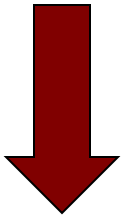
Знакомство с инструментальными средствами разработки программ на Assembler (на примере WinAsm Studio)

Язык ассемблера – это язык программирования со взаимно однозначным соответствием между его командами и командами процессора.

Язык ассемблера существует для каждого типа процессоров или целого семейства процессоров, так как команды на языке ассемблера должны соответствовать системе машинных команд и быть согласованы с архитектурой компьютера.



Язык ассемблера – язык низкого уровня.

Ассемблер – это программа, преобразовывающая исходные коды языка ассемблера в машинные команды.

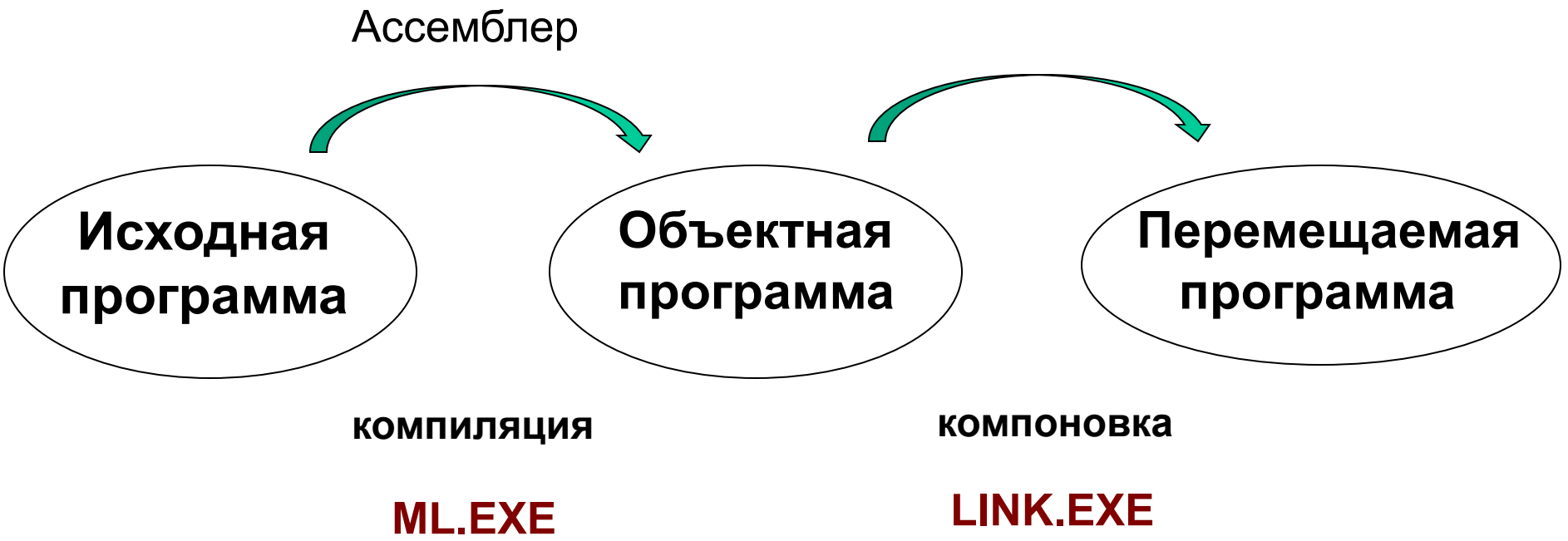
Для разработки программ на языке ассемблера для семейства процессоров Intel применяются два пакета программ:

1. Borland Turbo Assembler (TASM)
2. Microsoft Macro Assembler (MASM).

Microsoft Macro Assembler (MASM).

Технология разработки программ на языке ассемблера включает следующие этапы:

1. Постановка задачи и составление блок-схем.
2. Формирование текста программы с помощью редактора.
3. Создание исполняемого модуля, который можно запустить на выполнение под управлением операционной системы.
4. Выполнение программы.
5. Проверка результатов и выявление ошибок с помощью отладчика (выполнение программы в пошаговом режиме с контролем промежуточных результатов)


Ассемблер
Исходная программа
Объектная программа
Перемещаемая программа
компиляция
ML.EXE
компоновка
LINK.EXE

WinAsm Studio - специальное инструментальное средство для разработки программ на языке ассемблера, которое

- содержит встренный текстовый редактор, который ориентирован специально на написание программ на языке ассемблера (подсветка синтаксических конструкций языка, всплывающие подсказки и т.д.);
- имеет средства для разработки оконного интерфейса программы (создание окон и различных элементов управления (кнопки, поля ввода, выпадающие списки и пр.), меню программы);

WinAsm Studio - специальное инструментальное средство для разработки программ на языке Ассемблера, которое

- скрывает от программиста особенности компиляции и компоновки программы;
- содержит встроенные средства отладки программ.

D: \ WinAsm \ WinAsm.exe

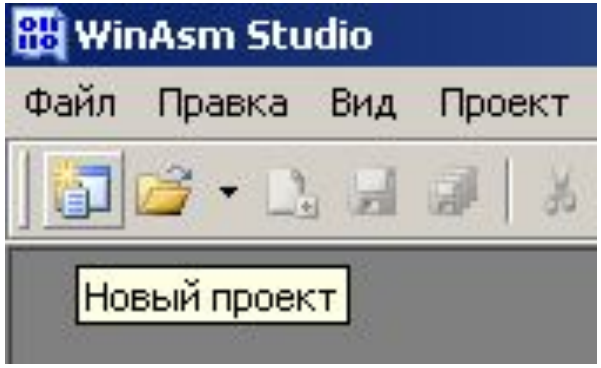
Проект - совокупность нескольких исходных программ на языке ассемблера, файлов заголовков и т.д., связанных между собой логикой алгоритма результирующей программы.

