Міністерство освіти і науки України
Харківський національний автомобільно-дорожній університет
Кафедра інформаційних технологій та мехатроніки

Конспект лекцій

“Програмування на мові Ассемблера”

З дисципліни “Програмування”

(Розділ “Програмування на мові Ассемблера”)

Для студентів напряму підготовки 6.050201 “Системна інженерія”
Семестр 4

Розробник доцент кафедри ІПМ Симбірський Г.Д.

Харків, 2016

Лекция 1

Архитектура компьютера

Цель лекции

Изучить архитектуру компьютера, центральный процессор, состав и назначение регистров.

Основные вопросы лекции:

1. Центральный процессор.
2. Регистры – состав и назначение.
3. Прочие регистры – состав и назначение.
4. Организация памяти в компьютере.

1. Центральный процессор

Выполнение компьютером программ означает исполнение специальной микросхемой (процессором) команд (инструкций) и хранение результатов в специальных разделах процессора (регистрах).

Например, выполнение инструкций

mov eax. 2
add eax. 3

приводит к тому, что в регистре **eax** оказывается число 5. Первая команда посылает в этот регистр число 2, а вторая – прибавляет к содержимому регистра число 3.

Центральный процессор (ЦП) - устройство, непосредственно предназначенное для выполнения вычислительных операций. Процессор работает под управлением программы, выполняя вычисления или принимая логические решения, необходимые для обработки информации.

Большинство современных центральных процессоров строятся на базе 32-битной архитектуры Intel-совместимых процессоров IA-32 (Intel Architecture), которая является третьим поколением базовой архитектуры x86.

Все процессоры, которыми мы пользуемся, называются процессорами x86. Под процессорами x86 подразумевают следующие модели процессоров: 8086, 80186, 80286, 80386, 80486, 80586 (Pentium I), 80686 (Pentium II) и т. д.

Процессоры 8086-80286 - 16-разрядные, все остальные - 32- разрядные.

Процессоры Intel с поддержкой технологии EM64T и процессоры AMD с поддержкой технологии AMD64 являются 64-разрядными процессорами.

Функционально центральный процессор можно разделить на две части (рис. 1):

- операционную, содержащую арифметико-логическое устройство (АЛУ) и микропроцессорную память (МПП) - регистры общего назначения;
- интерфейсную, содержащую адресные регистры, устройство управления, регистры памяти для хранения кодов команд, выполняемых в ближайшие такты; схемы управления шиной и портами. Обе части ЦП работают параллельно, причем интерфейсная часть опережает операционную, так что выборка очередной команды из памяти (ее запись в блок регистров команд и предварительный анализ) происходит во время выполнения операционной частью предыдущей команды. Такая организация ЦП позволяет существенно повысить его эффективное быстродействие.

Устройство управления (УУ) вырабатывает управляющие сигналы, поступающие по кодовым шинам инструкций в другие блоки вычислительной машины. УУ формирует управляющие сигналы для выполнения команд центрального процессора.

Арифметико-логическое устройство (АЛУ) предназначено для выполнения арифметических и логических операций преобразования информации.

Системная шина – набор проводников, по которым передаются сигналы, соединяющая процессор с другими компонентами на системной плате. Системная шина состоит из шины данных, шины адреса, шины управления.

Рис. 1 – Схема устройства центрального процессора

- Шина данных – служит для пересылки данных между процессором и оперативным запоминающим устройством (ОЗУ).
- Шина адреса – используется для передачи сигналов, с помощью которых определяется местоположение ячейки памяти для выполняемых процессором операций чтения/записи и ввода-вывода.
- Шина управления – служит для пересылки управляющих сигналов. Каждая линия этой шины имеет своё особое назначение, поэтому они могут быть как однонаправленными, так и двунаправленными.

Микропроцессорная память

Микропроцессорная память представляет собой набор регистров, которые условно можно разделить на 4 группы:

- регистры общего назначения;
- сегментные регистры;
- регистр счетчика команд;
- регистр признаков.

2. Регистры общего назначения

Регистр – устройство сверхбыстродействующей памяти в процессоре, служащее для временного хранения управляющей информации, операндов и результатов выполняемых операций. Совокупность регистров процессора называется набором регистров.

Набор регистров общего назначения 32-битной архитектуры центрального процессора включает в себя (рис. 2):

- 4 универсальных регистра: EAX, EBX, ECX, EDX;

- 2 индексных регистра: ESI, EDI;
- 2 регистра для работы со стеком: ESP, EBP.

Рис. 2 – Набор регистров общего назначения

Каждый из 32-разрядных **универсальных регистров** представляет собой логическое объединение, позволяющее отдельно обращаться к своей младшей 16-разрядной части: AX, BX, CX, DX (рис. 3).

Каждый 16-разрядный регистр позволяет независимо обращаться к старшему (H) и младшему (L) байту. Соответствующие 8-разрядные регистры имеют имена AH, AL, BH, BL, CH, CL, DH, DL.

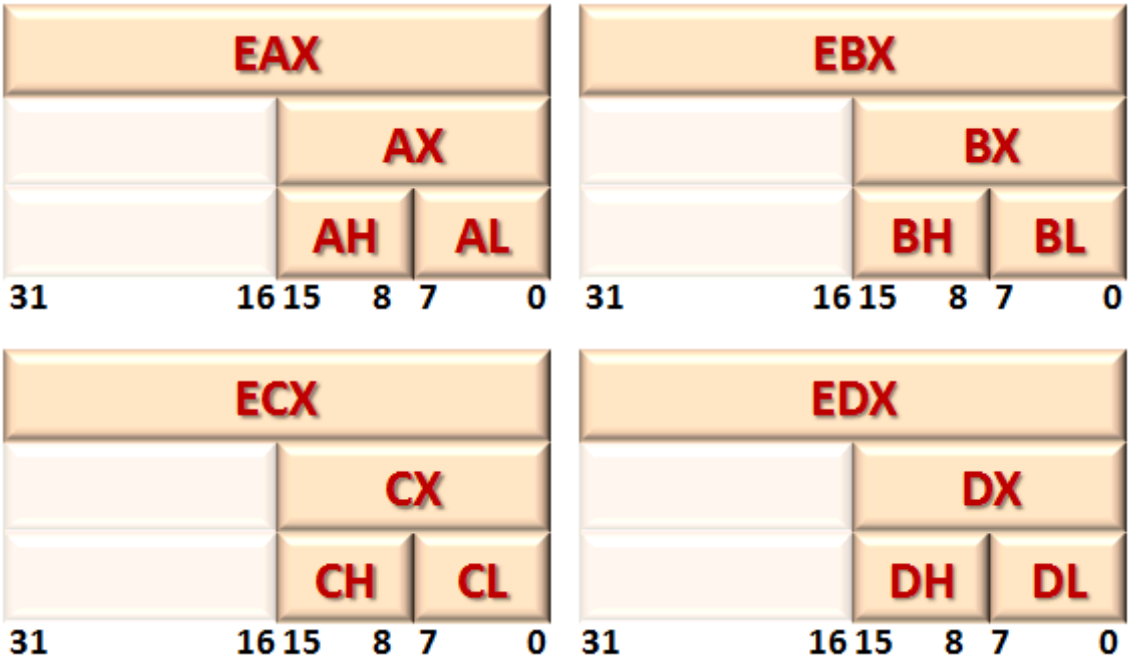


Рис. 3 –Универсальные регистры

Регистр EAX (аккумулятор) – автоматически применяется при операциях умножения, деления и при работе с портами ввода-вывода. Его использование в арифметических, логических и

некоторых других операциях позволяет увеличить скорость их выполнения. Использует для записи возвращаемого значения из процедуры.

Регистр EBX (регистр базы) – может содержать адреса элементов оперативной памяти. По умолчанию эти адреса будут представлять собой смещение в сегменте данных.

Регистр ECX (счетчик) – используется в операциях повторения, например в циклах, в строковых командах и т.д.

Регистр EDX (регистр данных) – является единственным элементом, который может хранить адреса портов ввода-вывода в командах типа IN (получить из порта) и OUT (вывести в порт). Без его помощи невозможно обратиться к портам с адресами в адресном пространстве больше 1 байта. Автоматически применяется также в операциях умножения и деления.

3. Прочие регистры – состав и назначение

Индексные регистры используются для выполнения косвенной адресации, а также автоматически используются в строковых командах. Каждый 32-разрядный индексный регистр представляет собой логическое объединение, позволяющее отдельно обратиться к своей младшей 16-разрядной части (рис. 4).

Регистр ESI (регистр индекса источника) может содержать адреса элементов в оперативной памяти. По умолчанию эти адреса будут представлять собой смещение в сегменте данных. При выполнении операций со строками в этом регистре содержится смещение строки источника в сегменте данных.

Регистр EDI (регистр индекса приемника) может содержать адреса элементов в оперативной памяти. По умолчанию эти адреса будут представлять собой смещение в сегменте данных. При выполнении операций со строками в этом регистре содержится смещение строки приемника в сегменте данных.

Рис. 4 – Индексные регистры

Регистры для работы со стеком используются для хранения вершины стека (ESP) и текущего элемента (базы) - EBP. Каждый 32-разрядный регистр для работы со стеком представляет собой логическое объединение, позволяющее отдельно обратиться к своей младшей 16-разрядной части (рис. 5).

Регистр EBP (указатель базы) может содержать адреса элементов в оперативной памяти. Эти адреса будут представлять собой смещение в сегменте стека.

Регистр ESP (указатель стека) используется для записи данных в стек и чтения их из стека. Фактически он содержит смещение в сегменте стека, которое определяет нужное слово памяти. Значения этого регистра автоматически меняются командами для работы со стеком типов push, pop, pushf, popf, call, ret.

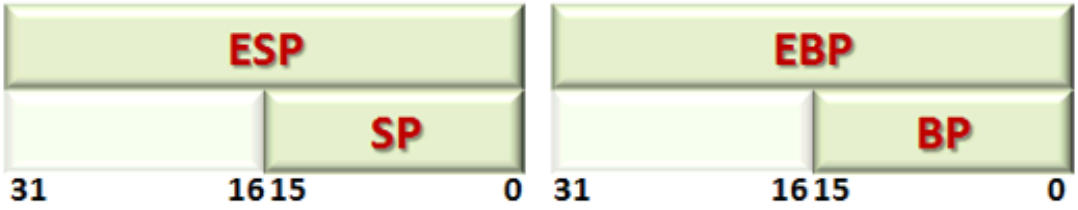


Рис. 5 – Регистры для работы со стеком

Сегментные регистры представляют собой набор 16-разрядных регистров (для 32-битной архитектуры центрального процессора) (рис. 6).

Сегмент - это логический элемент программы, который представляет собой независимый, поддерживаемый на аппаратном уровне блок памяти.

Рис. 6 – Сегментные регистры

Регистр CS (регистр сегмента кода) определяет стартовый адрес сегмента, в который помещается код выполняемой программы. Это единственный сегментный регистр, который нельзя загрузить непосредственно. Косвенно загрузить в регистр CS новое значение могут команды вида jxx, call, int, ret, iret.

Регистр DS (регистр сегмента данных) определяет стартовый адрес сегмента, в который помещаются данные для программы. По умолчанию смещения в сегменте данных задаются в регистрах EBX, ESI и EDI.

Регистр SS (регистр сегмента стека) определяет стартовый адрес сегмента, в который помещается стек для программы. По умолчанию смещения для сегмента стека задаются в регистрах ESP и EBP.

Регистры ES, FS, GS (регистры сегментов дополнительных данных) определяют стартовый адрес сегмента, в который помещаются дополнительные данные для программы. Например, в случае строковых команд, DS определяет сегмент для строки-источника, а ES – сегмент для строки-приемника. За исключением строковых команд, доступ к данным в сегменте ES обычно менее эффективен, чем в сегменте DS.

Регистр счетчика команд

Регистр EIP (указатель команд) содержит смещение в сегменте кода следующей выполняемой команды. Как только некоторая команда начинает выполняться, значение регистра EIP увеличивается на ее длину так, что будет адресовать следующую команду. Физический адрес команды в памяти выполняемой программы определяет пара регистров CS:EIP, то есть к физическому адресу начала сегмента кода добавляется смещение следующей команды в сегменте кода, хранящееся в регистре EIP.

Обычно команды выполняются в той последовательности, в которой они расположены в программе. Нарушают эту последовательность только команды переходов (они начинаются с буквы j: jxx), команды вызова подпрограммы (call), обработчиков прерываний (int) и возврата (ret, iret). Непосредственно содержимое EIP нельзя изменить или прочитать. Косвенно загрузить в регистр EIP новое значение могут только команды jxx, call, int, ret, iret. Регистр EIP является 32-битным. Младшая 16-битная часть регистра счетчика команд имеет имя IP.

Регистр признаков

Регистр признаков EFLAGS включает биты, каждый из которых устанавливается в единичное или в нулевое состояние при определенных условиях. Регистр EFLAGS 32-битный. Младшая 16-битная часть регистра признаков имеет имя FLAGS (рис. 7).

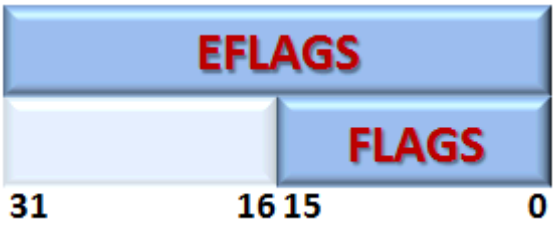


Рис. 7 – Регистр признаков

Все биты регистра признаков подразделяются на (рис. 8):

- **s** - биты состояния (STATUS);
- **c** - биты управления (CONTROL);
- **x** - системные биты (SYSTEM).

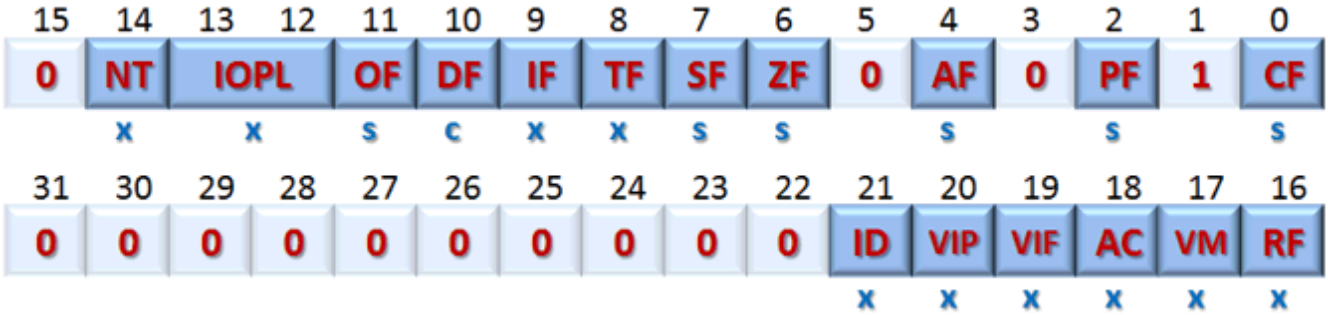


Рис. 8 – Схема битов регистра признаков

CF – бит переноса: устанавливается в 1, когда арифметическая операция генерирует перенос или выход за разрядную сетку результата. сбрасывается в 0 в противном случае. Этот флаг показывает состояние переполнения для бесзнаковых целочисленных арифметических действий. Он также используется в арифметических действиях с повышенной точностью. Может быть установлен командой STC или сброшен командой CLC.

PF – бит четности: устанавливается в 1, если результат последней операции имеет четное число единиц.

AF – бит вспомогательного переноса: устанавливается в 1, если арифметическая операция генерирует перенос из младшей тетрады битов (из 3 бита в 4), сбрасывается в 0 в противном случае. Этот флаг используется в двоично-десятичной арифметике.

ZF – бит нулевого значения: устанавливается в 1, если результат нулевой, сбрасывается в 0 в противном случае.

SF – знаковый бит: устанавливается равным старшему биту результата, который определяет знак в знаковых целочисленных операциях (0 – положительное число, 1 – отрицательное число).

TF – бит пошаговой отладки: устанавливается в 1 для включения режима пошаговой отладки программы, сбрасывается в 0 в противном случае.

IF – бит прерываний: при значении 1 микропроцессор реагирует на внешние аппаратные прерывания по входу INTR. При значении 0 микропроцессор игнорирует внешние прерывания.

DF – бит направления: управляет строковыми командами (MOVS, CMPS, SCAS, LODS, STOS). Если DF = 1 (команда STD), то содержимое индексных регистров ESI, EDI увеличивается, если DF = 0 (команда CLD), то содержимое индексных регистров ESI, EDI уменьшается.

OF – бит переполнения: устанавливается в 1, если целочисленный результат выходит за пределы разрядной сетки. Тем самым данный бит указывает на потерю старшего бита результата.

IOPL – уровень приоритета: 2-битовое поле, которое отображает уровень приоритета ввода-вывода для выполняемой в данное время программы или задачи. Действительный приоритет задачи может быть меньше или равен IOPL.

NT – флаг вложенной задачи: управляет последовательностью вызванных и прерванных задач. Установлен в 1, если текущая задача связана с предыдущей, сброшен в 0, если текущая задача не связана с другими задачами.

RF — флаг возобновления: используется при обработке прерываний от регистров отладки.

VM — флаг виртуального 8086: признак работы процессора в режиме виртуального 8086: 1 – процессор работает в режиме виртуального 8086, 0 – процессор работает в реальном или защищенном режиме.

AC — флаг контроля выравнивания: предназначен для разрешения контроля выравнивания при обращениях к памяти. Если требуется контролировать выравнивание данных и команд по адресам, кратным 2 или 4, то установка данных битов приведет к тому, что все обращения по некратным адресам будут вызывать исключительную ситуацию.

VIF — флаг виртуального прерывания: при определенных условиях (одно из которых – работа микропроцессора в V-режиме) является аналогом флага IF. Флаг VIF используется совместно с флагом VIP.

VIP — флаг отложенного виртуального прерывания: устанавливается в 1 для индикации отложенного прерывания. Используется совместно с VIF в виртуальном режиме.

ID — флаг поддержки идентификации процессора: используется для отображения поддержки микропроцессором инструкции CPUID.

Вопросы для самоконтроля

1. Как правильно записать действительное число на языке *C*++?
2. В результате решения задачи на ПК на екране появилось число 7.45600000*E*+01. Какое это число?
3. Как производится запись символьной константы в языке *C*++?
4. Как записывается идентификатор в *C*++?
5. Как записывается константа в *C*++?
6. Перечислите базовые типы данных в *C*++.
7. Дайте определение константы.
8. Дайте определение переменной.
9. Для чего необходимо объявлять переменные перед действиями с ними?
10. Каков формат объявления переменной?

Лекция 2

Организация памяти в компьютере

Цель лекции

Изучение основных принципов и особенностей организации памяти компьютера.

Основные вопросы лекции:

1. Память в компьютере.
2. Организация стека.
3. Организация памяти.
4. Основные модели памяти Ассемблера.

1. Память в компьютере

Память – способность компьютера обеспечивать хранение данных. Все объекты, над которыми выполняются команды, как и сами команды, хранятся в памяти.

Память состоит из ячеек, в каждой из которых содержится 1 бит информации, принимающий одно из двух значений: 0 или 1. Биты обрабатывают группами фиксированного размера. Для этого группы бит могут записываться и считываться за одну базовую операцию.

Группа из 8 бит называется **байт** (рис. 1).

Рис. 1 – **Структура одного байта памяти**

Байты последовательно располагаются в памяти компьютера.

- 1 килобайт (Кбайт) = 2^{10} = 1 024 байт
- 1 мегабайт (Мбайт) = 2^{10} Кбайт = 2^{20} байт = 1 048 576 байт
- 1 гигабайт (Гбайт) = 2^{10} Мбайт = 2^{30} байт = 1 073 741 824 байт

Для доступа к памяти с целью записи или чтения отдельных элементов информации используются идентификаторы, определяющие расположение в памяти этих элементов. Каждому идентификатору в соответствие ставится адрес. В качестве адресов используются числа из диапазона от 0 до 2^k-1 со значением k, достаточным для адресации всей памяти компьютера. Все 2^k адресов составляют адресное пространство компьютера.

Способы адресации байтов

Существует прямой и обратный способы адресации байтов.

При **обратном** способе адресации байты адресуются слева направо, так что самый старший (левый) байт слова имеет наименьший адрес (рис.2).

Рис. 2 – Обратная адресация байтов

Прямым способом называется противоположная система адресации (рис.3). Компиляторы высокоуровневых языков поддерживают прямой способ адресации.

Рис. 3 – **Прямая** адресация байтов

Объект занимает целое слово. Поэтому для того, чтобы обратиться к нему в памяти, нужно указать адрес, по которому этот объект хранится.

2. Организация стека

Стек – это такая структура данных в памяти, которая используется для временного хранения информации. Программа может поместить слово в стек (команда **PUSH**) или извлечь его из стека (команда **POP**).

Данные в стеке упорядочиваются специальным образом. Извлекаемый из стека элемент данных – это всегда тот элемент, который был записан туда последним. Такая организация хранения данных сокращенно обозначается **LIFO** (Last In, First Out – последний поступивший удаляется первым). Если мы поместим в стек два элемента: сначала A, а затем B, то при первом обращении к стеку извлекается элемент B, а при следующем – A. Информация выбирается из стека в обратном по отношению к записи порядке.

В ЭВМ за стеком резервируется блок памяти, адресуемый регистром SS и указатель, называемый указателем стека SP (Stack Pointer). Указатель стека используется программой для того, чтобы фиксировать самый последний записанный в стек элемент данных. При выполнении команды POP или PUSH значение указателя стека соответственно увеличивается или уменьшается на 4 (рис. 4).

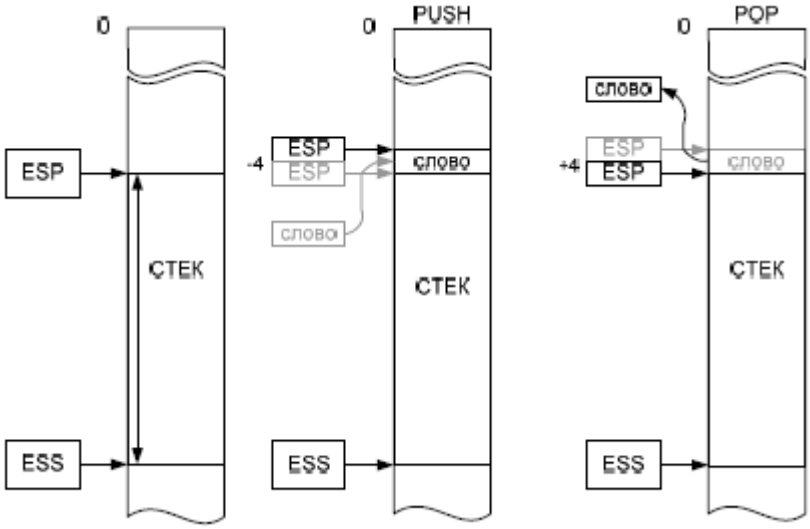


Рис. 4 **– Организация памяти в стеке**

3. Организация памяти

Физическая память, к которой микропроцессор имеет доступ по шине адреса, называется оперативной памятью (ОП) (или оперативным запоминающим устройством - ОЗУ). Механизм управления памятью полностью аппаратный, т.е. программа сама не может сформировать физический адрес памяти на адресной шине.

