

**Е.С. Абрамов, И.Д. Сидоров**

**МАШИННО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ   
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственноЕ

АВТОНОМНОЕ образовательноЕ учрежденИЕ   
высшего образования

«Южный федеральный университет»

**Инженерно-технологическая академия**

**МАШИННО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

**Учебное пособие**

Таганрог

Издательство Южного федерального университета

2016

УДК 004.431.4(075.8)

ББК 32.973Я73

А161

Печатается по решению редакционно-издательского совета

Южного федерального университета

**Рецензенты:**

главный научный сотрудник ГНИИИ ПТЗИ ФСТЭК, доктор технических наук Язов Юрий Константинович;

директор ООО «GigitalPro», кандидат технических наук Казарин Максим Николаевич

**А161 Абрамов, Е.С., Сидоров, И.Д.**

Машинно-ориентированное программирование: учебное пособие / Е.С. Абрамов, И.Д. Сидоров. – Таганрог: Издательство Южного федерального университета, 2016. – 89 с.

ISBN 978-5-9275-2065-7

Настоящее учебное пособие посвящено изучению основных принципов применения языка программирования Ассемблер. Большое количество наглядных примеров позволит освоить основные принципы составления программ на языке Ассемблер. А комплекс лабораторных работ по изучению основных приемов использования языка Ассемблер будет способствовать закреплению изученного материала. Учебное пособие по курсу "Машинно-ориентированное программирование" рассчитано на студентов, магистрантов и аспирантов направлений 10.03.01 «Информационная безопасность», 10.05.02 «Информационная безопасность телекоммуникационных систем», 10.05.03 «Информационная безопасность автоматизированных систем». Изученный материал позволит самостоятельно применять полученные знания на практике.

Табл. 6. Ил. 12. Библиогр. 5 назв.

ISBN 978-5-9275-2065-7 УДК 004.431.4(075.8)

ББК 32.973Я73

**©** ЮФУ, 2016

**©** Абрамов Е.С.,

Сидоров И.Д., 2016

**Содержание**

[Введение 6](#_Toc456278230)

[1. Устройство микропроцессора i386 7](#_Toc456278231)

[1.1. Основные сведения о микропроцессоре i386 7](#_Toc456278232)

[1.2. Виды адресации 9](#_Toc456278233)

[2. Защищённый режим микропроцессоров i386 14](#_Toc456278234)

[2.1. Общие сведения о защищенном режиме 14](#_Toc456278235)

[2.2. Дескриптор 15](#_Toc456278236)

[2.3. Таблицы дескрипторов 17](#_Toc456278237)

[2.4. Селектор 19](#_Toc456278238)

[2.5. Преобразование логического адреса в линейный 20](#_Toc456278239)

[2.6. Сегментный регистр 22](#_Toc456278240)

[2.6. Страничная адресация 23](#_Toc456278241)

[2.7. Каталоги и таблицы страниц 25](#_Toc456278242)

[2.8 Линейная адресная трансляция (4 кб страницы) 25](#_Toc456278243)

[2.9. Линейная адресная трансляция (4 мб страницы) 26](#_Toc456278244)

[2.10. Элементы каталогов и таблиц страниц 28](#_Toc456278245)

[3. Механизмы защиты 32](#_Toc456278246)

[Лабораторная работа № 1 38](#_Toc456278247)

[Лабораторная работа № 2 45](#_Toc456278248)

[Лабораторная работа № 3 48](#_Toc456278249)

[Лабораторная работа № 4 54](#_Toc456278250)

[Лабораторная работа № 5 58](#_Toc456278251)

[Лабораторная работа № 6 62](#_Toc456278252)

[Лабораторная работа № 7 68](#_Toc456278253)

[Лабораторная работа № 8 75](#_Toc456278254)

# Введение

Данное учебное пособие предназначено для теоретической и практической подготовки по курсу «Машинно-ориентированное программирование». Учебное пособие содержит теоретический материал по изучению основных регистров и команд для низкоуровневого языка программирования Ассемблер.

В учебном пособии приведен комплекс лабораторных работ, направленных на изучение студентами различных аспектов программирования на языке ассемблера, таких как: работа с арифметическими и логическими командами, командами передачи управления.

Учебное пособие ориентировано на студентов, магистрантов и аспирантов Института компьютерных технологий и информационной безопасности Южного федерального университета, обучающихся по направлениям 10.03.01 «Информационная безопасность», 10.05.02 «Информационная безопасность телекоммуникационных систем», 10.05.03 «Информационная безопасность автоматизированных систем». Изученный материал позволит самостоятельно применять полученные знания на практике.

Предполагается, что студенты знакомы с основами программирования на языке Си, а также имеют представление об операционных системах Windows и \*nix.

# 1. УСТРОЙСТВО МИКРОПРОЦЕССОРА I386

# 1.1. Основные сведения о микропроцессоре i386

В ходе изучения данного курса мы будем работать с Ассемблером процессора Intel 386, а также процессоров, совместимых с ним. Выбор 32-разрядного процессора обусловлен тем, что 16-разрядные процессоры на сегодняшний день сильно устарели, а для 64-разрядных нет бесплатных инструментов разработчика, удобных для работы студентов. Программная архитектура процессора Intel 386 представлена на рис. 1.

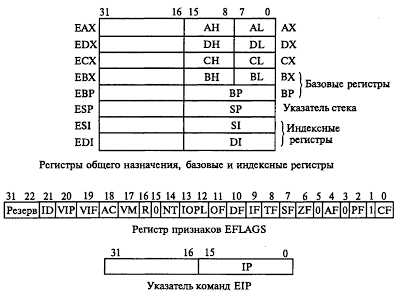


Рис.1. Программная архитектура процессора Intel 386

С точки зрения программиста на Ассемблере, центральный процессор представляет собой устройство, интерпретирующее и выполняющее поток команд, а также ячейки памяти, называемые регистрами. Микропроцессор Intel 386 имеет в своем составе:

* **регистры общего назначения** (EAX, EBX, ECX, EDX) – предназначены для хранения операндов и промежуточных результатов выполняемых операций;
* **регистры-указатели стека** (ESP, EBP) и **индексные регистры** (ESI, EDI) – используются для хранения базовых адресов и индексов при реализации некоторых способов адресации сложных структур данных;
* **указатель команд** (EIP);
* **регистр признаков** (EFLAGS) – содержит признаки результата выполненной операции и флаги управления.

Название регистров зависит от их назначения:

EAX/AX/AH/AL (accumulator register) – аккумулятор, используется для обмена информацией с внешними устройствами;

EBX/BX/BH/BL (base register) – регистр базы, используется для косвенной адресации;

ECX/CX/CH/CL (counter register) – счётчик, используется в командах организации циклов;

EDX/DX/DH/DL (data register) – регистр данных, используется для расширения целых чисел в командах умножения и деления;

ESI/SI (source index register) – индекс источника, используется для косвенной адресации и в строковых (цепочечных) командах;

EDI/DI (destination index register) – индекс приёмника (получателя), используется для косвенной адресации и в строковых (цепочечных) командах;

ESP/SP (stack pointer register) – регистр указателя стека, указывает на первый заполненный байт в стеке;

EBP/BP (base pointer register) – регистр указателя базы кадра стека, используется для организации кадра стека в процедурах/функциях на языках высокого уровня.

Также модель процессора включает в себя сегментные регистры – CS,SS,DD,ES,FS,GS. Они не используются при выполнении лабораторных работ, так как за их заполнение отвечает операционная система Windows или Linux.

Показанные на рис. 1 регистры EIP (указатель на текущую выполняемую команду) и EFLAGS (флаги) не могут быть непосредственно использованы в командах ассемблера. Доступ к ним возможен косвенным образом.

# 1.2. Виды адресации

Все имеющиеся способы адресации можно условно разделить на три группы: регистровая, непосредственная и с указанием адреса в памяти. При этом адрес в памяти можно задавать по-разному: прямым указанием символического обозначения ячейки памяти, указанием регистра, в котором хранится требуемый адрес, или и того и другого. Таким образом, третья группа включает, в сущности, целый ряд способов адресации. Они обычно носят названия: прямая, базовая, индексная, базово-индексная, а также базовая, индексная или базово-индексная со смещением [6].

***Регистровая адресация***

Операнд (байт или слово) находится в регистре. Способ применим ко всем программно-адресуемым регистрам процессора [6].

***Примеры****:*

push DS; *Сохранение  DS  в  стеке*

mov BP,SP; *Пересылка  содержимого  SP  в  ВР*

***Непосредственная адресация***

Операнд (байт или слово) может быть представлен в виде числа, адреса, кода ASCII, а также иметь символьное обозначение.

***Примеры:***

mov AX, 4C00h; *Операнд* – *16-ричное число*

mov DX, offset mas; *Смещение массива mas заносится в DX*

mov DL,'!'; *Операнд* – *код ASCII символа '!'*

num=9; *Число 9 получает  обозначение  num*

mov CX, num; *Число, обозначенное тип, загружается в СХ*

***Прямая адресация памяти***

В команде указывается символическое обозначение ячейки памяти, над содержимым которой требуется выполнить операцию.

***Пример:***

movDL, meml; *Содержимое байта памяти с символическим именем meml пересылается в DL*

Если нужно обратиться к ячейке памяти с известным абсолютным адресом, то этот адрес можно непосредственно указать в качестве операнда. Предварительно необходимо настроить какой-либо сегментный регистр на начало того участка памяти, в котором находится искомая ячейка.

***Пример:***

mov AX, 0; *Настроим сегментный регистр ES на*

mov ES,AX; *самое начало памяти (адрес 0)*

mov AX,ES:[0]; *АХ=содержимое слова с адресом 0000h:0000h*

mov DX,ES:[2]; *ОХ=содержимое слова с адресом 0000h:0002h*

Заметим, что в этом случае сегментный регистр надо указывать обязательно. Все остальные способы адресации относятся к группе косвенной адресации памяти.

***Базовая и индексная адресация памяти***

Относительный адрес ячейки памяти находится в регистре, обозначение которого заключается в квадратные скобки. При использовании регистров ВХ или ВР адресацию называют базовой, при использовании регистров SI или DI – индексной. При адресации через регистры ВХ, SI или DI в качестве сегментного регистра подразумевается DS; при адресации через ВР – регистр SS. Таким образом, косвенная адресация через регистр ВР предназначена для работы со стеком. Однако при необходимости можно явно указать требуемый сегментный регистр. Еще раз отметим, что во всех базовых и индексных способах адресации операндом является содержимое ячейки памяти, адрес которой находится в том или ином регистре или вычисляется сложением содержимого двух регистров [6].

***Примеры:***

mov AL, [ВХ]; *Сегментный адрес предполагается в DS, смещение в ВХ*

mov DL,ES:[ВХ]; *Сегментный адрес в ES, смещение в ВХ*

mov DX,[ВР]; *Сегментный адрес в SS, смещение в ВР*

mov AL, [DI]; *Сегментный адрес в DS, смещение в DI*

***Базовая и индексная адресации памяти со смещением***

Относительный адрес операнда определяется суммой содержимого регистра (ВХ, ВР, SI или DI) и указанного в команде числа, которое называют смещением [6].

***Пример:***

mas db  1,2,5,3,7,9,8,3,4;*Массив символов*

mov BX,2;*ВХ=индекс элемента в массиве*

mov DL,mas[BX];*В DL заносится третий элемент массива*

В этом примере относительный адрес адресуемого элемента массива mas вычисляется как сумма содержимого ВХ (т.е. содержимое ВХ равно 2) и значения символического обозначения mas, которое совпадает с относительным адресом начала массива mas. В результате в регистр DL будет загружен элемент массива mas с индексом 2, т.е. число 5. Предполагается, что базовый адрес сегмента, в который входит массив mas, загружен в DS. Такой же результат даст такая последовательность команд [6]:

mov BX, offset mas; *ВХ=относительный адрес ячейки mas*

mov DL, 2[BX]

Здесь относительный адрес адресуемого элемента массива mas вычисляется как сумма содержимого регистра ВХ и дополнительного смещения, задаваемого константой 2. Последняя команда может быть записана в следующем виде [6]:

mov DL, [BX+2]

mov DL, [ВХ]+2

Адресация с помощью регистров SI и DI осуществляется аналогично. При использовании регистра ВР следует помнить, что в качестве сегментного регистра по умолчанию подразумевается регистр SS [6].

***Базово-индексная адресация памяти***

Относительный адрес операнда определяется суммой содержимого базового и индексного регистров. Допускается использование следующих пар:

[BX][SI]

[BX][DI]

[ВР][SI]

[ВР][DI]

Если в качестве базового регистра выступает ВХ, то в качестве сегментного подразумевается DS (первые две команды); при использовании в качестве базового регистра ВР сегментным регистром по умолчанию назначается SS (вторые две команды). При необходимости можно явно указать требуемый сегментный регистр.

***Примеры:***

mov ВХ,[ВР][SI];*В ВХ засылается слово из стека (сегментный адрес в SS), а смещение вычисляется как сумма содержимого ВР и SI*

mov BX,ES:[BP][SI];*В ВХ засылается слово из сегмента, адрес которого находится в ES, а смещение вычисляется как сумма содержимого ВР и SI*

mov ES:[BX+DI],АХ;*В ячейку памяти, сегментный адрес которой хранится в ES, а смещение равно сумме содержимого ВХ и DI, пересылается содержимое АХ*

***Базово-индексная адресация памяти со смещением***

Относительный адрес операнда определяется суммой трех величин: содержимого базового и индексного регистров, а также дополнительного смещения. Допускается использование тех же пар регистров, что и в базово-индексном способе; так же действуют правила определения сегментных регистров.

***Примеры:***

mov mas [ВХ][SI], 10;*Число 10 пересылается в ячейку памяти, сегментный адрес которой хранится в DS, асмещение равно сумме содержимого ВХ и SI и смещения ячейки mas*

mov AX,[BP+2+DI];*В АХ пересылается из стека слово,смещение которого равно сумме ВР, DI и "добавки" 2*

Значительная часть рассмотренных выше способов адресации служит для обращения к ячейкам памяти. Таким образом, один и тот же конечный результат можно получить с помощью различных способов адресации. Например, все три приведенные ниже команды:

mov DL, mas+3

mov DL, mas[BX]; *В BХ заранее занесено число 3*

mov DL, [SI][ВХ]; *В BХ заранее занесено число 3, а в SI* – *смещение mas*

приведут к загрузке в регистр DL четвертого элемента массива mas (если выполняются описанные в комментариях условия). Однако команды с использованием различных способов адресации занимают различный объем памяти и выполняются за разное время. Так, первая из приведенных выше команд потребует для выполнения 15 машинных тактов, вторая – 18, а третья – 16. Разница невелика, однако при многократном выполнении команд в циклах суммарный эффект может быть значителен. С другой стороны, первые две команды занимают в памяти по 4 байта, а третья – только 2. Таким образом, тщательный выбор способов адресации позволяет в какой-то степени оптимизировать программы по времени выполнения или требуемой памяти, а иногда и по тому и по другому.

# 2. ЗАЩИЩЁННЫЙ РЕЖИМ МИКРОПРОЦЕССОРОВ I386

# 2.1. Общие сведения о защищенном режиме

Что такое защищенный режим и почему он так называется? Дело в том, что реальный режим процессора Intel –однозадачная среда. Это означает, что в данный момент времени в ней может выполняться только одна задача. Безусловно, можно сэмулировать многозадачность и в реальном режиме, но все дело в том, что именно в защищенном режиме вся многозадачность реализована аппаратно.

