| + X \cdot Y \cdot \Sigma |B_i| + \Sigma C_i$

8. $|\Sigma A_i + \Sigma B_i| \cdot X \cdot Y$

9. $|\Sigma (A_i + B_i)| \cdot X + X^2$

10. $|X \cdot \Sigma A_i + Y \cdot \Sigma B_i|$

11. $X \cdot \Sigma A_{ij} + Y \cdot \Sigma B_i$

12. $X \cdot \Sigma A_i + X^2 \cdot \Sigma B_{ij}$

13. $X \cdot \Sigma (A_{ij} + Y \cdot \Sigma B_i) + Y \cdot \Sigma C_i$

14. $X \cdot \Sigma (A_i \cdot B_{ij}) + Y \cdot \Sigma C_i$

15. $X \cdot \Sigma (|A_i| \cdot |B_{ij}|) - \Sigma (|C_i|) \cdot Y$

16. $\Sigma (|A_{ij}|) \cdot X + Y \cdot \Sigma (B_i)$

17. $X \cdot \Sigma |A_i| + X \cdot Y \cdot \Sigma |B_i| + \Sigma C_{ij}$

18. $(|\Sigma A_i + \Sigma B_{ij}|) \cdot X \cdot Y$

19. $|\Sigma (A_i + B_{ij})| \cdot X + X^2$

20. $|X \cdot \Sigma A_i + Y \cdot \Sigma B_{ij}|$

10.3. Лабораторная работа № 3

Задание выполняется на C/C++ с использованием ассемблерных вставок.

Программа должна выводить на консоль таблицу значений функции на заданном отрезке (в случае, если в какой-либо точке значение функции не существует, в соответствующей ячейке таблицы печатается прочерк).

Значения параметров *a*, *b*, *c*, отрезок [X_1, X_2] и шаг *dX* имеют вещественный тип и должны запрашиваться с клавиатуры. Ввод/вывод с консоли выполните с помощью функций **printf** и **scanf**, вызов которых тоже должен происходить внутри ассемблерных вставок. Ввод данных из файла не требуется, хотя приветствуется.

1.
$$F=\begin{cases} ax^2+bx+c, & x<0, b\neq 0 \\ \dfrac{x+a}{x-c}, & x>0, b=0 \\ \dfrac{x}{c}, & \text{иначе} \end{cases}$$

2.
$$F=\begin{cases} \dfrac{1}{ax}-b, & x+5<0, c=0 \\ \dfrac{x-a}{x}, & x+5>0, c\neq 0 \\ \dfrac{10x}{c-4}, & \text{иначе} \end{cases}$$

3.
$$F=\begin{cases} ax^2+bx+c, & a<0, c\neq 0 \\ \dfrac{-a}{x-c}, & a>0, c=0 \\ a(x+c), & \text{иначе} \end{cases}$$

4. $$F=\begin{cases} -ax+b, & c<0, x\neq 0 \\ \dfrac{x-a}{-c}, & c>0, x=0 \\ \dfrac{bx}{c-a}, & \text{иначе} \end{cases}$$

5. $$F=\begin{cases} a-\dfrac{x}{10+b}, & x<0, b\neq 0 \\ \dfrac{x-a}{x-c}, & x>0, b=0 \\ 3x+\dfrac{2}{c}, & \text{иначе} \end{cases}$$

6. $$F=\begin{cases} ax^2+b^2x, & c<0, b\neq 0 \\ \dfrac{x+a}{x+c}, & c>0, b=0 \\ \dfrac{x}{c}, & \text{иначе} \end{cases}$$

7. $$F=\begin{cases} -ax^2-b, & x<5, c\neq 0 \\ \dfrac{x-a}{x}, & x>5, c=0 \\ -\dfrac{x}{c}, & \text{иначе} \end{cases}$$

8. $$F=\begin{cases} -ax^2, & c<0, a\neq 0 \\ \dfrac{x-a}{xc}, & c>0, a=0 \\ \dfrac{x}{c}, & \text{иначе} \end{cases}$$

9. $$F=\begin{cases} ax^2+b^2x, & a<0, x\neq 0 \\ x-\dfrac{a}{x-c}, & a>0, x=0 \\ 1+\dfrac{x}{c}, & \text{иначе} \end{cases}$$

10. $$F=\begin{cases} ax^2-bx+c, & x<3, b\neq 0 \\ \dfrac{x-a}{x-c}, & x>3, b=0 \\ \dfrac{x}{c}, & \text{иначе} \end{cases}$$