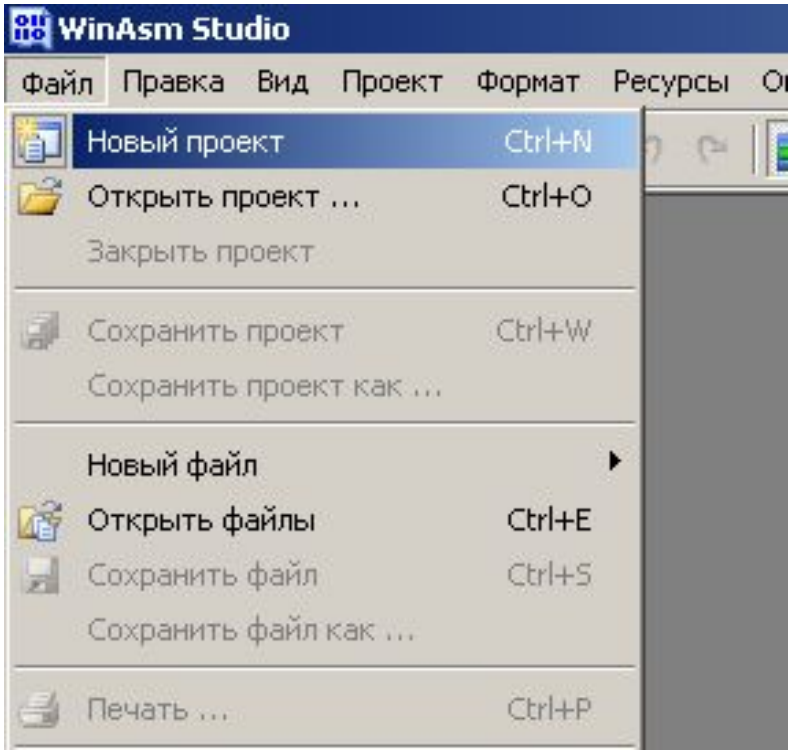
Программы на языке ассемблера имеют расширение “***.asm**”

Файл проекта, представляющий собой файл с информацией о файлах, входящих в проект, опциях используемых для запуска компилятора и компоновщика и прочей служебной информацией, имеет расширение “***.wap**”

Процедура создания проекта в WinAsm

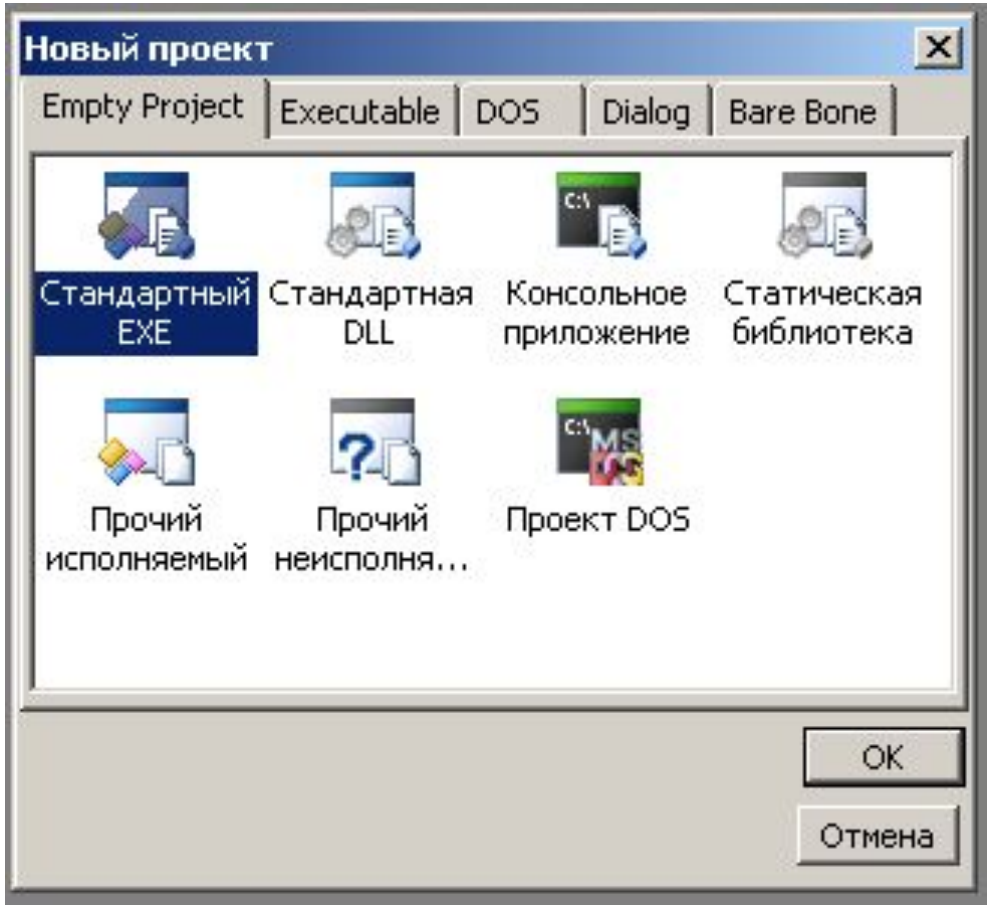
1. Создание нового проекта





Процедура создания проекта в WinAsm

2. Выбор варианта нового проекта.