Микропроцессор аппаратно поддерживает несколько моделей использования оперативной памяти:

- сегментированная модель;
- страничная модель;
- плоская модель.

3.1. В **сегментированной модели** память для программы делится на непрерывные области памяти, называемые сегментами. Программа может обращаться только к данным, которые находятся в этих сегментах. Сегмент представляет собой независимый, поддерживаемый на аппаратном уровне блок памяти.

Сегментация - механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния.

Каждая программа в общем случае может состоять из любого количества сегментов, но непосредственный доступ она имеет только к 3 основным сегментам и к 3 дополнительным сегментам, обслуживаемых 6 сегментными регистрами. К основным сегментам относятся:

- Сегмент кодов (**.CODE**) – содержит машинные команды для выполнения. Обычно первая выполняемая команда находится в начале этого сегмента, и операционная система передает управление по адресу данного сегмента для выполнения программы. Регистр сегмента кодов (**CS**) адресует данный сегмент.
- Сегмент данных (.DATA) – содержит определенные данные, константы и рабочие области, необходимые программе. Регистр сегмента данных (DS) адресует данный сегмент.
- Сегмент стека (**.STACK**). Стек содержит адреса возврата как для программы (для возврата в операционную систему), так и для вызовов подпрограмм (для возврата в главную программу). Регистр сегмента стека (**SS**) адресует данный сегмент. Адрес текущей вершины стека задается регистрами **SS:ESP.**

Регистры дополнительных сегментов (ES, FS, GS), предназначены для специального использования.

Для доступа к данным внутри сегмента обращение производится относительно начала сегмента линейно, т.е. начиная с 0 и заканчивая адресом, равным размеру сегмента. Для обращения к любому адресу в программе, компьютер складывает адрес в регистре сегмента и смещение - расположение требуемого адреса относительно начала сегмента. Например, первый байт в сегменте кодов имеет смещение 0, второй байт – 1 и так далее.

Таким образом, для обращения к конкретному физическому адресу ОЗУ необходимо определить адрес начала сегмента и смещение внутри сегмента.

Физический адрес принято записывать парой этих значений, разделенных двоеточием:

сегмент:смещение.

Регистры дополнительных сегментов (ES, FS, GS), предназначены для специального использования.

На рис. 5 графически представлены регистры SS, DS и S. Последовательность регистров и сегментов на практике может быть иной. Три сегментных регистра содержат начальные адреса соответствующих сегментов, и каждый сегмент начинается на границе параграфа.

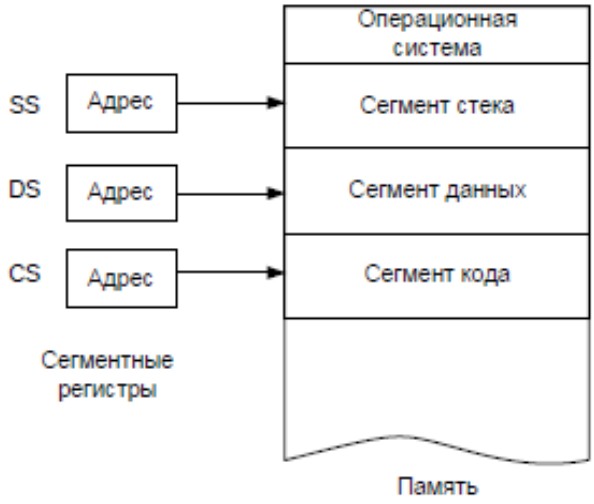


Рис. 5 – **Размещение сегментов программы в операционной памяти**

Операционная система размещает сегменты программы в ОП по определенным физическим адресам, а значения этих адресов записывает в определенные места, в зависимости от режима работы микропроцессора:

- в реальном режиме адреса помещаются непосредственно в сегментные регистры (**cs, ds, ss, es, gs, fs**);
- в защищенном режиме - в специальную системную дескрипторную таблицу (Элементом дескрипторной таблицы является дескриптор сегмента. Каждый сегмент имеет дескриптор сегмента -8 байт. Существует три дескрипторные таблицы. Адрес каждой таблицы записывается в специальный системный регистр).

Для доступа к данным внутри сегмента обращение производится относительно начала сегмента линейно, т.е. начиная с 0 и заканчивая адресом, равным размеру сегмента. Этот адрес называется смещением (**offset**) (рис. 6).

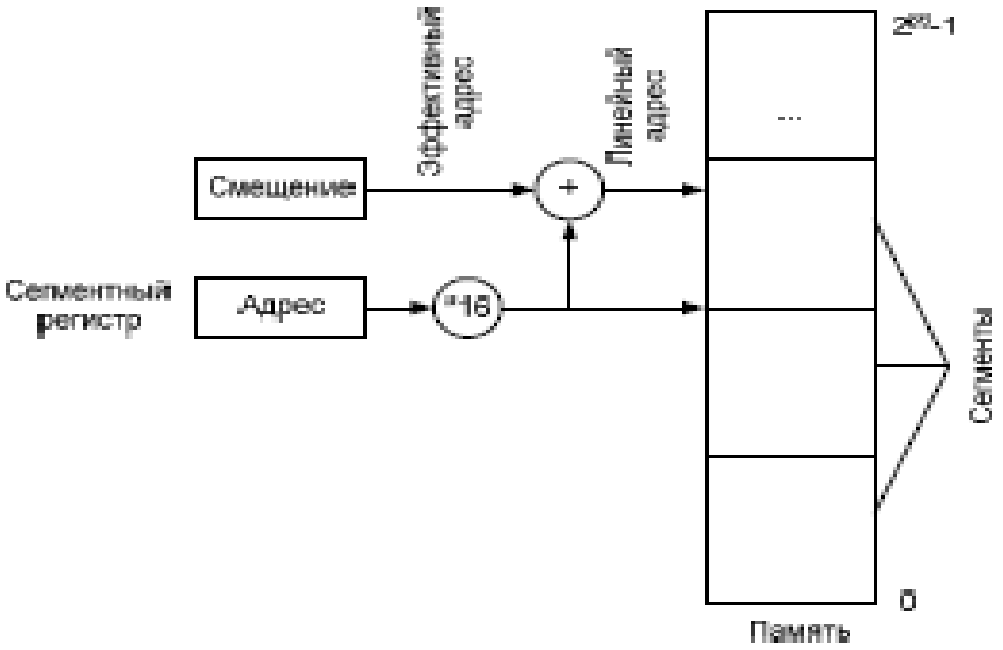


Рис. 6 **– Схема доступа к данным внутри сегмента памяти**

Двухбайтовое смещение (16-бит) может быть в пределах от 0000h до FFFFh или от 0 до 65535. Для обращения к любому адресу в программе, компьютер складывает адрес в регистре сегмента и смещение. Например, первый байт в сегменте кодов имеет смещение 0, второй байт – 01 и так далее.

Таким образом, для обращения к конкретному физическому адресу ОП необходимо определить адрес начала сегмента и смещение внутри сегмента.

Физический адрес принято записывать парой этих значений, разделенных двоеточием:

segment : offset .

Например, 0040:001Ch; 0000:041Ch; 0020:021Ch; 0041:000Ch.

В качестве примера адресации допустим, что регистр сегмента данных содержит 045Fh, и некоторая команда обращается к ячейке памяти внутри сегмента данных со смещением 0032h. Несмотря на то, что регистр сегмента данных содержит 045Fh, он указывает на адрес 045F0, то есть на границу параграфа. Действительный адрес памяти поэтому будет следующий:

Адрес в DS:	**045F0**
Смещение:	**0032.**
Реальный адрес:	**04622.**
Адрес указывается как DS:	**0032h.**

ОС строит для каждого исполняемого процесса соответствующую таблицу дескрипторов сегментов и при размещении каждого из сегментов в ОП или внешней памяти в дескрипторе отмечает его текущее местоположение (бит присутствия).

Дескриптор содержит поле адреса, с которого сегмент начинается и поле длины сегмента. Благодаря этому можно осуществлять контроль:

1) размещения сегментов без наложения друг на друга;

2) обращается ли код исполняющейся задачи за пределы текущего сегмента.

В дескрипторе содержатся также данные о правах доступа к сегменту (запрет на модификацию, можно ли его предоставлять другой задаче) и защита.

Достоинства сегментной организации памяти:

1) общий объем виртуальной памяти превосходит объем физической памяти;

2) возможность размещать в памяти как можно больше задач (до определенного предела) увеличивает загрузку системы, т. е. более эффективно использовать ресурсы системы.

Недостатки:

1) увеличивается время на доступ к искомой ячейке памяти, т.к. необходимо вначале прочитать дескриптор сегмента, после чего, используя его данные, можно вычислить физический адрес (для уменьшения этих потерь используется кэширование - дескрипторы, с которыми работа идет в данный момент размещаются в сверхоперативной памяти - в специальных регистрах процессора);

2) фрагментация;

3) потери памяти на размещение дескрипторных таблиц;

4) потери процессорного времени на обработку дескрипторных таблиц.

3.2. Страничная модель памяти – это надстройка над сегментной моделью. ОЗУ делится на блоки фиксированного размера, кратные степени 2, например 4 Кб. Каждый такой блок называется страницей. Основное достоинство страничного способа распределения памяти - минимально возможная фрагментация. Однако такая организация памяти не использует память достаточно эффективно за счет фиксированного размера страниц.

3.3. Плоская модель памяти предполагает, что задача состоит из одного сегмента, который, в свою очередь, разбит на страницы.

Достоинства:

- при использовании плоской модели памяти упрощается создание и операционной системы, и систем программирования;
- уменьшаются расходы памяти на поддержку системных информационных структур.

!!!В абсолютном большинстве современных 32(64)-разрядных операционных систем (для микропроцессоров Intel) используется плоская модель памяти.

4. Задание модели памяти в программе

Директива **.MODEL** определяет модель памяти, используемую программой. После этой директивы в программе находятся директивы объявления сегментов **(.DATA, .STACK, .CODE, SEGMENT**).

Синтаксис задания модели памяти

.MODEL модификатор **МодельПамяти** СоглашениеОВызовах.

Параметр **МодельПамяти** является обязательным.

Таблица 1. Основные модели памяти Ассемблера

Модель памяти	**Адресация кода**	**Адресация данных**	**Операционная система**	**Чередование кода и данных**
TINY	NEAR	NEAR	MS-DOS	Допустимо
SMALL	NEAR	NEAR	MS-DOS, Windows	Нет
MEDIUM	FAR	NEAR	MS-DOS, Windows	Нет
COMPACT	NEAR	FAR	MS-DOS, Windows	Нет
LARGE	FAR	FAR	MS-DOS, Windows	Нет
HUGE	FAR	FAR	MS-DOS, Windows	Нет
FLAT	NEAR	NEAR	Windows NT, Windows 2000, Windows XP, Windows Vista	Допустимо

Модель **tiny** работает только в 16-разрядных приложениях MS-DOS. В этой модели все данные и код располагаются в одном физическом сегменте. Размер программного файла в этом случае не превышает 64 Кбайт.

Модель **small** поддерживает один сегмент кода и один сегмент данных. Данные и код при использовании этой модели адресуются как **near** (ближние).

Модель **medium** поддерживает несколько сегментов программного кода и один сегмент данных, при этом все ссылки в сегментах программного кода по умолчанию считаются дальними (far), а ссылки в сегменте данных — ближними (**near**).

Модель **compact** поддерживает несколько сегментов данных, в которых используется дальняя адресация данных (**far**), и один сегмент кода с ближней адресацией (**near**).

Модель **large** поддерживает несколько сегментов кода и несколько сегментов данных. По умолчанию все ссылки на код и данные считаются дальними (**far**).

Модель **huge** практически эквивалентна модели памяти **large**.

Особого внимания заслуживает модель памяти **flat**, которая используется только в 32-разрядных операционных системах. В ней данные и код размещены в одном 32-разрядном сегменте. Для использования в программе модели **flat** перед директивой **.model flat** следует разместить одну из директив:

- .386
- .486
- .586
- .686

Желательно указывать тот тип процессора, который используется в машине, хотя это не является обязательным требованием. Операционная система автоматически инициализирует сегментные регистры при загрузке программы, поэтому модифицировать их нужно только в случае если требуется смешивать в одной программе 16-разрядный и 32-разрядный код. Адресация данных и кода является ближней (**near**), при этом все адреса и указатели являются 32-разрядными.

Параметр **модификатор** используется для определения типов сегментов и может принимать значения **use16** (сегменты выбранной модели используются как 16-битные) или **use32** (сегменты выбранной модели используются как 32-битные).

Параметр **СоглашениеОВызовах** используется для определения способа передачи параметров при вызове процедуры из других языков, в том числе и языков высокого уровня (C++, Pascal). Параметр может принимать следующие значения:

- C,
- BASIC,
- FORTRAN,
- PASCAL,
- SYSCALL,
- STDCALL.

При разработке модулей на ассемблере, которые будут применяться в программах, написанных на языках высокого уровня, обращайте внимание на то, какие соглашения о вызовах поддерживает тот или иной язык. Используются при анализе интерфейса программ на ассемблере с программами на языках высокого уровня.

Лекция 3

Основные понятия языка ассемблера

Цель лекции

Изучить основные понятия языка ассемблера: команды, операнды, данные и др.

Основные вопросы лекции:

1. Синтаксис команд ассемблера.
2. Структура программы на ассемблере.
3. .
4. .

1. Синтаксис команд ассемблера

Каждая машинная команда состоит из двух частей:

- операционной - определяющей, "что делать";
- операндной - определяющей объекты обработки, "с чем делать".

Машинная команда микропроцессора, записанная на языке ассемблера, представляет собой одну строку, имеющую следующий синтаксический вид:

метка команда/директива операнд(ы) ;комментарии

При этом **обязательным** полем в строке является команда или директива.

Метка, команда/директива и операнды (если имеются) разделяются, по крайней мере, одним символом пробела или табуляции.

Если команду или директиву необходимо продолжить на следующей строке, то используется символ обратный слеш: \.

По умолчанию язык ассемблера не различает заглавные и строчные буквы в написании команд или директив.

Примеры строк кода:

```
Count   db      1           ; Имя, директива, один операнд
        mov     eax,0       ; Команда, два операнда
        cbw                 ; Команда
```

Метка в языке ассемблера может содержать следующие символы:

- все буквы латинского алфавита;
- цифры от 0 до 9;
- спецсимволы: _, @, $, ?.

В качестве первого символа метки может использоваться точка, но некоторые компиляторы не рекомендуют применять этот знак. В качестве меток нельзя использовать зарезервированные имена Ассемблера (директивы, операторы, имена команд).

Первым символом в метке должна быть буква или спецсимвол (но не цифра). Максимальная длина метки – 31 символ. Все метки, которые записываются в строке, не содержащей директиву ассемблера, должны заканчиваться двоеточием **: .**

Команда указывает транслятору, какое действие должен выполнить микропроцессор. В сегменте данных команда (или директива) определяет поле, рабочую область или константу. В сегменте кода команда определяет действие, например, пересылка (**mov**) или сложение (**add**).

Директивы - операторы, позволяющие управлять процессом ассемблирования и формирования листинга. Они действуют только в процессе ассемблирования программы и, в отличие от команд, не генерируют машинных кодов.

Операнд – объект, над которым выполняется машинная команда или оператор языка программирования.

Команда может иметь один-два операнда, или вообще не иметь операндов. Число операндов неявно задается кодом команды.

Примеры:

- Нет операндов: **ret** **; Вернуться**
- Один операнд: **inc ecx** **; Увеличить ecx**
- Два операнда: **add eax,12** **; Прибавить 12 к eax**

Метка, команда (директива) и операнд не обязательно должны начинаться с какой-либо определенной позиции в строке. Однако рекомендуется записывать их в колонку для большего удобства чтения программы.

В качестве операндов могут выступать

- идентификаторы;
- цепочки символов, заключенных в одинарные или двойные кавычки;
- целые числа в двоичной, восьмеричной, десятичной или шестнадцатеричной системе счисления.

Идентификаторы – последовательности допустимых символов, использующиеся для обозначения таких объектов программы, как коды операций, имена переменных и названия меток.

Правила записи идентификаторов:

- Идентификатор может состоять из одного или нескольких символов.
- В качестве символов можно использовать буквы латинского алфавита, цифры и некоторые специальные знаки: _, ?, $, @.
- Идентификатор не может начинаться символом цифры.
- Длина идентификатора может быть до 255 символов.
- Транслятор воспринимает первые 32 символа идентификатора, а остальные игнорирует.

Комментарии отделяются от исполняемой строки символом **;** . При этом все, что записано после символа точка с запятой и до конца строки, является комментарием. Использование комментариев в программе улучшает ее ясность, особенно там, где назначение набора команд непонятно. Комментарий может содержать любые печатные символы, включая пробел. Комментарий может занимать всю строку или следовать за командой на той же строке.

2. Структура программы на ассемблере

Программа, написанная на ассемблере MASM, может состоять из нескольких частей, называемых модулями, в каждом из которых могут быть определены один или несколько сегментов данных, стека и кода. Любая законченная программа на ассемблере должна включать один главный, или основной, модуль, с которого начинается ее выполнение.

Модуль может содержать программные сегменты, сегменты данных и стека, объявленные при помощи соответствующих директив. Кроме того, перед объявлением сегментов нужно указать модель памяти при помощи директивы .MODEL.

Задание 1.3. В среде **Microsoft Visual Studio 2010** создайте и запустите на языке ассемблера проект с простейшей «ничего не делающей» программой:

```
; prog.asm
.686P
.MODEL FLAT, STDCALL
.DATA
.CODE
START:
RET
END START
```

В данной программе представлена всего одна команда микропроцессора. Эта команда **RET.** Она обеспечивает правильное окончание работы программы. В общем случае эта команда используется для выхода из процедуры.

Остальная часть программы относится к работе транслятора.

.686P - разрешены команды защищенного режима Pentium 6 (Pentium II).

Данная директива выбирает поддерживаемый набор команд ассемблера, указывая модель процессора. Буква P, указанная в конце директивы, сообщает транслятору о работе процессора в защищенном режиме.

.MODEL FLAT, - плоская модель памяти. Эта модель памяти используется в операционной системе Windows. **STDCALL** – используемое соглашение о вызовах процедур.

.DATA - блок программы, содержащий данные. В MS-DOS это называлось сегментом. В Windows для прикладной программы сегменты отсутствуют, точнее, есть один большой плоский сегмент. Такие блоки здесь называются **секциями.**

Секции можно задать различные свойства. Например, запретить запись в нее, или сделать секцию доступной для других программ, используя стандартные директивы сегментации.

Данная программа не использует стек, поэтому секция **.STACK** отсутствует.

.CODE - блок программы, содержащей код.

START - метка. В ассемблере метки играют большую роль, что не скажешь о современных языках высокого уровня.

END START - конец программы и сообщение компилятору, что начинать выполнение программы надо с метки START.

Каждая программа должна содержать директиву END, отмечающую конец исходного кода программы. Все строки, которые следуют за директивой END, игнорируются. Если вы опустите директиву END, то генерируется ошибка.

Метка, указанная после директивы END, сообщает транслятору имя главного модуля, с которого начинается выполнение программы. Если программа содержит один модуль, метку после директивы END можно не указывать.

Операционная система Windows является многозадачной средой. Каждая задача имеет свое адресное пространство и свою очередь сообщений. Более того, даже в рамках одной программы может быть осуществлена многозадачность – любая процедура может быть запущена как самостоятельная задача.

Программирование в Windows основывается на использовании функций API (Application Program Interface, т.е. интерфейс программного приложения). Их количество достигает двух тысяч. Программа для Windows в значительной степени состоит из таких вызовов. Все взаимодействие с внешними устройствами и ресурсами операционной системы будет происходить посредством таких функций.

Операционная система Windows использует плоскую модель памяти. Другими словами, всю память можно рассматривать как один сегмент. Для программиста на языке ассемблера это означает, что адрес любой ячейки памяти будет определяться содержимым одного 32-битного регистра, например EBX. Следовательно, мы фактически не ограничены в объеме данных, кода или стека (объеме локальных переменных). Выделение в тексте программы сегмента кода и сегмента данных является теперь простой формальностью, улучшающей читаемость программы.

Вызов функций API. В файле помощи любая функция API представлена в виде (например):

```
int MessageBox (HWND hWnd, LPCTSTR lpText,
                LPCTSTR lpCaption, UINT uType);
```

Данная функция выводит на экран окно с сообщением и кнопкой (или кнопками) выхода. Смысл параметров:

hWnd -дескриптор окна, в котором будет появляться окно-сообщение,

lpText - текст, который будет появляться в окне,

lpCaption - текст в заголовке окна,

uType - тип окна, в частности можно определить количество кнопок выхода.

Теперь о типах параметров. Практически все параметры API-функций в действительности 32-битные целые числа:

HWND — 32-битное целое,

LPCTSTR — 32-битный указатель на строку,

UINT — 32-битное целое. К имени функций часто добавляется суффикс "A" для перехода к более новым версиям функций.

Кроме того, при использовании MASM необходимо также в конце имени добавить @16 – количество байт, которое занимают в стеке переданные аргументы. Для функций Win32 API это число можно определить как количество аргументов **n**, умноженное на 4 (байта в каждом аргументе): **4*n**. Для вызова функции используется команда **CALL** ассемблера.

При этом все аргументы функции передаются в нее через стек (команда PUSH). Направление передачи аргументов: СЛЕВА НАПРАВО— СНИЗУ ВВЕРХ. В соответствии с этим, первым будет помещаться в стек аргумент **uType**.

Таким образом, вызов указанной функции будет выглядеть так:

CALL MessageBoxA@16.