Рассмотрим общие положения об организации памяти в защищенном режиме PM (*от англ. Protected Mode*).. В процессорах Intel организацию памяти разделяют на две части: сегментация (segmentation) и страничная организация (paging).

Сегментация позволяет изолировать модули кода, данных, стека и позволяет работать нескольким задачам и программам на одном процессоре (multitasking) и не конфликтовать между собой – это и есть по большому счету, вся революция по сравнению с процессором 8086 [7].

Страничная организация памяти, вообще говоря, сложнее сегментации, но у такой организации больше возможностей, и именно на ней работает Windows.

В процессоре нет такого бита, который бы четко отвечал за переключение между этими двумя режимами. Элементы сегментной организации, в любом случае, присутствуют всегда. А за включение страничной адресации отвечает флаг PG, бит 31 регистра CR0 [7].

Непосредственно сегмент может содержать в себе код, данные, стек, системные структуры данных (TSS, LDT, но о них позже). Если запущено сразу несколько программ, то каждой программе принадлежит своя, личная группа сегментов.

Память разделена на группы сегментов, у каждой группы есть свой владелец (программа). При любой попытке программы пресечь границы срабатывает механизм защиты и возникает исключение #GP (General Protection) [7].

Для того чтобы обратиться к любому байту в любом сегменте в памяти, мы должны сформировать логический адрес (дальний указатель).

Что такое логический адрес? Селектор и смещение. Селектор – это уникальный идентификатор сегмента. У каждого сегмента есть свой селектор. Селектор содержится в сегментом регистре (да да, те самые DS, CS, ES …). Зная селектор и зная смещение, мы можем получить линейный адрес – место, где реально расположен нужный нам байт [7].

Существует также такое понятие, как физическое адресное пространство процессора. Физический адрес – это адрес, который процессор может выставить на адресную шину, и в случае сегментной организации памяти (но не страничной) совпадает с линейным адресом [7].

# 2.2. Дескриптор

После того как программа или программы загружены в оперативную память, каждая программа владеет группой сегментов (данные, код, стек). Возникает вопрос: откуда процессор знает, где какой сегмент? Где начало сегмента? И где его конец? И что это за сегмент: данных? кода? А может быть, стека? Ответить на этот вопрос может только одна структура данных: дескриптор. Эта структура, описывающая сегмент, приведена на рис. 2 [8].

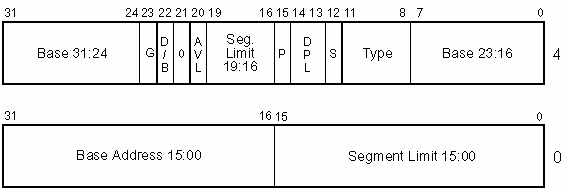


Рис.2. Дескриптор сегмента

Значения полей дескриптора

**Адрес базы**: адрес нулевого байта описываемого сегмента в 4 Гб линейном адресном пространстве (адрес, с которого начинается сегмент) [8].

**Лимит сегмента**: определяет размер сегмента. Реальный лимит сегмента зависит от бита гранулярности (G-granularity); если бит гранулярности сброшен (0), то 20-битное значение и будет тем самым лимитом сегмента; если бит гранулярности установлен (1), то всё 20-битное значение автоматически увеличивается в 1000h раз. Если при G=0 мы измеряем размер в байтах, то при G=1 – в 4Кб единицах [8].

Например, если G=1 и поле «Лимит сегмента» = 0000Fh (15 байт), то реальный лимит данного сегмента равен 0Fh•1000h=0F000h.

Следует отметить, что если поле «Лимит сегмента» содержит значение, равное 0, то это значит, что описываемый сегмент имеет размер в 1 байт (а не ноль!) при G=0, и размер в 4Кб при G=1. То есть сегмент никак не может иметь нулевую длину, минимум – 1 байт, максимум – 0хFFFFFh•4Кб = 4 Гб.

Есть еще один бит, от которого зависит смысловое значение этого поля. Бит направления роста сегмента (B-big)); если этот бит сброшен (0), то разрешены все смещения от 0 до лимита; если установлен (1) – то все, кроме от 0 до лимита.

В противном случае (если мы попытаемся обратится за пределы лимита) – возникнет исключение главной защиты (#GP).

**Тип:** определяет тип сегмента, права доступа к сегменту и направление роста сегмента (помните бит B). Значение этого поля зависит от значения поля «Тип дескриптора» (S-descriptor type). Значение этого поля различно для разных типов сегментов (кода, данных и системного).

**S (descriptor type)** – флаг «типдескриптора». Означает только одно: если сброшен (0), то описываемый сегмент – системный, если установлен (1) – это сегмент данных или кода.

**DPL (descriptor privilege level)** – уровень привилегий дескриптора. Определяет уровень привилегий сегмента. Это поле – двухразрядное, соответственно может принимать только четыре различных значения (от 0 до 3). Самый высокий – нулевой уровень привилегий (ядро ОС). Это поле нужно для контроля за доступом к сегменту.

**P (segment present flag)** – флаг присутствия сегмента. Если установлен, значит сегмент присутствует непосредственно в памяти; если сброшен – отсутствует. Используется для работы механизма подкачки (swaping).

**G (Granularity)** – флаг «гранулярности»: (см. «Лимит сегмента»), этот флаг влияет на лимит сегмента: если сброшен – лимит измеряется в байтах, если установлен – в 4 Кб единицах.

**AVL (Available and reserved bits)** – зарезервировано. Это два бита (21-20 во втором двойном слове). Бит 20 может использоваться как угодно. Бит 21 всегда должен быть равен 0.

***D/B.*** Зависит от типа сегмента (в зависимости от этого он называется либо D, либо B). Учитывая то, что про него уже было сказано при описании лимита сегмента, следует отметить еще вот что. Определяет разрядность сегмента. Если установлен – значит сегмент 32-разрядный, если сброшен – 16-разрядный. Это общий случай.

**Частный случай. Сегмент кода.** Для сегмента кода данный флаг называется D и устанавливает длину по умолчанию для эффективных адресов и операндов в сегменте. Если установлен, то в сегменте допустимы 32-битные адреса и 32-битные ИЛИ 8-битные операнды. Если сброшен, то 16-битные адреса и 16-битные ИЛИ 8-битные операнды. Но можно использовать префикс 66h (для изменения разрядности операндов) и префикс 67h (для изменения разрядности адресов).

**Частный случай. Сегмент стека.** Для сегмента стека данный флаг называется B (big) флаг и устанавливает длину *указателя стека* (регистр ESP) для команд push, pop и call. Если установлен, то используется 32-разрядный указатель стека (т.е. регистр ESP использован по максимуму), если сброшен, то используется только 16 бит (регистр SP).

# 2.3. Таблицы дескрипторов

Информация о положении, размере и свойствах сегмента хранится в дескрипторе [9].

Находиться дескриптор может в трех местах:

* глобальная таблица дескрипторов (GDT – Global Descriptor Table);
* локальная таблица дескрипторов (LDT – Local Descriptor Table);
* таблица дескрипторов прерываний (IDT – Interrupt Descriptor Table).

Эти таблицы находятся в оперативной памяти (там же, где и собственно сами программы). Поэтому строить таблицы дескрипторов придется программистам уровня ядра ОС.

***Глобальная таблица дескрипторов (GDT):***

Каждая ОС имеет одну и только одну GDT. Таблица GDT – сама по себе не сегмент. Это структура данных в линейном адресном пространстве (в памяти). Начало таблицы GDT хранится в регистре GDTR. Регистр GDTR – это самый обыкновенный регистр, такой же обыкновенный, как EAX, EIP, ES, только вот его функция заключается не в хранении каких-то промежуточных данных, а в хранении фиксированного числа – начала таблицы GDT. Название регистра GDTR запомнить очень легко: GDT – это таблица, R – регистр [9].

GDTR: 32-битный линейный базовый адрес|16-битный лимит таблицы.

GDTR имеет размер 48 бит. Он содержит не только адрес начала таблицы GDT в памяти, а еще и ее лимит. Лимит таблицы есть 16-битное значение, которое показывает величину таблицы в байтах + 1 (все как и в случае с лимитом сегмента: если лимит таблицы в GDTR равен 0, то на самом деле это означает что реально (в памяти) лимит равен одному байту) [9].

Сегментный дескриптор всегда занимает 8 байт (2 двойных слова). Следовательно, лимит таблицы дескрипторов – величина, равная 8N–1 байт.

Первый дескриптор в GDT не используется и называется «нулевой дескриптор» (null descriptor). При обращении к памяти через этот дескриптор сбрасывается исключение #GP.

Загрузить/считать значение регистра GDTR можно командами LGDT/SGDT. По умолчанию (после нажатия на кнопку Reset или включения компьютера) база GDT равна нулю, а лимит – FFFFh, т.е. фактически по умолчанию выделено максимум места, под FFFFh/8 = 8191 дескрипторов (минус один, учитывая пустой дескриптор) [9].

***Локальная таблица дескрипторов (LDT):***

В отличии от GDT может отсутствовать вообще. И в то же самое время их можно создать великое множество (GDT должна быть только одна). Каждая задача может иметь свою собственную LDT, в то же время несколько задач могут использовать одну LDT на всех [9].

LDT – это сегмент, а GDT – структура данных. Так как LDT – это сегмент, то значит у нее тоже есть свой дескриптор в GDT.

Так же как и у GDT, у LDT тоже есть свой регистр – LDTR. В отличии от GDTR этот регистр, помимо информации про базу и лимит LDT, содержит еще одно поле – сегментный селектор.

LDTR: сегментный селектор (16 бит); 32-битный линейный базовый адрес; 16-битный лимит сегмента.

Инструкции LLDT и SLDT позволяют писать/читать регистр LDTR. Точно так же, при сбросе, значение базы в LDTR падает в ноль, а лимит – в FFFFh.

# 2.4. Селектор

Материал изложен по [4]

Селектор – это 16-битная структура данных, которая является идентификатором сегмента (рис. 3) [9].

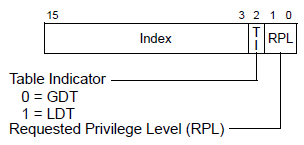


Рис.3. Селектор дескриптора сегмента

Поле ИНДЕКС (биты 3-15): указывает на один из 8192 дескрипторов в таблице дескрипторов (GDT или LDT).

Флаг TI (table indicator) (второй бит). Если он равен 0, то используется GDTR (т.е. дескриптор расположен в таблице GDT), если же установлен – LDTR.

RPL (Requested Privilege Level, биты 0–1) Запрашиваемый уровень привилегий.

# 2.5. Преобразование логического адреса в линейный

Как же процессор формирует адрес и знает куда обращаться? Рассмотрим, как найти линейный адрес в памяти, руководствуясь только двумя значениями – селектор и смещение, например CS и EIP? В CS ищем поле "Индекс" (см. на селектор выше). Смотрим в поле индекс и узнаем о местоположении нужного дескриптора в таблице дескрипторов. Далее нам нужно узнать адрес базы сегмента. Теперь осталось только одно: сложить этот адрес базы с EIP – и мы получим линейный адрес инструкции в памяти (который при сегментной организации совпадает с физическим, мы об этом говорили ранее). Селектор-->дескриптор-->база... +EIP = ЛИНЕЙНЫЙ адрес. Схема трансляции логического адреса в линейный для таблицы GDT приведена на рис. 4, а для таблицы LDT – на рис. 5.

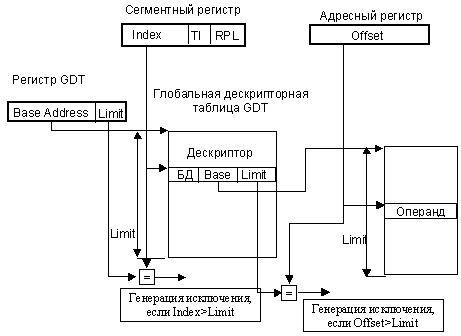


Рис. 4. Трансляция логического адреса в линейный для таблицы GDT

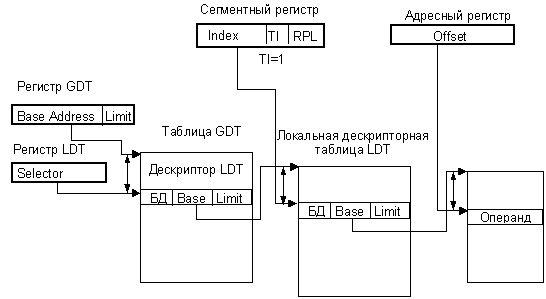


Рис. 5. Трансляция логического адреса в линейный для таблицы LDT

# 2.6. Сегментный регистр

В архитектуре процессоров Intel существуют шесть сегментных регистров:

CS, DS, SS, ES, GS и FS. Каждый из этих регистров отвечает за свой сегмент в памяти (кода, данных или стека). Итак, даже если программа состоит из тысячи сегментов, только 6 из них могут быть доступны одновременно. Другие сегменты станут доступны только после загрузки их селекторов в сегментные регистры.

Возникает вопрос: неужели процессор при каждом обращении к памяти все время повторяет одни и те же действия (ищет дескриптор, потом ищет базу, затем прибавляет к базе смещение...), и так при выполнении фактически каждой команды... Нет. Сегментный регистр – это 80-битный регистр, нам же доступны только младшие 16 бит, которые и называются селектором. Остальные 64 бита называются "Теневым регистром" (Shadow register) или "Дескрипторным кэшем" (Descriptor cache), они и содержат базу, которую процессор должен был бы высчитывать на каждом шаге. Кроме базы этот самый "теневой регистр" содержит еще и лимит, и права доступа. Как только мы загружаем в видимую часть (в селектор) соответствующее значение, процессор по селектору (а конкретно – по полю «индекс») извлекает базу, лимит и права доступа из дескриптора и заносит их в "теневую часть" сегментного регистра [9].

Если мы вдруг решим неожиданно поменять значение базы в дескрипторе для какого-либо сегмента, селектор которого в данный момент уже находится в сегментном регистре, то мы также должны позаботиться и о перезагрузке сегментного регистра, так как в теневой части останутся старые значения базы и лимита [9].

Загрузить сегментные регистры (явно или неявно) позволяют 16 команд [9]:

явно:

MOV –команда пересылки данных;

POP – значение из стека;

LDS – загрузить DS;

LES – загрузить ES;

LSS – загрузить SS;

LGS – загурзить GS;

LFS – загрузить FS.

неявное:

CALL, JMP, RET, SYSENTER, SYSEXIT, IRET, INTn, INTO, INT3. Чаще всего "неявные" команды изменяют значение именно CS-регистра, но в некоторых случаях и других.

# 2.7. Страничная адресация

Прежде всего, при использовании страничной адресации структуры из сегментной адресации (как-то: таблицы дескрипторов, селекторы, регистры таблиц дескрипторов) остаются на своих местах. Единственное место, где страничная адресация вклинивается в работу процессора, – это перевод линейного адреса в физический.