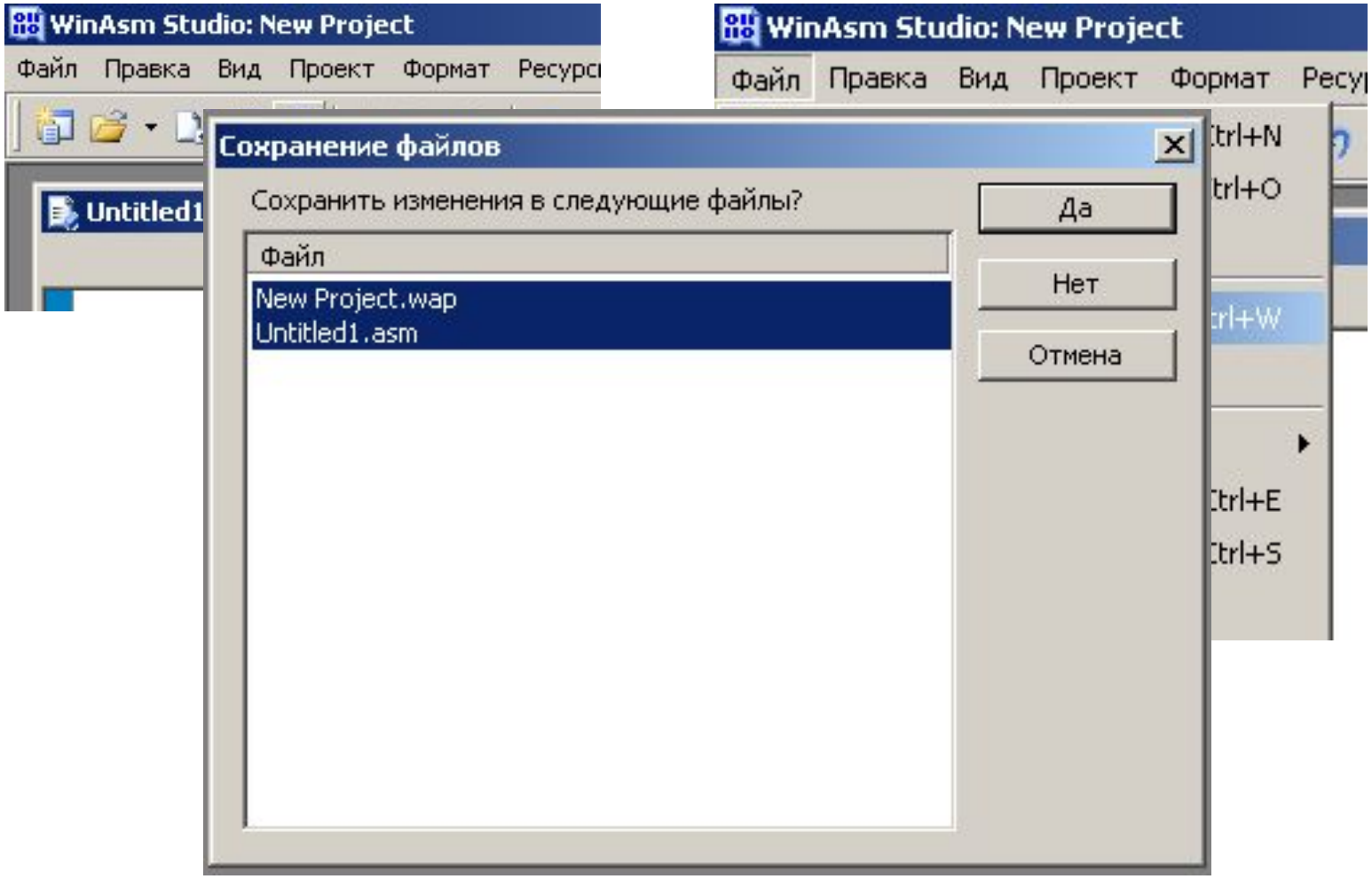


Процедура создания проекта в WinAsm

3. Окно нового проекта.

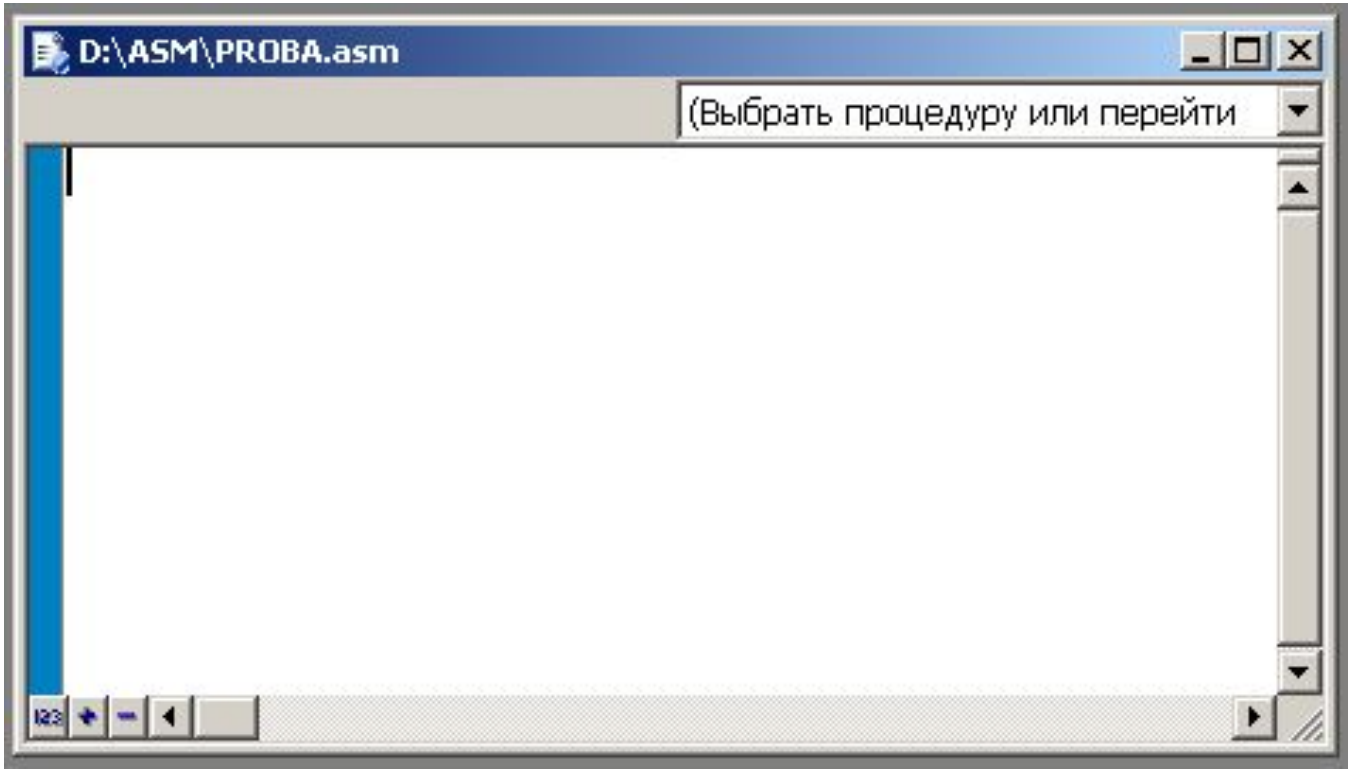
Процедура создания проекта в WinAsm

4. Сохранение нового проекта.