Результат выполнения любой API функции — это, как правило, целое число, которое возвращается в регистре EAX.

```
.686P
.MODEL FLAT, STDCALL
;include C:\masm32\include\user32.inc
MessageBoxA PROTO :DWORD, :DWORD, :DWORD,
:DWORD
.STACK 4096
.DATA
MB_OK EQU 0
STR1 DB "Моя первая программа",0
STR2 DB "Привет всем!",0
HW DD ?
.CODE
START:
INVOKE MessageBoxA, HW,
OFFSET STR2, OFFSET STR1, MB_OK
RET
END START
```

Директива **OFFSET** представляет собой «смещение в секции», или, переводя в понятия языков высокого уровня, «указатель» начала строки.

Директива **EQU** подобно **#define** в языке СИ определяет константу.

Транслятор языка **MASM** позволяет также упростить вызов функций с использованием макросредства – директивы **INVOKE**:

INVOKE функция, параметр1, параметр2,…

В этом случае нет необходимости добавлять @16 к вызову функции. Кроме того, параметры записываются точно в том порядке, в котором приведены в описании функции. Макросредствами транслятора параметры помещаются в стек. Для использования директивы INVOKE необходимо иметь описание прототипа функции с использованием директивы PROTO в виде:

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

Если в программе используется множество функций Win32 API, целесообразно воспользоваться директивой

include C:\masm32\include\user32.inc

Функция **MessageBoxA** вызывается из системной библиотеки **Windows user32.dll**.

include-файлы MASM с соответствующими названиями библиотек содержат описания прототипов всех функций данной библиотеки.

3. Типы данных в ассемблере

Данные – числа и закодированные символы, используемые в качестве операндов команд.

Основные типы данных в ассемблере

Тип	Директива	Количество байт
Байт	DB	1
Слово	DW	2
Двойное слово	DD	4
8 байт	DQ	8
10 байт	DT	10

Данные, обрабатываемые вычислительной машиной, можно разделить на 4 группы:

- целочисленные;
- вещественные.
- символьные;
- логические;

Целочисленные данные.

Целые числа в ассемблере могут быть представлены в 1-байтной, 2-байтной, 4-байтной или 8-байтной форме. Целочисленные данные могут представляться в знаковой и беззнаковой форме.

Беззнаковые целые числа представляются в виде последовательности битов в диапазоне от 0 до 2^n-1, где n- количество занимаемых битов.

Знаковые целые числа представляются в диапазоне -2^{n-1} … $+2^{n-1}$-1. При этом старший бит данного отводится под знак числа (0 соответствует положительному числу, 1 – отрицательному).

Вещественные данные.

Вещественные данные могут быть 4, 8 или 10-байтными и обрабатываются математическим сопроцессором.

Логические данные представляют собой бит информации и могут записываться в виде последовательности битов. Каждый бит может принимать значение 0 (ЛОЖЬ) или 1 (ИСТИНА). Логические данные могут начинаться с любой позиции в байте.

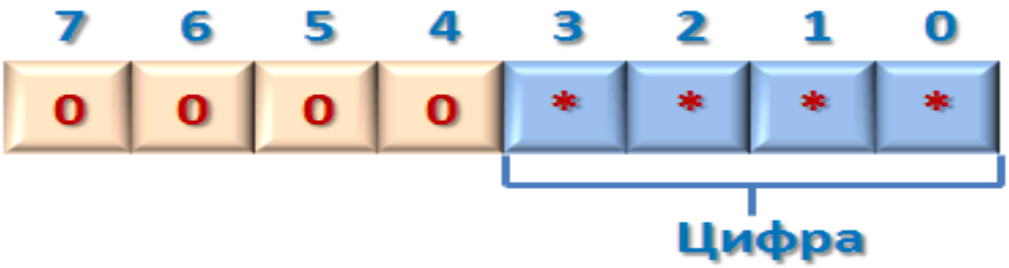
Символьные данные задаются в кодах и имеют длину, как правило, 1 байт (для кодировки ASCII) или 2 байта (для кодировки Unicode) .

Числа в двоично-десятичном формате

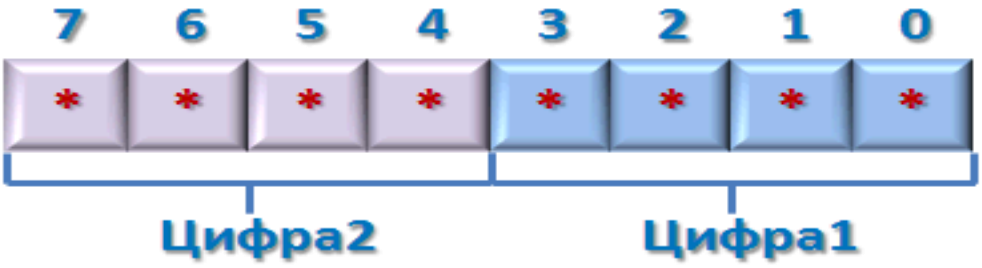
В двоично-десятичном коде представляются беззнаковые целые числа, кодирующие цифры от 0 до 9. Числа в двоично-десятичном формате могут использоваться в одном из двух видов:

- упакованном;
- неупакованном.

В **неупакованном** виде в каждом байте хранится одна цифра, размещенная в младшей половине байта (биты 3…0).



Упакованный вид допускает хранение двух десятичных цифр в одном байте, причем старшая половина байта отводится под старший разряд.



Числовые константы используются для обозначения арифметических операндов и адресов памяти. Для числовых констант в Ассемблере могут использоваться следующие числовые форматы:

Десятичный формат – допускает использование десятичных цифр от 0 до 9 и обозначается последней буквой d, которую можно не указывать, например, 125 или 125d. Ассемблер сам преобразует значения в десятичном формате в объектный шестнадцатеричный код и записывает байты в обратной последовательности для реализации прямой адресации:

a DB 12

Шестнадцатеричный формат – допускает использование шестнадцатеричных цифр от 0 до F и обозначается последней буквой h, например 7Dh. Так как ассемблер полагает, что с буквы начинаются идентификаторы, то первым символом шестнадцатеричной константы должна быть цифра от 0 до 9. Например, 0Eh.

a DB 0Ch

Двоичный формат – допускает использование цифр 0 и 1 и обозначается последней буквой b. Двоичный формат обычно используется для более четкого представления битовых значений в логических командах (AND, OR, XOR) :

a DB 00001100b

Восьмеричный формат – допускает использование цифр от 0 до 7 и обозначается последней буквой q или o, например, **253q**:

a DB 14q

Массивом называется последовательный набор однотипных данных, именованный одним идентификатором.

Цепочка - массив, имеющий фиксированный набор начальных значений.

Примеры инициализации цепочек:

M1 DD 0,1,2,3,4,5,6,7,8,9

M2 DD 0,1,2,3
DD 4,5,6,7
DD 8,9

Каждая из записей выделяет десять последовательных 4-байтных ячеек памяти и записывает в них значения 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Идентификатор **M1** определяет смещение начала этой области в сегменте данных **.DATA**.

Для инициализации всех элементов массива одинаковыми значениями используется оператор **DUP**:

Идентификатор Тип Размер DUP (Значение)

Идентификатор - имя массива;
Тип - определяет количество байт, занимаемое одним элементом;
Размер - константа, характеризующая количество элементов в массиве;
Значение - начальное значение элементов.
Например:

a DD 20 DUP (0)

описывает массив a из 20 элементов, начальные значения которых равны 0.

Если необходимо выделить память, но не инициализировать ее, в качестве поля **Значение** используется знак ?. Например:

b DD 20 DUP(?)

Символьные строки представляют собой набор символов для вывода на экран. Содержимое строки отмечается

- одиночными кавычками ' ', например, **'строка'**;
- двойными кавычками " ", например **"строка"**.

Символьная строка определяется только директивой **DB**, в которой указывается более одного символа в последовательности слева направо.

Символьная строка, предназначенная для корректного вывода, должна заканчиваться нуль-символом '\0' с кодом, равным 0:

Str DB 'Привет всем!', 0

Для перевода строки могут использоваться символы

- возврат каретки с кодом **13** (**0Dh**)
- перевод строки с кодом **10** (**0Ah**):
-

Stroka DB "Привет", 13, 10, 0

Лекция 4

Команды языка ассемблера

Цель лекции

Изучить команды языка ассемблера и их свойства**.**

Основные вопросы лекции:

1. Базовая система команд ассемблера.
2. Команды передачи данных.
3. Команды работы со стеком, ввода-вывода, арифметические.
4. Логические команды.
5. Прочие команды.

1. Базовая система команд микропроцессора

Для использования микропроцессора важно понимать, как работают команды: как обращаться к памяти (типы адресации), к регистрам, куда попадают результаты арифметических операций, как устроить условное ветвление (переходы) и т.д.

Каждая команда микропроцессора x86 состоит из одного, двух или более байт, причем первый байт – это **опкод** команды. Опкод определяет природу команды; по опкоду микропроцессор определяет, нужны ли дополнительные байты, и, если да, получает их в последующих циклах.

Поскольку опкод состоит из 8 бит, может существовать 256 разных опкодов. Ограничение в 256 команд было обойдено, так как некоторые опкоды (**0FEh**, **0FFh** и другие) служат шлюзами к следующим таблицам кодов. В приложении представлен полный список команд микропроцессора x86.

В таблице команды даны не в шестнадцатеричном машинном представлении (машинном коде), а в виде **мнемокодов** языка ассемблера, поскольку микропроцессор x86 обычно программируют на этом языке, обозначая адреса и числа легко запоминаемыми именами вместо чисел, с которыми в действительности оперирует микропроцессор.

После того как программа написана на языке ассемблера, стандартная программа, известная также под названием «ассемблер», преобразует мнемокоды и имена в числа. Таким образом, программа ассемблер переводит на объектный язык программы, составленные на входном языке.

В процессе этого перевода одна команда языка ассемблер может быть преобразована в 1, 2 или более байт объектного языка в зависимости от конкретной команды.

Базовую систему команд микропроцессора можно условно разделить на несколько групп по функциональному назначению (см. таб. 1).

- команды передачи данных;
- команды работы со стеком;
- команды ввода-вывода;
- арифметические команды;
- логические команды;
- сдвиговые команды;
- команды управления флагами;
- команды прерываний;
- команды передачи управления;
- команды поддержки языков высокого уровня;
- команды синхронизации работы процессора;
- команды обработки цепочки бит;
- строковые команды;

Кроме базовой системы команд процессора существуют также команды расширений:

X87 – расширение, содержащее команды математического сопроцес-сора (работа с вещественными числами);

MMX – расширение, содержащее команды для кодирования/деко-дирования потоковых аудио/видео данных;

SSE – расширение включает в себя набор инструкций, который производит операции со скалярными и упакованными типами данных;

SSE2 – модификация **SSE**, содержит инструкции для потоковой обработки целочисленных данных, что делает это расширение более предпочтительным для целочисленных вычислений, нежели использование набора инструкций MMX, появившегося ранее;

SSE3, SSE4 – содержат дополнительные инструкции расширения SSE.

В таблице приняты следующие обозначения:

r – регистр;

m – ячейка памяти;

c – константа;

8, **16**, **32** – размер в битах;

W – запись бита в регистре признаков;

U – значение бита в регистре признаков неизвестно после выполнения операции;

- – значение бита в регистре признаков не изменилось;

+ – значение бита в регистре признаков меняется в соответствии с результатом.

На все базовые команды процессора накладываются следующие ограничения:

1. Нельзя в одной команде оперировать двумя областями памяти одновременно.

Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения.

2. Нельзя оперировать сегментным регистром и значением непосредственно из памяти. Поэтому для выполнения такой операции нужно использовать промежуточный объект. Это может быть регистр общего назначения или стек.

3. Нельзя оперировать двумя сегментными регистрами. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных регистры общего назначения. Например,

```
mov ax,ds
mov es,ax ; es=ds
```

4. Нельзя использовать сегментный регистр **CS** в качестве операнда приемника, поскольку в архитектуре микропроцессора пара **CS:EIP** всегда содержит адрес команды, которая должна выполняться следующей. Изменение содержимого регистра **CS** фактически означало бы операцию перехода, а не модификации, что недопустимо.

5. Операнды команды, если это не оговаривается дополнительно в описании команды, должны быть одного размера.

Машинные коды для всех возможных сочетаний операторов команды

Описание машинного кода производится в шестнадцатеричном виде. При описании машинного кода используются следующие обозначения:

/цифра (от 0 до 7) показывает, что байт **mod r/m** кода операции использует только операнд **r/m**.

Поле **reg** содержит цифры которые обеспечивает расширение опткода;

/r — показывает, что байт **mod r/m** команды содержит как регистровый операнд, так и операнд **r/m;**

cb, cw, cd, cp, cq - одно-, двух-, четырех, шести-, восьмибайтное значение, следующее за полем кода операции и используемое для смещения в сегменте кода и возможно задает новое значение для регистра;

ib, iw, id, iq - одно-, двух-, четырех-, восьмибайтное непосредственное значение (число). Следует за опкод, **Mod R/M** или **SIB** (если таковые есть), при этом код операции определяет, является ли непосредственный операнд знаковым значением, а все слова, двойные и четверные слова приводятся в порядке «младший байт по младшему адресу»;

+rb, +rw, +rd, +rq, +i - код регистра от 0 до 7. Добавляется к байту слева от знака «+». В результате получается окончательный опкод.

2. Команды передачи данных

Основной командой передачи данных (таб. 1) является команда **MOV**, осуществляющая операцию присваивания:

MOV приемник, источник

Команда **MOV** присваивает значению операнда приемника значение операнда источника. В качестве **приемника** могут выступать регистр общего назначения, сегментный регистр или ячейка памяти, а в качестве **источника** могут выступать константа, регистр общего назначения, сегментный регистр или ячейка памяти. Оба операнда должны быть одного размера.

Команда обеспечивает копирование значения из операнда-источника в операнд- приемник.

Формат данной команды типичен для большинства арифметических и логических команд процессора. Оба операнда должны быть одного типа или иметь одинаковую длину.

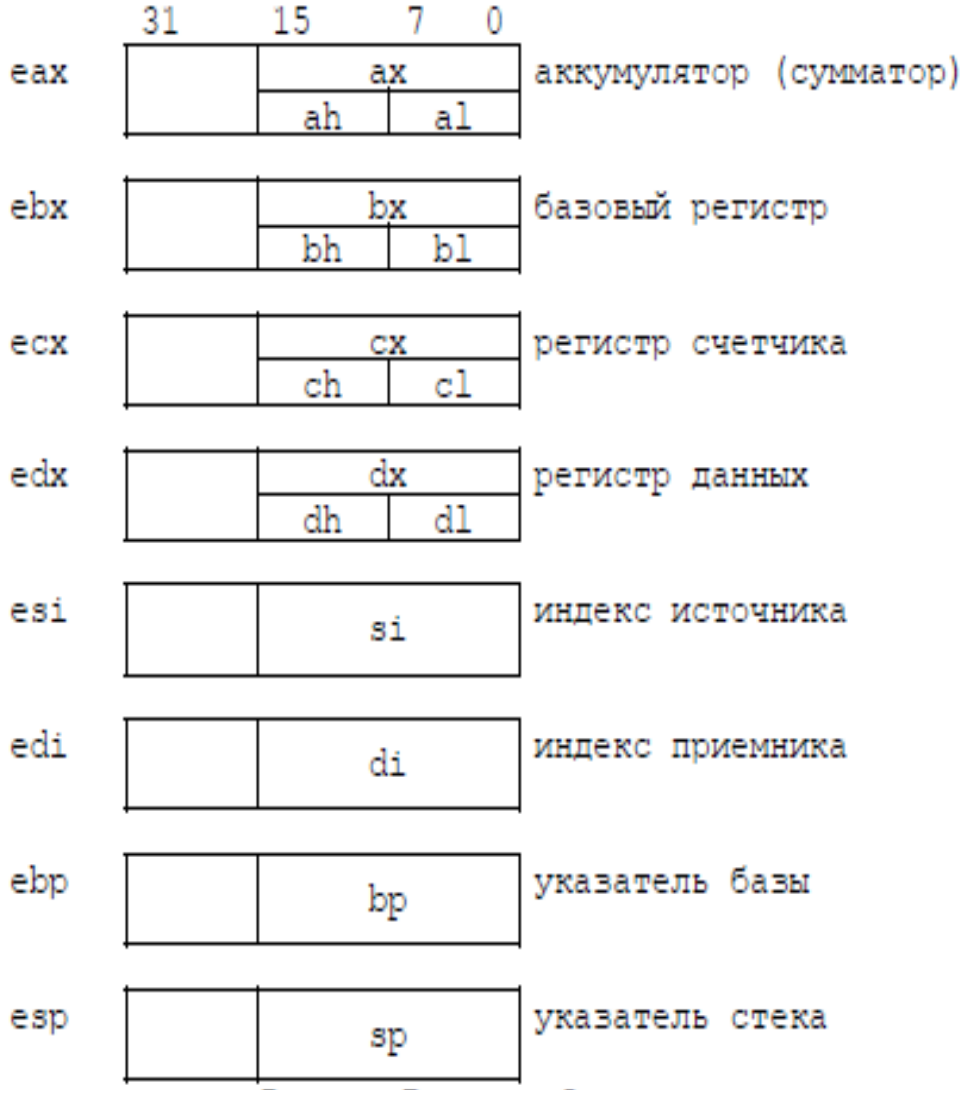


Рисунок – Регистры общего назначения

Примеры использования команды:

mov eax, ebx
mov result, ax **; result** – 16-битная переменная
mov ecx, count **; count** - 32-битная переменная
mov ah, 64 **; 64** – непосредственный операнд
mov var1, 100

Бывают случаи, когда не ясен размер операнда. Для такого уточнения можно использовать оператор переопределения типа **PTR**. Типы могут быть **byte**, **word**, **dword** и другие. Например, для получения младшего байта 32-битной переменной x можно использовать команду

mov al,byte ptr x.

Таблица 1 - Команды передачи данных

Команда	Операнды	Пояснение	Описание
MOV	r(m)8,r8 r(m)16,r16 r(m)32,r32 r8,r(m)8 r16,r(m)16 r32,r(m)32 r(m)8,c8 r(m)16,c16 r(m)32,c32	r(m)8=r8 r(m)16=r16 r(m)32=r32 r8=r(m)8 r16=r(m)16 r32=r(m)32 r(m)8=c8 r(m)16=c16 r(m)32=c32	Пересылка операндов
XCHG	r(m)8, r8 r8, r(m)8 r(m)16,r16 r16, r(m)16 r(m)32, r32 r32, r(m)32	r(m)8 ↔r8 r8 ↔r(m)8 r(m)16↔r16 r16 ↔r(m)16 r(m)32↔r32 r32 ↔r(m)32	Обмен операндов
BSWAP	r32	TEMP ← r32 r32[7..0]←TEMP[31..24] r32[15..8]←TEMP[23..16] r32[23..16]←TEMP[15..8] r32[31..24]←TEMP[7..0]	Перестановка байтов из порядка "младший – старший" в порядок "старший – младший"
MOVSX	r16, r(m)8 r32, r(m)8 r32, r(m)16	r16,r(m)8 DW ← DB r32,r(m)8 DD ← DB r32,r(m)16 DD ← DW	Пересылка с расширением формата и дублированием знакового бита
MOVZX	r16,r(m)8 r32,r/m8 r32,r/m16	r16,r(m)8 DW ← DB r32,r(m)8 DD ← DB r32,r(m)16 DD ← DW	Пересылка с расширением формата и дублированием нулевого бита
XLATXLATB	m8	AL=DS:[(E)BX+unsigned AL]	Загрузить в AL байт из таблицы в сегменте данных, на начало которой указывает EBX (BX); начальное значение AL играет роль смещения
LEA	r16, m r32, m	r16=offset m r32=offset m	Загрузка эффективного адреса
LDS	r16,m16 r32,m16	DS:r=offset m	Загрузить пару регистров из памяти
LSS		SS:r=offset m	
LES		ES:r=offset m	
LFS		FS:r=offset m	
LGS		GS:r=offset m	

3. Команды работы со стеком, ввода-вывода, арифметические

Команды установки единичного бита проверяют условие состояния битов регистра **EFLAGS** и, если условие выполняется, то младший бит операнда устанавливается в 1, в противном случае в 0. Анализ битов производится аналогично условным переходам.

Команда	Операнды	Пояснение
SETA, SETNBE	r(m)8	CF=0 и ZF=0
SETAE, SETNB, SETNC		CF=0
SETB, SETC, SETNAE		CF=1
SETBE, SETNA		CF=1 или ZF=1
SETE, SETZ		ZF=1
SETG, SETNLE		ZF=0 и SF=OF
SETGE, SETNL		SF=OF
SETL, SETNGE		SF!=OF
SETLE, SETNG		SF!=OF или ZF=1
SETNE, SETNZ		ZF=0
SETNO,		OF=0
SETNP, SETPO		PF=0
SETNS		SF=0
SETO		OF=1
SETP, SETPE		PF=1
SETS		SF=1

Команды работы со стеком

Команда	Операнды	Пояснение	Описание
PUSH	r(m)32 r(m)16 c32	ESP=ESP-4; SS:ESP=r(m)32/c SP=SP-2; SS:SP=r(m)16	Поместить операнд в вершину стека

POP	r(m)32
	r(m)16
		r(m)32=SS:ESP;
		ESP=ESP+4
		r(m)16=SS:SP;
		SP=SP+2;
			Извлечь операнд из вершины стека

PUSHA PUSHAD	r(m)32 r(m)16	-	Поместить в стек регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
POPA POPAD	-		Извлечь из стека содержимое и заполнить регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
PUSHF	-		Поместить в вершину стека регистр EFLAGS
POPF	-		Извлечь содержимое вершины стека и заполнить регистр EFLAGS

Команды ввода-вывода позволяют микропроцессору передавать данные в порты ввода-вывода, которые поддерживаются аппаратно и используют соответствующие своим предназначениям линии ввода-вывода процессора.

Аппаратное адресное пространство ввода-вывода процессора не является физическим адресным пространством памяти. Адресное пространство ввода-вывода состоит из 64 Кбайт индивидуально адресуемых 8-битных портов ввода-вывода, имеющих адреса **0...FFFFh**. Адреса **0F8h...0FFh** являются резервными.

Любые два последовательных 8-битных порта могут быть объединены в 16-битный порт, 4 последовательных 8-битных порта – в 32-битный порт.