Вспомним три понятия: логический, линейный и физический адрес. Логический адрес – это некий абстрактный, CS:EIP. Что он может нам сообщить? Только то, что если мы узнаем из CS поле «индекс», а затем по этому полю найдем в таблице дескрипторов соответствующий дескриптор, и к базе из этого дескриптора прибавим EIP, то только тогда получим линейный адрес. При использовании сегментной адресации физический адрес совпадает с линейным (физический – это адрес, который процессор выставляет уже на адресную шину). При сегментной адресации процессор просто берет линейный адрес и выставляет на адресную шину.

При использовании страничной адресации линейный адрес не совпадает с физическим, как в случае с сегментной адресацией. Процессор делит линейное адресное пространство на страницы фиксированного размера (длиной 4Кб, 2Мб или 4Мб), которые, в свою очередь, уже отображаются в физической памяти (или на диске). Когда программа обращается к памяти через логический адрес, процессор переводит его в линейный и затем, используя механизм страничной адресации, переводит его в соответствующий физический адрес. Если страницы в данный момент нет в физической памяти, то возникает исключение #PF. Обработчик этого исключения (#PF) должен выполнить соответствующие манипуляции по устранению данной проблемы, т.е. подгрузить страницу с жесткого диска (или наоборот – скинуть ненужную страницу на диск).

Что такое своп-файл в ОС Windows? Страницы памяти на жётском диске, которые после возникновения #PF должны быть загружены в ОЗУ (или наоборот). По сути #PF –это штатная ситуация работы процессора и ОС.

Как только страница была благополучно возвращена на место, то выполнение прерванной программы продолжается с той самой инструкции, которая вызвала #PF. Какая инструкция может вызвать #PF? Да в принципе любая, которая так или иначе обращается к памяти [10].

Страничная организация отличается от сегментной еще и тем, что все страницы имеют фиксированный размер (в сегментной организации размер сегментов абсолютно произволен). Также при сегментной адресации все сегменты обязательно должны присутствовать непосредственно в ОЗУ, а при страничной возможна ситуация, когда кусок сегмента находится в памяти, а другой кусок фактически того же сегмента – на диске [10].

В случае со страничной адресацией страницы, к которым процессор чаще всего обращается, кэшируются в процессоре, в области, которая называется буфер с ассоциативной выборкой (TLB – Table lookaside buffer). Там находятся не все такие страницы целиком, а только записи, которые необходимы для доступа к ним [10].

Страничная организация непосредственно в процессоре управляется тремя флажками [10]:

1. Флаг PG (paging): бит 31 в регистре CR0.
2. Флаг PSE (page size extensions): бит 4 в регистре CR4.
3. PAE (physical address extension) flag: бит 5 врегистре CR4.

Флаг PG разрешает страничную адресацию. Сразу после установки его в единицу страничная адресация включена [10].

Флаг PSE, если его установить, позволяет использовать страницы больших размеров (4Мб и 2Мб). Если сброшен – страницы имеют размер 4Кб [10].

Флаг PAE позволяет расширить физический адрес до 36 бит (стандартно он 32-битный). Данный флаг можно использовать только в режиме страничной адресации [10].

Следует отметить, что использовать 36-разрядную адресацию можно как с помощью PAE-флага, так и с помощью PSE-флага, это два разных метода, мы их рассмотрим дальше [10].

# 2.8. Каталоги и таблицы страниц

При трансляции линейного адреса в физический (при включенной страничной адресации) процессор использует 4 структуры данных:

1. Каталог страниц – массив 32-битных записей (PDE – page-directory entry), которые хранятся в 4 Кб странице.

2. Таблица страниц – массив 32-битных записей (PTE – page-table entry), которые также все расположены в одной 4 Кб странице.

3. Сама страница – 4 Кб, 2 Мб или 4 Мб участок памяти.

4. Указатель на каталог страниц – массив 64-битных записей, каждая из которых указывает на каталог страниц. Эта структура данных используется процессором только при использовании 36-битной адресации.

Все эти таблицы позволяют обращаться к 4 Кб или к 4 Мб страницам при 32-битной адресации и к 4 Кб, 2 Мб или 4 Мб страницам при 36-битной адресации. Флаг PSE-36 появился только в Pentium 3, а доступен он или нет можно узнать посредством CPUID. При использовании PSE-36 механизма доступны только страницы размером 4 Мб. Фактически так можно адресовать 64 Gb физического адресного пространства.

При страничной организации существуют 3 способа адресации: 32-разрядная, 36-разрядная с использованием флага PAE и 36-разрядная с использованием флага PSE.

Рассмотрим 32-разрядную адресацию.

# 2.9. Линейная адресная трансляция (4 Кб страницы)

Линейный адрес, который получается в результате сложения базы сегмента со смещением при сегментной организации при страничной организации с 4 Кб страницами, делится на три части [10]:

* Номер записи в каталоге страниц: биты 22-31.
* Номер записи в таблице страниц: биты 12-21.
* Смещение в самой 4 Кб странице: биты 0-11.

В регистре CR3 (рис. 6) только биты 12-31 содержат смещение каталога страниц в памяти. При этом младшие 12 битов всегда равны нулю.



Рис. 6. Регистр CR3

Может создаться ошибочное мнение, что каталог страниц – один единственный, и указывает на него регистр CR3 (как в случае с глобальной таблицей дескрипторов и регистром GDTR). На самом же деле все обстоит совсем не так, каталогов страниц может быть несколько, даже у каждой задачи свой, а регистр CR3 меняется после переключения задачи [11].

Схема формирования физического адреса в случае 4 Кб страниц показана на рис. 7.

# 2.10. Линейная адресная трансляция (4 Мб страницы)

Фактически за счет размера страниц мы ничего не теряем и опять же, можем адресовать 4 Гб, как и в случае с 4 Кб страницами. 4 Мб размер страницы получается в случае установки флагов PSE и PS. Флаг PSE находится в CR4, а флаг PS находится непосредственно в записи каталога страниц [10, 11].

В случае 4 Мб страниц линейный адрес делится на ДВЕ части (рис. 8):

* Номер записи в каталоге страниц (биты 22–31).
* Смещение в самой 4 Мб странице (биты 0–21).

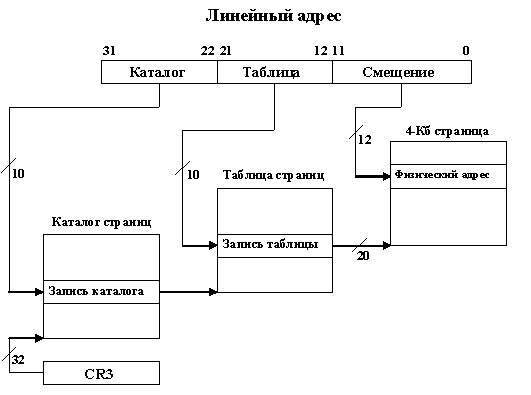


Рис. 7. Формирование физического адреса в случае   
4 Кб страниц

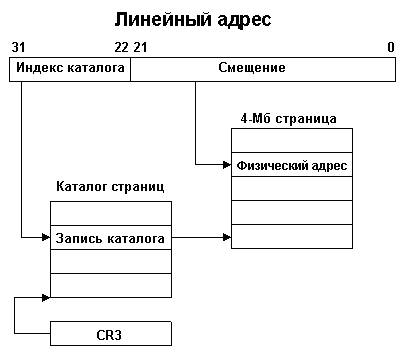


Рис. 8. Адресация при использовании 4 Мб страниц

***PCD-флаг (page level cache disabled)*** – четвертый бит, контролирует кэширование каталога страниц. Если он установлен, то кэширование не происходит. Если сброшен – каталог страниц может кэшироваться. Этот флаг влияет только на L1 и L2 (внутренние кэши процессора). Процессор игнорирует этот флаг, если страничная адресация отключена (флаг PG в CR0 сброшен) или если вообще отключен кэш (флаг CD в CR0) [11].

***PWT-флаг (page-level writes transparent)*** – третий бит, контролирует кеширование записи [11].

***Регистр CR3*** (главным образом, хранящий начало каталога страниц текущей задачи) также называют PDBR (page directory base register). Если мы собираемся использовать страничную адресацию, мы должны загрузить этот регистр в процессе инициализации. Загрузить его можно либо непосредственно командой mov, либо он сам загрузится при переключении задач [11].

Может возникнуть ситуация, когда каталог страниц в момент переключения отсутствует в физической памяти, поэтому ОС перед переключением задачи должна устранить это недоразумение. Каталог страниц должен находиться в памяти столько времени, сколько активна задача, использующая этот каталог страниц [11].

# 2.11. Элементы каталогов и таблиц страниц

При страничной организации линейный адрес (база из дескриптора + offset) не соответствует физическому (т.е. адресу, который процессор выставляет на адресную шину). Прежде чем попасть на адресную шину, он проходит ряд преобразований [11].

Итак, линейный адрес делится на три части в случае 4 Кб страниц, и на две в случае 4 Мб страниц.

Рассмотрим случай с 4 Кб страницами.

Биты 22–31 линейного адреса – это номер элемента (записи) в каталоге страниц. Адрес начала каталога страниц в памяти содержит регистр CR3. Каждый элемент в каталоге страниц «весит» 32 бита [11].

Поле «Адрес базы таблицы страниц» содержит адрес базы таблицы страниц. Сам элемент (его номер в таблице страниц) определяют биты 12–21 линейного адреса. Его структура представлена на рис. 9.

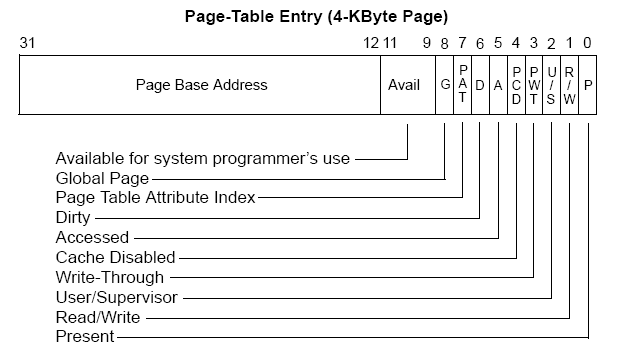


Рис. 9. Структура элемента таблицы страниц

Очень похож на неё элемент каталога страниц, но все же есть отличия. Структура элемента каталога для 4 Мб страниц представлена на рис. 10.

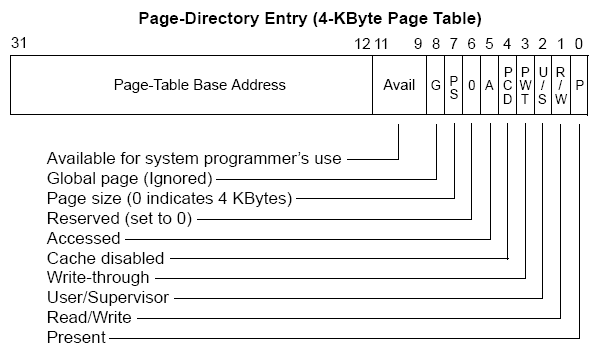


Рис. 10. Структура элемента каталога страниц

***P-флаг***, бит 0: указывает, находится ли страница (или таблица страниц) в физической памяти. Если он равен 1, то страница находится в памяти, и можно продолжать формирование физического адреса. Если же он сброшен, то процессор вызывает исключение #PF (Page Fault), обработчик которого должен загрузить ее в память. Процессор никогда не меняет этот флаг сам – этим должна заниматься ОС.

***R/W-флаг***, бит 1: определяет привилегии чтения/записи для страницы или группы страниц (в случае, когда элемент каталога страниц указывает на таблицу страниц). Если флаг сброшен, то страница доступна только для чтения. Если установлен – для чтения и записи. Данный флаг тесно связан с флагами U/S и WP из CR0.

***U/S-флаг***, бит 2: определяет привилегии пользователя/супервизора для страницы или группы страниц. Если флаг сброшен, то страница доступна только для супервизорского уровня привилегий, если установлен – для пользователя и супервизора. Данный флаг связан с флагами U/S и WP.

***PWT-флаг***, бит 3: контролирует кэширование страницы (write-through и write-back). Если флаг установлен, то write-through-кэширование разрешено, если сброшен – тогда разрешено write-back-кэширование. Процессор игнорирует этот флаг, если установлен CD (cache disabled) флаг в CR0.

***PCD-флаг***, бит 4: контролирует кэширование страницы. Если флаг установлен, то кэширование страницы отключено, если сброшен – разрешено. Так же как и с PWT-флагом, значение PCD-флага игнорируется, если установлен CD-флаг в CR0.

***A-флаг***, бит 5: (аналог флага А в дескрипторе) показывает, было ли произведено обращение к странице с момента загрузки ее в память. ОС при загрузке страницы в память в первый раз должна сбросить этот бит в ноль. Затем, после ее загрузки в память, процессор отлавливает момент, когда к ней в первый раз произойдет обращение и ***сам*** устанавливает этот флаг в единицу. Флаг А (также как и D) нужен для управления и контроля за страницами и таблицами страниц.

***D-флаг***, бит 6: показывает была ли произведена запись в страницу. Этот флаг игнорируется в элементах каталога страниц, которые указывают на таблицы страниц, при 4 Кб-страничной организации. ОС должна обнулить данный флаг как только страница загружена в память, затем процессор уже ***сам*** установит его, как только в страницу будет произведена первая запись и больше его трогать не будет. Данный флаг по сути схож с флагом A.

***PS-флаг***, бит 7 (для 4 Кб-страниц): определяет размер страницы. Если флаг сброшен, то страница 4 Кб и элемент каталога страниц указывает на таблицу страниц. Если установлен – страница 4 Мб (при 32-битной адресации) или 2 Мб (если используется EPA) и элемент каталога страниц указывает на саму страницу. Если элемент каталога страниц указывает на таблицу страниц, то все страницы, на которые ссылаются элементы таблицы страниц также являются 4 Кб-ми.

***PAT-флаг***, бит 7 в элементе таблицы страниц (для 4 Кб страниц) или бит 12 в элементе каталога страниц (для 4 Мб страниц): PAT – это таблица, расширение IA-32 архитектуры. На процессорах, не поддерживающих эту возможность, флаг зарезервирован и должен равняться нулю.

***G-флаг***, бит 8: появился в Pentium Pro-процессорах. Если флаг установлен, то это означает, что описываемая страница является глобальной.

Зарезервированные и доступные для программ биты: существуют во всех IA-32-процессорах. Биты 9,10 и 11 доступны для использования программно. Если бит P (бит присутствия сегмента) сброшен, то доступны все 32 бита). Для элемента каталога страниц, который указывает на таблицу страниц, бит 6 должен всегда равняться нулю. Если PSE- и PAE-флаги в CR4 установлены, то процессор будет сбрасывать #PF, если зарезервированные биты не сброшены в нули.

# 3. МЕХАНИЗМЫ ЗАЩИТЫ

Современные операционные системы способны одновременно выполнять десятки пользовательских процессов, защищённых от взаимного влияния. Это (и многое другое) обеспечивается благодаря механизму защиты.

В процессорах с архитектурой IA-32 защитные механизмы реализованы как на сегментном, так и на страничном уровнях (т.е. после включения страничной адресации предоставляются некоторые дополнительные возможности защиты, которые, вообще говоря, без страничной адресации не имеет смысла применять). Процессор различает четыре уровня сегментной защиты (3-0) и два – страничной (здесь номеров нет, но условно их два). Чем уровень ниже (численно), тем сегмент более привилегирован. Так, весь код ядра ОС и драйвера устройств помещают в сегменты с нулевым уровнем привилегий [12].