Процедура создания проекта в WinAsm

5. Окно редактирования текста программы.



Структура программы на языке ассемблера

Исходный текст программы разделяется на следующие секции:

- данные,
- неинициализированные данные,
- константы,
- код.

Структура программы на языке ассемблера

Секция данных (.DATA) содержит данные, доступные для чтения и записи, включается в exe-файл.

Неинициализированные данные (.DATA?) не имеют никакого содержания при запуске, не включены в exe-файл непосредственно, это только часть памяти, зарезервированной Windows. Эта секция доступна для чтения и записи.

	.DATA	.DATA?
+	Быстрая загрузка	Маленький объем исполняемого модуля
-	Большой объем исполняемого модуля	Медленная загрузка

Структура программы на языке ассемблера

Секция констант аналогична секции данных, но доступна только для чтения.

Секция кода содержит текст программы на языке ассемблера, реализующий требуемый алгоритм работы.

Шаблон программы на языке ассемблера

```
.386
.MODEL Flat, STDCALL
.DATA
<Инициализированные  данные>
.DATA?
<Неинициализированные  данные>
.CONST
<Константы>
.CODE
<Метка (точка входа в программу)>
  <Код программы>
end <Метка (точка входа в программу)>
```

Особенности шаблона программы на языке ассемблера

1) **Директивы установки типа процессора** – это директивы, которые определяют минимально возможный тип применяемого процессора.

.386

2) **Директивы выбора модели памяти**

.MODEL FLAT, STDCALL

Модели памяти, используемые в MASM

Модель	Описание
Tiny (тонкая)	Коды и данные вместе должны занимать не более 64 Кбайт.
Small (малая)	Код <= 64 Кбайт, данные <= 64 Кбайт. Один сегмент кодов, один сегмент данных.
Medium (средняя)	Данные <= 64 Кбайт, код любого размера. Много сегментов кодов, один сегмент данных.
Compact (компактная)	Код <= 64 Кбайт, данные любого размера. Один сегмент кодов, много сегментов данных.
Large (большая)	Код > 64 Кбайт, данные > 64 Кбайт. Много сегментов кодов и данных.
Huge (огромная)	Аналогична модели Large, за исключением того, что отдельные переменные, такие как массив, могут быть больше 64 Кбайт.
Flat (плоская)	Нет сегментов, 32-разрядная адресация используется как для кодов, так и для данных.

Особенности шаблона программы на языке ассемблера

1) **Директивы установки типа процессора** – это директивы, которые определяют минимально возможный тип применяемого процессора.

.386

2) **Директивы выбора модели памяти**
.MODEL FLAT, STDCALL

Ключевое слово **STDCALL** устанавливает порядок передачи параметров при вызове подпрограмм и функций справа налево.

Особенности шаблона программы на языке ассемблера

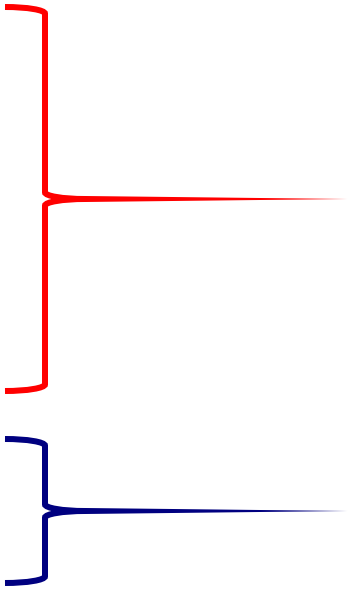
3) **Директивы, определяющие начала секций программы.**

.DATA

.DATA?

.CONST

.CODE



Секция кода

```
.code
start:
...
end start
```

Использование функций Windows API в программах на ассемблере

Windows API (Application Programming Interface)

Kernel32.dll содержит API функции, взаимодействующие с памятью и управляющие процессами.

User32.dll содержит API функции, контролирующие пользовательский интерфейс

Gdi32.dll содержит API функции, ответственные за графические операции (определение цветовой палитры создаваемых окон, элементов управления и т.д.).

Программы по мере необходимости связываются с библиотеками.

Связь осуществляется путем использования в тексте программы ссылки на одноименные файлы с расширением ***.LIB**, называемые **библиотеками импорта**.

Подключение библиотек импорта осуществляется директивой **includelib**.

Описание передаваемых в API функций параметров содержится в одноименых файлах с расширением ***.INC,** называемых **файлами для включения**.

Регистры – участки высокоскоростной памяти для хранения данных в процессоре, они непосредственно подключены к блоку управления и арифметико-логическому устройству, поэтому доступ из этих блоков к регистрам происходит значительно быстрее, чем доступ к внешней памяти.

Регистры общего назначения
– это 32-разрядные регистры EAX, EBX, ECX, EDX, в каждом из которых выделяют 16-тиразрядный регистр, состоящий из двух 8-разрядных частей, например, в EAX рассматривают регистр AX, в нем младшую часть – регистр AL и старшую часть - AH.

В общем случае функция, выполняемая тем или иным регистром, определяется командами, в которых он используется.