Команды ввода-вывода

Команда	Операнды	Пояснение	Описание
IN	AL,c8 AX,c8 EAX,c8 AL,DX AX,DX EAX,DX	AL= port byte AX= port word EAX= port dword AL= [DX-port] AX= [DX-port] EAX= [DX-port]	Ввод из порта
OUT	c8, AL c8, AX c8, EAX DX, AL DX, AX DX, EAX	port byte=AL port word=AX port dword=EAX [DX-port]=AL [DX-port]=AX [DX-port]=EAX	Вывод в порт
INSB INSW INSD	-	ES:(E)DI = [DX-port]	Вводит данные из порта, адресуемого DX в ячейку памяти ES:[(E)DI]. После ввода 1, 2 или 4-байтного слова данных EDI/DI корректируется на 1,2,4. При наличии префикса REP процесс продолжается, пока ECX>0
OUTSB OUTSW OUTSD	-	[DX-port]=DS:(E)SI	Выводит данные из ячейки памяти, определяемой регистрами DS:[(E)SI], в порт, адрес которого находится в DX. После вывода данных производится коррекция указателя ESI/SI на 1,2 или 4

Команды целочисленной арифметики

Команда	Операнды	Пояснение	Описание
ADD	r(m)8,c8	r(m)8=r(m)8+c8	Сложение целых чисел
ADC	r(m)16,c16 r(m)32,c32 r(m)8,r8 r(m)16,r16 r(m)32,r32 r8,r(m)8 r16,r(m)16 r32,r(m)32	r(m)16=r(m)16+c16 r(m)32=r(m)32+c32 r(m)8=r(m)8+r8 r(m)16=r(m)16+r16 r(m)32=r(m)32+r32 r8=r8+r(m)8 r16=r16+r(m)16 r32=r32+r(m)32	Сложение целых чисел с учетом флага переноса CF
INC	r(m)8	r/m8=r/m8±1	Увеличение на 1

DEC	r(m)16 r(m)32	r(m)16=r(m)16±1 r(m)32=r(m)32±1	Уменьшение на 1
SUB	r(m)8,c8	r(m)8=r(m)8-c8	Вычитание целых чисел
SBB	r(m)16,c16 r(m)32,c32 r(m)8,r8 r(m)16,r16 r(m)32,r32 r8,r(m)8 r16,r(m)16 r32,r(m)32	r(m)16=r(m)16-c16 r(m)32=r(m)32-c32 r(m)8=r(m)8-r8 r(m)16=r(m)16-r16 r(m)32=r(m)32-r32 r8=r8-r(m)8 r16=r16-r(m)16 r32=r32-r(m)32	Вычитание с учетом флага переноса CF
CMP	r(m)8,c8 r(m)16,c16 r(m)32,c32 r(m)8,r8 r(m)16,r16 r(m)32,r32 r8,r(m)8 r16,r(m)16 r32,r(m)32	r(m)8-c8 r(m)16-c16 r(m)32-c32 r(m)8-r8 r(m)16-r16 r(m)32-r32 r8-r(m)8 r16-r(m)16 r32-r(m)32	Сравнение целых чисел По результату сравнения устанавливаются флаги CF PF AF ZF SF OF
NEG	r(m)8 r(m)16 r(m)32	r(m)8=-r(m)8 r(m)16=-r(m)16 r(m)32=-r(m)32	Изменение знака числа
MUL	r(m)8 r(m)16 r(m)32	AX=AL*r(m)8 DX:AX=AX*r(m)16 EDX:EAX=EAX*r(m)32	Умножение без знака
IMUL	r(m)8 r(m)16 r(m)32 r16,r(m)16 r32,r(m)32 r16,r(m)16,c r32,r(m)32,c r16,c r32,c	AX=AL*r(m)8 DX:AX=AX*r(m)16 EDX:EAX=EAX*r(m)32 r16=r16*r(m)16 r32=r32*r(m)32 r16=r(m)16*c16 r32=r(m)32*c32 r16=r16*c16 r32=r32*c32	Умножение со знаком
DIV	r(m)8	AL=AX/r(m)8, AH=mod	Деление без знака
IDIV	r(m)16 r(m)32	AX=DX:AX/r(m)16, DX=mod EAX=EDX:EAX/r(m)32, EDX=mod	Деление со знаком

Особого внимания среди рассмотренных команд целочисленной арифметики заслуживает команда CMP, которая вычитает второй операнд из первого и не сохраняет результат, а устанавливает биты OF, SF, ZF, AF, PF, CF регистра признаков EFLAGS в соответствии с результатом. Команда CMP чаще всего предшествует командам знакового или беззнакового условных переходов.

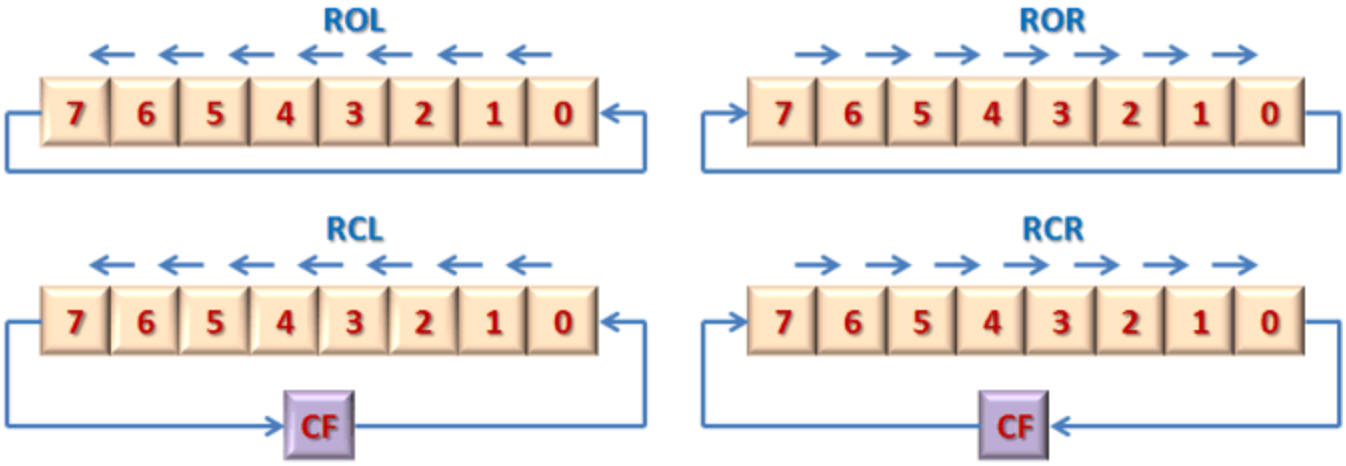
4. Логические команды

Команда	Операнды	Пояснение	Описание
AND	r(m)8,c8 r(m)16,c16 r(m)32,c32 r(m)8,r8 r(m)16,r16 r(m)32,r32 r8,r(m)8 r16,r(m)16 r32,r(m)32	r(m)8=r(m)8 Ф c8 r(m)16=r(m)16 Ф c16 r(m)32=r(m)32 Ф c32 r(m)8=r(m)8 Ф r8 r(m)16=r(m)16 Ф r16 r(m)32=r(m)32 Ф r32 r8=r8 Ф r(m)8 r16=r16 Ф r(m)16 r32=r32 Ф r(m)32	Логическое умножение (И), конъюнкция
OR			Логическое сложение (ИЛИ), дизъюнкция
XOR			Исключающее ИЛИ
NOT	r(m)8 r(m)16 r(m)32	r(m)8=~r(m)8 r(m)16=~r(m)16 r(m)32=~r(m)32	Логическое отрицание (НЕ), инверсия
TEST	r(m)8,c8 r(m)16,c16 r(m)32,c32 r(m)8,r8 r(m)16,r16 r(m)32,r32 r8,r(m)8 r16,r(m)16 r32,r(m)32	r(m)8 & c8 r(m)16 & c16 r(m)32 & c32 r(m)8 & r8 r(m)16 & r16 r(m)32 & r32 r8 & r(m)8 r16 & r(m)16 r32 & r(m)32	Логическое умножение без сохранения результата. В соответствии с результатом устанавливаются флаги PF ZF SF

Сдвиговые команды

Команда	Операнды	Пояснение	Описание
SHR	r(m)8 r(m)8,CL r(m)8,c r(m)16 r(m)16,CL r(m)16,c r(m)32 r(m)32,CL r(m)32,c	r(m)8 на 1 разряд r(m)8 на CL разрядов r(m)8 на c разрядов r(m)16 на 1 разряд r(m)16 на CL разрядов r(m)16 на c разрядов r(m)32 на 1 разряд r(m)32 на CL разрядов r(m)32 на c разрядов	Логический сдвиг вправо
SAR			Арифметический сдвиг вправо (старшие разряды заполняются значением знакового)
SHL SAL			Логический (арифметический) сдвиг влево
ROR			Циклический сдвиг вправо
ROL			Циклический сдвиг влево
RCR			Циклический сдвиг вправо через перенос
RCL			Циклический сдвиг влево через перенос

Команды циклического сдвига выполняются в соответствии со схемой



5. Прочие команды

Команды коррекции двоично-десятичных чисел не имеют операндов и используют операнд по умолчанию, хранящийся в регистре AX (паре регистров AH:AL).

Команды коррекции двоично-десятичных чисел

Команда	Пояснение	Описание
AAA	if((AL&0Fh)>9 || AF) { AH=AH+1; AL=(AL+6) & 0Fh; CF:=1; AF:=1;}	Коррекция AX после сложения двух неупакованных двоично-десятичных чисел
AAS	if((AL&0Fh)>9)|| AF) { AH=AH-1; AL=(AL-6)& 0Fh; CF=1; AF=1;}	Коррекция AX после вычитания двух неупакованных двоично-десятичных чисел
AAM	AH=AL/10; AL=AL%10;	Коррекция AX после умножения двух неупакованных двоично-десятичных чисел
AAD	AL=AH*10+AL; AH=0;	Коррекция AX перед делением двух неупакованных двоично-десятичных чисел

DAA	old_AL = AL;
	old_CF = CF;
	if(((AL & 0x0F)>9) || AF==1) {
	AL = AL + 6;
	CF = old_CF | CF;
	AF = 1;}
	else
	AF = 0;
	if((old_AL > 99h) || (CF==1)) {
	AL = AL + 60h;
	CF = 1;}
	else
	CF = 0;
		Коррекция AL после сложения двух упакованных двоично-десятичных чисел

DAS	old_AL = AL;
	old_CF = CF;
	if(((AL & 0x0F)>9) ||
	AF==1) {
	  AL = AL - 6;
	  CF = old_CF | CF;
	  AF = 1;}
	else
	  AF = 0;
	if((old_AL > 99h) ||
	(CF==1)) {
	  AL = AL + 60h;
	  CF = 1;}
	else
	  CF = 0;
		Коррекция AL после вычитания двух упакованных двоично-десятичных чисел

Команды преобразования типов предназначены для корректного изменения размера операнда, заданного неявно в регистре-аккумуляторе (EAX, AX, AL). Непосредственно после аббревиатуры команды операнд не указывается.

Команды преобразования типов

Команда	Пояснение	Описание
CBW	AX=(DW)AL	2 байта ← 1 байт
CWDE	EAX=(DD)AX	4 байта ← 2 байта
CWD	DX:AX=(DD)AX	4 байта ← 2 байта
CDQ	EDX:EAX=(DQ)EAX	8 байт ← 4 байта

Команды управления флагами предназначены для сброса или установки соответствующего бита регистра признаков EFLAGS. Команды управления флагами не имеют операндов.

Команды управления флагами

Команда	Пояснение	Описание
CLC	CF = 0	Сброс бита переноса
CLD	DF=0	Сброс бита направления
CMC	CF=!CF	Инверсия бита переноса
STC	CF=1	Установка бита переноса
STD	DF=1	Установка бита направления
STI	IF=1	Установка бита прерывания

Команды прерываний предназначены для управления программными прерываниями.

Прерывание – это, как правило, асинхронная остановка работы процессора, вызванная началом работы устройства ввода-вывода. Исключением являются синхронные прерывания, возникающие при определении некоторых предопределенных условий в процессе выполнения команды.

Когда поступает сигнал о прерывании, процессор останавливает выполнение текущей программы и переключается на выполнение обработчика прерывания, заранее записанного для каждого прерывания.

Архитектура IA-32 поддерживает 17 векторов аппаратных прерываний и 224 пользовательских.

Команда INT вызывает обработчик указанного операндом прерывания (константой). Операнд определяет номер вектора системного прерывания BIOS от 0 до 255, представленный в виде беззнакового 8-битного целого числа. При вызове обработчика прерывания в стеке сохраняются регистры EIP, CS и EFLAGS.

Прерывание по переполнению вызывается отдельной командой INTO и имеет вектор 04h.

Команды прерываний

Команда	Пояснение	Описание
INT c	EIP → стек CS → стек EFLAGS → стек переход к вектору c	Программное прерывание
INTO	OF=1	Прерывание по переполнению
IRET	EFLAGS ← стек CS ← стек EIP ← стек возврат	Возврат из обработчика прерывания

Команды передачи управления

Команда	Операнды	Пояснение	Описание
JMP	метка r(m)16 r(m)32	метка адрес в r(m)16 адрес в r(m)32	Безусловный переход на адрес, указанный операндом

Команды обращения к процедуре (функции)

Команда	Операнды	Пояснение	Описание
CALL	метка r(m)16 r(m)32	метка адрес в r(m)16 адрес в r(m)32	Вызов процедуры, указанной операндом
RET	- c16	- Удаляет из стека c16 байт	Возврат из процедуры

Команды поддержки языков высокого уровня

Команда	Операнды	Пояснение	Описание
ENTER	c16, c8	PUSH EBP MOV EBP, ESP	Подготовка стека при входе в процедуру. Константа c16 указывает количество байт, резервируемых в стеке для локальных идентификаторов, константа c8 определяет вложенность процедуры
LEAVE	-	POP EBP	Приведение стека в исходное состояние
BOUND	r16, m16&16 r32, m32&32	m16<r16<m16&16 m32<r16<m32&32	Проверка индекса массива: сравнивает значение в регистре, заданном первым операндом с двумя значениями, расположенными последовательно в ячейке памяти, адресуемой вторым операндом.

Команды организации циклов - используют регистр ECX по умолчанию в качестве счетчика числа повторений цикла. Каждый раз при выполнении команды LOOP значение регистра ECX уменьшается на 1, а затем сравнивается с 0. Если ECX=0, выполнение цикла заканчивается, и продолжает выполняться код программы, записанный после команды LOOP. Если ECX содержит ненулевое значение, то осуществляется переход по адресу операнда команды LOOP.

Команды организации циклов

Команда	Операнды	Пояснение	Описание
LOOP	метка	ECX=ECX-1; if(CX>=0) EIP=метка;	Переход если ECX>0
LOOPE LOOPZ		ECX=ECX-1; if(ECX>0 && ZF==1) EIP=метка;	Переход если ECX>0 и ZF=1
LOOPNE LOOPNZ		ECX=ECX-1; if(ECX>0 && ZF==0) EIP=метка;	Переход если ECX>0 и ZF=0

Команды условных переходов - проверяют состояние одного или нескольких битов регистра признаков и при выполнении условия осуществляют передачу программного управления в другую точку кода, задаваемую операндом. Указанный класс команд не запоминает информацию для возврата. Операнд определяет адрес команды, которой должно быть передано управление.

Команды условных переходов

Команда	Операнды	Пояснение	Описание
JCXZ	метка	if(ECX==0) EIP=метка;	Переход при ECX=0
JC		if(CF==1) EIP=метка;	Переход при переносе (CF=1)
JNC		if(CF==0) EIP=метка;	Переход при отсутствии переноса (CF=0)
JS		if(SF==1) EIP=метка;	Переход при отрицательном результате (SF=1)
JNS		if(SF==0) EIP=метка;	Переход при неотрицательном результате (SF=0)
JE JZ		if(ZF==1) EIP=метка;	Переход при нулевом результате (ZF=1)
JNE JNZ		if(ZF==0) EIP=метка;	Переход при ненулевом результате (ZF=0)
JP JPE		if(PF==1) EIP=метка;	Переход по четности (PF=1)
JNP JPO		if(PF==0) EIP=метка;	Переход по нечетности (PF=0)
JO JNZ		if(OF==1) EIP=метка;	Переход при переполнении (OF=1)
JNO JNZ		if(OF==0) EIP=метка;	Переход при отсутствии переполнения (OF=0)

Команды беззнаковых переходов предназначены для сравнения беззнаковых величин и, как правило, используются непосредственно после команды сравнения CMP:

cmp m1, m2 ; сравнение m1 и m2, m1-m2

В аббревиатурах команд используются следующие обозначения:

- A (above) - выше;
- B (below) - ниже;
- E (equal) - равно.

Команды беззнаковых переходов

Команда	Операнды	Пояснение	Описание

JB
JNAE
	метка	cmp m1,m2
		if(CF==1)
		EIP=метка;
			Переход если ниже:
			m1<m2

JBE
JNA
		cmp m1,m2
		if(CF==1 || ZF==1)
		EIP=метка;
			Переход если не выше: m1<=m2

JAE
JNB
		cmp m1,m2
		if(CF==0)
		EIP=метка;
			Переход если не ниже: m1>=m2

JA
JNBE
		cmp m1,m2
		if(CF==0 && ZF==0)
		EIP=метка;
			Переход если выше: m1>m2

Команды знаковых переходов предназначены для сравнения знаковых величин и, как правило, используются непосредственно после команды сравнения CMP:

cmp m1, m2 ; сравнение m1 и m2, m1-m2

В аббревиатурах команд используются следующие обозначения:

- L (less) - меньше;
- G (greater) - больше;
- E (equal) - равно.

Команды знаковых переходов

Команда	Операнды	Пояснение	Описание

JL
JNGE
	метка	cmp m1,m2
		if(SF != OF)
		EIP=метка;
			Переход если меньше:
			m1<m2

JLE
JNG
		cmp m1,m2
		if((SF != OF) || ZF==1)
		EIP=метка;
			Переход если не больше: m1<=m2

JGE
JNL
		cmp m1,m2
		if(SF==OF)
		EIP=метка;
			Переход если не меньше: m1>=m2

JG
JNLE
		cmp m1,m2
		if((SF==OF) && ZF==0)
		EIP=метка;
			Переход если больше: m1>m2

Команды синхронизации работы процессора

Команда	Описание
HLT	Остановка процессора до внешнего прерывания
LOCK	Префикс блокировки шины. Заставляет процессор сформировать сигнал LOCK# на время выполнения находящейся за префиксом команды. Этот сигнал блокирует запросы шины другими процессорами в мультипроцессорной системе
WAIT	Ожидание завершения команды сопроцессора. Большинство команд сопроцессора автоматически вырабатывают эту команду
NOP	Пустая операция
CPUID	Получение информации о процессоре. Возвращаемое значение зависит от параметра в EAX

Команды побитового сканирования

Команда	Операнды	Описание
BSR	r16,r(m)16 r32,r(m)32	Ищет 1 в операнде 2, начиная со старшего бита. Если 1 найдена, ее индекс записывается в операнд 1
BSF		Ищет 1 в операнде 2, начиная со младшего бита. Если 1 найдена, ее индекс записывается в операнд 1
BT	r(m)16,r16 r(m)32,r32 r(m)16,c8 r(m)32,c8	Тестирование бита с номером из операнда 2 в операнде 1 и перенос его значения во флаг CF.
BTC		Тестирование бита с номером из операнда 2 в операнде 1 и перенос его значения во флаг CF с инверсией.
BTR		Тестирование бита с номером из операнда 2 в операнде 1 и перенос его значения во флаг CF. Само значение бита сбрасывается в 0
BTS		Тестирование бита с номером из операнда 2 в операнде 1 и перенос его значения во флаг CF. Само значение бита устанавливается в 1

Строковые команды предназначены для обработки цепочек данных. Операнды в строковых командах задаются по умолчанию. Как правило,

- операнд-источник адресуется регистром ESI внутри сегмента, на который указывает DS.
- операнд-источник адресуется регистром EDI внутри сегмента, на который указывает ES.

Строковые команды

Команда	Операнды	Пояснение	Описание
MOVSB	1 байт	ES:EDI = DS:ESI	Копирование строки
MOVSW	2 байта		
MOVSD	4 байта		
LODSB	1 байт	AL = DS:ESI	Загрузка строки
LODSW	2 байта	AX = DS:ESI	
LODSD	4 байта	EAX = DS:ESI	
STOSB	1 байт	ES:EDI = AL	Сохранение строки
STOSW	2 байта	ES:EDI = AX	
STOSD	4 байта	ES:EDI = EAX	
SCASB	1 байт	поиск AL в ES:EDI	Поиск данных в строке
SCASW	2 байта	поиск AX в ES:EDI	
SCASD	4 байта	поиск EAX в ES:EDI	
CMPSB	1 байт	поиск DS:ESI в ES:EDI	Поиск данных в строке
CMPSW	2 байта		
CMPSD	4 байта		

Перед выполнением строковых команд содержимое индексных регистров ESI, EDI должно быть проинициализировано. Сегментные регистры DS, ES должны указывать на соответствующие сегменты данных.

Для повторения выполнения строковых команд используются префиксы. Например, чтобы скопировать строку по байтам, используется команда

REP MOVSB

Здесь префикс REP сообщает процессору о том, что команда MOVSB должна повторяться. Максимальное количество повторений задается до вызова строковой команды в регистре ECX. Каждое выполнение строковой команды уменьшает содержимое регистра ECX на 1 и результат сравнивает с 0. В случае если ECX=0 выполнение повторяющейся строковой команды прекращается, и продолжается выполнение оставшейся части программы. Каждое повторение строковой команды также изменяет содержимое используемых индексных регистров (ESI, EDI) на размер операнда, заданный в команде (1, 2 или 4 байта). Направление модификации индексных регистров задается битом направления DF:

- **DF=0**: содержимое используемых индексных регистров увеличивается на размер операнда при каждом повторении.
- **DF=1**: содержимое используемых индексных регистров уменьшается на размер операнда при каждом повторении.

Префиксы, используемые для повторения строковых команд, представлены в таблице.

Префиксы строковых команд

Префикс	Команды, с которыми используется	Описание
REP	MOVS* LODS* STOS*	Повторение
REPE REPZ	SCAS* CMPS*	Повторение пока операнды равны
REPNE REPNZ	SCAS* CMPS*	Повторение пока операнды не равны

Лекция 5

Этапы создания программы на языке ассемблера

Цель лекции

Изучить команды языка ассемблера и их свойства**.**

Основные вопросы лекции:

1. Подготовка текста программы.
2. Структура программы на языке ассемблера.
3.
4.

Разработка программы включает несколько этапов:

1. Подготовка (изменение) исходного текста программы,
2. Ассемблирование программы (получение объектного кода),
3. Компоновка программы (получение исполняемого файла программы),
4. Запуск программы,
5. Отладка программы.

Обычно эти этапы циклически повторяются, потому что при нахождении ошибок при ассемблировании, компоновке или отладке прграммы приходится вновь возвращаться к первому этапу и изменять код для устранения ошибок.