Механизм защиты обеспечивает проверку каждого обращения к памяти до цикла чтения/записи. Любое нарушение границ приводит к генерации исключения.

Следует отметить, что все проверки производятся параллельно с процедурой преобразования адресов, поэтому задержка на проверку отсутствует. Условно защиту IA-32 можно разделить на 6 разделов [12]:

* проверка границ;
* проверка типов;
* проверка уровней привилегий;
* ограничения на адресацию;
* ограничения на использование инструкций;
* ограничения на точки входа в процедуры.

Весь механизм защиты процессора Intel обеспечивается 3-мя флагами (плюс еще 2 дополнительных при использовании страничной адресации) и 5-ю полями

Флаги, обеспечивающие защитный механизм [13]:

***Тип дескриптора (S)*** – бит 12 во втором двойном слове дескриптора сегмента. Отвечает за тип сегмента: системный, кода или данных.

***Флаг гранулярности (G)*** – бит 23 во втором двойном слове дескриптора сегмента. Вместе с полем "Лимит" дескриптора и флагом E (флаг направления роста) определяют размер сегмента.

***Флаг направления роста сегмента (E)*** – бит 10 во втором двойном слове дескриптора сегмента. Работает на пару с флагом G и полем "Лимит".

***Флаг пользователь/супервизор (U/S)*** – бит 2 элемента таблицы или каталога страниц. Определяет тип страницы: пользовательская или супервизорная.

***Флаг чтения/записи (R/W)*** – бит 1 элемента таблицы или каталога страниц. Определяют тип доступа к странице: только для чтения или для чтения/записи. Флаг R/W - второй (он же последний) дополнительный механизм защиты при использовании страничной адресации.

Поля, обеспечивающие защитный механизм [13]:

***Поле "Тип"*** – биты 8 – 11 во втором двойном слове дескриптора сегмента. Определяет тип сегмента кода, данных или системного сегмента.

***Поле "Лимит"*** – биты 0 – 15 первого двойного слова и биты 16-19 второго двойного слова дескриптора сегмента. Вмести с флагом G и E определят размер сегмента.

***Поле "Уровень привилегий дескриптора" (DPL)*** – биты 13 и 14 во втором двойном слове дескриптора сегмента. На самом деле поле DPL задает уровень привилегий описываемого им сегмента, дескриптору уровень привилегий ни к чему. Чем меньшее значение содержит это поле, тем описываемый сегмент более привилегирован. Сегмент с нулевым уровнем привилегий может исполнять ЛЮБЫЕ инструкции, обращаться к ЛЮБЫМ данным. В сегментах с 0-м уровнем должно располагаться ядро ОС, другие части ОС – в сегментах с 1-м уровнем, драйвера – со 2-м, пользовательские программы – с 3-м. На деле программисты из Microsoft при проектировании ОС Windows решили "упростить" модель до двухуровневой: все, что не пользовательские программы, то в 0-м уровне, пользовательские – в третьем. 1-й и 2-й уровни не используются. По сути, если бы Intel IA32 эксплуатировался только под ОС Windows, то для полей DPL, RPL и CPL хватило бы одного бита вместо двух [13].

***Поле "Запрашиваемый уровень привилегий" (RPL)*** – биты 0 и 1 любого (!) сегментного селектора.

***Поле "Текущий уровень привилегий" (CPL)*** – биты 0 и 1 сегментного регистра CS (!). Фактически определяет уровень привилегий текущей исполняемой программы (процедуры), уровень привилегий исполняемого в данный кода.

Поле "Лимит" дескриптора сегмента призвано уберечь программы (процедуры) от обращения к памяти за пределами, установленными этим полем. Эффективное (реальное) значение лимита зависит от флага гранулярности G. Для сегментов данных лимит также зависит от флага E (направление роста) и флага B дескриптора.

Если бит G=0, то лимит сегмента совпадает со значением одноименного поля дескриптора; очевидно, что в данном случае сегмент может иметь длину от 0 до FFFFFh(1Мб). Если флаг G=1, то реальный лимит сегмента равен содержимому поля "Лимит" дескриптора, умноженному на 4Кб (FFFh). Очевидно, сегмент в этом случае может занимать от 4Кб до 4Гб.

Важно: если бит G=1, то младшие 12 бит адреса не проверяются на превышение лимита! Если создать дескриптор сегмента с G=1 и полем "Лимит", равным 0, то смещения от 0 до FFFh будут доступны для обращения. Процессор игнорирует младшие 12 бит при проверке на превышение лимита.

Итак, процессор сгенерирует исключение #GP (номер 13) в следующих случаях:

* обращение к байту по смещению большему, чем эффективный лимит;
* обращение к слову по смещению большему, чем (эффективный лимит-1);
* обращение к двойному слову по смещению большему, чем (эффективный лимит-3);
* обращение к учетверенному слову по смещению большему, чем (эффективный лимит-7).

У сегментов, имеющих тип 4 или 6, поле "Лимит" в дескрипторе определяет НИЖНЮЮ границу сегмента, а верхняя граница равна FFFFh (1Мб), если бит B=0 и FFFFFFFFh (4Гб), если бит B=1. Возникает вопрос: если у нас определена нижняя граница сегмента (поле "Лимит") и верхняя (1Мб или 4Гб), то зачем тогда поле база в дескрипторе? Поле "Лимит" хоть и определяет нижнюю границу, но все равно отсчитывается от базы. Фактически нижняя граница у растущих вниз сегментов равна: содержимое поля "Лимит" дескриптора + содержимое поля "База" дескриптора.

Процессор, помимо проверки лимитов сегментов, проверяет также лимиты таблиц дескрипторов, информация о которых содержится в регистрах GDTR, IDTR, LDTR и TR. #GP будет сгенерировано, если произойдет попытка обращения к дескриптору, лежащему за пределами таблицы дескрипторов.

***Проверка типа сегмента***

Процессор проверяет бит S (бит 12 второго двойного слова) и поле «Тип» (биты 8 – 11 второго двойного слова) в следующих двух случаях [14]:

* при загрузке селектора в сегментный регистр;
* при обращении к сегменту.

Состояние #GP возникнет если:

* в регистр CS загрузить селектор на дескриптор, который описывает отличный от сегмента кода сегмент;
* в регистры DS,ES,FS или GS загрузить селектор на дескриптор, который описывает нечитаемый сегмент кода;
* в регистр SS загрузить селектор на дескриптор, описывающий отличный от «читаемый сегмент данных» сегмент;
* в регистр LDTR загрузить селектор, не указывающий на LDT;
* в регистр TR загрузить селектор, не указывающий на TSS;
* при записи в исполняемый сегмент (т.е. писать в сегмент кода нельзя через CS, через DS – пожалуйста);
* при записи в сегмент данных доступный только для чтения;
* при чтении из нечитабельного исполнимого сегмента;
* дескрипторы в таблице IDT должны быть шлюзами прерывания, ловушки или задачи.

Бит S и поле «Тип» в дескрипторе говорят процессору, что с сегментом делать можно, а чего нельзя. Если кто-то попытается сделать то, чего делать нельзя, то будет вызван обработчик #GP.

***Нулевой селектор***

Если в CS или SS загрузить селектор, который указывает на нулевой дескриптор (т.е. 16-разрядное число, биты 15 – 3 которого равны 0), то #GP возникнет немедленно.

Если нулевой селектор загрузить в DS,ES,FS или GS, то #GP возникнет при попытке обращения к данным через такой селектор.

Нулевой селектор придуман в качестве подстраховки от ненужных обращений к сегментам.

***Уровни привилегий***

Иногда (в частности – в ОС Windows) их называют кольца защиты (ProtectionRings). Всего уровней привилегий 4: 0 – самый привилегированный, 3 – наименее привилегированный. В нулевом кольце располагается код ядра ОС, драйвера, обработчики исключений (тот же #GP). Однако ОС Windows, как и \*nix тоже, используют только два кольца: 0 и 3. В третьем – пользовательские приложения и служебные сервисы ОС, в нулевом – все остальное.

Оговорим, что далее запись «уровень привилегий задачи A больше уровня привилегий задачи B», значит, что численное значение привилегий задачи А меньше чем у B.

Всего существует три типа уровней привилегий: это CPL (Current Privilege Level), RPL (Requested Privilege Level) и DPL (Descriptor Privilege Level).

CPL – это текущий уровень привилегий, иначе говоря – уровень привилегий кода (задачи), исполняющегося в данный момент. Он находится в поле «уровень привилегий» селектора, загруженного в данный момент в регистр CS.

DPL – уровень привилегий сегмента. Хранится в дескрипторе каждого сегмента в таблице дескрипторов.

RPL – уровень привилегий запроса. Определяется содержимым поля «уровень привилегий» сегментного регистра (за исключением CS).

Запрос на обращение успешно выполняется, если верно следующее неравенство:

MAX(CPL,RPL)<=DPL

То есть наименьший из запрашиваемого уровня привилегий и уровня привилегий кода должен быть больше или равен уровню привилегий дескриптора того сегмента, к которому мы пытаемся получить доступ.

# Лабораторная работа № 1

***Пакет masm32 и отладчик ollydbg***

**Цель работы:** Ознакомиться со структурой программы на Ассемблере в ОС Windows, научится использовать Ассемблер ml, линкер link и отладчик ollydbg.

**Описание работы**

Программа на языке Ассемблера представляет собой текстовый файл. Каждая строка данного файла содержит символьное представление (мнемокод) одной машинной команды, либо является директивой Ассемблера или макрокомандой. Команды и директивы состоят из латинских букв, прописных или строчных. Русские буквы могут использоваться только в комментариях.

Строка программы, представляющая машинную команду, в общем случае имеет следующий формат:

*[метка:] операция [операнд1[,операнд2[,операнд3]]] [;комментарий]*

Поле «метка» служит для удобства ссылки на данную команду. При указании метки в качестве аргумента, например для команды перехода, Ассемблер на этапе трансляции сам рассчитает необходимое смещение и подставит его в команду. Метка должна заканчиваться символом ‘:’.

Поле «операция» служит для указания мнемокода команды (например, mov, shl).

Поля операнд 1, операнд 2, операнд 3 используются для указания операндов команды. В тех случаях, когда команда имеет два операнда, операнд 1 называют приёмником, а операнд 2 – источником.

Поле «комментарий» служит для записи комментариев. Комментарием считаются все символы, начиная с ‘;’ и заканчивая концом строки.

В программах на Ассемблере допустимы следующие виды констант:

Десятичные числа представляют собой последовательность цифр от 0 до 9;

Двоичные числа – последовательность цифр 0 и 1, заканчивающуюся литерой b или B;

Восьмеричные числа – последовательность цифр от 0 до 7, заканчивающуюся литерами Q,q,O или o.

Шестнадцатеричные числа – последовательность шестнадцатеричных цифр от 0 до 9 и от A до F ( или от a до f), заканчивающуюся литерой h или H.

Ниже приведена программа для ОС Windows. Для любой системы задача простейшей программы состоит из двух частей – корректно получить управление и так же корректно вернуть управление.

**.386** ;используем набор инструкций i386

**.modelflat, stdcall** ;используем плоскую модель памяти

;функции вызываем как stdcall

**includelibkernel32.lib**

;подключаем библиотеку основных функций Windows

**ExitProcessPROTO,:DWORD**

;Прототип функции завершения процесса

**.code**

;Секция кода

**WinmainPROC** ;Точка входа в программу

**push 0** ;Передаём код возврата 0 – корректное завершение

**callExitProcess** ;Завершаем программу

**Winmainendp** ;Закрываем процедуру Winmain

**endWinmain** ;Конец файла

Подготовка программы на Ассемблере к запуску состоит из следующих этапов:

1. Создание файла с текстом программы, например с помощью редактора notepad. Этот файл обычно имеет расширение asm.

2. Трансляция файла. На этом этапе из текстового файла мы получаем объектный файл с расширением obj. Этот файл уже содержит двоичные команды, но ещё не может быть загружен ОС. Трансляция осуществляется следующей командой:

ml /c /coffprimer.asm

Ключ /c обозначает, что необходимо только транслировать программу, ключ /coff указывает, что создаваемый объектный файл должен иметь формат coff (CommonObjectFileFormat).

3. Линковка программы. На этом этапе созданный объектный файл объединяется с другими объектными файлами, а также с библиотечными функциями. При этом должны быть разрешены все символические имена (т.е., если где-то в программе вызывается функция, то её код должен присутствовать в линкуемых файлах или подключаемых библиотеках), иначе произойдёт ошибка компоновки. В результате получается выполнимый файл, имеющий расширение exe. Этот файл уже может быть загружен ОС. Линковка осуществляется следующей командой:

link /SUBSYSTEM:WINDOWS /LIBPATH:путь\_к\_libdirprimer.obj

Аргумент /SUBSYSTEM:WINDOWS означает, что в результате линковки будет сформировано приложение для ОС Windows.

Аргумент /LIBPATH:путь\_к\_libdir указывает путь к директории, хранящей lib-файлы. Обычно это C:\masm32\lib.

4. Запуск программы. Если запуск производится без отладчика, то можно просто запустить программу командой primer.exe или двойным щелчком по exe-файлу в графической оболочке.

При запуске программы без отладчика, если программа завершится с ошибкой, мы не сможем узнать, что именно произошло – стандартное сообщение очень мало информативно (рис. 11).

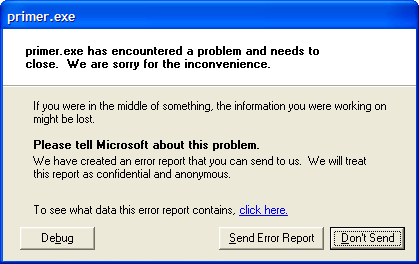


Рис. 11. Стандартное сообщение о некорректно   
завершившейся программе

Так как в первых лабораторных работах не будет производится вывод результатов, для работы понадобится отладчик ollydbg.

Его основные возможности:

* Пошаговое выполнение – отладчик способен выполнять по одной команде, заходя или не заходя в функции.
* Точки останова – пользователь может установить точку останова на выбранной команде, и отладчик остановит выполнение программы, когда будет достигнута указанная команда.
* Просмотр информации – можно получить содержимое регистров, оперативной памяти, получить информацию о потоках, открытых окнах, загруженных библиотеках, и т.д.

Необходимо отметить, что отладчик не может менять программу. Для внесения изменений необходимо прервать выполнение программы, изменить исходный текст, пересобрать программу и снова открыть ее в отладчике.

Вид отладчика, с открытой для отладки программой, приведён на рис.12.

