Ассемблер NASM для процессоров с архитектурой IA-32
Адресация
Вызов внешних подпрограмм

Материалы

Документация по ассемблеру NASM на русском (http://www.codenet.ru/progr/asm/nasm/)

Юров В. И. и др. Assembler: учебник для вузов. СПб: Питер, 2008. 636 с.

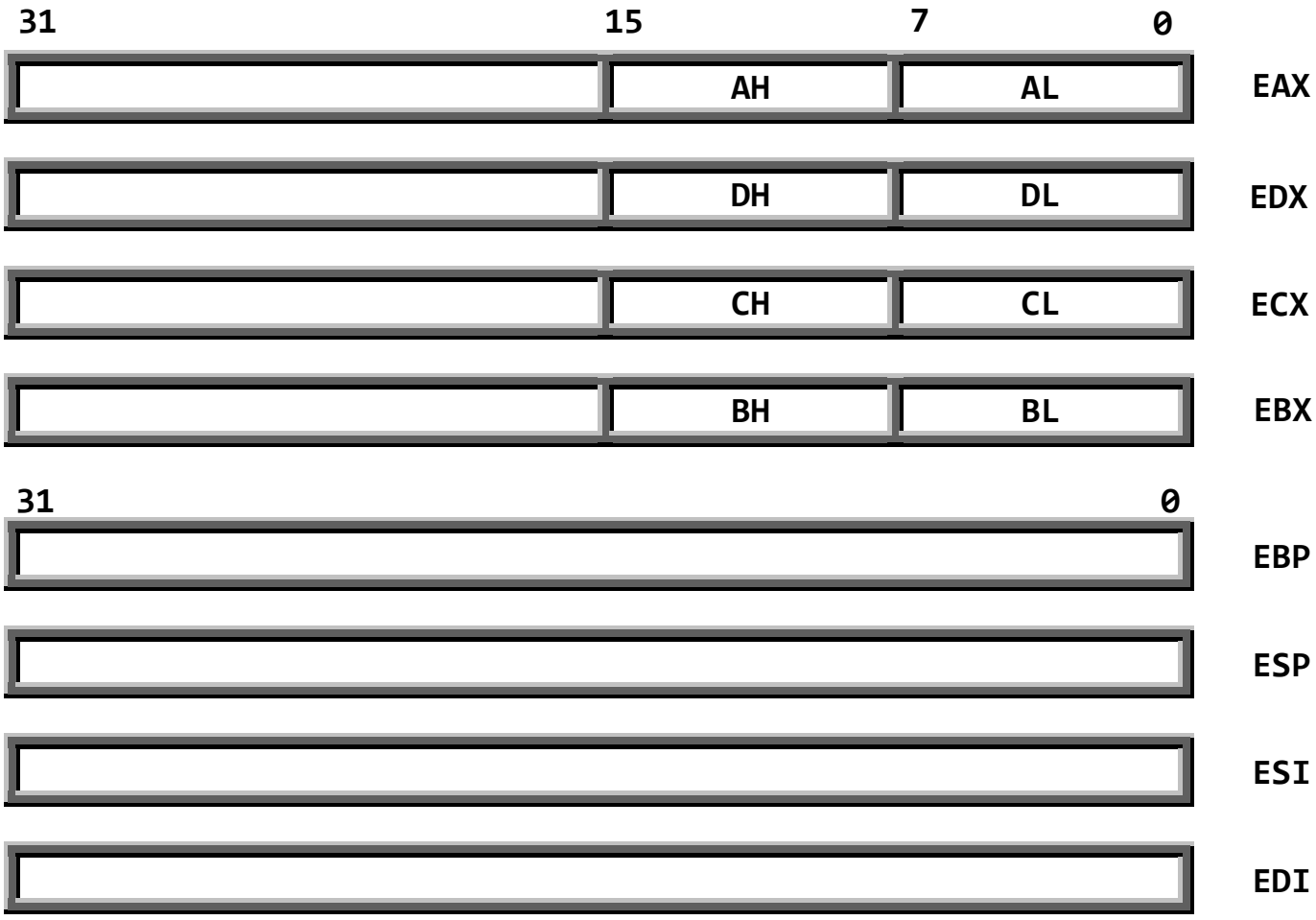
Полная документация по функциям **GlobalAlloc** (http://msdn.microsoft.com/en-us/library/aa366574%28v=vs.85%29.aspx) и **GlobalFree** (http://msdn.microsoft.com/en-us/library/aa366579%28v=vs.85%29.aspx).