1. Подготовка текста программы

Данные правила относятся не только к программированию на языке ассемблера, но и к программированию на других языках. Может быть, их трудно понять, не имея навыка в программировании, но «незнание основ не освобождает от ответственности».

Начинайте с комментариев

Начните с написания инструкции для пользователя — для чего создается и каковы возможности вашей программы. А теперь немного усложните вашу инструкцию по применению вашей программы, подразумевая под «пользователем» программиста, использующего написанный вами код — зачастую этим программистом-исследователем будете вы сами.

Акт записи на обычном языке описания того, что делает программа и что делает каждая функция в программе, является критическим шагом в мыслительном процессе. Хорошо построенное, грамматически правильное предложение — признак ясного мышления. Если вы не можете это записать, то велика вероятность того, что вы не полностью продумали задачу или метод ее решения.

Плохая грамматика и построение предложения являются также показателем поверхностного мышления. Поэтому первый шаг в написании любой программы - записать, что именно и как делает программа.

Итак, комментарии для вашей программы уже готовы. Теперь возьмите ваше описание по использованию и добавьте вслед за каждым абзацем блоки кода, реализующие функции, описанные в этом абзаце. Оправдание: «У меня не было времени, чтобы добавить комментарии» на самом деле означает — «Я писал этот код без проекта системы и у меня нет времени воспроизвести его».

Если создатель программы не может воспроизвести идеи, воплощенные в программный проект, то кто же тогда сможет?

Работа программиста состоит из двух частей: разработать приложение для пользователя и сделать возможным дальнейшее сопровождение программы. Единственный способ решить вторую часть задачи - комментировать код.

Причем комментарии должны описывать не только то, что делает код, но и предположения, принятый подход и причины, по которым вы выбрали именно его. Кроме того, необходимо, чтобы комментарии также соответствовали коду.

Хотя вы можете считать, что ваш код полностью очевиден и может служить примером ясности, без правильных комментариев понять его постороннему достаточно трудно. Парадокс заключается в том, что спустя неделю вы сами можете оказаться в роли этого постороннего.

Старайтесь использовать следующий подход при написании комментариев:

- перед каждой функцией или методом размещается одно или два предложения со следующей информацией:
  - что делает программа;
  - возникающие при этом предположения о программе;
  - что должно содержаться во входных параметрах;
  - что должно содержаться во выходном параметре в случае успешного или неудачного завершения;
  - все возможные выходные значения;
- перед каждой не совсем очевидной частью функции следует поместить одно или два предложения, объясняющие выполняемые действия;
- любой интересный алгоритм заслуживает подробного описания;
- любая нетривиальная ошибка, устраненная в коде, должна комментироваться, при этом нужно привести номер ошибки и описать сделанное исправление;
- правильно размещенные операторы диагностики, проверки условий, а также соглашения об именах переменных могут также служить хорошими комментариями и передавать содержание кода;
- писать комментарии так, будто сами собираетесь заниматься его поддержкой через пять лет;
- если возникла мысль «это хитро сделано» или «это ловкий трюк» - лучше переписать данную функцию, а не комментировать ее.

Читайте код

Все писатели — это читатели. Вы учитесь, когда смотрите, что делают другие писатели. Читатель может найти ошибки, которые вы не увидели, и подать мысль, как улучшить код. Лучше присесть с коллегой и просто разобрать код строка за строкой, объясняя, что и как делается, получить какую-то обратную связь и совет.

Для того чтобы подобное упражнение принесло пользу, автор кода не должен делать никаких предварительных пояснений. Читатель должен быть способен

понимать код в процессе чтения. Если вам пришлось объяснять что-то вашему читателю, то это значит, что ваше объяснение должно быть в коде в качестве комментария. Добавьте этот комментарий, как только Вы его произнесли; не откладывайте этого до окончания просмотра.

Разлагайте сложные проблемы на задачи меньшего размера

На самом деле это также и правило литературного стиля. Если очень трудно объяснить точку зрения за один раз, то разбейте изложение на меньшие части и по очереди объясняйте каждую. То же самое назначение у глав в книге и параграфов в главе.

Программа должна писаться не менее двух раз

Хороший код программы должен быть сначала написан, а затем отредактирован в целях улучшения (под «редактированием» имеется в виду «исправление»). Редактирование, как правило, приводит к сокращению кода, а небольшие программы выполняются быстрее.

Оптимизация программ на языке ассемблера

Итак ваша программа заработала, а теперь постарайтесь переделать ее так, чтобы она стала максимально компактной и в тоже время максимально быстродействующей.

Такая оптимизация достигается в три этапа:

- Алгоритмическая оптимизация то есть подбор алгоритма, который выполняет вашу задачу более быстрым способом и позволит сократить не пять, а пятьдесят операторов;
- Подстройка программы под конкретное оборудование;
- Замена некоторых ассемблерных команд на машинный код. Тщательный анализ машинного кода, вырабатываемого транслятором, позволяют прийти к выводу, что некоторые коды человек может выработать более оптимально, чем программа.

2. Структура программы на языке ассемблера

Как вы увидите в следующих лекциях, язык ассемблера заставляет нас помещать определенное количество строк в качестве заголовка программ, которые мы пишем. Другими словами, нам нужно каждый раз записывать несколько псевдооператоров, которые сообщают языку ассемблера основную информацию. В качестве рекомендации на будущее ниже приведен абсолютный минимум, необходимый для программ, которые вы пишите:

```
.686P              ; директива описания типа микропроцессора
.model flat        ; плоская модель памяти  - для 32-разрядной Windows
.code
start:
                   ;тело  программы
ret
.data
                   ;данные вашей программы
 end start          ; конец программы и сообщение компилятору, что
                  ;начинать выполнение  надо с метки START.
```

Разберем ее подробнее. В первой строке **.686P** – это директива описания типа микропроцессора (может быть еще и .8086, .8087, .186, .286, .287, .386, .387, .486, .586, .mmx с добавлением или без добавления букв **P** (привилегированные команды) и **C** или **N** (непривилегированные команды)). Если не указывать тип микропроцессора, то программа будет сгенерирована в кодах i8086.

Директива описания типа микропроцессора

Директива	Назначение
.8086	Разрешены инструкции базового процессора i8086 (и идентичные им инструкции процессора i8088). Запрещены инструкции более поздних процессоров.
.186 .286 .386 .486 .586 .686	Разрешены инструкции соответствующего процессора x86 (x=1,…,6). Запрещены инструкции более поздних процессоров.
.187 .287 .387 .487 .587	Разрешены инструкции соответствующего сопроцессора x87 наряду с инструкциями процессора x86. Запрещены инструкции более поздних процессоров и сопроцессоров.
.286c .386c .486c .586c .686c	Разрешены НЕПРИЛЕГИРОВАННЫЕ инструкции соответствующего процессора x86 и сопроцессора x87. Запрещены инструкции более поздних процессоров и сопроцессоров.
.286p .386p .486p .586p .686p	Разрешены ВСЕ инструкции соответствующего процессора x86, включая привилегированные команды и инструкции сопроцессора x87. Запрещены инструкции более поздних процессоров и сопроцессоров.
.mmx	Разрешены инструкции MMX-расширения.
.xmm	Разрешены инструкции XMM-расширения.
.K3D	Разрешены инструкции AMD 3D.

Модель памяти задается директивой **.model**. Строка **.model flat** говорит, что будет создаваться exe-файл для 32-разрядной операционной системы Windows.

Плоская (**flat**) модель памяти 32-разрядной Windows располагает три сегмента (сегмент кода, стека и данных) в едином четырехгигабайтном адресном пространстве, позволяя вообще **забыть о существовании сегментов**. Но для 16-разрядных приложений MS-DOS и Windows 3.x максимально допустимый размер сегментов составляет всего лишь 64 килобайта, что явно недопустимо для большинства приложений.

В крошечной (**tiny**) модели памяти сегмент кода, стека и данных также расположены в едином 64-килобайтном адресном пространстве, но в отличие от плоской модели это адресное пространство чрезвычайно ограничено в размерах, поэтому и код, и стек, и данные более серьезных приложений приходилось размещать в нескольких сегментах (модели памяти **small, medium, compact, large, huge, tchuge**).

В этих моделях памяти, например, для вызова функции недостаточно было знать ее смещение, а требовалось указать еще и сегмент, в котором функция была расположена.

Команды передачи управления **переходы** (**jmp**) и **вызовы** (**call**) в этих моделях памяти могут быть близкими (**near**) и дальними (**far**). Если вы пишете программы для 32/64-разрядной операционной системы Windows, то о других моделях памяти кроме **flat** можно забыть со спокойной совестью.

Для адресации четырех гигабайтов виртуальной памяти, выделенной в распоряжение процесса, Windows использует два селектора, один из которых загружается в сегментный регистр **CS**, а другой в регистры **DS**, **ES** и **SS**. Оба селектора ссылаются на один и тот же базовый адрес памяти, равный нулю, и имеют идентичные лимиты, равные четырем гигабайтам. Windows использует еще и регистр **FS**, в который загружается селектор сегмента, содержащего информационный блок потока **TIB.**

Фактически существует всего один сегмент, вмещающий в себя и код, и данные, и стек процесса. Благодаря этому передача управления коду, расположенному в стеке, осуществляется близким (**near**) вызовом или переходом. Отличия между регионами кода, стека и данных заключаются в атрибутах принадлежащих им страниц: страницы кода допускают чтение и исполнение, страницы данных чтение и запись, а страницы стека чтение, запись и исполнение одновременно.

Допустим у нас есть логический адрес **0137:00456789h**. Чтобы этот адрес перевести в линейный – в селекторе **137** находится соответствующий ему дескриптор в таблице дескрипторов: база = **0**, граница = **0FFFFFFFh**, следовательно, линейный адрес равен **0** (база) **+ 00456789h.**

Однако линейный адрес не является физическим адресом. Для его получения используется **третья ступень** – страничная адресация. То есть 20 старших бит линейного адреса используются для выбора 4 Кбайт памяти из каталога страниц, оставшиеся 12 бит представляют смещение внутри полученной страницы (в качестве упражнения рекомендую написать небольшую программу под 32-разрядной Windows, которая будет показывать сегментные регистры, значения дескрипторов для каждого селектора, базу, границу, RPL и т.п).

В 32-разрядной Windows и сегмент кода, и сегмент данных и стека приложения имеют одинаковые базу и границу (**0** и **0FFFFFFFh**). Это называется плоской (**FLAT**) моделью памяти. Хотя **cs** и **ds** имеют разные значения и дескрипторы, они указывают на одно и то же линейное адресное пространство **0..0FFFFFFFh**. Следовательно, логические адреса **cs:12345678** и **ds:12345678** совпадают.

Лекция 6

Создание программ на языке ассемблера в среде Masm32 с отладчиком Ollydbg

Цель лекции: изучить начальные сведения о программировании на языке ассемблера в среде Masm32**.**

Основные вопросы лекции:

1. Этапы создания программ на языке Ассемблера.
2. Подготовка текста программы в Masm32.
3. Получение объектного кода в Masm32.
4. Получение исполняемого файла программы в Masm32.
5. Отладка программы.
6. Регистры 32-разрядного МП Intel.
7. Способы адресации 32-разрядного МП Intel.
8. Отладчик OllyDbg.

1. Этапы создания программ на языке Ассемблера

Ассемблер, как язык программирования, является набором мнемонических обозначений машинных кодов, которые понимают вычислительные устройства. Пока последние будут существовать, язык ассемблера будет актуален. Несмотря на бурное развитие математического обеспечения и алгоритмических языков высокого уровня, интерес к языку ассемблера постоянно растет.

Язык ассемблера помогает раскрыть секреты аппаратного и программного обеспечения. Большинство программистов работают с языками высокого уровня, где отдельный оператор превращается в множество микропроцессорных команд. Каждая команда языка ассемблера имеет взаимное однозначное соответствие с машинными командами, что дает основание считать его языком низкого уровня и отличает от таких языков высокого уровня, как Pascal, BASIC и C ++, для которых характерна трансляция одного оператора в множество машинных команд.

Чаще всего язык ассемблера используется для непосредственного управления операционной системой и для прямого доступа к аппаратуре. На ассемблере пишут оптимальные драйверы и небольшие утилиты, прошивки BIOS, загрузчики и ядра ОС, "движки" игрушек, вирусы, компиляторы и многие иное. Ассемблер служит инструментом для связи с нетрадиционными внешними устройствами. Также он необходим при оптимизации критических блоков в приложениях с целью повышения их быстродействия.

Известно, что понять работу компьютера можно, только освоив ассемблер и управление компьютером на самом нижнем уровне. Знание ассемблера и умение работать с компьютером на самом нижнем уровне обязательны для системного программиста.

Язык ассемблера существует для каждого типа процессоров или семейства микропроцессоров.

Ассемблер может создавать листинг программы с номерами строк, адресами переменных, операторами и таблицей перекрестных ссылок символов и переменных. Наряду с ассемблером используется программа, называемая компоновщиком (**linker**), которая объединяет отдельные файлы, созданные ассемблером, в единую исполняемую программу.

При изучении языка ассемблера и разработке реальных программ на этом языке под операционную систему (ОС) Windows процесс распределения ячеек памяти для сохранения кодов команд и чисел выполняется автоматизировано.

Этапы создания программы на языке ассемблера следующие:

- подготовка (или внесение изменений) исходного текста программы;
- ассемблирование программы (получение объектного кода)
- компоновка программы (получение исполняемого файла программы);

- отладка программы (исправление ошибок).

Обычно эти этапы циклически повторяются, поскольку при обнаружении ошибок на всех этапах приходится возвращаться к первому этапу и вносить изменения в текст программы для исправления ошибок.

Ассемблеры бывают двух типов: однофазные и двухфазные.

Однофазные ассемблеры могут обрабатывать только такие программы, в которых символы появляются в поле названия до того, как на них дается ссылка в поле операндов.

Двухфазные ассемблеры транслируют программы в два этапа:

- в **первой фазе** трансляции ассемблер последовательно считает каждое предложение начальной программы, частично ее транслирует и строит полную таблицу символов;

во **второй фазе** ассемблер заканчивает трансляцию программы, используя таблицу символов первой фазы как входную информацию.

Для обоих типов ассемблеров символ, записанный в поле операндов, обязательно должен быть определен в поле названия, иначе будет сообщение об ошибке.

2. Подготовка текста программы в Masm32

Текст программы на языке ассемблера записывается в один или несколько текстовых файлов. Имена файлов и их расширения могут быть любые, но принято использовать расширение ***.asm**. Эти файлы являются текстовыми, их можно подготовить с помощью любого текстового редактора, например **Блокнота** или **NotePad**, и хранить в виде обычных файлов в формате ASCII. Использование **Word** нежелательно, потому что такой текст содержит большое количество служебных символов, которые являются невидимыми и приводят к ошибкам трансляции.

Возможно при подготовке текста использовать саму среду **Masm32.**

Структура программы под Win32 следующая:

```
TITLE <имя программы
.386                    ;директива определения типа микропроцессора
.model flat             ;задание линейной модели памяти
.data                   ;директива определения данных,
                        ;которые определены
.data?                  ;неинициализированные данные
.code                   ;директива начала кода программы
label:                  ;метка начала программы <code>
ret                     ;возвращение управления ОС
end label               ;окончание программы
```

3. Получение объектного кода в Masm32

Подготовленный текст программы с расширением ***.asm** является входными данными для программ, которые называются ассемблерами (например, программы **Masm**, **Tasm**, **Wasm**, **Fasm**). Задача ассемблера - превратить текст программы в форму двоичных команд, которые может выполнить МП. После ассемблирования получают файлы объектных модулей, имеющие расширение ***.obj**. Процесс ассемблирования называют трансляцией.

Для трансляции и линкования простых программ удобно использовать среду **masm32**. Ассемблер **Masm32** (**www.masm32.com**) необходимо устанавливать на системный диск **c:/**. Если поставить ассемблер **Masm32** не на системный диск, а на другой логический диск, то в таком случае необходимо конкретно указывать пути подключения библиотек расположения API-функций.

Программа **ml.exe** из Masm32 выполняет трансляцию исходного текста программы в промежуточный объектный файл. Программа **link.exe** выполняет компоновку объектного (объектных) файла (файлов) и библиотек в единую исполняемую программу. Эти exe-файлы находятся в папке **[bin]**.

Для трансляции и линкования программы необходимо запустить программу **qeditor** (файл qeditor.exe) из **masm32**, вставить в окно или набрать свою программу, сохранить ее с расширением **asm** и выбрать команды **Project→Assemble & Link**. Если ошибок не будет, то будет создан **exe**-файл.

4. Получение исполняемого файла программы в Masm32

После успешной трансляции исходного текста ассемблерной программы результат в виде объектного файла передается компоновщику link.exe, выполняющему объединение объектных модулей в один файл. Сегменты, определенные в программе, группируются в соответствии с инструкциями, содержащимися в объектном файле. Вся информация о размещении сегментов записывается в заголовок исполняемого файла.

Результатом компоновки является исполняемый файл, имеющий расширение ***.exe.**

В процессе трансляции исходного текста программы выполняются следующие действия:

1. Анализируются директивы условного ассемблирования, и в случае истинности указанных в них условий выполняются те или иные шаги.
2. Развертываются макросы.
3. Вычисляются константные выражения, при этом они замещаются вычисленными значениями.
4. Декодируются команды и операнды, которые не находятся в памяти Например, декодируется команда **mov EAX, 2008**, поскольку она не имеет операндов, расположенных в памяти.
5. Сохраняются смещения переменных в памяти относительно смещения в сегментах, в которых эти переменные расположены.
6. Сегменты и их атрибуты размещаются в объектном файле.
7. В объектный файл помещаются перемещаемые адреса (Relocatable addresses).
8. При необходимости создается файл листинга.
9. Непосредственно программе link.exe передаются некоторые директивы (например, INCLUDELIB).

Схема ассемблирования, компоновки и выполнения программы приведен на рис. 1.

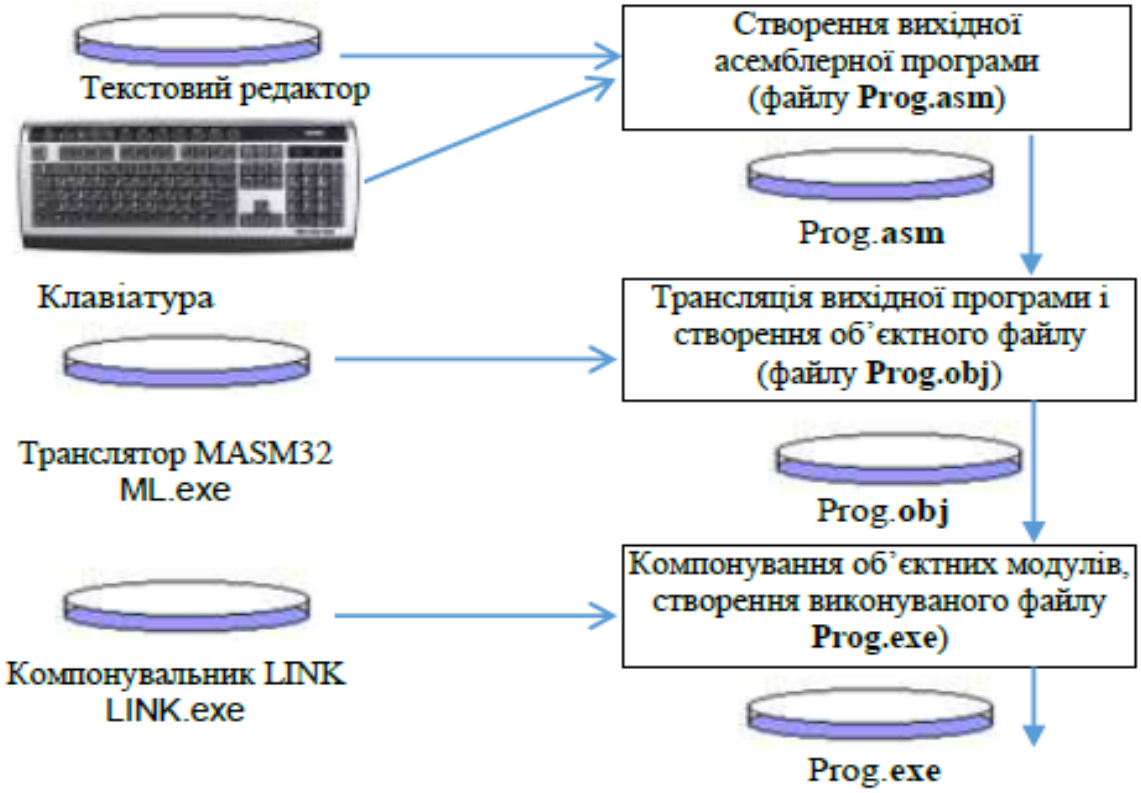


Рис. 1 - Схема ассемблирования, компоновки и выполнения программы

Структура программы (не только на ассемблере) определяется несколькими факторами:

- архитектурой процессора;
- особенностями той операционной системы, под управлением которой эта программа будет выполняться;

- правилами работы выбранного компилятора - разные компиляторы ставят различные требования к исходному тексту программы.

MASM и TASM поддерживают упрощенную сегментацию. Суть такой сегментации в том, что директивы .CODE и .DATA могут появляться в тексте программы несколько раз. Транслятор затем собирает код и данные вместе. Основной целью такого подхода является возможность приблизить в тексте программы данные к тем строкам, где они используются.

5. Отладка программы

Любая программа требует отладки (исправления ошибок). Современные отладочные программы для ассемблера платформы x86, например программы Dbg_x86 (http://www.microsoft.com/whdc/devtools/ debugging), SoftICE (уже не поддерживается, но еще используется), Emu8086 v.4.04, OllyDbg 1.10 (http://cracklab.ru/), OllyDbg 2 (http://www.ollydbg.de/), Syser (www.syser.com) и др. позволяют в процессе выполнения программы контролировать значения регистров общего назначения или переменных и изменять их.

С помощью отладчика можно, например, просматривать содержимое различных участков памяти, выполнять программу шаг за шагом, менять программный код, сохранять изменения в exe-файле и др.

Для отладки профессиональных ассемблерных программ, которые применяют последние версии команд параллельной обработки вещественных чисел или новые API-функции, которые постоянно появляются, используют также среду Visual Studio.

При написании программ необходимо использовать справочник **MSDN** (**http://msdn.microsoft.com/ru-ru/library**).

6. Регистры 32-разрядного МП Intel

Основные регистры процессора архитектуры IA-32, с которыми работают приложения, показаны на рис. 2.