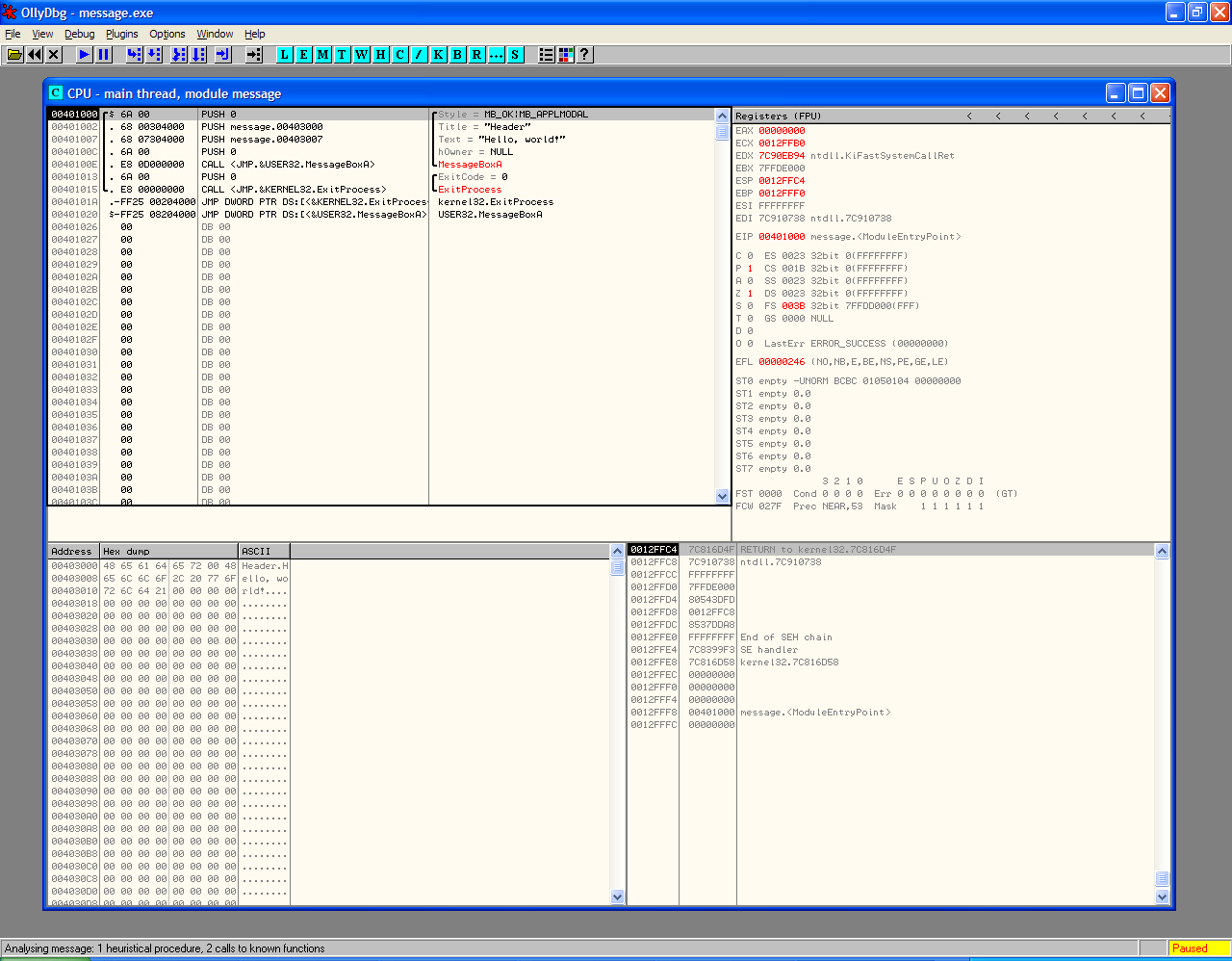


Рис. 12. Отладчик OllyDbg с запущенной под ним программой

Основным окном при отладке программы в OllyDbg является окно CPU. Оно показывает:

* дизассемблированный код программы с подсказками (например, аргументами стандартных функций) и подсвеченной текущей командой;
* содержимое регистров процессора;
* шестнадцатеричный и ASCII дамп области данных;
* стек выполняемой программы.

Начать отладку можно, выбрав отлаживаемый файл с помощью пункта меню File-Open. Для установки/снятия точек останова можно использовать клавишу F2, для пошагового выполнения с заходом в функции (StepInto) – F7, а для пошагового выполнения без захода в функции (StepOver) – F8. Запуск программы на выполнение осуществляется клавишей F9, а сброс (выполнение программы с начала) – комбинацией Ctrl-F2.

С помощью пунктов меню View доступна следующая информация:

* Executablemodules – показывает список загруженных модулей (обычно саму программу и используемые динамические библиотеки);
* Memory – карта адресного пространства;
* Threads – потоки выполняемой программы;
* Windows – окна, принадлежащие выполняемой программе;
* Handles – открытые программой хендлы;
* CallStack – стек вызовов функций для данной программы;
* Breakpoints – установленные точки останова;
* Watches – отслеживаемые значения переменных.

**Контрольные вопросы**

1. Каков общий вид строки программы на Ассемблере?

2. Каким образом в программе объявляются шестнадцатеричные константы?

3. Что происходит с программой на этапе трансляции? На этапе компоновки?

4. Почему объектный файл не может быть загружен операционной системой?

5. Для чего нужен отладчик? Какие основные возможности он предоставляет программисту?

6. В чём различие между командами отладчика StepInto и StepOver?

**Задание к лабораторной работе**

В качестве задания необходимо ввести, оттранслировать, скомпоновать и запустить программу, приведённую ниже. Необходимо уметь запускать программу с отладчиком и без, выполнять трассировку программы, установку/снятие точек останова, а также знать назначение всех компонент окна отладчика CPU. Также необходимо понимать суть процедур трансляции и компоновки, знать назначение аргументов Ассемблера и компоновщика.

**Дополнительное задание** – уберите из программы-примера строку callExitProcess и запустите её в таком виде. Результат поясните.

Программа-пример:

.386

.modelflat, stdcall

includelibkernel32.lib

ExitProcessPROTO ,:DWORD

.data

numbers dd 0,-2,5,8,-3,7,4

.const

s = 7

.code

Winmain PROC

lea ebx,numbers

xor edx,edx

mov ecx,s

cycle:

mov eax,[ebx]

cmp eax,0

jl not\_inc

inc edx

not\_inc: add ebx,4

loop cycle

push 0

call ExitProcess

Winmain endp

end Winmain

# Лабораторная работа № 2

***Арифметические команды***

**Цель работы:** Научиться использовать арифметические команды ассемблера.

**Описание работы.**

Команды двоичной арифметики и команды пересылки данных приведены в табл. 1. Напоминаем, что для команд, имеющих два операнда, первый операнд называется приёмником, второй – источником.

Таблица 1

Команды пересылки данных и двоичной арифметики

|  |  |
| --- | --- |
| Команда | Описание |
| mov пр,ист | Пересылает операнд из источника в приёмник |
| xchg пр,ист | Меняет местами приёмник и источник |
| add пр,ист | Суммирует приёмник и источник, результат сохраняет в приёмник |
| adc пр,ист | Суммирует приёмник, источник и флаг переноса, результат сохраняет в приёмник |
| inc пр | Увеличивает приёмник на 1 |
| sub пр,ист | Вычитает из приёмника источник, результат сохраняет в приёмник |
| sbb пр,ист | Вычитает из приёмника источник и флаг переноса, результат сохраняет в приёмник |
| dec пр | Уменьшает приёмник на 1 |
| mul мн2 | Умножает al, ax, eax (в зависимости от размера операнда) на множитель 2, результат помещает соответственно в ax, dx:ax, edx:eax. Предназначена для беззнаковых чисел |
| imul мн2 | Аналог команды mul для чисел со знаком |
| div делитель | Делит содержимое ax, dx:ax, edx:eax на делитель, при этом частное записывается соответственно в al, ax, eax; а остаток от деление – в ah, dx, edx. Предназначена для чисел без знака |
| idiv делитель | Аналог команды div для чисел со знаком |
| neg пр | Инвертирует знак операнда |
| xadd пр,ист | Заменяет пару команд  xchg пр, ист  add пр, ист |

**Задания к лабораторной работе**

В данной работе необходимо вычислить значение заданного выражения. Переменные, участвующие в выражении, задаются в области данных в виде двойных слов. Для упрощения программы можно считать, что исходные значения не превосходят по модулю 10 000. В конце вычислений результат должен быть сохранён в ячейку памяти. Для хранения промежуточных результатов используются регистры.

Например, вычисление выражения 3•A–B+500 выглядит следующим образом:

.386

.model flat, stdcall

includelib kernel32.lib

ExitProcess PROTO ,:DWORD

.data

A dd 5

B dd 7

res dd ?

.code

Winmain PROC

mov eax,A

mov ebx,3

mul ebx

sub eax,B

add eax,500

mov res,eax

push 0

call ExitProcess

Winmainendp

endWinmain

С помощью отладчика определяем, что после выполнения команды movres, eax в ячейке res имеем FC 01 00 00. Учитывая, что процессоры Интел используют обратный порядок хранения данных, в ячейке res сохранено число 1FCh = 508=3•5–7+500.

Выражения для вычисления:

1. X/Y + 5•Z=

2. A–B+A•C–3=

3. (3–A)/B+C=

4. (A/7–4)•B–C=

5. ((A+B)/2 + C)•5=

6. X•Y–Z/2=

7. 3•P–4•Q+1=

8. (14–Z)•2•X=

Контрольные вопросы

* 1. В чем различие между командами add и adc?
  2. Между различие между командами sub и sbb?

3. В чем различие между командами mul и imul?

4. В каких случаях сбрасывает исключение команда div?

5. В каких случаях сбрасывает исключение команда idiv?

6. Приведите примеры операндов команды add, уставливающие CF=0, OF=1?

7. Приведите примеры операндов команды add, уставливающие CF=1, OF=0?

# Лабораторная работа № 3

***Циклические и разветвляющиеся программы***

**Цель работы:** Научиться использовать команды условного и безусловного переходов, команды организации циклов.

**Описание работы.**

В данной работе будут рассмотрены вопросы программирования разветвляющихся и циклических программ. Для организации разветвляющегося кода используются команды условного и безусловного перехода.

Команда безусловного перехода имеет следующий синтаксис:

*jmp метка*

Эта команда осуществляет безусловный переход на команду, обозначенную меткой.

Команды условного перехода имеют следующий синтаксис:

*jxxx метка*

Эти команды осуществляют переход на указанную метку, если проверяемое условие истинно. Если же условие ложно, выполняется следующая команда.

Для того, чтобы принять решение о переходе, необходимо сформировать условие, которое будет проверяться командой условного перехода. Это можно сделать любой командой, изменяющей состояние арифметических флагов, или командами cmp/test, специально предназначенными для генерации условий. Также команды условного перехода могут анализировать значение регистра ecx.

Команда cmp работает аналогично команде sub, с той разницей, что результат вычитания не записывается в приёмник. Однако она изменяет значения флагов и затем они могут анализироваться командами условного перехода. Также работает и команда test, аналогичная and.

Рассмотрим, как реализовать привычную конструкцию языка высокого уровня:

*if (условие) {*

*//Действие*

*}*

*//Продолжение программы*

На языке ассемблера можно записать

*;проверка условия*

*jxxxcontinue ;переход, если условие ложно*

*;Команды, выполняемые, если условие истинно*

*continue:*

*;Продолжение программы*

Для удобства восприятия разобьем команды условного перехода на группы.

1. Команды, предназначенные для совместного использования с командой cmp (табл. 2). Эти команды обычно используют в программе следующим образом:

cmp оп1, оп2 ;проверка отношения двух операндов

jxxx метка ;переход, если условие истинно

Таблица 2

Команды условного перехода по результату сравнения

|  |  |  |  |
| --- | --- | --- | --- |
| Типы операндов | Мнемокод команды условного перехода | Критерий перехода | Значения флагов для осуществления перехода |
| Любые | Je | оп1==оп2 | zf=1 |
| Любые | Jne | оп1!=оп2 | zf=0 |
| Со знаком | jl/jnge | оп1<оп2 | sf<>of |
| Со знаком | jle/jng | оп1<=оп2 | sf<>of ИЛИ zf=1 |
| Со знаком | jg/jnle | оп1>оп2 | sf=of И zf=0 |
| Со знаком | jge/jnl | оп1>=оп2 | sf=of |
| Без знака | jb/jnae | оп1<оп2 | cf=1 |
| Без знака | jbe/jna | оп1<=оп2 | cf=1 ИЛИ zf=1 |
| Без знака | ja/jnbe | оп1>оп2 | cf=0 И zf=0 |
| Без знака | jae/jnb | оп1>=оп2 | cf=0 |

В мнемокодах легко ориентироваться, если запомнить следующие обозначения:

* e – equal – равно
* g – greater – больше
* l – less – меньше
* a – above – выше (в смысле больше)
* b – below – ниже (в смысле меньше)
* n – not – не (отрицание условия)

При этом следует помнить, что обозначения greater/less относятся к числам со знаком, above/below – к числам без знака, equal – и к тем, и к другим.

2. Команды, осуществляющие условный переход в зависимости от значения флагов, приведены в табл. 3.

Таблица 3

Команды условного перехода по флагам

|  |  |  |
| --- | --- | --- |
| Имя флага | Переход, если флаг = 1 | Переход, если флаг = 0 |
| CF | jc | jnc |
| PF | jp | jnp |
| ZF | jz | jnz |
| SF | js | jns |
| OF | jo | jno |

3. Команды условного перехода по значению регистра cx/ecx. Команды jcxz/jecxz осуществляют переход, если значение регистра cx/ecx равно нулю.

Для организации циклов используются следующие команды:

1. Команда loop-метка работает следующим образом:

Если значение в регистре ecx равно нулю, перейти на следующую команду, иначе уменьшить ecx на 1 и осуществить переход на указанную метку.

2. Команда loope-метка работает следующим образом:

Если значение в регистре ecx равно нулю или флаг ZF=1, перейти на следующую команду, иначе уменьшить ecx на 1 и осуществить переход на указанную метку.

3. Команда loopne-метка работает следующим образом:

Если значение в регистре ecx равно нулю или флаг ZF=0, перейти на следующую команду, иначе уменьшить ecx на 1 и осуществить переход на указанную метку.

Командой loop удобно организовывать циклы, повторяющиеся заданное число раз. Например:

*movecx,число\_итераций*

*cycle:*

*……….. ;Тело цикла*

*loopcycle*

Команды loope и loopne можно использовать для организации цикла, который досрочно прерывается по достижению какого-либо условия (например, нахождения в массиве элемента с указанным значением).

При использовании команд условного перехода и организации циклов часто бывает необходимо временно сохранить значения регистра флагов или регистра ecx. Это можно сделать с помощью команд pushf/pushecx, а восстановление значений регистров осуществить командами popf/popecx.

В данной лабораторной работе необходимо будет работать с последовательностями чисел. Это удобно делать, используя индексную, или базово-индексную, адресацию. При использовании индексной адресации в индексный регистр (esi или edi) заносится адрес начала массива, а в конце каждой итерации этот адрес увеличивается на значение, равное размеру в байтах одного элемента массива. При использовании базово-индексной адресации адрес начала массива заносится в базовый регистр (ebx или ebp), а в индексном регистре находится смещение текущего элемента относительно начала массива. Этот режим удобен при работе с несколькими массивами одновременно – индекс остаётся неизменным, меняется только база (адрес начала массива).

**Контрольные вопросы**

1. Покажите, как на языке Ассемблера реализовать конструкцию языка высокого уровня:

*if (условие) {*

*//Действия, если условие истинно*

*}*

*else {*

*//Действия, если условия ложно*

*}*

2. Какие существуют группы команд условного перехода?

3. Можно ли обойтись без команды безусловного перехода? Ответ аргументируйте.

4. Покажите, как можно на языке Ассемблера организовать вложенные циклы.