Документация по отладчику OllyDbg на cracklab.ru (http://cracklab.ru/art/ollydbg.php) и habrahabr.ru (http://habrahabr.ru/post/93402/).

Адресация. Группы регистров

В прошлых практических работах было дано общее описание регистровой памяти и основные приемы работы с ней. Ниже представлено более подробное описание назначения основных регистров, которые стоит или не стоит использовать. Регистр является устройством временного хранения данных и используется с целью облегчения арифметических, логических и пересылочных операций. Процессоры с архитектурой IA-32 содержат 14 программно доступных регистров (общего назначения и сегментные), указатель команд (**EIP**) и регистр флагов (**EFLAGS**).

Регистры общего назначения (РОН)



Восемь регистров общего назначения (РОН) имеют длину 32 бита и могут хранить любые значения, однако каждый из них имеет свое назначение. Они поддерживают операнды-данные длиной 1, 8, 16, 32 и — при использовании двух регистров — 64 бит; битовые поля от 1 до 32 бит; операнды-адреса длиной 16 и 32 бит. Эти регистры называются **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, **EBP**, **ESP**. Доступ к младшим 16 битам этих регистров выполняется независимо при использовании соответствующих имен 16-битных регистров: **AX**, **BX**, **CX**, **DX**, **SI**, **DI**, **BP** и **SP**.

Также могут использоваться индивидуально младший (биты 0-7) и старший (биты 8-15) байты регистров **AX**, **BX**, **CX**, **DX**. Им соответствуют обозначения **AH**, **DH**, **CH**, **BH** и **AL**, **DL**, **CL**, **BL**.

В свою очередь РОН также делятся на подгруппы: регистры данных, индексные регистры и регистры-указатели.

Регистры данных (относятся к РОН)

EAX — регистр-аккумулятор, используется для накопления данных (“accumulator”);

EBX — регистр базы (“base”);

ECX — регистр-счетчик (“counter”);

EDX — регистр данных (“data”).

Эти регистры используются для хранения данных и выполнения различных арифметических и логических операций. Некоторые команды неявно используют определенный регистр. Например, **ECX** может играть роль счетчика цикла.

Индексные регистры (относятся к РОН)

Индексные регистры предназначены для хранения индексов при работе с массивами.

ESI (source index) содержит индекс источника;

EDI (destination index) содержит индекс приёмника.

Эти регистры можно использовать и как регистры общего назначения.

Регистры-указатели (относятся к РОН)

Основным назначением регистров-указателей является хранение индексов (смещений) относительно некоторого начала массива (базы) при выборке операндов из памяти. При этом адрес базы обычно хранится в одном из базовых регистров (**EBX** или **EBP**). Обращаться с ними нужно более осторожно, чем с предыдущими группами регистров, иначе можно получить непредвиденный результат (недопустимое обращение к памяти, выход за границы массивы и т. п.).

В данную группу регистров входят следующие:

EBP — указатель базы (“base pointer”);

ESP — указатель стека (“stack pointer”).

Регистр **EBP** служит указателем базы при работе с данными в стеке. Может использоваться произвольным образом в большинстве арифметических и логических операций или для временного хранения каких-либо данных. **ESP** указывает на вершину стека. Он используется командами, которые работают со стеком. **ESP** не стоит использовать для других целей.