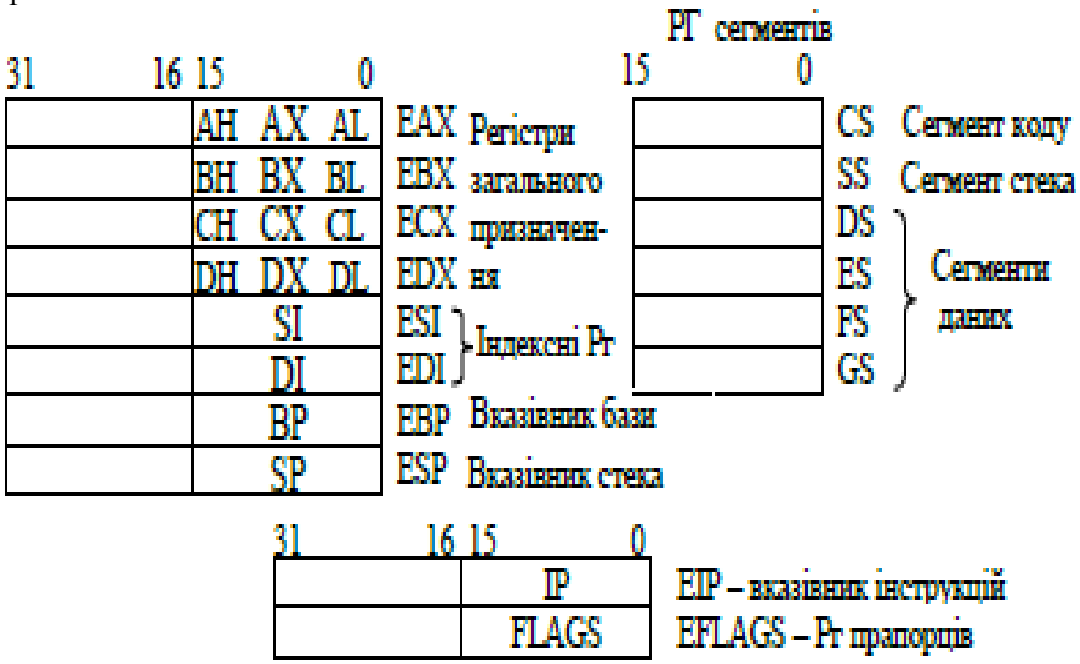


Рис. 2 - Основные регистры процессора архитектуры IA-32

Эти регистры принадлежат к видимой для приложений части архитектуры x86 и представляют собой расширение набора регистров 16-разрядных микропроцессоров 8086/8088 и 80286.

В 32-разрядных микропроцессорах регистры общего назначения EAX, EBX, ECX, EDX состоят из двух 16-битных половинок, младшая из которых тоже делится на две половины. К

последним можно независимо обращаться по символическим именам AH, BH, CH, DH (старшие байты - High) и AL, BL, CL, DL (младшие байты - Low).

МП платформы x86 имеют аккумуляторную структуру. Это означает, что все операции выполняются через аккумулятор, то есть через регистр EAX.

Регистры-указатели ESP (Stack Pointer - указатель стека), EBP (Base Pointer - базовый регистр или указатель базы) и индексные регистры ESI (Source Index - индекс источника), EDI (Destination Index - индекс назначения) допускают 32-разрядное обращения.

Адрес текущей инструкции хранится в 32-битном указателе команд EIP (Instruction Pointer).

Регистры в командах могут адресоваться явно. В ряде команд существует неявное использование регистров:

- EAX - умножение, деление, ввод и вывод **двойного** слова;
- AX - умножение, деление, ввод и вывод слова;
- AL - умножение, деление, ввод и вывод байта; десятичная арифмеметика, трансляция (XLAT)
- AH - умножение и деление байта;
- EBX - трансляция;
- ECX - счетчик циклов и указатель длины строчных операций;
- CL - смещение с указанием переменной;
- EDX - умножение и деление слова, ввод и вывод с косвенной адресацией;
- SP - операции со стеком;
- ESI, EDI - строчные операции.

В микропроцессорах IA-32 все эти регистры имеют префикс E (Extended - расширенный).

Подробнее обо всех регистрах (таб. 1)и их назначении будет указано в следующих работах.

Таблица 1 - Размер регистров и их назначение

Тип	Біти	Регістри							
General	8	al	cl	dl	bl	ah	ch	dh	bh
	16	ax	cx	dx	bx	sp	bp	si	di
	32	eax	ecx	edx	ebx	esp	ebp	esi	edi
	64	rax	rcx	rdx	rbx	rsp	rbp	rsi	rdi
Segment	16	es	cs	ss	ds	fs	gs		
Control	32	cr0		cr2	cr3	cr4			
Debug	32	dr0	dr1	dr2	dr3			dr6	dr7
FPU	80	st0	st1	st2	st3	st4	st5	st6	st7
MMX	64	mm0	mm1	mm2	mm3	mm4	mm5	mm6	mm7
SSE	128	xmm0 xmm1 xmm2 xmm3 xmm4 xmm5 xmm6 xmm7							
AVX	256	Для 32-разрядной ОС:YMM0 — YMM07 Для 64-разрядной ОС:YMM0 — YMM15							

7. Способы адресации 32-разрядного МП Intel

При выполнении любой программы процессор обращается к памяти, в которой хранятся команды и данные. Способ или метод определения в команде адреса операнда или адреса перехода называется режимом адресации или просто адресацией.

В наиболее простом режиме адресации, называемом прямой адресацией, адрес находится в самой команде. Однако использование этого режима, кстати, предусмотренного в большинстве современных микропроцессоров, приводит к чрезмерной длине команд, особенно в условиях,

когда постоянно увеличивается емкость памяти. Поэтому сегодня в микропроцессорах применяется много других режимов адресации, направленных на достижение следующих целей:

1. Определение адреса памяти наименьшим количеством битов, что сокращает длину команд;

2. Вычисление адреса по текущей команде (так называемая относительная адресация), что обеспечивает загрузку программ без модификаций в любой участок памяти;

3. Осуществление доступа к ячейкам памяти, адреса которых вычисляются при выполнении программы, что упрощает доступ к регулярным структурам данных;

4. Обращение к адресам в форме, удобной для таких структур данных, как массивы и стеки.

Режимы адресации значительно расширяют гибкость и удобство пользования системой команд.

Система команд 32-разрядных микропроцессоров предусматривает 11 режимов адресации. При этом только в двух случаях операнды не связаны с памятью. Это операнд, который берется из каждого 8-, 16- или 32-битного регистра микропроцессора, и непосредственный операнд (8, 16 или 32 бит), который содержится в самой команде.

Непосредственная адресация. Регистр сегмента не используется. Например,

MOV EAX, 12345678h; загрузка в EAX const = 12345678_{16}.

Регистровая адресация. Например, **MOV EAX, ECX.**

Другие девять режимов так или иначе обращаются к памяти.

При обращении к памяти эффективный адрес вычисляется с использованием следующих компонентов:

• **смещения** (Displacement или Disp) - 8-, 16- или 32-битного числа, включенного в команду;

• **базы** (Base) - содержания базового регистра. Обычно используется для указания на начало некоторого массива;

• **индекса** (Index) - содержимого индексного регистра. Обычно используется для выбора элемента массива;

• **масштаба** (Scale) - множителя (1, 2, 4 или 8), указанного в коде инструкции. Этот элемент, который используется для указания размера элемента массива, доступен только при 32-битной адресации.

Эффективный адрес вычисляется по формуле

EA = Base + Index*Scale + Disp.

Отдельные составляющие в этой формуле могут отсутствовать.

Различия 16- и 32-битных режимов адресации приведены в табл. 2.

Таблиця 2 - Различия 16- и 32-битных режимов адресации

Компонент	16-бітова адресація	32-бітова адресація
Базовий регістр	BX або BP	32-бітовий регістр загального призначення
Індексний регістр	SI або DI	32- бітовий регістр загального призначення, крім ESP
Масштаб	Немає (завжди 1)	1,2, 4 або 8 (число)
Зміщення	0, 8 або 16 біт	0, 8 або 32 біт

Процессор может работать с 32- или 16-битной адресацией. 16-битная адресация работает

так же, как и в микропроцессорах 8086 и 80286, причем как компоненты адреса используются младшие 16 бит соответствующих регистров.

8. Отладчик OllyDbg

OllyDbg — бесплатный 32-х битный отладчик для операционных систем **Windows**, предназначенный для анализа и модификации откомпилированных исполняемых файлов и библиотек, работающих в режиме пользователя (рис. 3).

OllyDbg выгодно отличается от классических отладчиков (таких, как SoftICE) простым интерфейсом, подсветкой специфических структур кода, простотой в установке и запуске. Для того, чтобы разобраться в принципе работы OllyDbg, достаточно базовых знаний в области языка ассемблера.

Будем использовать версию программы 1.10.

OllyDbg самый популярный отладчик в последнее время. Он удобен тем, что дизассемблирует программу и позволяет вести ее анализ, когда исходники недоступны. Отладчик отображает значения регистров, показывает содержимое памяти, распознает процедуры, API-функции, переходы, строковые и цифровые константы, имеется возможность переименовывать переменные и делать комментарии в дизассемблированном коде. Сделанные Вами патчи кода отладчик умеет записывать прямо в исполняемый файл.

Возможности OllyDbg:

- Поддерживаемые процессоры: вся серия 80x86, Pentium и совместимые; расширения MMX, 3DNow! и SSE до версии SSE4 включительно (SSE5 пока не поддерживается).
- Поддерживаемые форматы данных: hex-код, ASCII, юникод, 16- и 32-битные целые числа со знаком и без знака, 32-, 64- и 80-битные числа с плавающей запятой (float).
- Способы отображения дизассемблированного кода: MASM, IDEAL, HDA.
- Мощный анализатор кода, распознающий процедуры, циклы, ветвления, таблицы, константы и текстовые строки.
- Развёрнутая система поиска: поиск всех возможных констант, команд, последовательностей команд, текстовых строк и ссылок в коде на данный адрес.
- Распознание и расшифровка более двух тысяч типичных функций WinAPI и языка C.
- Распознание и расшифровка PE-заголовка.
- Эвристический анализ стэка, распознание адресов возврата в родительскую процедуру, SEH-блоки.
- Простые, условные и протоколирующие точки останова (брейкпойнт).
- Пошаговая отладка с протоколированием хода выполнения (run trace).
- Индивидуальный файл конфигурации (UDD) для каждого отлаживаемого приложения.
- Возможность расширения функционала всемозможными плагинами, написанными сторонними разработчиками.

На рис. 3 выделено 9 областей, представляющих из себя инструменты информирования и интерактивности, которые предоставляет OllyDbg.

- 1 (красным) - виртуальные адреса;
- 2 (зеленым) - машинный код;
- 3 (голубым) - дизассемблированный листинг (команды ассемблера);
- 4 (оранжевым) - комментарии отладчика;
- 5 (фиолетовым) - регистры общего назначения;
- 6 (желтым) - EIP регистр (показывает виртуальный адрес следующей выполняемой команды);
- 7 (коричневым) - флаги и регистр флагов;
- 8 (синим) - дамп памяти;
- 9 (малиновым) - стэк.

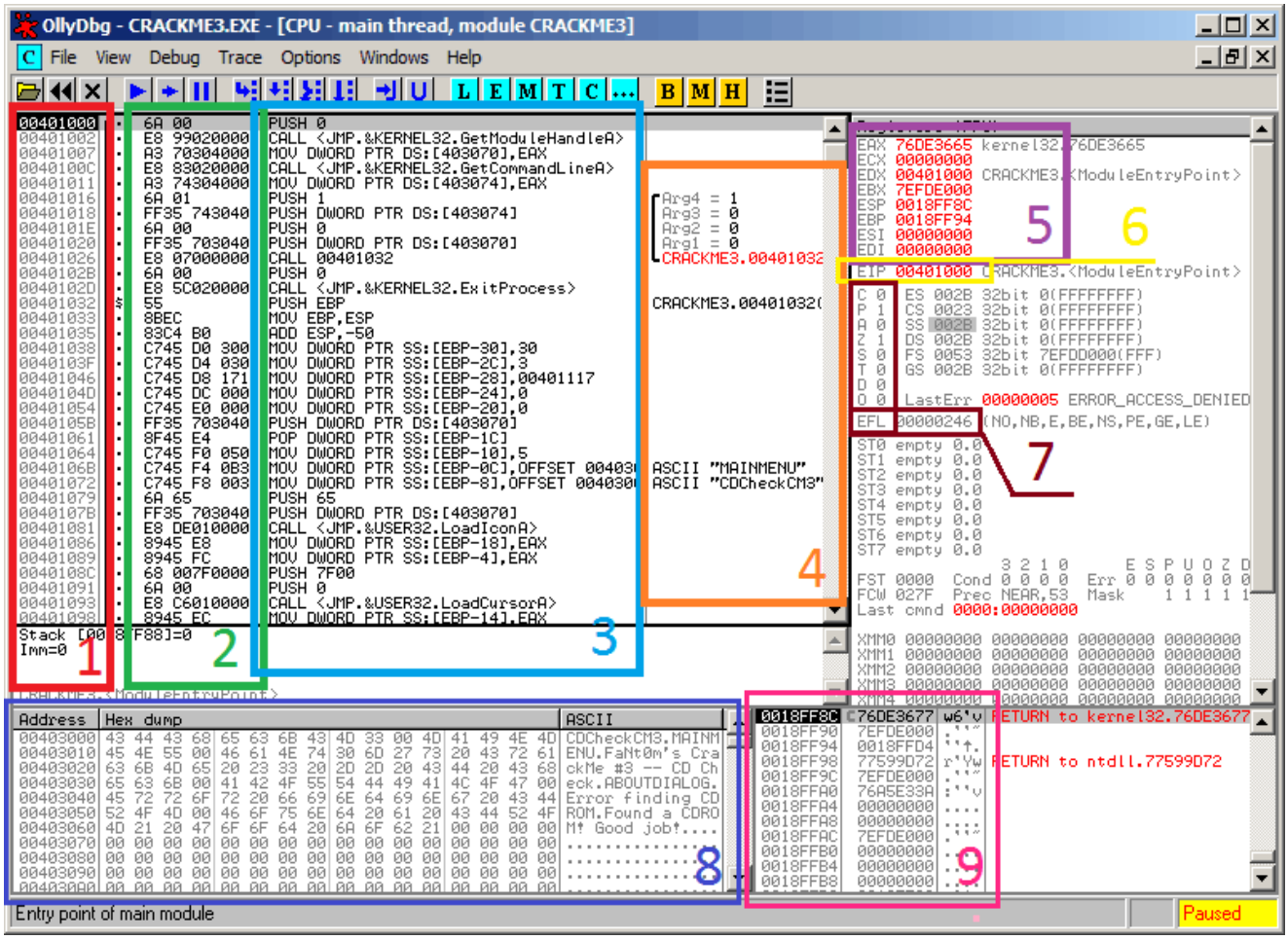


Рис. 3 – Окно отладчика OllyDbg

Пример 1. Выполнить ассемблирование и компоновку программы, выводящей на консоль результат вычитания двух целых чисел. Выполнить отладку программы с помощью отладчика OllyDbg.

Программа имеет следующий вид:

```
.686                                    ; директива определения типа микропроцессора
.model  flat, stdcall                   ; задание линейной модели памяти и соглашения
                                        ;Windows
option casemap:none                     ; отличие строчных и прописных букв
include \masm32\include\windows.inc     ; файлы структур, констант …
include \masm32\include\kernel32.inc    ; системные функции применений…
include \masm32\include\\user32.inc     ; файлы интерфейса …
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
a1 dd 12
c1 dd 97
titl db "Вывод через функцию MessageBox",0 ; название упрощенного окна
st1 dw ?,0                              ; буфер выведения сообщения dw
                                        ;для небольших чисел
ifmt  db "Вывод чисел с памяти через MessageBox:", 0dh,0ah,
"a = 12", 0dh,0ah, 'c = 97', 0dh,0ah,
"a - c = -%d", 0dh,0ah,0ah,             ; задание превращения символа
.code
_start:
mov eax,a1
```

```
sub eax,c1
not eax
inc eax
invoke  wsprintf, ADDR st1, ADDR ifmt, eax
invoke MessageBox, 0, addr st1,addr titl,MB_ICONINFORMATION
invoke ExitProcess, 0 ;
END   _start                                    ;окончание программы с именем _start
```

Открыть папку **masm32** на диске **c:** и запустить на выполнение программу qeditor.exe из корневого каталога **masm32**. В открывшемся окне редактора с названием **c:\ masm32** набрать текст программы.

После этого выполнить команды **File→Save as** (Сохранить как) и в открывшемся окне сохранить исследуемый файл под именем **6-1** (расширение, возможно, ввести не удастся) (рис. 4).

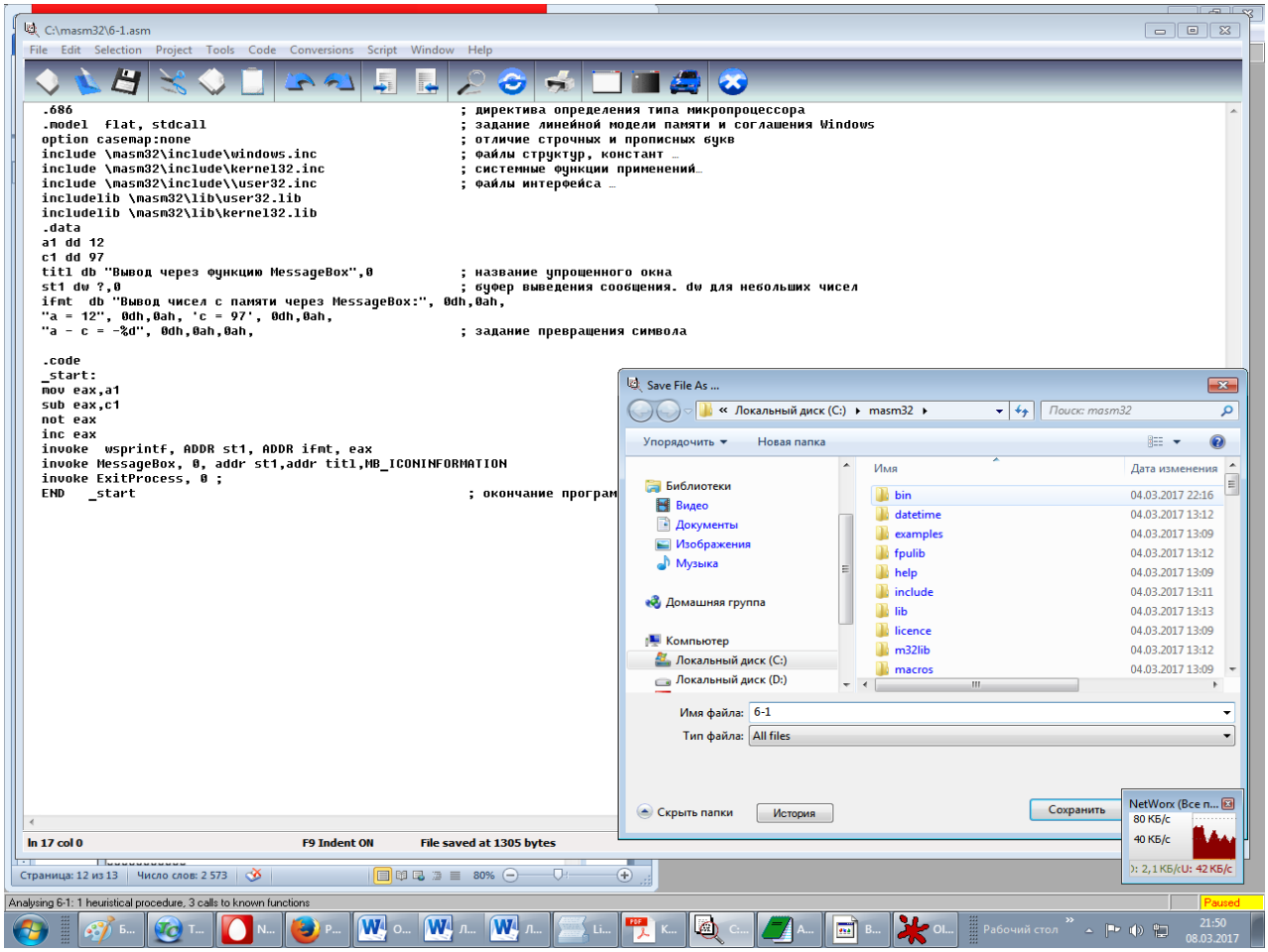


Рис. 4 – Сохранение файла **5-1**

После этого необходимо переименовать в файловом менеджере Total Commander в папке **masm32** файл **6-1** в файл **6-1.asm**.

Вернуться в окно программы **masm32** и для ассемблирования и компоновки исследуемой программы выполнить команды **Project→Assemble & Link**.

Кроме этого, необходимо проставить комментарии к тем строкам, где их не было.

В открывшемся окне операционной системы **Windows** (рис. 5) будет краткий отчет об успешном или нет ассемблировании и линковании.