11. $$F=\begin{cases} ax^2+bc, & x<1, b\neq 0 \\ \dfrac{x-a}{x-c}, & x>1.5, b=0 \\ \dfrac{x}{c}, & \text{иначе} \end{cases}$$

12. $$F=\begin{cases} ax^3+b^2+c, & x<0.6, b+c\neq 0 \\ \dfrac{x-a}{x-c}, & x>0.6, b+c=0 \\ \dfrac{x}{c}+\dfrac{x}{a}, & \text{иначе} \end{cases}$$

13. $$F=\begin{cases} ax^2+b, & x-1<0, b-x\neq 0 \\ \dfrac{x-a}{x}, & x-2>0, b+x=0 \\ \dfrac{x}{c}, & \text{иначе} \end{cases}$$

14. $$F=\begin{cases} -ax^3-b, & x+c<0, a\neq 0 \\ \dfrac{x-a}{x-c}, & x+c>0, a=0 \\ \dfrac{x}{c}+\dfrac{c}{x}, & \text{иначе} \end{cases}$$

15. $$F=\begin{cases} -ax^2+b, & x<0, b\neq 0 \\ \dfrac{x}{x-c}+5.5, & x>0, b=0 \\ \dfrac{-x}{c}, & \text{иначе} \end{cases}$$

16. $$F=\begin{cases} a(x+c)^2-b, & x=0, b\neq 0 \\ \dfrac{x-a}{-c}, & x>0, b=0 \\ a+\dfrac{x}{c}, & \text{иначе} \end{cases}$$

17. $$F=\begin{cases} ax^2-cx+b, & x+10<0, b\neq 0 \\ \dfrac{x-a}{x-c}, & x+10>0, b=0 \\ \dfrac{-x}{a-c}, & \text{иначе} \end{cases}$$

18. $$F=\begin{cases} ax^3+bx^2, & x<0, b\neq 0 \\ \dfrac{x-a}{x-c}, & x>0, b=0 \\ \dfrac{x+5}{c(x-10)}, & \text{иначе} \end{cases}$$

19. $$F=\begin{cases} a(x+7)^2-b, & x<5, b\neq 0 \\ \dfrac{x-ac}{ax}, & x>5, b=0 \\ \dfrac{x}{c}, & \text{иначе} \end{cases}$$

20. $$F=\begin{cases} -\dfrac{2x-c}{cx-a}, & x<0, b\neq 0 \\ \dfrac{x-a}{x-c}, & x>0, b=0 \\ -\dfrac{x}{c}+\dfrac{-c}{2x}, & \text{иначе} \end{cases}$$

СПИСОК ЛИТЕРАТУРЫ

1. *Гриффитс А.* GCC. Настольная книга пользователей, программистов и системных администраторов. – К.: ООО «ТИД «ДС», 2004. – 624 с.
2. *Зубков С.В.* Assembler. Язык неограниченных возможностей. – М.: ДМК, 1999. – 640 с.
3. *Магда Ю.С.* Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. – 410 с.
4. *Юров В.Ю.* Assembler: учебник для вузов. – 2-е изд. – СПб.: Питер, 2008. – 637 с.
5. URL: http://ru.wikibooks.org/wiki/Ассемблер_в_Linux_для_программистов_C (дата обращения: 20.11.2010).
6. URL: http://www.linuxinsight.com/advanced_linux_programming.html (дата обращения: 20.11.2010).
7. URL: http://www.linuxcenter.ru/lib/articles/programming/gas.phtml (дата обращения: 20.11.2010).
8. URL: http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html (дата обращения: 20.11.2010).

СОДЕРЖАНИЕ

Учебное издание

Богаченко *Надежда Федоровна*
Лавров *Дмитрий Николаевич*
Ракицкий *Юрий Сергеевич*

АССЕМБЛЕР В ПРИМЕРАХ И ЗАДАЧАХ

Учебно-методическое пособие
Часть 1

Санитарно-гигиенический сертификат
№ 77.99.60.953 Д 001101.01.10 от 26.01.2010

Издаётся в авторской редакции.
Макет подготовлен в Издательстве ОмГУ

Технический редактор *М.В. Быкова*
Оформление обложки *З.Н. Образова*

Подписано в печать 03.02.2011. Формат бумаги 60х84 1/16.
Печ. л. 4,5. Усл.-печ. л. 4,1. Уч.-изд. л. 4,0. Тираж 125 экз. Заказ 43.

Издательство Омского государственного университета
644077, Омск-77, пр. Мира, 55а
Отпечатано на полиграфической базе ОмГУ
644077, Омск-77, пр. Мира, 55а