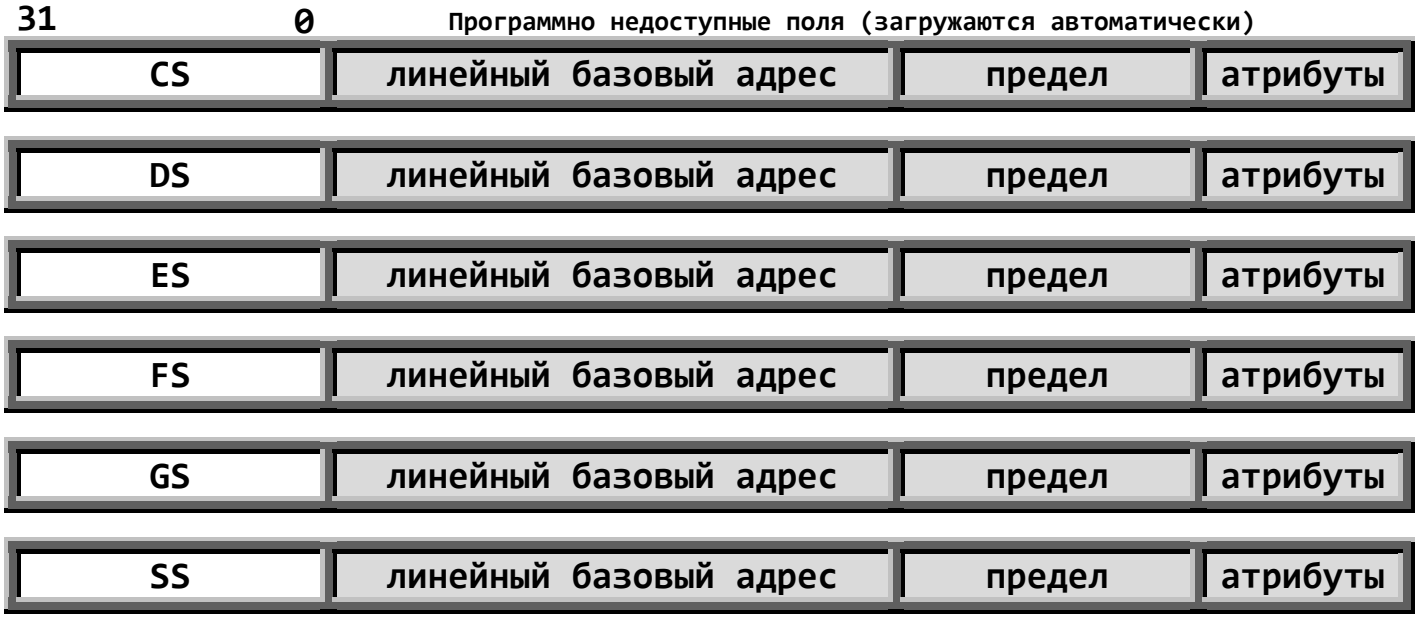
Примечание. Нельзя обратиться по имени к старшим или младшим битам регистров **ESI**, **EDI**, **EDP**, **ESP**, в отличие от **EAX**, **EBX**, **ECX**, **EDX**.

Следует отметить, что регистры могут быть неравнозначны и при использовании определенных инструкций могут иметь **специальное значение**:

- **EAX** — аккумулятор, операнд-источник или приемник результата (некоторые инструкции могут быть короче на один байт при использовании **EAX**);
- **EBX** — указатель на данные в сегменте **DS**;
- **ECX** — счетчик для цепочечных (например, **MOVS**) и циклических (с префиксом **REP** и т.п.) инструкций;
- **EDX** — адрес порта ввода-вывода для инструкций **IN** / **INS**, **OUT** / **OUTS**;
- **ESI** — указатель на операнд-источник в сегменте **DS** для цепочечных инструкций;
- **EDI** — указатель на операнд-приемник в сегменте **ES** для цепочечных инструкций;
- **EBP** — указатель на данные в сегменте **SS**.

Сегментные регистры

Процессор с архитектурой IA-32 включает 6 непосредственно доступных 16-битных регистров селекторов сегментов. С каждым сегментным регистром ассоциирован программно-недоступный кэш дескриптора соответствующего сегмента, содержащий базовый адрес сегмента в линейном адресном пространстве, предел сегмента и атрибуты сегмента. Этот кэш заполняется при загрузке значения в сегментный регистр. В реальном режиме предел сегмента всегда **0FFFFh**, атрибуты игнорируются, а базовый адрес вычисляется сдвигом значения селектора на 4 бита влево. В защищенном режиме кэш заполняется соответствующими значениями из дескрипторной таблицы.



Сегментные регистры предназначены для обеспечения сегментной адресации.