При этом с каждым регистром связано некоторое стандартное его назначение:

- регистр **EAX** служит для временного хранения данных (регистр аккумулятор), часто используется при выполнении операций сложения, вычитания, сравнения и других арифметических и логических операций;
- регистр **EBX** служит для хранения адреса некоторой области памяти (базовый регистр), а также используется как вычислительный регистр;

В общем случае функция, выполняемая тем или иным регистром, определяется командами, в которых он используется.

При этом с каждым регистром связано некоторое стандартное его назначение:

- регистр **ECX** иногда используется для временного хранения данных, но в основном служит счетчиком, в нем хранится число повторений одной команды или фрагмента программы;
- регистр **EDX** используется главным образом для временного хранения данных, часто служит средством пересылки данных между разными программными системами, а также используется в качестве расширителя аккумулятора для вычислений повышенной точности и при умножении и делении.

Регистры указатели – это 16-разрядные регистры EBP (указатель базы), ESI (индекс источника), EDI (индекс результата), ESP (указатель стека), EIP (указатель команд). Они содержат величину смещения, используемую при расчете адресов команд и данных.

ESI (индекс отправителя) указывает смещение адреса начала данных, которые должны быть перемещены.

EDI (индекс результата) указывает смещение адреса, куда перемещаются данные.

EIP (указатель команд) хранит смещение относительно начала сегмента кода *следующей* команды.

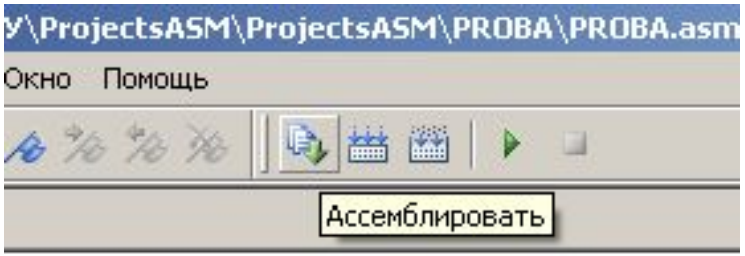
Регистры сегментов – это 16-разрядные регистры, которые позволяют организовать память в виде совокупности четырех различных сегментов.

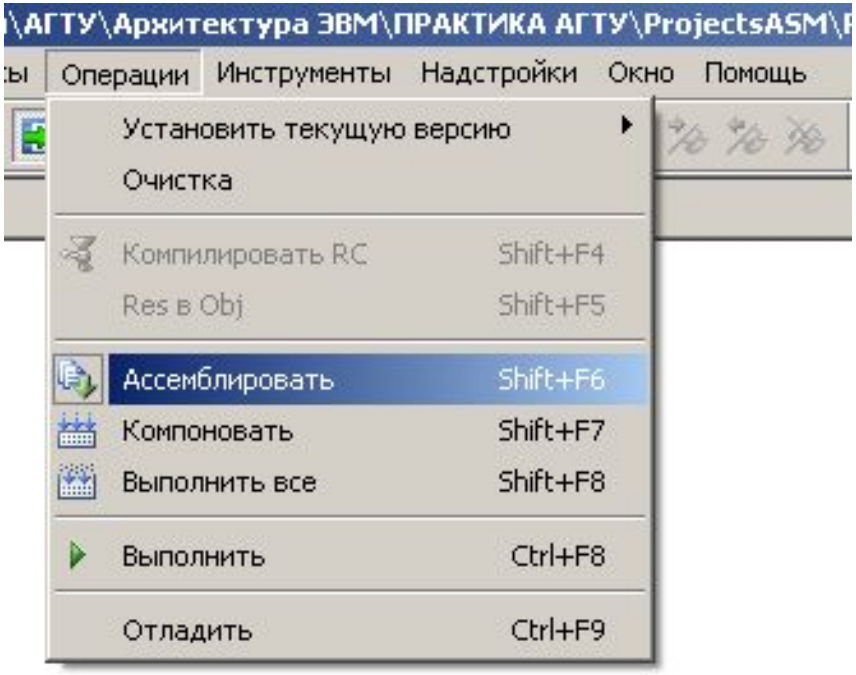
CS – регистр программного сегмента (сегмента кода) определяет адрес начала участка ОП, содержащего выполняемые процессором команды;
DS – регистр информационного сегмента (сегмента данных) определяет адрес начала участка ОП для хранения данных;
SS – регистр стекового сегмента (сегмента стека) определяет часть памяти, используемой как системный стек;
ES – регистр расширенного сегмента (дополнительного сегмента) указывает дополнительную область памяти, используемую для хранения данных.