**Задания к лабораторной работе**

Во всех заданиях необходимо организовать работу со следующими входными данными:

inpdd 0,4, – 13,3,…. ;Некоторое количество двойных слов со знаком или без

N = 4 ;Размер массива

1. Найти максимум значений (числа без знака).

2. Найти минимум значений (числа со знаком).

3. Найти число, наименьшее по модулю (числа со знаком).

4. Вычислить среднее арифметическое (числа без знака).

5. Найти сумму модулей чисел (числа со знаком).

6. Найти количество положительных чисел (числа со знаком).

7. Найти индекс первого вхождения нуля в последовательность чисел.

8. Переставить числа в обратном порядке.

9. Заменить все положительные числа на 1, отрицательные – на –1 (числа со знаком).

10. Поменять местами пары соседних чисел.

# Лабораторная работа № 4

***Логические команды***

**Цель работы:** Научиться использовать команды логических операций.

**Описание работы**

Логическими называются операции, которые работают со своими операндами не как с числами в двоичной системе счисления, а как с последовательностью бит. Список логических команд приведён в табл. 4.

Таблица 4

Логические команды

|  |  |
| --- | --- |
| Команда | Описание |
| andпр, ист | Выполняет побитовую конъюнкцию над приёмником и источником, результат записывает в приёмник |
| testпр, ист | Полностью аналогична команде and, однако не заносит результат в приёмник |
| or пр, ист | Выполняет побитовую дизъюнкцию над приёмником и источником, результат записывает в приёмник |
| xor пр, ист | Выполняет побитовое суммирование по модулю два над приёмником и источником, результат записывает в приёмник |
| not пр | Выполняет побитовую отрицание над приёмником |
| shl оп, счётчик | Сдвигает операнд влево логически на указанное число разрядов |
| shr оп, счётчик | Сдвигает операнд вправо логически на указанное число разрядов |
| sal оп, счётчик | Сдвигает операнд влево арифметически на указанное число разрядов |
| sar оп, счётчик | Сдвигает операнд вправо арифметически на указанное число разрядов |
| rol оп, счётчик | Циклически сдвигает операнд влево на указанное число разрядов |
| ror оп, счётчик | Циклически сдвигает операнд вправо на указанное число разрядов |
| rcl оп, счётчик | Циклически сдвигает операнд влево через флаг CF на указанное число разрядов |
| rcr оп, счётчик | Циклически сдвигает операнд вправо через флаг CF на указанное число разрядов |
| shld оп1, оп2, счётчик | Сдвигает операнды влево на указанное число разрядов, заполняя справа операнд 1 битами, вытесненными слева из операнда 2 |
| shrd оп1, оп2, счётчик | Сдвигает операнды вправо на указанное число разрядов, заполняя слева операнд 1 битами, вытесненными справа из операнда 2 |

Пояснения:

* Для команд сдвига величина сдвига *счётчик* либо задаётся непосредственно в команде, либо заносится в регистр cl. При этом в обоих случаях величина сдвига не должна превышать 31 (фактически, значение сдвига задают пять младших бит регистра cl, остальные биты игнорируются).
* Команда логического сдвига вправо отличается от команды арифметического сдвига вправо тем, что при логическом сдвиге вправо в старший бит регистра заносится 0, а при арифметическом сдвиге вправо – значение знакового бита исходного числа.
* Команды циклического сдвига через флаг CF работают следующим образом – на освободившиеся место заносится значение флага CF, а «вытесненный» из регистра бит становится новым значением флага CF.

Приведём примеры использования логических команд. Напомним, что биты в двоичных числах нумеруются справа налево, начиная с нуля.

1. Установить 3 бит регистра eax в 1.

mov ebx,1

shl ebx,3

or eax,ebx

2. Установить 15 бит регистра eax в 0

mov ebx,1

shl ebx,15

not ebx

andeax,ebx

3. Проверить, установлен ли 8 бит регистра eax

mov ebx,1

shl ebx,8

test eax,ebx

jziszero ;Если бит сброшен – переход на label1

….

iszero:…

**Контрольные вопросы**

1. Каким образом можно установить указанный бит в 1? Установить в 0? Инвертировать?

2. Чем отличается логический сдвиг от арифметического?

3. Чем отличается команды ror/rol и rcr/rcl?

4. Для чего нужна команда test?

**Задания к лабораторной работе**

Во всех заданиях необходимо организовать работу со следующими входными данными:

inpdd 0,4,-13,3,…. ;Некоторое количество двойных слов

N = 4 ;Размер массива

1. Пусть задан массив resdbNdup (0). Заполните этот массив следующим образом: если соответствующее двойное слово из inp имеет чётное число бит, установленных в 1, запишите в массив 0, если нечётное – 1.

2. Пусть задан массив resdbNdup (0). Заполните этот массив следующим образом: в каждую ячейку запишите число бит соответствующего двойного слова массива inp, установленных в 1.

3. Пусть задан массив ruledbNdup (0). Сдвиньте каждое двойное слово массива inp циклически влево на число разрядов, заданное соответствующей ячейкой массива rule.

4. Рассматривая массив inp как битовую последовательность, найдите длину самой длинной последовательности единиц.

5. Дополните массив inp битами контроля чётности по следующему правилу: после каждых семи бит исходного массива идёт восьмой бит, установленный в 0, если в семи битах присутствует нечётное число единиц, или установленный в 1, если чётное. Результат запишите в новый массив байтов. Если число битов не кратно семи, дополните массив справа нулями.

6. Рассматривая массив inp как битовую последовательность, разделите его на 4-битовые участки и найдите результат «исключающего или» всех этих участков.

7. Рассматривая массив как битовую последовательность, разделите её на участки по два бита и подсчитайте число диграмм (00, 01, 10, 11).

# Лабораторная работа № 5

**Работа со строками**

**Цель работы:** Научиться программировать строковые операции с использованием Ассемблера. Научиться выводить результат работы с помощью функции MessageBoxA.

**Описание работы**

Строки в языке Ассемблера задаются в виде ASCIIZ, т.е. представляют собой последовательность символов, заканчивающуюся нулём. Строку можно объявить в области данных, например, следующим образом:

my\_stringdb “Thisisthestring!”,0

В строках можно использовать русские или латинские буквы, цифры, знаки препинания. Перевод строки можно задать байтами с кодами 13, 10, табуляцию – байтом с символом 9:

my\_string2 db “Строка с ”,13,10,”переводом на новую строку”,0

my\_string3 db “Строка с ”,9,”табуляцией”,0

Для удобства операций со строками можно использовать команды цепочечной обработки данных (приведены в табл.5), а также команды loop/loope/loopne, которые рассматривались в лабораторной работе № 3.

Напомним, что команды цепочечной обработки данных в качестве источника рассматривают область памяти по адресу – ds:esi, а в качестве приёмника – es:edi. При работе эти команды автоматически корректируют значение индексных регистров, увеличивая (если флаг направления DF=0) или уменьшая (DF=1) их на размер элемента. Если к цепочечной команде добавляется постфикс b – байт, то значение индексных регистров изменяется на 1, если постфикс w – слово, то на 2, и если постфикс d – двойное слово, то на 4. Флаг направления можно установить командой std и сбросить командой cld.

Таблица 5

Команды цепочечной обработки данных

|  |  |
| --- | --- |
| Команда | Описание |
| movs | Копирует приёмник с источником |
| cmps | Сравнивает (аналогично cmp) приёмник с источником |
| scas | Сравнивает элементы строки с элементом, находящегося в al/ax/eax |
| lods | Загружает элемент строки в al/ax/eax |
| stos | Сохраняет элемент из al/ax/eax в строку |

Вывод информации пользователю можно осуществить с помощью функции MessageBoxA, отображающей окно сообщения. Литера A в конце имени указывает, что данная функция предназначена для обработки ASCII-строк. Функция MessageBoxA описана следующим образом:

int MessageBox(  
    HWND hWnd,

    LPCTSTR lpText,

    LPCTSTR lpCaption,

    UINT uType

);

Назначение параметров

hWnd

[in] Хендл окна-владельца. Наша программа не имеет окна, поэтому тут передаём 0.

lpText

[in] Указатель на строку-сообщение.

lpCaption

[in] Указатель на строку-заголовок.

uType

[in] Константы (какие кнопки и какой значок рисовать). Чтобы отобразить окно без значков и с кнопкой ok, передаём 0.

Для корректного подключения MessageBoxA в программу на Ассемблере необходимо добавить следующие строки:

includelib user32.lib

MessageBoxA PROTO ,:DWORD, :DWORD, :DWORD, :DWORD

Пример вызова функции MessageBoxA:

push 0

push offset MsgBoxCaption

push offset MsgBoxText

push 0

callMessageBoxA

Предполагается, что MsgBoxCaption и MsgBoxText – ASCIIZ-строки, описанные в секции данных программы.

**Контрольные вопросы**

1. Что такое ASCIIZ-строка?

2. Как можно задать строку в секции данных программы на Ассемблере?

3. Где находятся операнд-источник и операнд-приёмник в строковых командах?

4. Для чего служит флаг направления DF?

5. Для чего предназначена функция MessageBoxA? Какие аргументы ей передаются?

**Задания к лабораторной работе**

Во всех заданиях необходимо организовать работу со строками, описанными в секции данных, и вывести результат с помощью MessageBoxA.

«Словом» называется последовательность алфавитно-цифровых символов, ограниченная с двух сторон пробелами либо началом/концом строки.

«Предложением» называется последовательность слов, завершающаяся точкой либо концом строки.

«Палиндромом» называется слово, которое читается одинаково и слева направо, и справа налево.

1. Дана строка, состоящая из алфавитно-цифровых символов и пробелов. Необходимо заменить в строке строчные латинские буквы на прописные. Указание – используйте команду xlat.

2. Дана строка, состоящая из алфавитно-цифровых символов и пробелов. Необходимо заменить в строке прописные латинские буквы на строчные. Указание – используйте команду xlat.

3. Дана строка, состоящая из алфавитно-цифровых символов и пробелов. Необходимо инвертировать порядок знаков в слове.

4. Дана строка, состоящая из алфавитно-цифровых символов и пробелов. Необходимо инвертировать порядок слов в строке.

5. Дана строка, состоящая из алфавитно-цифровых символов и пробелов. Необходимо вывести слова, являющиеся палиндромами, в одной строке, разделенные пробелом.

6. Дана строка, состоящая из маленьких латинских букв, пробелов и точек. Необходимо заменить первую букву первого слова в предложении на прописную для всех предложений.

7. Даны две строки. Необходимо сравнить их лексикографически. Вывести «больше», если первая строка больше второй, «меньше», если меньше, «равно», если строки идентичны.

8. Дана строка, состоящая из алфавитно-цифровых символов и пробелов. Необходимо удалить лидирующие и концевые пробелы. Если слова разделены больше чем одним пробелом, оставить только один пробел.

# Лабораторная работа № 6

***Организация функций***

**Цель работы:**Изучить организацию функций на языке Ассемблера, способы передачи параметров в функцию и возврата результатов. Научится реализовывать программы повышенной сложности с помощью функциональной декомпозиции.

**Описание работы**

Подпрограммой называется поименованная или иным образом идентифицированная часть компьютерной программы, содержащая описание определённого набора действий. Подпрограмма может быть многократно вызвана из разных частей программы. Для оформления и использования подпрограмм в языках программирования существуют специальные синтаксические средства.

Если подпрограмма возвращает некоторое значение как результат своей работы, её называют функцией, если не возвращает – процедурой.

Подпрограммы часто используются для многократного выполнения стереотипных действий над различными данными. Подпрограмма обычно имеет доступ к объектам данных, описанным в основной программе (по крайней мере, к некоторым из них), поэтому для того, чтобы передать в подпрограмму обрабатываемые данные, их достаточно присвоить, например, глобальным переменным. Но такой путь не особенно удобен и чреват ошибками.

Для обеспечения контролируемой передачи параметров в подпрограмму и возврата результатов из неё используется механизм параметров. В языках высокого уровня параметры описываются при описании подпрограммы (в её заголовке) и могут использоваться внутри процедуры аналогично переменным, описанным в ней. При вызове процедуры значения каждого из параметров указываются в команде вызова (обычно после имени вызываемой подпрограммы).

В Ассемблере организация передачи параметров подпрограмме осуществляется вручную. Существуют следующие способы передачи параметров в функцию:

1. Через регистры – в описании функции оговаривается, какой регистр какой параметр содержит. Перед вызовом параметры загружаются в соответствующие регистры, затем извлекаются оттуда подпрограммой. Если регистры содержали промежуточные данные, их необходимо сохранить, например в стеке. (Таким образом реализованы, например, системные вызовы Linux).

2. Через общую область секции данных – может применяться только при вызове функций в пределах одной программы. Не очень удобный способ.

3. Через стек – этот способ очень широко применяется различными компиляторами языков высокого уровня.

При передаче параметров через стек необходимо оговорить два вопроса:

В каком порядке аргументы помещаются в стек. Аргументы в стек можно помещать в прямом порядке (начиная с первого аргумента, заканчивая последним); либо в обратном (начиная с первого, заканчивая последним). Предпочтительным является обратный порядок аргументов, так как он позволяет легко реализовать функции, где число и размер операндов зависят от первого операнда (например, printf).

Какой код очищает стек. Это может делать либо код процедуры (вызываемый код) с помощью команды retn; либо код основной программы (вызывающий код) с помощью команды addesp,n. Когда стек чистит вызывающий код, это приводит к тому, что каждый вызов функции сопровождается командой очистки стека. Если же стек очищает вызываемый код, то команда очистки стека будет только одна, и код получится более компактным.

Различные варианты передачи параметров через стек представлены в табл.6.

Таблица 6

Варианты передачи параметров через стек

|  |  |  |
| --- | --- | --- |
| Имя | Порядок помещения аргументов в стек | Какой код очищает стек |
| c | Обратный | Вызывающий |
| pascal | Прямой | Вызываемый |
| stdcall | Обратный | Вызываемый |

Пример передачи параметров через стек stdcall-функции:

push param\_n

…

push param\_2

pushparam\_1

callstdcall\_func

Возврат значения функцией обычно организован через регистр. Например, стандарт stdcall предполагает, что возвращаемое значение будет находиться в регистре eax.

При написании функции, работающей с параметрами через стек, очень удобно сохранить адрес, с которого начинаются параметры, в регистре указателя кадра стека (ebp).

Пример (предполагается, что параметры и локальные переменные имеют размер двойного слова):

movebp, esp

moveax, [ebp+4] ;Загрузка параметра №1

movebx, [ebp+8] ;Загрузка параметра №2

Таким образом можно легко работать с параметрами и при этом выделять место в стеке под локальные переменные. К ним можно обращаться тем же способом:

movecx, [ebp-4] ;Чтение первой локальной переменной

movedx, [ebp-8] ;Чтение второй локальной переменной

При этом перед работой с локальными переменными необходимо выделить под них место командой:

subesp, размер\_переменных

а после работы – очистить стек командой:

addesp, размер\_переменных

В языке Ассемблера подпрограмма может никак не оформляться – достаточно описать метку, по которой будет происходить переход на подпрограмму. Однако в синтаксисе предусмотрена специальная конструкция для описания подпрограмм:

имя\_процедуры proc

.... ;команды процедуры

имя\_процедуры endp

Процедуры можно разместить до текста основной программы или после него – это не имеет особого значения. Можно даже поместить процедуру посреди кода основной программы, но тогда её придётся «обходить» командой jmp.