CS — регистр сегмента кода (“code segment”). Хранит селектор сегмента кода. Процессор извлекает очередную инструкцию для исполнения, формируя логический адрес из селектора в **CS** и смещения в регистре **EIP**. Значение этого регистра нельзя изменить непосредственно, оно меняется в командах межсегментного перехода (**FAR JMP**), межсегментного вызова (**FAR CALL**), при вызове обработчика прерывания (**INT**) и при возврате из дальней процедуры (**RETF**) или обработчика прерывания (**IRET**).

Регистры **DS**, **ES**, **FS** и **GS** хранят селекторы сегментов данных.

DS — регистр сегмента данных (“data segment”). Если инструкция обращается к памяти, но содержит только смещение, то считается, что она обращается к данным в сегменте **DS**.

ES — регистр дополнительного сегмента данных (“enhanced segment”). Сегмент **ES** может использоваться без явного указания в цепочечных командах.

FS и **GS** — используются при обращении к памяти только при явном использовании в инструкции префиксов этих сегментов.

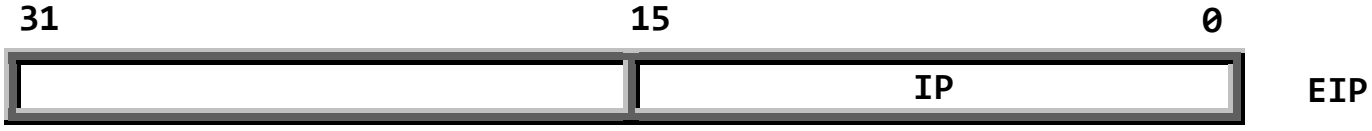
SS — регистр сегмента стека (“stack segment”). Регистр **SS** хранит селектор сегмента стека. Стек используется для передачи параметров подпрограммам и для сохранения адреса возврата при вызове подпрограммы или обработчика прерывания. Вершиной стека считается байт, логический адрес которого образуется из селектора в регистре **SS** и смещения в регистре **ESP**. Программа может непосредственно изменить значение **SS**, что дает ей возможность переключаться между несколькими стеками. Причем на время выполнения команды **MOV SS,xxxx** и одной команды следующей за ней (обычно это **MOV ESP,xxxx**) запрещаются маскируемые и блокируются немаскируемые прерывания.

Значения селекторов могут быть загружены при исполнении программы и являются специфичными для задачи. Это значит, что регистры сегментов в защищенном режиме перезагружаются автоматически при переключении микропроцессора на другую задачу. В то же время, использование сегментных регистров зависит от того, какая модель адресного пространства

используется. При использовании сплошной модели в сегментные регистры загружается один и тот же селектор сегмента с базой **0** и пределом **0FFFFFFFFh**, обеспечивая доступ ко всему линейному адресному пространству. При использовании сегментированной модели значения селекторов в сегментных регистрах независимы, так что программа может одновременно обращаться к шести сегментам.

Данную группу регистров необходимо было явно использовать в 16-разрядном программировании. В Win32 (начиная с Windows 95 и выше) такой проблемы уже не существует. Сегменты всё еще существуют, но о них не нужно заботиться, поскольку они уже не 64 КБ (как при 16-ти разрядах), а 4 ГБ. **Обращаться к регистрам этой группы не стоит** — это может повлечь непредвиденные ситуации.