Региистр, соде овия, уста ндой, или биты назы

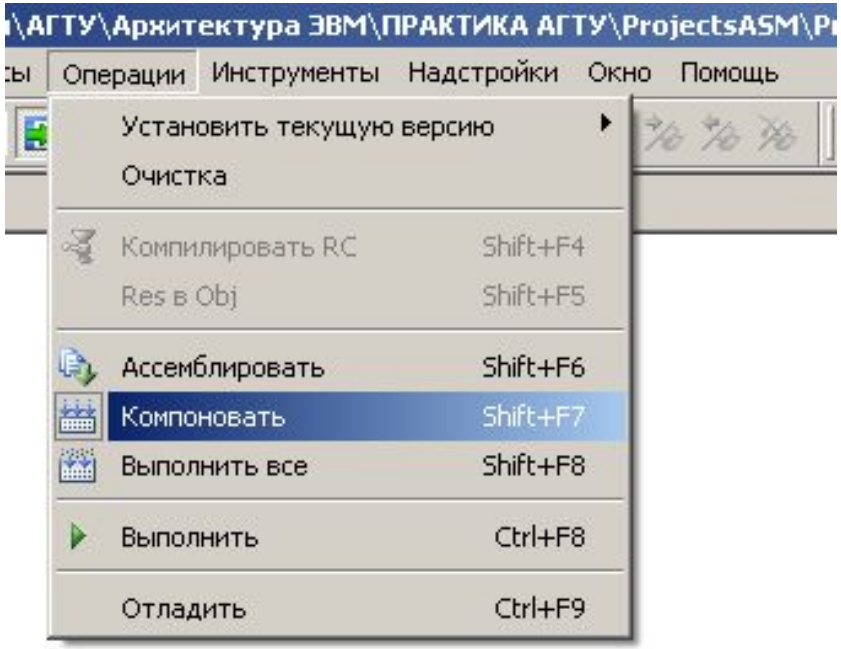
Номер бита	Условное обозначение	Назначение
0	**CF**	**Признак переноса**: этот флаг устанавливается в единицу, если имеет место перенос или заем из старшего бита результата, он полезен для проведения операций над числами длиной в несколько слов, которые сопряжены с переносами и заемами из слова в слово (слово – 2 байта).
2	**PF**	**Признак четности**: этот флаг устанавливается в единицу, если результат имеет четное число единиц
6	**ZF**	**Признак нулевого результата**: ZF=1, если результат равен нулю.
7	**SF**	**Признак знака**: SF=1, если старший бит результата равен единице. Иными словами, SF=0 для положительных чисел, и SF=1 для отрицательных чисел
11	**OF**	**Признак переполнения**: равен единице, если возникает арифметическое переполнение, то есть когда объем результата превышает размер ячейки назначения.

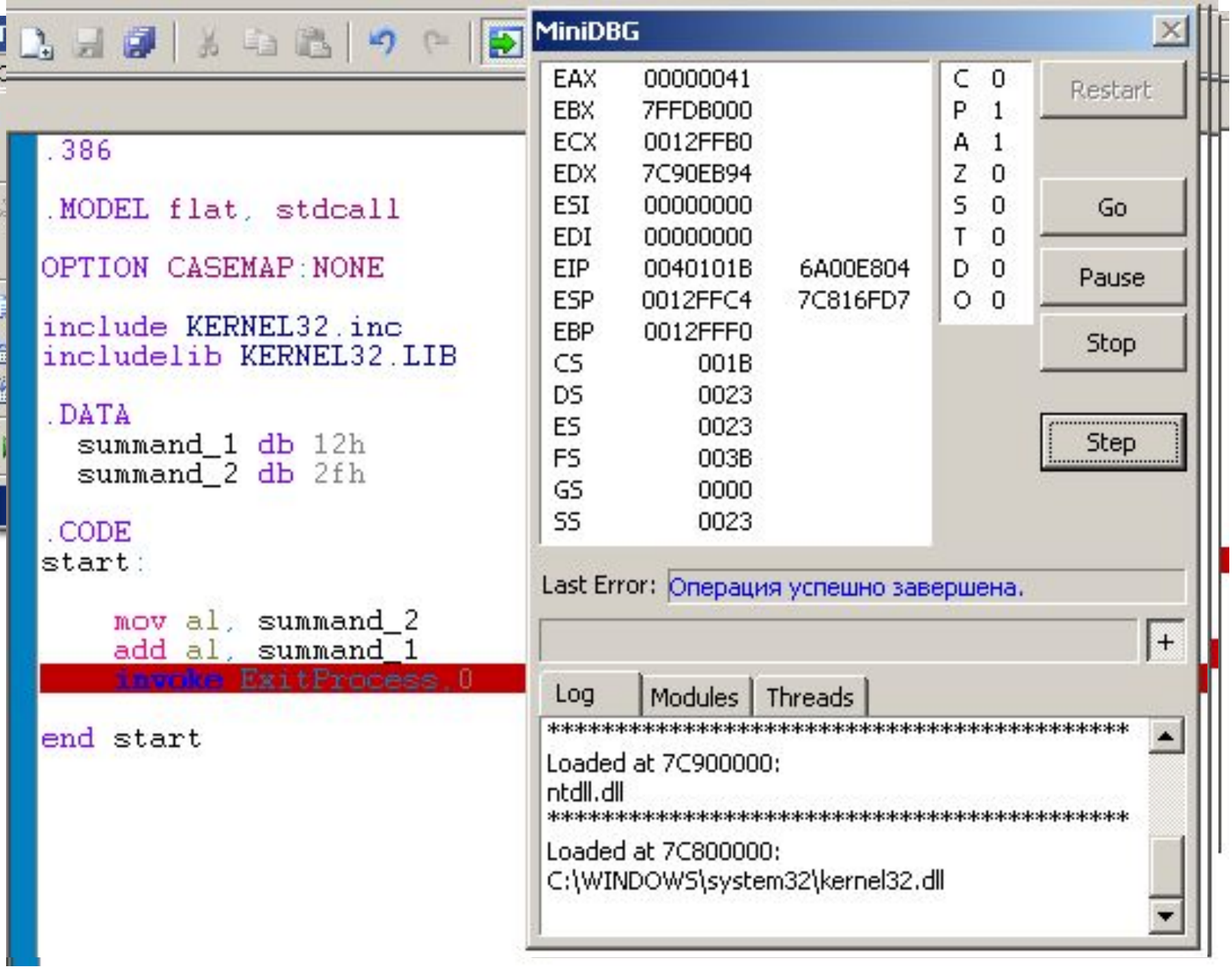
```
.386
 .MODEL flat, stdcall
 include KERNEL32.inc
 includelib KERNEL32.LIB
 .DATA
  summand_1 db 12h
  summand_2 db 2fh
 .CODE
start:
 mov al, summand_2
 add al, summand_1
 invoke ExitProcess,0
end start
```


\ProjectsASM\ProjectsASM\PROBA\PROBA.asm
Окно Помощь
Ассемблировать


\АГТУ\Архитектура ЭВМ\ПРАКТИКА АГТУ\ProjectsASM\
Операции Инструменты Надстройки Окно Помощь
Установить текущую версию
Очистка
Компилировать RC Shift+F4
Res в Obj Shift+F5
Ассемблировать Shift+F6
Компоновать Shift+F7
Выполнить все Shift+F8
Выполнить Ctrl+F8
Отладить Ctrl+F9