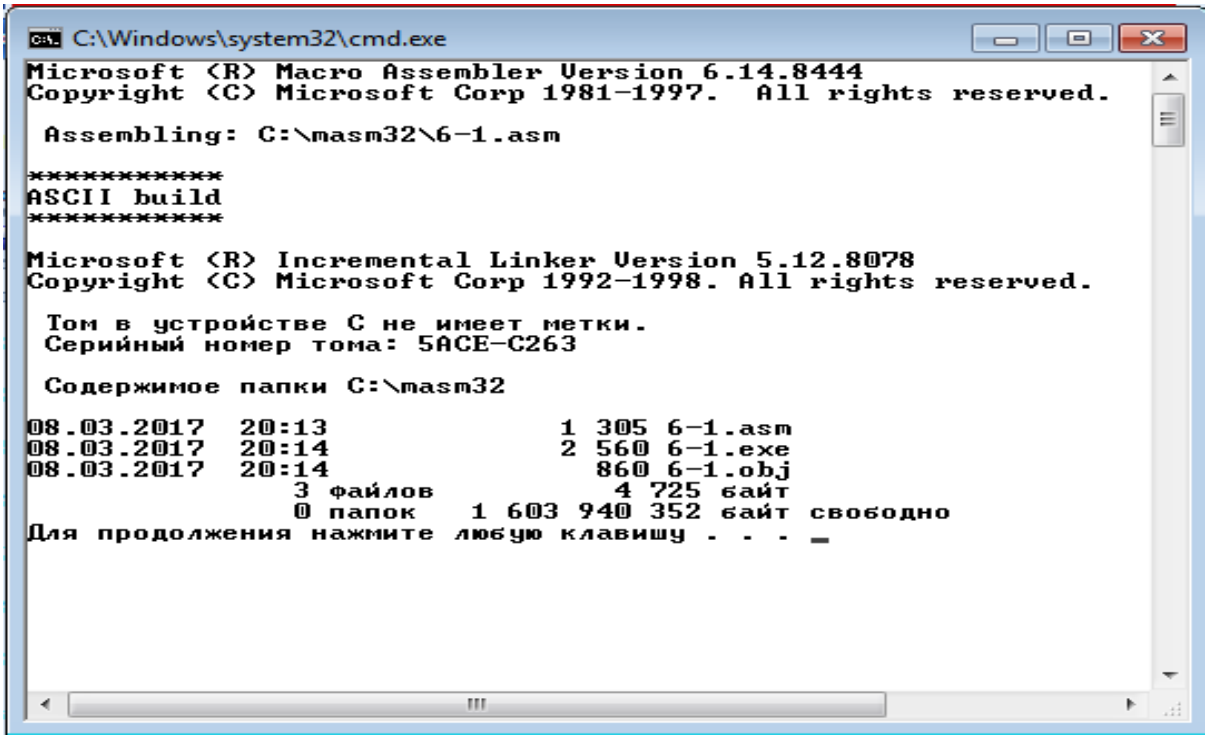


Рис. 5 – Окно операционной системы Windows с сообщением
об успешной компоновке файла **5-1.asm**

При отсутствии ошибок будут сформированы и помещены в папку **masm32** файлы **6-1.obj** и **6-1.exe.** В результате выполнения файла **6-1.exe** получим результат исследуемой программы – окно с вычисленной разностью (рис. 6):

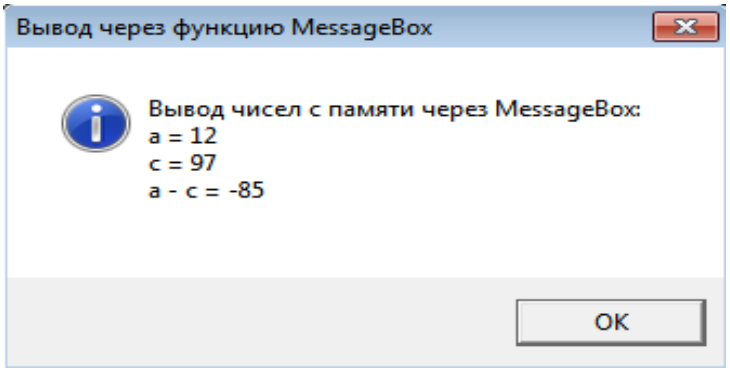


Рис. 6 – Окно с консольным выводом результата выполнения файла **5-1.exe**

Для более глубокого анализа данной программы в случае ее успешной компоновки или же для отладки в случае выявления ошибок используем отладчик **OllyDbg**, для запуска которого необходимо в папке **masm32** открыть папку **OLLYDBG** и выполнить файл **OLLYDBG**.

В открывшемся окне **OllyDbg 6-1.exe** (рис. 7) самостоятельно проанализируйте содержание регистров микропроцессора. Результаты анализа письменно изложите в отчете по работе.

Лекция 7

Создание программ на языке ассемблера в среде Masm32 с отладчиком Ollydbg

Цель лекции: изучить начальные сведения о программировании на языке ассемблера в среде Masm32**.**

Основные вопросы лекции:

1. Этапы создания программ на языке Ассемблера.
2. Подготовка текста программы в Masm32.
3. Получение объектного кода в Masm32.
4. Получение исполняемого файла программы в Masm32.
5. Отладка программы.
6. Регистры 32-разрядного МП Intel.
7. Способы адресации 32-разрядного МП Intel.
8. Отладчик OllyDbg.

```
.386                                ; директива определения типа микропроцессора
.model flat                         ; задание линейной модели памяти
.data                               ; директива определения данных
a1B byte 11h                        ; запись в 8-разрядную ячейку памяти
                                    ; с именем a1B числа 11h = 8A1
a2B db 22h                          ; запись в 8-разрядную ячейку с именем a2B числа 22h
b1W word 3A3Bh                      ; запись в 16-разрядную ячейку памяти числа 3A3Bh
b2W dw 4A4Bh                        ; запись в 16-разрядную ячейку памяти числа 4A4Bh
c1D dword 5A5B5C5Dh                     ; запись в 32-разрядную ячейку памяти
c2D dd 6A6B6C6Dh                        ; запись в 32-разрядную ячейку памяти
d1Q dq 7A7B7C7D7E7F7A7Bh                ; 8 байтов
d2Q qword 8A8B8C8D8E8F8A8Bh             ; 8 байтов
e1t tbyte 9A9B9C9D9E9F9A9B9C9Dh         ; 10 байтов
.code                                   ; директива начала кода программы
_start:                                 ; метка начала программы с именем start
ret                                     ; возвращение управления ОС
end _start                              ; окончание программы с именем _start
```

1. Этапы создания программ на языке Ассемблера

Ассемблер, как язык программирования, является набором мнемонических обозначений машинных кодов, которые понимают вычислительные устройства. Пока последние будут существовать, язык ассемблера будет актуален. Несмотря на бурное развитие математического обеспечения и алгоритмических языков высокого уровня, интерес к языку ассемблера постоянно растет.

Язык ассемблера помогает раскрыть секреты аппаратного и программного обеспечения. Большинство программистов работают с языками высокого уровня, где отдельный оператор превращается в множество микропроцессорных команд. Каждая команда языка ассемблера имеет взаимное однозначное соответствие с машинными командами, что дает основание считать его языком низкого уровня и отличает от таких языков высокого уровня, как Pascal, BASIC и C ++, для которых характерна трансляция одного оператора в множество машинных команд.

Чаще всего язык ассемблера используется для непосредственного управления операционной системой и для прямого доступа к аппаратуре. На ассемблере пишут оптимальные драйверы и небольшие утилиты, прошивки BIOS, загрузчики и ядра ОС, "движки" игрушек, вирусы, компиляторы и многие иное. Ассемблер служит инструментом для связи с нетрадиционными внешними устройствами. Также он необходим при оптимизации критических блоков в приложениях с целью повышения их быстродействия.

Известно, что понять работу компьютера можно, только освоив ассемблер и управление компьютером на самом нижнем уровне. Знание ассемблера и умение работать с компьютером на самом нижнем уровне обязательны для системного программиста.

Язык ассемблера существует для каждого типа процессоров или семейства микропроцессоров.

Ассемблер может создавать листинг программы с номерами строк, адресами переменных, операторами и таблицей перекрестных ссылок символов и переменных. Наряду с ассемблером используется программа, называемая компоновщиком (**linker**), которая объединяет отдельные файлы, созданные ассемблером, в единую исполняемую программу.

При изучении языка ассемблера и разработке реальных программ на этом языке под операционную систему (ОС) Windows процесс распределения ячеек памяти для сохранения кодов команд и чисел выполняется автоматизировано.

Этапы создания программы на языке ассемблера следующие:

- подготовка (или внесение изменений) исходного текста программы;
- ассемблирование программы (получение объектного кода)
- компоновка программы (получение исполняемого файла программы);
- отладка программы (исправление ошибок).

Обычно эти этапы циклически повторяются, поскольку при обнаружении ошибок на всех этапах приходится возвращаться к первому этапу и вносить изменения в текст программы для исправления ошибок.

Ассемблеры бывают двух типов: однофазные и двухфазные.

Однофазные ассемблеры могут обрабатывать только такие программы, в которых символы появляются в поле названия до того, как на них дается ссылка в поле операндов.

Двухфазные ассемблеры транслируют программы в два этапа:

- в **первой фазе** трансляции ассемблер последовательно считает каждое предложение начальной программы, частично ее транслирует и строит полную таблицу символов;

во **второй фазе** ассемблер заканчивает трансляцию программы, используя таблицу символов первой фазы как входную информацию.

Для обоих типов ассемблеров символ, записанный в поле операндов, обязательно должен быть определен в поле названия, иначе будет сообщение об ошибке.

2. Подготовка текста программы в Masm32

Текст программы на языке ассемблера записывается в один или несколько текстовых файлов. Имена файлов и их расширения могут быть любые, но принято использовать расширение ***.asm**. Эти файлы являются текстовыми, их можно подготовить с помощью любого текстового редактора, например **Блокнота** или **NotePad**, и хранить в виде обычных файлов в формате ASCII.

Использование **Word** нежелательно, потому что такой текст содержит большое количество служебных символов, которые являются невидимыми и приводят к ошибкам трансляции.

Возможно при подготовке текста использовать саму среду **Masm32.**

Структура программы под Win32 следующая:

```
TITLE <имя программы
.386                    ;директива определения типа микропроцессора
.model flat             ;задание линейной модели памяти
.data                   ;директива определения данных,
                        ;которые определены
.data?                  ;неинициализированные данные
.code                   ;директива начала кода программы
label:                  ;метка начала программы <code>
ret                     ;возвращение управления ОС
end label               ;окончание программы
```

3. Получение объектного кода в Masm32

Подготовленный текст программы с расширением ***.asm** является входными данными для программ, которые называются ассемблерами (например, программы **Masm**, **Tasm**, **Wasm**, **Fasm**). Задача ассемблера - превратить текст программы в форму двоичных команд, которые может выполнить МП. После ассемблирования получают файлы объектных модулей, имеющие расширение ***.obj**. Процесс ассемблирования называют трансляцией.

Для трансляции и линкования простых программ удобно использовать среду **masm32**. Ассемблер **Masm32** (**www.masm32.com**) необходимо устанавливать на системный диск **c:/**. Если поставить ассемблер **Masm32** не на системный диск, а на другой логический диск, то в таком случае необходимо конкретно указывать пути подключения библиотек расположения API-функций.

Программа **ml.exe** из Masm32 выполняет трансляцию исходного текста программы в промежуточный объектный файл. Программа **link.exe** выполняет компоновку объектного (объектных) файла (файлов) и библиотек в единую исполняемую программу. Эти exe-файлы находятся в папке **[bin].**

Для трансляции и линкования программы необходимо запустить программу **qeditor** (файл qeditor.exe) из **masm32**, вставить в окно или набрать свою программу, сохранить ее с расширением **asm** и выбрать команды **Project→Assemble & Link**. Если ошибок не будет, то будет создан **exe**-файл.

4. Получение исполняемого файла программы в Masm32

После успешной трансляции исходного текста ассемблерной программы результат в виде объектного файла передается компоновщику link.exe, выполняющему объединение объектных модулей в один файл. Сегменты, определенные в программе, группируются в соответствии с инструкциями, содержащимися в объектном файле. Вся информация о размещении сегментов записывается в заголовок исполняемого файла.

Результатом компоновки является исполняемый файл, имеющий расширение ***.exe.**

В процессе трансляции исходного текста программы выполняются следующие действия:

1. Анализируются директивы условного ассемблирования, и в случае истинности указанных в них условий выполняются те или иные шаги.
2. Развертываются макросы.
3. Вычисляются константные выражения, при этом они замещаются вычисленными значениями.
4. Декодируются команды и операнды, которые не находятся в памяти Например, декодируется команда **mov EAX, 2008**, поскольку она не имеет операндов, расположенных в памяти.
5. Сохраняются смещения переменных в памяти относительно смещения в сегментах, в которых эти переменные расположены.

6. Сегменты и их атрибуты размещаются в объектном файле.

7. В объектный файл помещаются перемещаемые адреса (Relocatable addresses).

8. При необходимости создается файл листинга.

9. Непосредственно программе link.exe передаются некоторые директивы (например, INCLUDELIB).

Схема ассемблирования, компоновки и выполнения программы приведен на рис. 1.

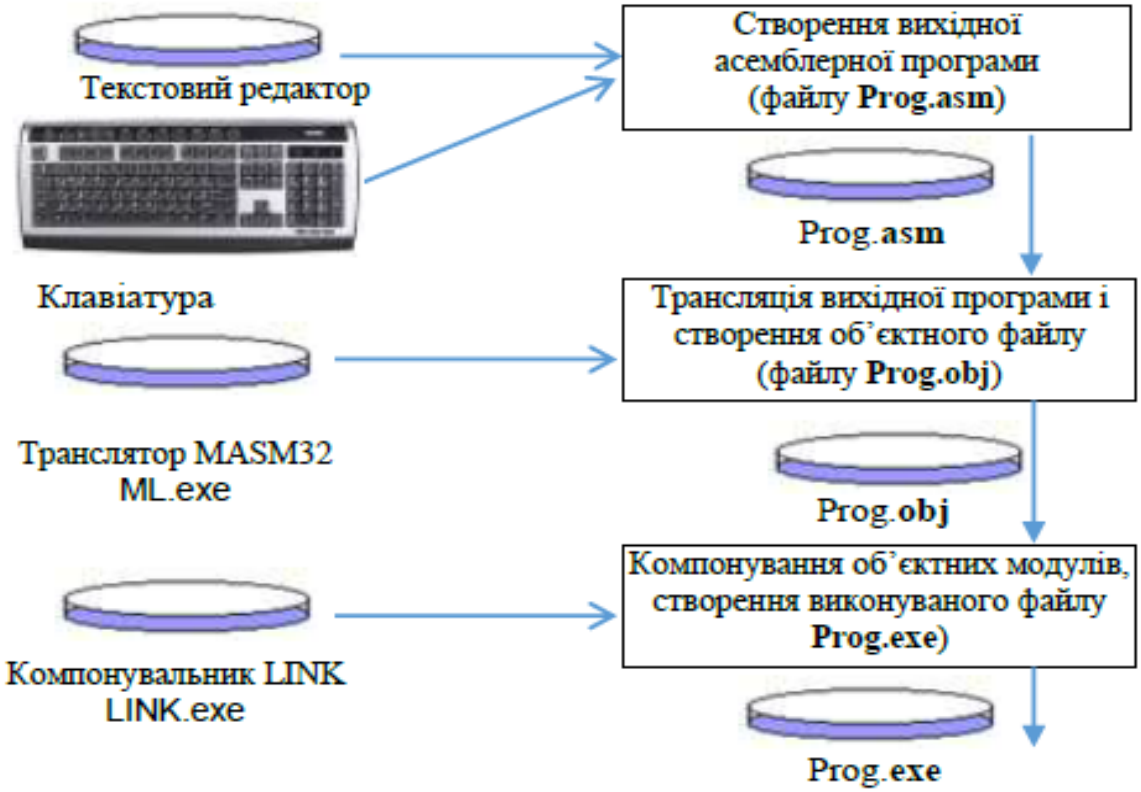


Рис. 1 - Схема ассемблирования, компоновки и выполнения программы

Структура программы (не только на ассемблере) определяется несколькими факторами:

- архитектурой процессора;
- особенностями той операционной системы, под управлением которой эта программа будет выполняться;
- правилами работы выбранного компилятора - разные компиляторы ставят различные требования к исходному тексту программы.

MASM и TASM поддерживают упрощенную сегментацию. Суть такой сегментации в том, что директивы .CODE и .DATA могут появляться в тексте программы несколько раз. Транслятор затем собирает код и данные вместе. Основной целью такого подхода является возможность приблизить в тексте программы данные к тем строкам, где они используются.

5. Отладка программы

Любая программа требует отладки (исправления ошибок). Современные отладочные программы для ассемблера платформы x86, например программы Dbg_x86 (http://www.microsoft.com/whdc/devtools/ debugging), SoftICE (уже не поддерживается, но еще используется), Emu8086 v.4.04, OllyDbg 1.10 (http://cracklab.ru/), OllyDbg 2 (http://www.ollydbg.de/), Syser (www.syser.com) и др. позволяют в процессе выполнения программы контролировать значения регистров общего назначения или переменных и изменять их.

С помощью отладчика можно, например, просматривать содержимое различных участков памяти, выполнять программу шаг за шагом, менять программный код, сохранять изменения в exe-файле и др.

Для отладки профессиональных ассемблерных программ, которые применяют последние версии команд параллельной обработки вещественных чисел или новые API-функции, которые постоянно появляются, используют также среду Visual Studio.

При написании программ необходимо использовать справочник **MSDN** (**http://msdn.microsoft.com/ru-ru/library**).

6. Регистры 32-разрядного МП Intel

Основные регистры процессора архитектуры IA-32, с которыми работают приложения, показаны на рис. 2.

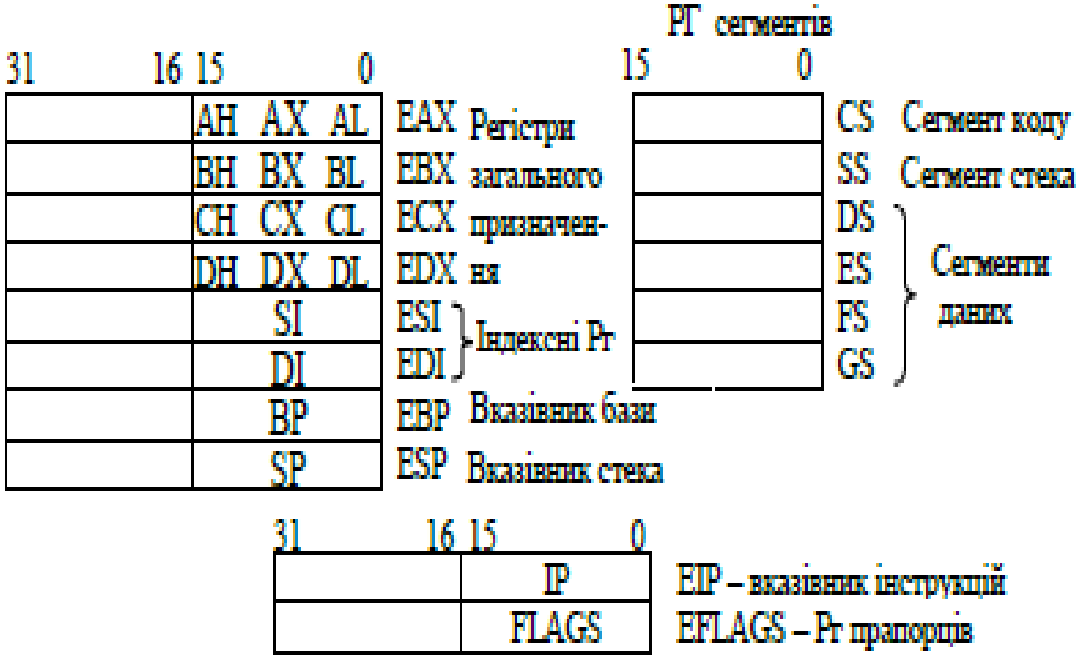


Рис. 2 - Основные регистры процессора архитектуры IA-32

Эти регистры принадлежат к видимой для приложений части архитектуры x86 и представляют собой расширение набора регистров 16-разрядных микропроцессоров 8086/8088 и 80286.

В 32-разрядных микропроцессорах регистры общего назначения EAX, EBX, ECX, EDX состоят из двух 16-битных половинок, младшая из которых тоже делится на две половины. К последним можно независимо обращаться по символическим именам AH, BH, CH, DH (старшие байты - High) и AL, BL, CL, DL (младшие байты - Low).

МП платформы x86 имеют аккумуляторную структуру. Это означает, что все операции выполняются через аккумулятор, то есть через регистр EAX.

Регистры-указатели ESP (Stack Pointer - указатель стека), EBP (Base Pointer - базовый регистр или указатель базы) и индексные регистры ESI (Source Index - индекс источника), EDI (Destination Index - индекс назначения) допускают 32-разрядное обращения.

Адрес текущей инструкции хранится в 32-битном указателе команд EIP (Instruction Pointer).

Регистры в командах могут адресоваться явно. В ряде команд существует неявное использование регистров:

- EAX - умножение, деление, ввод и вывод **двойного** слова;
- AX - умножение, деление, ввод и вывод слова;
- AL - умножение, деление, ввод и вывод байта; десятичная арифмеметика, трансляция (XLAT)
- AH - умножение и деление байта;
- EBX - трансляция;
- ECX - счетчик циклов и указатель длины строчных операций;
- CL - смещение с указанием переменной;

- EDX - умножение и деление слова, ввод и вывод с косвенной адресацией;
- SP - операции со стеком;
- ESI, EDI - строчные операции.

В микропроцессорах IA-32 все эти регистры имеют префикс E (Extended - расширенный). Подробнее обо всех регистрах (таб. 1)и их назначении будет указано в следующих работах.

Таблица 1 - Размер регистров и их назначение

Тип	Біти	Регістри
General	8	al cl dl bl ah ch dh bh
	16	ax cx dx bx sp bp si di
	32	eax ecx edx ebx esp ebp esi edi
	64	rax rcx rdx rbx rsp rbp rsi rdi
Segment	16	es cs ss ds fs gs
Control	32	cr0 cr2 cr3 cr4
Debug	32	dr0 dr1 dr2 dr3 dr6 dr7
FPU	80	st0 st1 st2 st3 st4 st5 st6 st7
MMX	64	mm0 mm1 mm2 mm3 mm4 mm5 mm6 mm7
SSE	128	xmm0 xmm1 xmm2 xmm3 xmm4 xmm5 xmm6 xmm7
AVX	256	Для 32-разрядной ОС:YMM0 — YMM07 Для 64-разрядной ОС:YMM0 — YMM15

7. Способы адресации 32-разрядного МП Intel

При выполнении любой программы процессор обращается к памяти, в которой хранятся команды и данные. Способ или метод определения в команде адреса операнда или адреса перехода называется режимом адресации или просто адресацией.

В наиболее простом режиме адресации, называемом прямой адресацией, адрес находится в самой команде. Однако использование этого режима, кстати, предусмотренного в большинстве современных микропроцессоров, приводит к чрезмерной длине команд, особенно в условиях, когда постоянно увеличивается емкость памяти. Поэтому сегодня в микропроцессорах применяется много других режимов адресации, направленных на достижение следующих целей:

1. Определение адреса памяти наименьшим количеством битов, что сокращает длину команд;

2. Вычисление адреса по текущей команде (так называемая относительная адресация), что обеспечивает загрузку программ без модификаций в любой участок памяти;

3. Осуществление доступа к ячейкам памяти, адреса которых вычисляются при выполнении программы, что упрощает доступ к регулярным структурам данных;

4. Обращение к адресам в форме, удобной для таких структур данных, как массивы и стеки.

Режимы адресации значительно расширяют гибкость и удобство пользования системой команд.

Система команд 32-разрядных микропроцессоров предусматривает 11 режимов адресации. При этом только в двух случаях операнды не связаны с памятью. Это операнд, который берется из каждого 8-, 16- или 32-битного регистра микропроцессора, и непосредственный операнд (8, 16 или 32 бит), который содержится в самой команде.

Непосредственная адресация. Регистр сегмента не используется. Например,

MOV EAX, 12345678h; загрузка в EAX const = 12345678_{16}.