Указатель команд

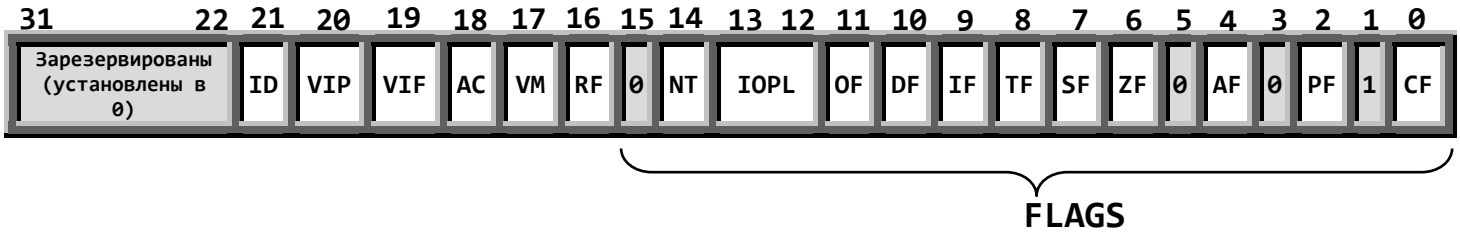


EIP — указатель команд, указатель инструкции (instruction pointer). Является 32-разрядным регистром. **EIP** содержит смещение следующей команды, подлежащей выполнению. Относительный адрес отсчитывается от начала сегмента исполняемой задачи. Указатель команд **EIP** непосредственно недоступен (это его сокращенное название, а не мнемоническое обозначение, используемое в языке программирования), но он управляется явно командами управления потоком, прерываниями и исключениями (**JMP**, **CALL**, **RET**, **IRET**, **команды условного перехода**). Получить текущее значение **EIP** можно, если выполнить команду **CALL**, а затем прочитать слово на вершине стека.

Младшие 16 бит регистра **EIP** обозначаются **IP** и могут быть использованы процессором независимо при исполнении 16-битного кода.

Регистр флагов

Регистр **EFLAGS** содержит группу флагов состояния, управления и системных флагов:



Регистр флагов содержит отдельные биты: флаги управления и признаки результата.

Неопределенные биты зарезервированы, то есть на данный момент они не имеют значения, однако могут быть использованы для специальных целей в следующих версиях процессора. Термин «установлен» означает значение **1**, а термин «сброшен» - значение **0**.

Некоторые из флагов могут быть изменены специально предназначенными для этой цели инструкциями. Для изменения или проверки группы флагов можно воспользоваться командами:

- **LAHF** / **SAHF** — загрузка / сохранение младших 8 битов регистра флагов в регистре **AH**;
- **PUSHF** / **POPF** — помещение / извлечение из стека младших 16 битов регистра флагов;
- **PUSHFD** / **POPFD** — помещение / извлечение из стека 32-битного регистра **EFLAGS**.

Флаги статуса

Флаги статуса (признаки результата) устанавливаются после выполнения арифметических и логических команд (таких как **ADD**, **SUB**, **MUL**, **DIV**):

CF (carry) — флаг переноса. **CF=1** (установлен), если операция привела к переносу из старшего бита при сложении или к займу для старшего бита при вычитании, иначе сброшен. Для 8-, 16-, 32-

разрядных операций этот бит устанавливается при переносе из битов **7**, **15** и **31** соответственно. Для беззнаковых операций флаг сигнализирует о переполнении. Значение этого флага может быть изменено непосредственно при помощи инструкций: **CLC** — сбросить **CF** в **0**, **STC** — установить **CF** в **1**, **CMC** — инвертировать **CF**. Также используется в операциях сдвига.

PF (parity) — флаг четности. **PF=1** (установлен), если младшие восемь бит операнда содержат четное число единиц (проверка на четность) иначе сброшен. На этот флаг влияют только младшие восемь бит независимо от длины операнда.

AF (auxiliary; adjust) — флаг вспомогательного переноса. Используется для упрощения сложения и вычитания упакованных двоично-десятичных чисел. Независимо от длины операнда (8, 16 или 32 бит) флаг **AF** установлен, если операция привела к займу из бита **3** при вычитании или переносу из бита **3** при сложении, иначе он сброшен.

ZF (zero) — флаг нулевого результата. **ZF=1** (установлен), если все биты результата равны нулю, иначе сброшен.

SF (sign) — флаг знака. **SF=1** (установлен), если установлен старший бит результата, иначе он сброшен. Для 8-, 16- и 32-разрядных операций этот флаг отражает состояние **7**, **15** и **31** бита соответственно. Для знаковых чисел старший бит отражает знак числа: **0** — неотрицательное, **1** — отрицательное.

OF (overflow) — флаг переполнения. **OF=1** (установлен), если получен результат за пределами допустимого диапазона значений. То есть если операция привела к переносу (займу) в знаковый (самый старший) бит результата, но не привела к переносу (займу) из самого старшего бита, или наоборот. Для операций над числами со знаком сигнализирует о переполнении.

Флаг направления

DF (direction) — флаг направления. Управляет направлением обработки строк данных, поведением цепочных инструкций (**MOVS**, **CMPS**, **SCAS**, **LODS**, **STOS**): **DF=0** — от младших адресов к старшим, **DF=1** — от старших адресов к младшим (для специальных строковых команд). Когда флаг сброшен, при выполнении цепочечной команды происходит автоинкремент адресов источника и приемника. Когда флаг установлен — автодекремент. Флаг можно непосредственно установить при помощи инструкции **STD** и сбросить при помощи **CLD**.