\ProjectsASM\ProjectsASM\PROBA\PROB
Окно Помощь
Компоновать


\АГТУ\Архитектура ЭВМ\ПРАКТИКА АГТУ\ProjectsASM\
Операции Инструменты Надстройки Окно Помощь
Установить текущую версию
Очистка
Компилировать RC Shift+F4
Res в Obj Shift+F5
Ассемблировать Shift+F6
Компоновать Shift+F7
Выполнить все Shift+F8
Выполнить Ctrl+F8
Отладить Ctrl+F9


.386
.MODEL flat, stdcall
OPTION CASEMAP:NONE
include KERNEL32.inc
includelib KERNEL32.LIB
.DATA
summand_1 db 12h
summand_2 db 2fh
.CODE
start:
mov al, summand_2
add al, summand_1
invoke ExitProcess,0
end start
MiniDBG
EAX 00000041
EBX 7FFDB000
ECX 0012FFB0
EDX 7C90EB94
ESI 00000000
EDI 00000000
EIP 0040101B 6A00E804
ESP 0012FFC4 7C816FD7
EBP 0012FFF0
CS 001B
DS 0023
ES 0023
FS 003B
GS 0000
SS 0023
C 0
P 1
A 1
Z 0
S 0
T 0
D 0
O 0
Restart
Go
Pause
Stop
Step
Last Error: Операция успешно завершена.
Log
Modules
Threads
Loaded at 7C900000:
ntdll.dll
Loaded at 7C800000:
C:\WINDOWS\system32\kernel32.dll

Синтаксис ассемблера

Все конструкции языка ассемблера можно разделить на 4 вида:

Команды (**инструкции**) – представляют собой символические аналоги машинных команд. *Например*, mov.

Макрокоманды – это оформляемые определенным образом предложения текста программы, замещаемые во время компиляции другими предложениями.

Директивы

Комментарии – содержат любые символы. Позволяют хранить примечания программиста к тексту исходной программы. Комментарии начинаются с символа *точка с запятой* “;”.

Формат команд и макрокоманд:

[имя метки] : [операция] [операнд(ы)] ; [комментарий]

Имя метки – символьный идентификатор строки программы.

Операция – символическое обозначение машинной команды или макрокоманды.

Операнд(ы) – части команды, макрокоманды или директивы, обозначающие объекты, над которыми производятся действия.

Пример:

```
metka_1:   adc al, var2 ;
```

складываем с учетом флага CF
содержимое регистра
и переменную

Формат директивы:

[имя] [директива] [операнд(ы)] ; [комментарий]

Пример:

summand_1 **db** 12h

~~**mov ax, summand_1**~~

Основные директивы размещения данных

Директива	Описание	Количество байт
DB (BYTE)	Объявить байт	1
DW (WORD)	Объявить слово	2
DD(DWORD)	Объявить двойное слово	4
DF (FWORD)	Объявить тройное слово	6
DQ (QWORD)	Объявить учетверенное слово	8
DT (TBYTE)	Объявить десять байтов (упятеренное слово)	10

Примеры:

summa db 10h ; выделяется байт, в него записывается число 10h

char1 DB ‘A’ ; выделяется байт, в него записывается 8-разрядный код символа A

list **db** 10h, 20h, 30h ,40h ; выделяются 4 байта, в них записывается число 40302010h

list **dd** 10203040h ; выделяются 4 байта, в них записывается число 10203040h

Для присвоения значений константам применяются **директивы объявления констант:**

1) Директива равенства – сопоставляет с именем константы числовое значение. Это значение может быть переопределено в программе. Формат директивы равенства: имя = выражение. Например, const1 = 50.
2) Директива EQU – сопоставляет с именем константы числовое значение или строку символов. Описанная таким образом константа не может быть переопределена в ходе программы. Формат использования:
имя EQU число
имя EQU <число1, число2, …>

Команды пересылки данных

1) **MOV** – копирует данные из одного операнда в другой. Формат команды:

MOV операнд-получатель, операнд-отправитель

Варианты отправителя и получателя:

MOV reg, reg

MOV mem, reg

MOV reg, mem

MOV mem, immed

MOV reg, immed

где reg – регистр ЦП, mem – место в памяти (например, переменная),
immed – непосредственное значение (например, 2Bh).

Недостаток команды MOV: отсутствие возможности использовать одновременно **два операнда памяти**, то есть чтобы переслать данные из одной переменной в другую, необходимо сначала из одной переменной поместить данные в какой-либо регистр, а затем из регистра переслать данные во вторую переменную.

2) **XCHG** (от exchange) – обменивает содержимое двух регистров или содержимое регистра и переменной. Возможны следующие **варианты использования**:

XCHG reg, reg

XCHG reg, mem

XCHG mem, reg

Отсутствует возможность использования двух операндов памяти.

Арифметические команды

1) Команды инкремента и декремента

INC операнд – команда инкремента (значение операнда увеличивается на единицу).

DEC операнд – команда декремента (значение операнда уменьшается на единицу).

В этом случае в качестве операнда рассматривается регистр процессора или участок памяти.

2) Команды сложения

ADD операнд-получатель, операнд-отправитель (складывает операнд-отправитель с операндом-получателем и помещает результат в операнд-получатель; исходный операнд-отправитель при этом не изменяется).

2) Команды сложения

ADC операнд-получатель, операнд-отправитель

Особенностью команды ADC является то, что ЦП в процессе ее выполнения складывает операнд-отправитель с операндом-получателем, но дополнительно производит операцию сложения полученного результата со значением флага CF (флаг переноса) из регистра флагов процессора.

3) Команда вычитания

SUB операнд-получатель, операнд-отправитель

(вычитает из операнда-получателя операнд-отправитель и помещает результат в операнд-получатель).