Если код подпрограммы находится в одном исходном файле, а вызывающий код – в другом, то в таком случае необходимо дать указания транслятору, чтобы он включил в объектные файлы специальную информацию для компоновщика. Это можно сделать с помощью директив public и extrn. В модуле, в котором находится код процедуры, необходимо объявить её директивой public:

publicmy\_proc ,

а в вызывающем модуле объявить эту метку как внешнюю:

extrnmy\_proc

После этого можно будет откомпилировать исходные файлы по отдельности и объединить объектные файлы с помощью компоновщика.

**Контрольные вопросы**

1. Что такое подпрограмма? Для чего нужны подпрограммы?

2. Какие вы знаете способы передачи параметров в функцию?

3. Какой командой может выполнить очистку стека вызывающий код/вызываемый код?

3. Как можно эффективно организовать работу с параметрами, переданными в стеке?

4. Как можно поместить код подпрограммы и вызывающий код в разные исходные файлы?

**Задания к лабораторной работе**

Во всех заданиях предполагается разработка программы, состоящей из основной части и одной или нескольких подпрограмм. В отчёте необходимо для каждой подпрограммы указать её название, назначение, перечень передаваемых и возвращаемых значений, алгоритм работы.

Передавать параметры функциям необходимо через регистры или через стек. При этом при передаче через регистры необходимо указать, в каком из регистров какой параметр находится; а при передаче через стек – в каком порядке параметры помещаются в стек и кто производит его очистку.

1. Разработать программу, находящую минимальную строку из набора строк (в лексикографическом смысле). Указание: разработать функцию, производящую лексикографическое сравнение двух строк.

2. Разработать программу, находящую максимальную строку из набора строк (в лексикографическом смысле). Указание: разработать функцию, производящую лексикографическое сравнение двух строк.

3. Разработать программу, выводящую на экран содержимое двойного слова в шестнадцатеричном виде с помощью MessageBoxA. Указание: разработать функцию, преобразующую отдельный байт.

4. Разработать программу, переводящую строковое представление шестнадцатеричного числа в двоичное. Строка задана в виде константы. Указание: разработать функцию, преобразующую пару шестнадцатеричных цифр в отдельный байт.

5. Разработать программу, получающую целочисленные длины трёх отрезков и выводящую с помощью MessageBoxA «Да», если из них можно составить треугольник, и нет, если нельзя. Указание: разработать процедуру, проверяющую, превосходит ли сумма двух чисел третье число.

6. Задана замкнутая ломаная без самопересечений. Проверить, является ли она выпуклым многоугольником. Указание: разработать функцию, проверяющую, по какую сторону отрезка лежит точка.

7. Разработать программу, разбивающую строку на слова, и выводящую только слова, являющиеся палиндромами. Указание: разработать две подпрограммы, одна из которых разбивает строку на две части – первое слово и остаток строки, вторая – проверяет, является ли строка палиндромом.

8. Напишите программу, реализующую пирамидальную сортировку (heapsort) двойных слов со знаком. Указание: разработайте процедуру, выполняющую пропихивание («pushdown») элемента в пирамиду.

# Лабораторная работа № 7

***Использование Ассемблера в ОС Linux***

**Цель работы:** Получить представление о программировании на Ассемблере в ОС Linux, научиться писать программы в синтаксисе AT&T, научиться использовать системные вызовы ОС Linux.

**Описание работы**

Ранее в качестве основного средства трансляции программ на Ассемблере использовался пакет masm (Макроассемблер). Этот пакет использует синтаксис ассемблерных команд архитектуры x86, принятый компанией Intel. В ОС Linux (и иных \*nix-подобных ОС) используется GNU-ассемблер (gas, as). Этот Ассемблер использует синтаксис компании AT&T. Он отличается от синтаксиса Intel следующим образом:

* Непосредственные опеpанды AT&T пишутся после `$'; непосредственные операнды Intel не выделяются (Intel `push 4' соответствует AT&T `pushl $4'). Регистровые операнды AT&T пишутся после `%'; регистровые операнды Intel не выделяются. Абсолютные операнды AT&T (по сравнению с операндами, заданными относительно текущей инструкции) jump/call пишутся после `\*'; они не выделяются в синтаксисе Intel.
* В синтаксисах AT&T и Intel используются противоположные порядки исходных операндов и целевых операндов. Intel `add eax, 4' соответствует `addl $4, %eax'. Порядок операндов в синтаксисе AT&T совместим с предыдущими Unix-ассемблерами.
* В синтаксисе AT&T размер операндов определяется последним знаком имени кода операции. Суффиксами кодов операций являются `b', `w', и `l', определяющие байт (8-бит), слово (16-бит) и long (32-бит) указатели. В синтаксисе Intel это указывается префиксами операндов памяти (не самими кодами операций) при помощи `byte ptr', `word ptr' и `dword ptr'. Так, Intel `mov al, byte ptr FOO' соответствует `movb FOO, %al' в синтаксисе AT&T.
* В синтаксисе AT&T длинные переходы и вызовы записываются как lcall/ljmp $SECTION, $OFFSET, а в синтаксисе Intel – call/jmp far SECTION:OFFSET. Также, инструкция дальнего возврата в синтаксисе AT&T пишется lret $STACK-ADJUST, а в синтаксисе Intel – ret far STACK-ADJUST.

Сборка и запуск программы, написанной на Ассемблере, в ОС Linux выполняются следующим образом:

(предположим, что в файле hello.s находится ассемблерный листинг)

as -s hello.s -o hello.o

ld -s hello.o -o hello

./hello

Первая команда – это вызов ассемблера, который транслирует листинг в объектный файл, вторая – вызов линкера, который создаёт выполнимый файл, третья – запуск программы на выполнение.

В gas, как и в masm, необходимо явно объявить в программе точку входа. Если в masm выполнение программы начинается с процедуры winmain, то в gas – с метки \_start. Эта метка должна быть объявлена как глобальная, чтобы линкер смог установить точку входа в приложение на неё (подробнее см. пример).

Код в программе для gas пишется в секции .text, данные объявляются в секции .data. Ключевые слова для объявления данных – .byte объявляет байт, .word – слово, .long – двойное слово, .double – число с плавающей точкой (длина 8 байт), .string – строку. Для удобства ссылок на данные можно использовать метки.

Примеры объявления данных:

.data

A: .long 5

.byte 4,7

S: .string “abc”

Взаимодействие программы с операционной системой осуществляется с помощью так называемых системных вызовов (syscalls). Эти вызовы реализуются обработчиком прерывания 0x80. Параметры передаются через регистры следующим образом: в eax помещается номер системного вызова, параметры помещаются в регистры ebx, ecx, edx, esi, edi, ebp, результат возвращается в регистре eax. Если результат больше нуля либо равен ему, то вызов завершился успешно, если меньше нуля – произошла ошибка. Чтобы получить код ошибки, необходимо инвертировать знак содержимого регистра eax. Номера системных вызовов можно найти в файле asm/unistd.h, коды ошибок – asm/errno.h. Обычно include-файлы находятся в директории /usr/include.

Далее приводятся описания системных вызовов, необходимых для выполнения лабораторной работы. При описании системного вызова указываются номер системного вызова (загружается в регистр eax), его имя, синтаксис (типы передаваемых параметров и возвращаемых значений) и семантика (назначение системного вызова, передаваемых параметров и возвращаемых значений).

**1. sys\_exit**

Синтаксис: int sys\_exit(int status)

Семантика: Завершает текущий процесс. status – код возврата.

**2. sys\_fork**

Синтаксис: int sys\_fork()

Семантика: разветвляет текущий процесс, т.е. создаёт его точную копию.

Если после вызова eax=0, мы находимся в дочернем процессе, иначе – в родительском, и тогда eax содержит pid дочернего процесса.

**3. sys\_read**

Синтаксис: ssize\_t sys\_read(unsigned int fd, char \* buf, size\_t count)

Семантика: Считать данные из заданного дескриптора. Параметры – номер дескриптора, указатель на буфер в памяти, количество байт для чтения. Возвращает количество прочитанных байт.

**4. sys\_write**

Синтаксис: ssize\_t sys\_write(unsigned int fd, const char \* buf, size\_t count)

Семантика: Пишет данные в заданный дескриптор. Параметры – номер дескриптора, указатель на буфер, количество байт для записи. Возвращает количество записанных байт.

**5. sys\_open**

Синтаксис: int sys\_open(const char \* filename, int flags, int mode)

Семантика: Открывает файл для чтения/записи. Параметры – указатель на имя файла, флаги (0-чтение, 1-запись, 2-чтение/запись), права доступа для вновь создаваемого файла. Возвращает номер дескриптора

**6. sys\_close**

Синтаксис: sys\_close(unsigned int fd)

Семантика: Закрыть дескриптор. Параметр – номер дескриптора.

**7. sys\_creat**

Синтаксис: int sys\_creat(const char \* pathname, int mode)

Семантика: Создать файл. Параметры – указатель на имя файла, права для вновь создаваемого файла. Фактически этот вызов является вызовом sys\_open с параметром flags=01102.

**8. sys\_link**

Синтаксис: int sys\_link(const char \* oldname, const char \* newname)

Семантика: создаёт жёсткую ссылку на указанный файл. Параметры – указатель на старое имя, указатель на новое имя.

**9. sys\_unlink**

Синтаксис: int sys\_unlink(const char \* pathname)

Семантика: уничтожает жёсткую ссылку, а если это была последняя ссылка, то и файл тоже. Параметр – указатель на имя файла.

**10. sys\_time**

Синтаксис: int sys\_time(int \* tloc)

Семантика: возвращает время в секундах с начала эпохи Unix (1 сентября 1970 г). Параметр – указатель на 32-разрядное целое, в которое будет записано число секунд. Возвращает число секунд.

**11. sys\_lseek**

Синтаксис: off\_t sys\_lseek(unsigned int fd, off\_t offset, unsigned int origin)

Семантика: устанавливает позицию чтения/записи в файле. Параметры – дескриптор файла, смещение, позиция откуда смещаться (начало, текущая позиция, конец). Возвращает смещение позиции от начала файла.

**12. sys\_getpid**

Синтаксис: int sys\_getpid(void)

Семантика: возвращает pid нашего процесса.

Пример программы для ОС Linux:

.text #Секция кода

.globl \_start

#метка \_start должна быть объявлена как глобальная

#Линкер устанавливает на неё точку входа

\_start:

movl $0x5,%eax #Открыть файл

movl $filename,%ebx #указатель на имя

movl $01102,%ecx

#Открыть для чтения/записи,

#если не существует – создать,

#если существует – стереть всё содержимое

movl $0644,%edx

#Права доступа к создаваемому файлу

int $0x80 #Ядро Линукса

movl %eax,%ebx #Дескриптор файла

movl $len,%edx #Длина строки

movl $message,%ECX #указатель на строку

movl $0x4,%EAX #Писать в дескриптор

int $0x80 #Ядро Линукса

movl $0x6,%eax #Закрыть дескриптор

int $0x80 #Ядро Линукса

movl $0x1,%eax #Выйти из программы

xorl %ebx,%ebx #Код возврата 0

int $0x80 #Ядро Линукса

.data #Секция данных

message: .string "Hello, world!\n" #Данные

len = .-message-1 #Длина данных, исключая концевой \0

filename: .string "myfile.txt" #Имяфайла

**Контрольные вопросы**

1. Как будет выглядеть в синтаксисе AT&T команда moveax, 5?

2. Поясните назначение суффиксов b,w,l, приписываемых к команде.

3. Для чего служит прерывание int $0x80?

4. Каким образом передаются параметры системному вызову? Каким образом возвращается значение?

5. Как можно узнать, что при выполнении системного вызова произошла ошибка?

4. Что показывает метка \_start? Почему её необходимо объявлять как глобальную?

5. Объявите строку со значением “qwerty”, на которую можно ссылаться с помощью метки string1.

**Задания к лабораторной работе**

Во всех заданиях имя файла (файлов) задаётся в виде строки в секции данных программы. Допустимо (хотя и не рекомендуется) читать и записывать файлы по одному символу. Выводить числовые значения, если не указано иное, необходимо в шестнадцатеричном виде.

1. Перевести текстовый файл из формата Windows в формат Linux (заменить все пары символов 0x0d 0x0a на 0x0a)

2. Перевести текстовый файл из формата Linux в формат Windows (заменить все символы 0x0a на пары 0x0d 0x0a).

3. Вывести на стандартный вывод размер файла на диске.

4. Создать файл, создать жёсткую ссылку на него, оригинал уничтожить.

5. Создать форк-процесс, при этом уничтожить процесс-родитель, процессу-потомку выдать на стандартный вывод свой pid.

6. Выдать на стандартный вывод текущее время (вывод осуществить в секундах, без форматирования).

7. Закодировать файл по правилу "кодирование длинных серий" (RLE).

При таком кодировании первый байт обозначает число повторений второго байта, третий – четвёртого и т.д. Если длина серии превосходит 255, то серия кодируется как несколько серий.

Пример (в шестнадцатеричном виде):

Входной файл: 00 7e 7e 0a 11 11 11

Выходной файл: 01 00 02 7e 01 0a 03 11

8. Декодировать файл по правилу "кодирование длинных серий" (RLE).

Пример (в шестнадцатеричном виде):

Входной файл: 01 00 02 7e 01 0a 03 11

Выходной файл: 00 7e 7e 0a 11 11 11

# Лабораторная работа № 8

***Использование Ассемблера в ОС Windows***

**Цель работы:** Научиться писать программы, использующие основные возможности ОС Windows.

**Описание работы**

В операционной системе Windows взаимодействие программы с ОС осуществляется с помощью вызова функций Windows API. Эти функции экспортируются из стандартных динамических библиотек Windows. Вызов функций осуществляется по формату stdcall , т.е. параметры помещаются в стек в обратном порядке (начиная с последнего), командой call осуществляется вызов функции, стек очищает вызываемый код. В регистре eax функция возвращает результат выполнения.

Возникновение ошибки можно определить, если возвращаемое значение равно специальной константе, например INVALID\_HANDLE\_VALUE.

Функции WindowsAPI, работающие со строками, представлены в двух вариантах:

Вариант, работающий с ASCII строками, оканчивается на литеру A. Вариант, работающий с UNICODE-строками – на литеру W. В наших лабораторных работах мы будем использовать ASCII-строки.

**Пример программы на Ассемблере, использующей Windows API:**

.386

.model flat, stdcall

includelib kernel32.lib

includelib user32.lib

ExitProcess PROTO ,:DWORD

MessageBoxA PROTO ,:DWORD, :DWORD, :DWORD, :DWORD

.data

MsgBoxCaption db "Заголовок",0

MsgBoxText db "Текстсообщения!",0

.const

NULL equ 0

MB\_OK equ 0

.code

Winmain PROC

push MB\_OK

push offset MsgBoxCaption

push offset MsgBoxText

push NULL

call MessageBoxA

push 0

call ExitProcess

Winmain endp

end Winmain

Из примера видно, что для использования функций в программе необходимо подключить библиотеку, в которой находятся эти функции, с помощью директивы includelib, и объявить сами функции с помощью директивы PROTO.