Системные флаги и поле IOPL

Системные флаги (флаги управления) и поле **IOPL** влияют на процесс исполнения задачи, и поэтому не должны изменяться прикладной программой. Назначение этих флагов описано ниже.

Эти флаги меняют режим работы процессора.

TF (trap) — флаг ловушки, флаг трассировки. Используется отладчиком для выполнения программы по шагам. Установка флага **TF** переводит процессор в пошаговый режим для отладки. Процессор автоматически генерирует исключение #1 после каждой команды, что позволяет проверить программу на исполнение каждой команды. Когда флаг **TF** сброшен, то ловушка по исключению #1 возникает в точках адресов останова, загружаемых в регистры отладки **DR0**–**DR3**.

IF (interrupt enable) — флаг разрешения прерываний. Установка флага (**IF=1**) позволяет процессору воспринимать запросы внешних маскируемых прерываний. Снятие этого бита запрещает такие прерывания. Флаг не влияет на обработку, как немаскируемых внешних прерываний, так и исключений.

IOPL (I/O privilege level field) — уровень привилегий ввода-вывода. Это двухбитное поле используется в защищенном режиме. Биты **IOPL** показывают наивысшее значение текущего уровня привилегий (**CPL**), позволяющее выполнять команды ввода-вывода, не приводя к исключению #13 или обращению к битовой карте разрешения ввода-вывода. Это поле показывает также наивысшее значение **CPL**, которое позволяет изменять бит **IF** с помощью команд **STI** или **CLI**, а также при выборке нового значения из стека в регистр **EFLAGS**. Это поле может быть изменено инструкциями **POPF** или **IRET** только если текущий уровень привилегий задачи равен **0**.

NT (nested task) — флаг вложенной задачи. Если при переключении задач происходит вложение

задач, то этот флаг устанавливается в **1**. Совместно с полем «Связь TSS» в сегменте состояния задачи обеспечивает корректное вложение задач.

RF (resume) — флаг возобновления. Временно приостанавливает обработку исключений отладки (т. е. возвращает к нормальному исполнению программы) так, что исполнение команды может быть повторено после обработки исключения для отладки, не вызывая немедленно обработку другого исключения для отладки.

VM (virtual 8086-mode) — режим виртуального микропроцессора 8086. Бит обеспечивает для задачи функционирование в режиме виртуального МП 8086. Бит **VM** может быть установлен только двумя способами: при восстановлении флагов из стека по инструкции **IRET** на нулевом уровне привилегий и переключением на задачу, в **TSS** которой в образе **EFLAGS** бит **VM** выставлен.

AC (alignment check) — флаг контроля выравнивания. Разрешает контроль выравнивания для текущей задачи. Контроль выравнивания производится, если **CR0.AM=1** и **EFLAGS.AC=1** и **CR0.PE=1** и **CPL=3**. Контроль выравнивания требует, чтобы при обращениях к памяти двойное слово обязательно должно начинаться с адреса, кратного **4**, а 16-битное слово — с адреса, кратного **2**, иначе генерируется нарушение контроля выравнивания (исключение #17).

VIF (virtual interrupt) — виртуальный флаг прерывания. Виртуальный образ флага **IF**, используется совместно с флагом **VIP**. Процессор распознает **VIF**, если **CR4.VME=1** или **CR4.PVI=1** (разрешено расширение виртуального режима) и **IOPL<3**.

VIP (virtual interrupt pending) — виртуальный флаг задержки прерывания. Системное ПО устанавливает этот флаг, если требуется отложить обработку прерывания. Используется совместно с **VIF**. Процессор читает этот флаг, но никогда не изменяет его. Флаг распознается, если **CR4.VME=1** или **CR4.PVI=1** (разрешено расширение виртуального режима) и **IOPL<3**.

ID (identification) — флаг поддержки CPUID. Если программа может изменить этот флаг (т. е. процессор хранит то значение, которое программа запишет в этот флаг), то процессор поддерживает инструкцию CPUID. Инструкцию CPUID поддерживают не только Pentium и P6, но и некоторые модели i486.