Команды передачи управления

Практически в любой программе есть точки, в которых необходимо принять решение о том, какая команда будет выполняться следующей. Это решение может быть двух видов:

1) безусловным, то есть произойдет переход не к следующей по порядку команде, а к **указанной**;

2) условный переход, то есть произойдет переход не к следующей по порядку команде, а к той, которая будет выполняться **исходя из анализа некоторых условий**.

Команда безусловного перехода:

JMP метка – заставляет процессор продолжать выполнение программы с места, отмеченного меткой, которая указывается в самой команде JMP.

Команды условного перехода:

Команды условного перехода в качестве анализа условий могут либо рассматривать соотношения между операндами, либо состояние флагов процессора (разрядов регистра флагов).

Для анализа соотношения между операндами перед командой условного перехода должна быть выполнена **команда сравнения операндов**:

CMP операнд1, операнд2

Команда CMP, как и команда SUB, выполняет вычитание операндов (из операнд1 вычитается операнд2), в результате выполнения вычитания процессор выставляет флаги в регистре флагов, но не записывает полученный результат вычитания на место первого операнда.

По результатам анализа флагов возможно произвести необходимый условный переход.

Команда условного перехода	Критерий условного перехода	Значение флагов для перехода
JE	операнд1=операнд2	ZF=1
JNE	операнд1<>операнд2	ZF=0
JL или JNGE	операнд1<операнд2	SF<>OF
JLE или JNG	операнд1<=операнд2	SF<>OF или ZF=1
JG или JNLE	операнд1>операнд2	SF=OF и ZF=0
JGE или JNL	операнд1=>операнд2	SF=OF
JB или JNAE	операнд1<операнд2	CF=1
JBE или JNA	операнд1<=операнд2	CF=1 или ZF=1
JA или JNBE	операнд1>операнд2	CF=0 и ZF=0
JAE или JNB	операнд1=>операнд2	CF=0

j – jump, e – equal, n – not, g – greater, l – less, a – above, b – below.

Формат команды условного перехода:

Jcc метка

где **cc** – код конкретного условия, анализируемого командой.

Пример:

```
CMP AL, AH     ; сравниваем значения AL и AH
JE metka1   ; если равны AL=AH, то переход
                        ; к команде перед которой стоит
          ; metka1
JL metka2
JG metka3
```

Другим вариантом команд условного перехода являются команды, которые просто **анализируют состояние определенных флагов процессора**.

Обозначение этих команд состоит также из символа “**J**” (jump) и одной буквы, **отражающей название флага**, перед которой может быть вставлен символ отрицания “N” (not).

Название флага	Номер бита в регистре флагов	Команда условного перехода	Значение флага для осуществления перехода
Переноса CF	1	JC	CF = 1
Четности PF	2	JP	PF = 1
Нуля ZF	6	JZ	ZF = 1
Знака SF	7	JS	SF = 1
Переполнения OF	11	JO	OF = 1
Переноса CF	1	JNC	CF = 0
Четности PF	2	JNP	PF = 0
Нуля ZF	6	JNZ	ZF = 0
Знака SF	7	JNS	SF = 0
Переполнения OF	11	JNO	OF = 0

Пример:

JC metka ; если флаг переноса равен **1**, то
; переход к команде,
; перед которой стоит **metka**

Две **команды**, предназначенные специально для **работы с регистром ECX**:

JCXZ metka ; (Jump if CX is Zero) если регистр
; CX содержит ноль, то переход
; к команде, перед которой
; стоит metka

JECXZ metka; (Jump if ECX is Zero) переход,
; если регистр ECX содержит
; ноль

Для организации цикла лучше всего использовать команду **LOOP**.

Формат:

LOOP metka

Эта команда вычитает из регистра ECX единицу. Если в результате выполнения декремента регистр ECX не принимает нулевое значение, то ЦП передает управление команде, перед которой находится **metka**.

Дополнительные операторы, директивы и команды

Оператор **OFFSET** – возвращает расстояние (смещение) переменной от начала сегмента.

Пример:

```
MOV EBX, OFFSET summand_1;  помещает в регистр
                ; EBX смещение переменной
                ; summand_1 от начала
                ; сегмента данных,
```

Операнд-приемник должен быть обязательно 32-разрядным, поскольку для указания смещения используются 32-разрядные числа.

Оператор **PTR** позволяет выделить из указанной переменной необходимое количество байт и поместить их в необходимое место.

Пример использования:

```
…
.DATA
var1 DB 12h
…
.CODE
mov al, byte PTR var1
…
```

Директива **LABEL** устанавливает метку и присваивает этой метке определенный размер без размещения данных.

В примере метка var_1 объявлена перед переменной var_2 и имеет длину равную 16 бит:

```
...
.DATA
  var_1 LABEL word
  var_2 DD 12345678h  ; размещаем в памяти
                      ; двойное слово
.CODE
  mov AX, var_1  ;  AX = 5678h
  mov CX, var_1[2]   ;  CX = 1234h
```

Команда **NEG** позволяет осуществить перевод чисел в противоположные им по знаку.

Формат команды:
NEG [операнд]

Полученное значение будет записано в этот же операнд.

Практические задания.

Задание 1. Разработать алгоритм и написать программу на языке ассемблера для сложения двух целых положительных чисел размером 1 байт. Исходные числа задаются в самой программе.

В алгоритме предусмотреть вариант получения результата с разрядностью, превышающей разрядность слагаемых.

Произвести отладку программы при различных значениях слагаемых, в том числе и для случая увеличения разрядности результата.

Практические задания.

Задание 2. Разработать алгоритм и написать программу на языке ассемблера для сложения двух целых положительных чисел размером N байт. Размер слагаемых и сами слагаемые задаются в самой программе.

В алгоритме предусмотреть вариант получения результата с разрядностью, превышающей разрядность слагаемых.

Произвести отладку программы при различных значениях слагаемых, в том числе и для случая увеличения разрядности результата.

БЛАГОДАРЮ ЗА ВНИМАНИЕ!