Регистровая адресация. Например, **MOV EAX, ECX.**

Другие девять режимов так или иначе обращаются к памяти.

При обращении к памяти эффективный адрес вычисляется с использованием следующих компонентов:

• **смещения** (Displacement или Disp) - 8-, 16- или 32-битного числа, включенного в команду;

• **базы** (Base) - содержания базового регистра. Обычно используется для указания на начало некоторого массива;

• **индекса** (Index) - содержимого индексного регистра. Обычно используется для выбора элемента массива;

• **масштаба** (Scale) - множителя (1, 2, 4 или 8), указанного в коде инструкции. Этот элемент, который используется для указания размера элемента массива, доступен только при 32-битной адресации.

Эффективный адрес вычисляется по формуле

EA = Base + Index*Scale + Disp.

Отдельные составляющие в этой формуле могут отсутствовать.

Различия 16- и 32-битных режимов адресации приведены в табл. 2.

Таблиця 2 - Различия 16- и 32-битных режимов адресации

Компонент	16-бітова адресація	32-бітова адресація
Базовий регістр	BX або BP	32-бітовий регістр загального призначення
Індексний регістр	SI або DI	32- бітовий регістр загального призначення, крім ESP
Масштаб	Немає (завжди 1)	1,2, 4 або 8 (число)
Зміщення	0, 8 або 16 біт	0, 8 або 32 біт

Процессор может работать с 32- или 16-битной адресацией. 16-битная адресация работает так же, как и в микропроцессорах 8086 и 80286, причем как компоненты адреса используются младшие 16 бит соответствующих регистров.

8. Отладчик OllyDbg

OllyDbg — бесплатный 32-х битный отладчик для операционных систем **Windows**, предназначенный для анализа и модификации откомпилированных исполняемых файлов и библиотек, работающих в режиме пользователя (рис. 3).

OllyDbg выгодно отличается от классических отладчиков (таких, как SoftICE) простым интерфейсом, подсветкой специфических структур кода, простотой в установке и запуске. Для того, чтобы разобраться в принципе работы OllyDbg, достаточно базовых знаний в области языка ассемблера.

Будем использовать версию программы 1.10.

OllyDbg самый популярный отладчик в последнее время. Он удобен тем, что дизассемблирует программу и позволяет вести ее анализ, когда исходники недоступны. Отладчик отображает значения регистров, показывает содержимое памяти, распознает процедуры, API-функции, переходы, строковые и цифровые константы, имеется возможность переименовывать переменные и делать комментарии в дизассемблированном коде. Сделанные Вами патчи кода отладчик умеет записывать прямо в исполняемый файл.

Возможности OllyDbg:

- Поддерживаемые процессоры: вся серия 80x86, Pentium и совместимые; расширения MMX, 3DNow! и SSE до версии SSE4 включительно (SSE5 пока не поддерживается).
- Поддерживаемые форматы данных: hex-код, ASCII, юникод, 16- и 32-битные целые числа со знаком и без знака, 32-, 64- и 80-битные числа с плавающей запятой (float).
- Способы отображения дизассемблированного кода: MASM, IDEAL, HDA.
- Мощный анализатор кода, распознающий процедуры, циклы, ветвления, таблицы, константы и текстовые строки.
- Развёрнутая система поиска: поиск всех возможных констант, команд, последовательностей команд, текстовых строк и ссылок в коде на данный адрес.
- Распознание и расшифровка более двух тысяч типичных функций WinAPI и языка C.
- Распознание и расшифровка PE-заголовка.
- Эвристический анализ стэка, распознание адресов возврата в родительскую процедуру, SEH-блоки.
- Простые, условные и протоколирующие точки останова (брейкпойнт).
- Пошаговая отладка с протоколированием хода выполнения (run trace).
- Индивидуальный файл конфигурации (UDD) для каждого отлаживаемого приложения.
- Возможность расширения функционала всемозможными плагинами, написанными сторонними разработчиками.

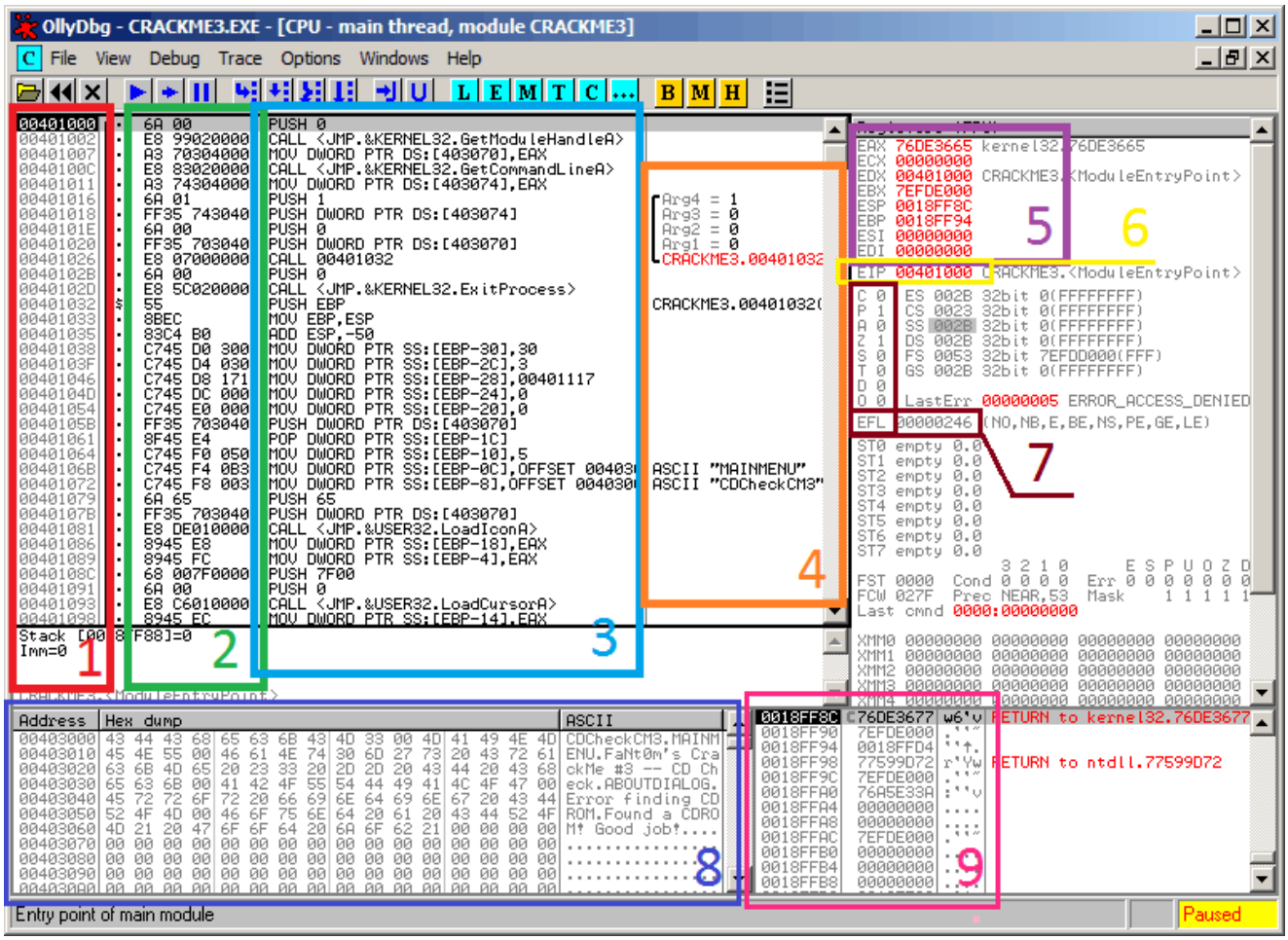


Рис. 3 – Окно отладчика OllyDbg

На рис. 3 выделено 9 областей, представляющих из себя инструменты информирования и интерактивности, которые предоставляет OllyDbg.

- 1 (красным) - виртуальные адреса;
- 2 (зеленым) - машинный код;

- 3 (голубым) - дизассемблированный листинг (команды ассемблера);
- 4 (оранжевым) - комментарии отладчика;
- 5 (фиолетовым) - регистры общего назначения;
- 6 (желтым) - EIP регистр (показывает виртуальный адрес следующей выполняемой команды);
- 7 (коричневым) - флаги и регистр флагов;
- 8 (синим) - дамп памяти;
- 9 (малиновым) - стэк.

Пример 1. Выполнить ассемблирование и компоновку программы, выводящей на консоль результат вычитания двух целых чисел. Выполнить отладку программы с помощью отладчика OllyDbg.

Программа имеет следующий вид:

```
.686                                ; директива определения типа микропроцессора
.model  flat, stdcall               ; задание линейной модели памяти и соглашения
                                    ;Windows
option casemap:none                 ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\kernel32.inc ; системные функции применений…
include \masm32\include\\user32.inc ; файлы интерфейса …
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
a1 dd 12
c1 dd 97
titl db "Вывод через функцию MessageBox",0 ; название упрощенного окна
st1 dw ?,0                          ; буфер выведения сообщения dw
                                    ;для небольших чисел
ifmt  db "Вывод чисел с памяти через MessageBox:", 0dh,0ah,
"a = 12", 0dh,0ah, 'c = 97', 0dh,0ah,
"a - c = -%d", 0dh,0ah,0ah,         ; задание превращения символа
.code
_start:
mov eax,a1
sub eax,c1
not eax
inc eax
invoke  wsprintf, ADDR st1, ADDR ifmt, eax
invoke MessageBox, 0, addr st1,addr titl,MB_ICONINFORMATION
invoke ExitProcess, 0 ;
END   _start                        ;окончание программы с именем _start
```

Открыть папку **masm32** на диске **c:** и запустить на выполнение программу qeditor.exe из корневого каталога **masm32**. В открывшемся окне редактора с названием **c:\ masm32** набрать текст программы.

После этого выполнить команды **File→Save as** (Сохранить как) и в открывшемся окне сохранить исследуемый файл под именем **5-1** (расширение, возможно, ввести не удастся) (рис. 4).

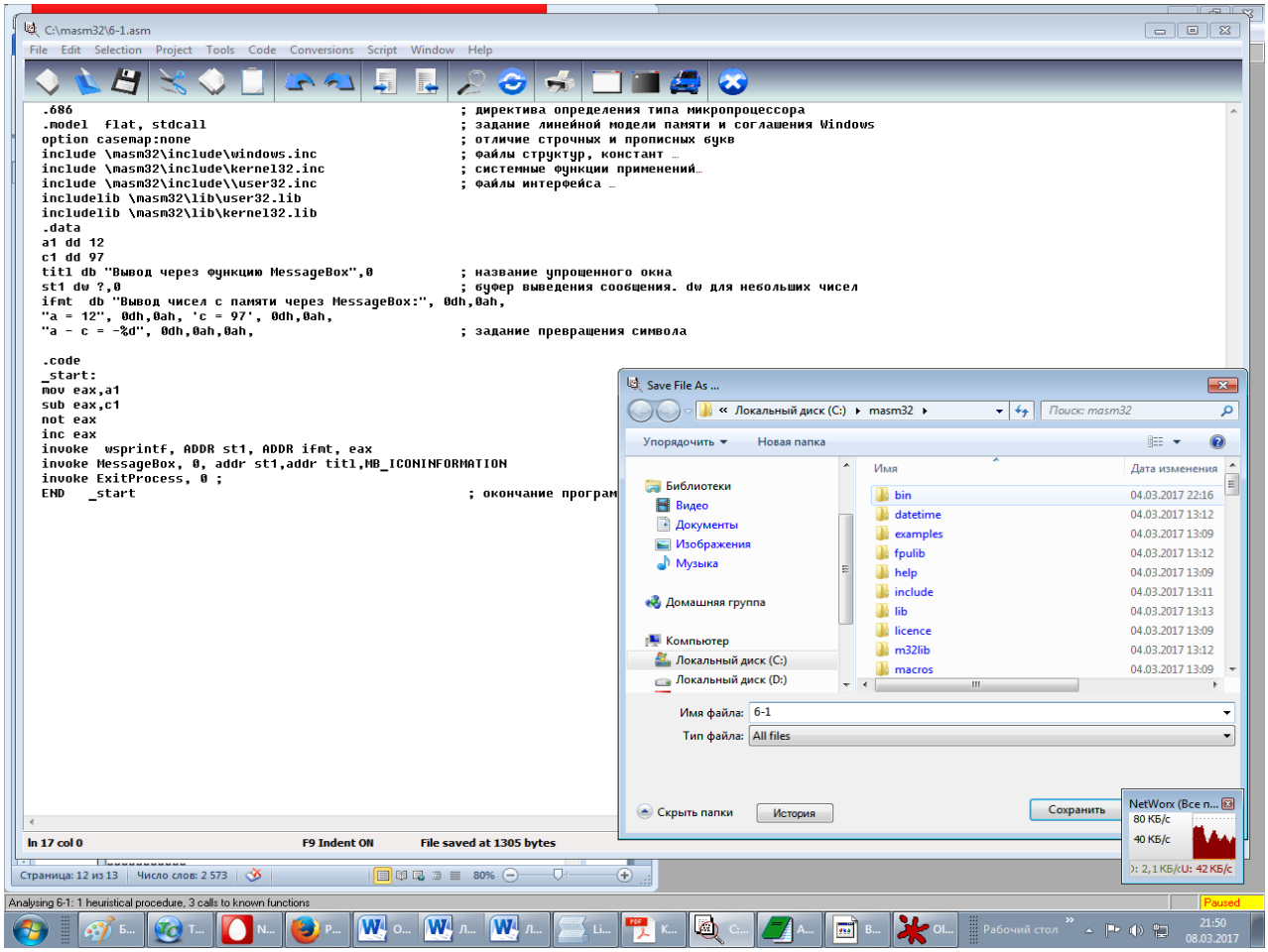


Рис. 4 – Сохранение файла **5-1**

После этого необходимо переименовать в файловом менеджере Total Commander в папке **masm32** файл **5-1** в файл **5-1.exe**.

Вернуться в окно программы **masm32** и для ассемблирования и компоновки исследуемой программы выполнить команды **Project→Assemble & Link**.

Кроме этого, необходимо проставить комментарии к тем строкам, где их не было.

В открывшемся окне операционной системы **Windows** (рис. 5) будет краткий отчет об успешном или нет ассемблировании и линковании.

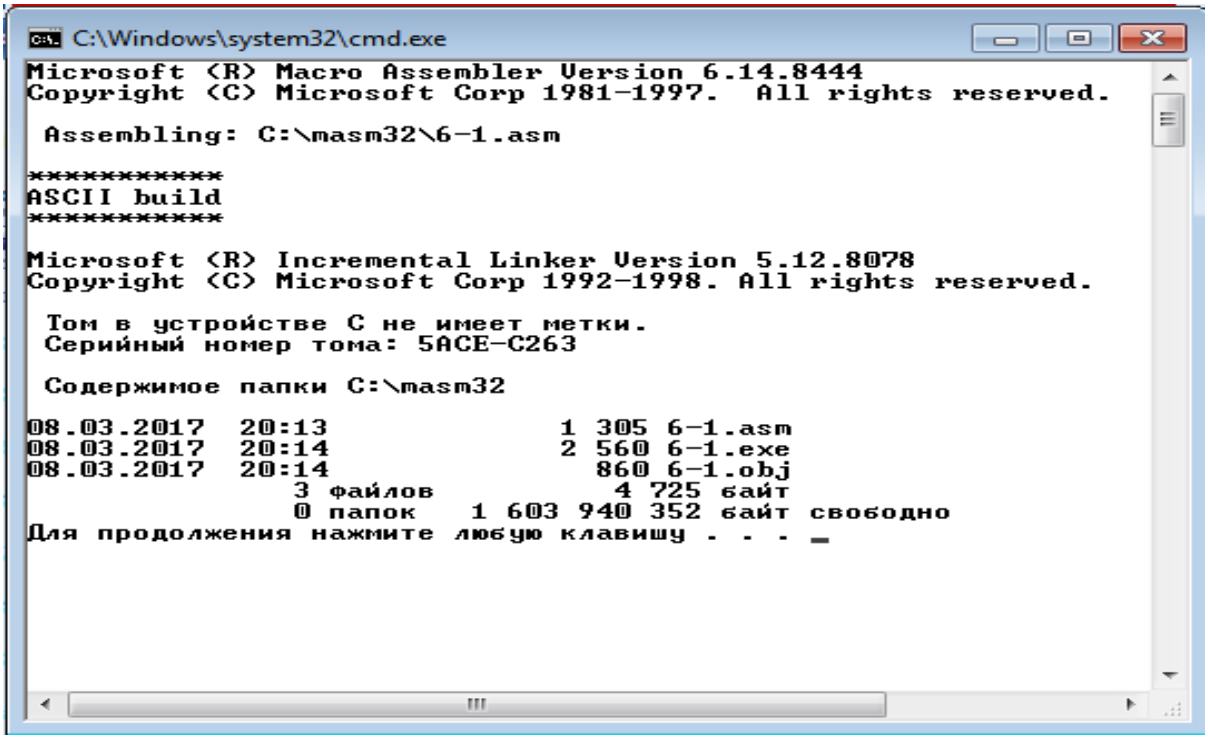


Рис. 5 – Окно операционной системы Windows с сообщением
об успешной компоновке файла **5-1.asm**

При отсутствии ошибок будут сформированы и помещены в папку **masm32** файлы **6-1.obj** и **6-1.exe.** В результате выполнения файла **6-1.exe** получим результат исследуемой программы – окно с вычисленной разностью (рис. 6):

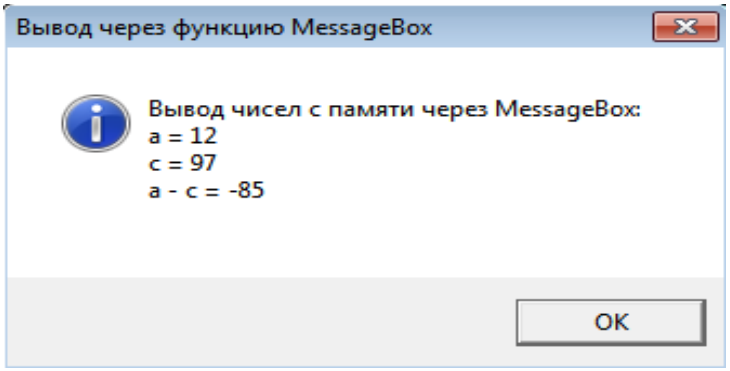


Рис. 6 – Окно с консольным выводом результата выполнения файла **6-1.exe**

Для более глубокого анализа данной программы в случае ее успешной компоновки или же для отладки в случае выявления ошибок используем отладчик **OllyDbg**, для запуска которого необходимо в папке **masm32** открыть папку **OLLYDBG** и выполнить файл **OLLYDBG.exe**.

В открывшемся окне **OllyDbg 6-1.exe** (рис. 7) самостоятельно проанализируйте содержание регистров микропроцессора. Результаты анализа письменно изложите в отчете по работе.

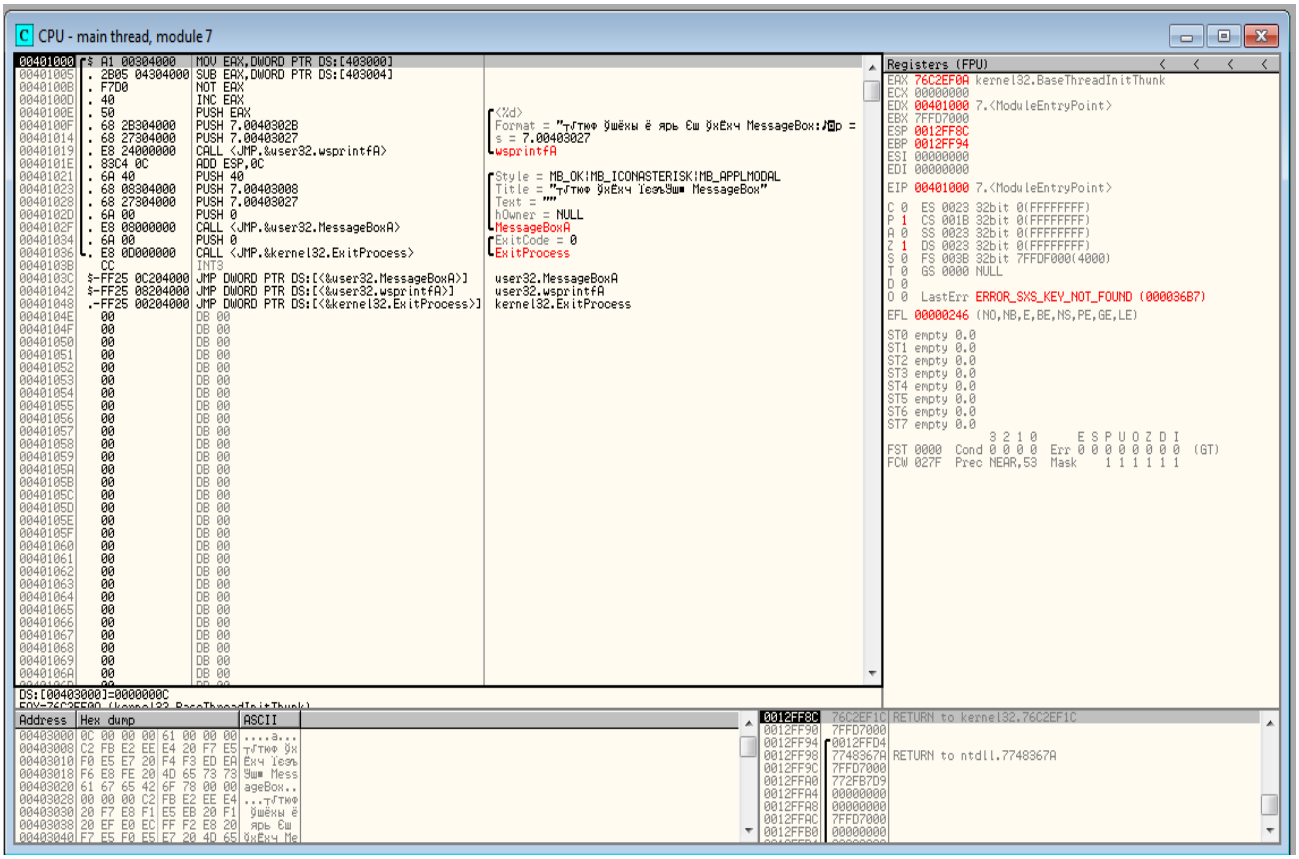


Рис. 7 – Окно программы **OllyDbg 6-1.exe**