Адресация. Обращение по ссылке и по значению

В языках высокого уровня при объявлении простой переменной обращение к ней ведется по значению. Кроме того, существует специальный тип данных — указатель, который содержит адрес ячейки памяти, и в случае его использования обращение будет происходить по ссылке.

На языке ассемблера ситуация иная — прямое обращение к имени переменной является обращением к адресу этой переменной в памяти. Для обращения к значению ячейки по какому-либо адресу на диалекте NASM (и многих других) используются квадратные скобки. Ниже представлен пример программы, которая выводит адрес и значение переменной:

```
extern ExitProcess
%include "io.inc"
section .text use32
..start:
      push dword x          ; печать адреса переменной x
      call outi
      call outln
      add [x],dword 256     ; изменение значения x
      push dword [x]  ; печать значения переменной x
      call outi
      push dword 0
      call ExitProcess
section .data
      x dw 256
```

Адресация. Эффективный адрес

Для большинства вычислительных систем эффективный адрес (“effective address”, EA) — адрес, по которому происходит обращение к памяти; численно совпадает с виртуальным адресом. Вычисление EA выполняется следующим образом:

```
база + ( индекс * коэффициент ) + смещение
```

Где:

- ☼ в качестве базы может выступать любой РОН;
- ☼ индексом может быть любой РОН, кроме **ESP**;
- ☼ коэффициент масштабирования может принимать численное значение **1**, **2**, **4** или **8**;
- ☼ смещение задается константой.

Примечание: можно встретить и более полную форму вычисления эффективного адреса — перед базой указан сегмент, представленный одним из сегментных регистров; для прикладных задач (вроде практических работ) сегмент не учитывают.

База, индекс и смещение могут использоваться в различных комбинациях — некоторые из компонентов могут равняться нулю. Коэффициент масштабирования может использоваться только вместе с индексом.

Ниже показан пример вычисления адреса **ebx+esi*2+2** на языке ассемблера:

```
mov eax,ebx   ; eax := ebx (копируем адрес базы для вычислений)
mov edi,esi   ; edi := esi (копируем текущий индекс для вычислений)
imul edi,2    ; edi := edi * 2
add eax,edi   ; eax := eax + edi
add eax,2     ; eax := eax + 2
```

Однако записывать такое количество инструкций зачастую неудобно, поэтому для вычисления EA была добавлена специальная инструкция **LEA** (“load effective address”). Приведенный выше пример можно записать с использованием одной команды **LEA**:

```
lea eax,[ebx+esi*2+2]
```

На диалекте NASM инструкция **MOV** может использоваться точно так же как и **LEA** (однако **LEA** в некоторых случаях эффективнее), то есть допустим следующий код:

```
mov eax,[ebx+esi*2+2]
```

Адресация. Описание массивов

Инициализированные статические массивы на диалекте NASM записываются так же, как и простые переменные, например:

```
array db 1,1,2,3,5,8,13,21      ; массив из N элементов
array_length equ $-array        ; количество элементов массива
```

Неинициализированные статические массивы описываются как и неинициализированные переменные:

```
buffer resb 64          ; выделить 64 байта
wordvar resw 2          ; выделить 2 слова
```

Для динамического выделения памяти необходимо выполнить системный вызов. Такой привилегией обладают лишь функции ОС. Для работы с динамической памятью в Windows используются следующие внешние подпрограммы.

`GlobalAlloc(<флаги>, <кол-во>)` — запрос к Windows на выделение указанного количества байтов памяти; флаги указывают различные параметры выделения памяти. **`GlobalFree(<указатель>)`** — запрос к Windows для пометки области памяти, на которую ссылается указатель, как свободной.

Примечание: на сайте MSDN расположена полная документация функций `GlobalAlloc` и `GlobalFree`.

Доступ к массивам данных может быть организован с помощью обычных циклов (инструкция **`loop`** или только условные переходы), однако наиболее эффективным является использование эффективных адресов ячеек массива и итерация по ним.

Отладчик программ на языке ассемблера

Для отладки программ, написанных на языке ассемблера рекомендуется использовать бесплатный отладчик OllyDbg 2.0. Документация по отладчику на русском языке:

- ☼ cracklab.ru;
- ☼ habrahabr.ru.