Команды трансляции, сборки и запуска для примера:

>c:\masm32\bin\ml /c /coff hello.asm

>c:\masm32\bin\link /SUBSYSTEM:Windows /LIBPATH:c:\masm32\lib hello.obj

>hello.exe

Далее приводится описание функций, необходимых для выполнения лабораторной работы:

**CreateFile**

CreateFile – создаёт или открывает файл. Возвращает хендл на открытый файл (или код ошибки)

HANDLE CreateFile(

LPCTSTR lpFileName,

DWORD dwDesiredAccess,

DWORD dwShareMode,

LPSECURITY\_ATTRIBUTES lpSecurityAttributes,

DWORD dwCreationDisposition,

DWORD dwFlagsAndAttributes,

HANDLE hTemplateFile

);

Параметры

lpFileName

[in] указатель на строку, содержащую имя файла

dwDesiredAccess

[in] Права доступа к файлу – GENERIC\_READ, GENERIC\_WRITE

dwShareMode

[in] Режим разделения доступа – передаём 0.

lpSecurityAttributes

[in] Атрибуты безопасности. Передаём 0.

dwCreationDisposition

[in] Что делать, если файл существует/не существует

|  |  |
| --- | --- |
| Константа | Значение |
| CREATE\_ALWAYS | Создать, если существует – уничтожить и создать заново |
| CREATE\_NEW | Создать новый, если существует – вернуть ошибку |
| OPEN\_ALWAYS | Открыть, если не существует – создать новый |
| OPEN\_EXISTING | Открыть, если не существует – вернуть ошибку |
| TRUNCATE\_EXISTING | Если существует – уничтожить и создать заново, если нет – вернуть ошибку. |

dwFlagsAndAttributes

Передаём FILE\_ATTRIBUTE\_NORMAL

HANDLE hTemplateFile

Передаём 0

Примеры:

Открытие для чтения

#include<windows.h>

#include <stdio.h>

HANDLE hFile;

hFile = CreateFile(TEXT("myfile.txt"), // file to open

GENERIC\_READ, // open for reading

FILE\_SHARE\_READ, // share for reading

NULL, // default security

OPEN\_EXISTING, // existing file only

FILE\_ATTRIBUTE\_NORMAL, // normal file

NULL); // no attr. template

if (hFile == INVALID\_HANDLE\_VALUE)

{

printf("Could not open file (error %d)\n", GetLastError());

return 0;

}

Открытие для записи:

#include <windows.h>

#include <stdio.h>

HANDLE hFile;

hFile = CreateFile(TEXT("myfile.txt"),// file to create

GENERIC\_WRITE, // open for writing

0, // do not share

NULL, // default security

CREATE\_ALWAYS, // overwrite existing

FILE\_ATTRIBUTE\_NORMAL | // normal file

FILE\_FLAG\_OVERLAPPED, // asynchronous I/O

NULL); // no attr. template

if (hFile == INVALID\_HANDLE\_VALUE)

{

printf("Could not open file (error %d)\n", GetLastError());

return 0;

}

**ReadFile**

Читаем данные из файла.

BOOL ReadFile(

HANDLE hFile,

LPVOID lpBuffer,

DWORD nNumberOfBytesToRead,

LPDWORD lpNumberOfBytesRead,

LPOVERLAPPED lpOverlapped

);

Параметры

hFile

[in] Хендл, из которого читаем.

lpBuffer

[out] Указатель на буфер для обмена данными.

nNumberOfBytesToRead

[in] Размер буфера.

lpNumberOfBytesRead

[out] указатель на переменную типа двойное слово, в которую запишем, сколько байт прочитали.

lpOverlapped

Здесь передаётся 0

Возвращаемые значения.

Если завершено успешно, возвращается ненулевое значение. Если неудачно, то 0.

Пример:

//Попытка синхронной операции чтения.

bResult = ReadFile(hFile,

&inBuffer,

nBytesToRead,

&nBytesRead,

NULL) ;

// Проверяем, не конец ли файла.

if (bResult&&nBytesRead == 0)

{

//Достигли конца файла.

}

**SetFilePointer**

перемещает указатель чтения/записи для открытого файла

DWORD SetFilePointer(

HANDLE hFile,

LONG lDistanceToMove,

PLONG lpDistanceToMoveHigh,

DWORD dwMoveMethod

);

Параметры

hFile

[in] Хендлфайла

lDistanceToMove

Насколько двигаем

lpDistanceToMoveHigh

Здесь передаём 0

dwMoveMethod

[in] Откуда двигаем

|  |  |
| --- | --- |
| Константа | Значение |
| FILE\_BEGIN | От начала |
| FILE\_CURRENT | От текущего значения |
| FILE\_END | От конца |

**WriteFile**

функция записи данных в файл

BOOL WriteFile(

HANDLE hFile,

LPCVOID lpBuffer,

DWORD nNumberOfBytesToWrite,

LPDWORD lpNumberOfBytesWritten,

LPOVERLAPPED lpOverlapped

);

Параметры

hFile

[in] Хендл файла

lpBuffer

[in] Указатель на буфер

nNumberOfBytesToWrite

[in] Размер буфера

lpNumberOfBytesWritten

[out] Указатель на переменную типа двойное слово, в которую пишется, сколько записали байт.

lpOverlapped

[in] Здесь передаём 0.

Возвращаемые значения

Если успешно, то возвращаем не 0, если неудачно, то 0.

**CloseHandle**

функция, закрывающая открытый хендл

BOOL CloseHandle(

HANDLE hObject

);

Параметры

HANDLE hObject

– хендл объекта

Возвращаемые значения

Возвращает ненулевое значение при успехе, 0 – при неудаче.

**MessageBox**

выводит окошко сообщения

int MessageBox(        
    HWND hWnd,

    LPCTSTR lpText,

    LPCTSTR lpCaption,

    UINT uType

);

Параметры

hWnd

[in] Хендл окна-владельца. Тут передаём NULL.

lpText

[in] Указатель на строку-сообщение.

lpCaption

[in] Указатель на строку-заголовок.

uType

[in] Константы (какие кнопки и какой значок рисовать). Передаём 0.

Возвращаемые значения

Ничего не возвращает.

**GetSystemInfo**

Функция получения информации о системе.

void GetSystemInfo(

LPSYSTEM\_INFO lpSystemInfo

);

Параметры

LPSYSTEM\_INFO lpSystemInfo

Указатель на структуру SYSTEM\_INFO

Писание структуры SYSTEM\_INFO

typedef struct \_SYSTEM\_INFO {

union {  
 DWORD dwOemId;  
 struct {  
 WORD wProcessorArchitecture;  
 WORD wReserved;  
 };  
 };  
 DWORD dwPageSize;  
 LPVOID lpMinimumApplicationAddress;  
 LPVOID lpMaximumApplicationAddress;  
 DWORD\_PTR dwActiveProcessorMask;  
 DWORD dwNumberOfProcessors;  
 DWORD dwProcessorType;  
 DWORD dwAllocationGranularity;  
 WORD wProcessorLevel;  
 WORD wProcessorRevision;

} SYSTEM\_INFO;

Описание членов структуры

dwOemId

Устаревший член структуры, оставленный для совместимости

wProcessorArchitecture

***Архитектура процессора***

|  |  |
| --- | --- |
| Константа | Значение |
| PROCESSOR\_ARCHITECTURE\_AMD64 9 | x64 (AMD or Intel) |
| PROCESSOR\_ARCHITECTURE\_IA32\_ON\_WIN64 10 | WOW64 |
| PROCESSOR\_ARCHITECTURE\_IA64 6 | Intel Itanium Processor Family (IPF) |
| PROCESSOR\_ARCHITECTURE\_INTEL 0 | x86 |
| PROCESSOR\_ARCHITECTURE\_UNKNOWN 0xffff | Unknown processor. |

wReserved

Зарезервированное поле.

dwPageSize

Размер страницы памяти.

lpMinimumApplicationAddress

Указатель на нижнюю границу памяти, доступной приложению

lpMaximumApplicationAddress

Указатель на верхнюю границу памяти, доступную приложению.

dwActiveProcessorMask

Маска, представляющая множество процессоров, работающих в системе (бит 0 обозначает процессор 0, бит 31 – процессор 31)

dwNumberOfProcessors

Число процессоров в системе.

dwProcessorType

Устаревшее поле типа процессора оставлено для совместимости.

|  |  |
| --- | --- |
| Константа | Значение |
| PROCESSOR\_INTEL\_386 | 386 |
| PROCESSOR\_INTEL\_486 | 486 |
| PROCESSOR\_INTEL\_PENTIUM | 586 |
| PROCESSOR\_INTEL\_IA64 | 2200 |
| PROCESSOR\_AMD\_X8664 | 8664 |

dwAllocationGranularity

Гранулярность начального адреса, с которого происходит выделение памяти.

wProcessorLevel

Уровень процессора, зависящий от архитектуры. Для процессоров x86 зависит от производителя, для процессоров IA64 всегда равен 1.

wProcessorRevision

Версия (ревизия) процессора. Значение поля зависит от архитектуры.

**GetLocalTime**

Функция GetLocalTime из библиотеки Kernel32.dll возвращает текущие дату и время в формате UTC (Universal Time Coordinated). Её прототип:

VOID GetLocalTime(

LPSYSTEMTIME lpSystemTime // Адрес структуры SystemTime

);

Параметры

Тип данных LPSYSTEMTIME – это указатель на структуру SystemTime:

Описаниеструктуры SYSTEMTIME

typedef struct \_SYSTEMTIME {

WORD wYear; // Год

WORD wMonth; // Месяц

WORD wDayOfWeek; // День недели

WORD wDay; // День месяца

WORD wHour; // Часы

WORD wMinute; // Минуты

WORD wSecond; // Секунды

WORD wMilliseconds; // Миллисекунды

} SYSTEMTIME;

Как видно из описания, все поля структуры – это 16-разрядные целые числа. Месяцы нумеруются, начиная с единицы, т.е. для января wMonth=1, и так далее. Номера дней недели начинаются с нуля, т.е. для воскресенья wDayOfWeek=0, для понедельника wDayOfWeek=1, и так далее.

Функция GetLocalTime не возвращает никаких значений.

**GlobalMemoryStatus**

Функция GlobalMemoryStatus из библиотеки Kernel32.dll предоставляет информацию об использовании физической и виртуальной памяти компьютера. Её прототип:

VOID GlobalMemoryStatus(

LPMEMORYSTATUS lpBuffer // Адрес структуры MemoryStatus

);

Параметры

lpBuffer – это указатель на структуру MemoryStatus:

typedef struct \_MEMORYSTATUS {

DWORD dwLength;// Размер структуры

DWORD dwMemoryLoad;// Процент использования памяти

DWORD dwTotalPhys;// Физическая память, байт

DWORD dwAvailPhys;// Свободная физическая память, байт

DWORD dwTotalPageFile; // Размер файла подкачки, байт

DWORD dwAvailPageFile; // Свободных байт в файле подкачки

DWORD dwTotalVirtual; // Виртуальная память, используемая процессом

DWORD dwAvailVirtual; // Свободная виртуальная память

} MEMORYSTATUS, \*LPMEMORYSTATUS;

Функция не возвращает никаких значений.

**Sleep**

Функция Sleep из библиотеки Kernel32.dll приостанавливает выполнение приложения на указанное количество миллисекунд. Вот её прототип:

VOID Sleep(

DWORD dwMilliseconds

);

Параметры

dwMilliseconds – время «засыпания» программы в миллисекундах

Возвращаемое значение

Функция ничего не возвращает

**Контрольные вопросы**

1. Каким образом программа пользователя взаимодействует с ОС Windows?

2. Каким образом передаются параметры функции, имеющей тип вызова stdcall?

3. Как можно определить, что при выполнении функции произошла ошибка?

4. Для чего применяется директива includelib?

5. Для чего применяется директива PROTO?

**Задания к лабораторной работе**

1. Открыть файл, заменить в нем Unix переводы строки (0x0a) на Windows (0x0d 0x0a), а Windows – на Unix.

2. Открыть файл, наложить операцией XOR константную строку, результат записать в новый файл.

3. Закодировать файл по правилу «Кодирование длинных серий» (см. л/р № 7).

4. Раскодировать файл по правилу «Кодирование длинных серий» (см. л/р № 7).

4. Отсортировать файл, содержащий двойные слова. Алгоритм сортировки – простой выбор.

6. Отсортировать файл, содержащий двойные слова. Алгоритм сортировки – простые вставки.

7. Вывести с помощью MessageBox текущие дату и время.

8. Вывести с помощью MessageBox размер некоторого файла.

9. Вывести с помощью MessageBox информацию о системе (несколько полей на выбор студента).

10. Вывести с помощью MessageBox информацию об использовании памяти.

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

* 1. Юров В.И. Assembler: учебник для вузов. – 2-е изд. – СПб.: Питер, 2010. – 637 с.
  2. Юров В.И. Assembler. Практикум. – 2-изд. – СПб.: Питер, 2006. – 399 с.
  3. Абашев А.А., Жуков И.Ю., Иванов М.А., Метлицкий Ю.В., Тетерин И.И. Ассемблер в задачах защиты информации. – М.: КУДИЦ-Образ, 2004. – 544 с.
  4. Аблязов Р. З. Программирование на Ассемблере на платформе x86-64. – М.: ДМК Пресс, 2011. – 308 с.
  5. Майк Предко. PIC-микроконтроллеры. Архитектура и программирование. – М.: ДМК Пресс, 2010. – 512 с.
  6. Архитектура МП 80486 // http://birmaga.ru.
  7. Процессор Intel в защищенном режиме #1 // http://www.wasm.ru/article/187 – Дата публикации 15.10.2002.
  8. Процессор Intel в защищенном режиме #2 //http://www.wasm.ru/article/188 – Дата публикации 15.10.2002.
  9. Процессор Intel в защищенном режиме #3 //http://www.wasm.ru/article/189 – Дата публикации 15.10.2002.
  10. Процессор Intel в защищенном режиме #5 //http://www.wasm.ru/article/191 – Дата публикации 15.10.2002.
  11. Процессор Intel в защищенном режиме #6 //http://www.wasm.ru/article/192 – Дата публикации 15.10.2002.
  12. Процессор Intel в защищенном режиме #11 //http://www.wasm.ru/article/246 – Дата публикации 15.10.2002.
  13. Процессор Intel в защищенном режиме #12 //http://www.wasm.ru/article/248 – Дата публикации 15.10.2002.
  14. Процессор Intel в защищенном режиме #13 //http://www.wasm.ru/article/251 – Дата публикации 15.10.2002.

**ДЛЯ ЗАПИСЕЙ**

**ДЛЯ ЗАПИСЕЙ**

Учебное издание

**Абрамов Евгений Сергеевич**

**Сидоров Игорь Дмитриевич**

**Машинно-ориентированное   
программирование**

Учебное пособие

Ответственный за выпуск Ищукова Е.А.

Редактор Надточий З.И.

Корректор Надточий З.И.

Подписано в печать 15.09.2016

Заказ № Тираж 50 экз.

Формат 60х84 1/16. Усл. печ. л. – 5,25. Уч.-изд. л. – 5,00.

Издательство Южного федерального университета

344091, г. Ростов-на-Дону, пр. Стачки, 200/1.

Тел. (863)2478051.

Отпечатано в Секторе обеспечения полиграфической   
продукцией кампуса в г. Таганроге отдела полиграфической, корпоративной и сувенирной продукции ИПК КИБИ МЕДИА ЦЕНТРА ЮФУ.

ГСП 17 А, Таганрог, 28, Энгельса, 1.

Тел. (8634)371717.