Некоммерческое акционерное общество

АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ

Кафедра информационные системы

ПРОГРАММИРОВАНИЕ В АССЕМБЛЕРЕ

Конспект лекций
для студентов специальности 5В060200 - Информатика

Алматы 2015

СОСТАВИТЕЛИ: А.Т. Купарова. Программирование в ассемблере. Конспект лекций для студентов специальности 5В060200 - Информатика. – Алматы: АУЭС, 2015 –71с.

Настоящий конспект лекций составлен в соответствии с программой курса «Программирование в ассемблере» для студентов специальности 5В060200 - Информатика.

Ил.6, табл.5

Рецензент: к.п.н., доц. А.М. Саламатина.

Печатается по плану издания некоммерческого акционерного общества «Алматинский университет энергетики и связи» на 2015 г.

Содержание

Введение

Машинно-ориентированное программирование появилось одновременно с созданием электронных вычислительных машин. Сначала это были программы в машинных кодах, затем появился язык программирования Assembler (Автокод). Этот стиль программирования предполагает досконального знание возможностей конкретной архитектуры ЭВМ и операционной системы и используется до сих пор тогда, когда другие стали бессильны, или нужно получить максимальное быстродействие в рамках той или иной операционной системы с использованием архитектуры данной ЭВМ. Для того чтобы использовать эффективно все возможности компьютера, применяют символический аналог машинного языка – язык ассемблера. История развития вычислительной техники неразрывно связана с историей развития системного программного обеспечения. Современные компьютерные системы наряду с прикладным программным обеспечением всегда содержат системное, которое обеспечивает организацию вычислительного процесса.

Целью курса «Программирование в ассемблере» является изучение особенностей организации и функционирования ЭВМ, для грамотного программирования на любом алгоритмическом языке, умения отлаживать реальные задачи и действительно понимать, как работает ЭВМ. Изучение базовых понятий языка ассемблер, архитектуру компьютера на основе процессора Intel, основные аспекты современного программирования на ассемблере, включая системное и прикладное программирование для DOS и Windows.

В настоящих методических указаниях рассматривается язык ассемблер - как символическое представление машинного языка. Все процессы в машине на самом низком, аппаратном уровне приводятся в действие только командами (инструкциями) машинного языка. Отсюда понятно, что язык ассемблера для каждого типа компьютера свой. Это касается и внешнего вида программ, написанных на ассемблере и идей, отражением которых этот язык является.

Лекция №1. Архитектура ЭВМ

Цель лекции: изучение архитектурных особенностей IBM PC.

Содержание лекции: понятие архитектуры ЭВМ; распределение адресного пространства.

Понятие архитектуры ЭВМ

Архитектура ЭВМ - это абстрактное представление ЭВМ, которое отражает ее структурную, схемотехническую и логическую организацию. Понятие архитектуры ЭВМ является комплексным и включает в себя:

1) Структурную схему ЭВМ.
2) Организацию и разрядность интерфейсов ЭВМ.
3) Набор и доступность регистров.

4) Организацию и способы адресации памяти.
5) Способы представления и форматы данных ЭВМ.
6) Набор машинных команд ЭВМ.
7) Форматы машинных команд.
8) Обработку нештатных ситуаций (прерываний).

Таким образом, понятие архитектуры включает в себя практически всю необходимую для программиста информацию о компьютере. Все современные ЭВМ обладают некоторыми общими и индивидуальными свойствами архитектуры. Индивидуальные свойства присущи только конкретной модели компьютера и отличает ее от больших и малых собратьев. Наличие общих архитектурных свойств обусловлено тем, что большинство типов существующих машин принадлежат 4 и 5 поколениям ЭВМ, так называемой, фон-неймановской архитектуры. К числу общих архитектурных свойств и принципов можно отнести:

1) Принцип хранимой программы.
2) Принцип микропрограммирования.
3) Линейное пространство памяти.
4) Последовательное выполнение программ.
5) Безразличие к целевому назначению данных.

Совокупность всех взаимно связанных блоков, образующих функциональное полное работоспособное средство обработки информации с помощью компьютера, называется компьютерной системой. Основные компоненты стандартной компьютерной системы: системный блок, видеомонитор, клавиатура, печатающее устройство, дисководы и различные средства для асинхронной связи и управления игровыми программами. На рис.1 показана схема компьютера на базе микропроцессоров семейства Intel P6, к которым относятся Pentium Pro/II,III. На схеме представлены: центральный процессор, оперативная память, внешние устройства. Все компоненты соединены между собой через системную шину. Системная шина имеет дополнительную шину - шину расширения. Основу микропроцессора составляют блок микропрограммного управления, исполнительное устройство, регистры. Остальные компоненты микропроцессора выполняют вспомогательные функции. ЭВМ являются преобразователями информации. В них исходные данные задачи преобразуются в результат ее решения. В соответствии с используемой формой представления информации машины делятся на два класса: непрерывного действия - аналоговые и дискретного действия - цифровые. В силу универсальности цифровой формы представления информации цифровые электронные вычислительные машины представляют собой наиболее универсальный тип устройства обработки информации. Основные свойства ЭВМ - автоматизация вычислительного процесса на основе программного управления, огромная скорость выполнения арифметических и логических операций, возможность хранений большого количества различных данных, возможность решения широкого круга математических задач и задач обработки данных. Особое значение ЭВМ

состоит в том, что впервые с их появлением человек получил орудие для автоматизации процессов обработки информации. Управляющие ЭВМ - предназначены для управления объектом или производственным процессом. Для связи с объектом их снабжают датчиками. Непрерывные значения сигналов с датчиков преобразуются с помощью аналогово - цифровых преобразователей в цифровые сигналы, которые вводятся в ЭВМ в соответствии с алгоритмом управления. После анализа сигналов формируются управляющие воздействия, которые с помощью цифро - аналоговых преобразователей преобразуются в аналоговые сигналы. Через исполнительные механизмы изменяется состояние объекта.

Универсальные ЭВМ - предназначены для решения большого круга задач, состав которых при разработке ЭВМ не конкретизируется. Пример современных архитектурных линий ЭВМ: персональные ЭВМ (IBM PC и Apple Macintosh - совместимые машины), машины для обработки специфической информации (графические станции Targa, Silicon Graphics), большие ЭВМ (мэйнфреймы IBM, Cray, ЕС ЭВМ). Общее назначение системного ПО - обеспечивать интерфейс между программистом или пользователем и аппаратной частью ЭВМ (операционная система, программы-оболочки) и выполнять вспомогательные функции (программы-утилиты). Современная операционная система обеспечивает следующее:

1) Управление процессором путем передачи управления программам.
2) Обработка прерываний, синхронизация доступа к ресурсам.
3) Управление памятью.
4) Управление устройствами ввода-вывода.
5) Управление инициализацией программ, межпрограммные связи.
6) Управление данными на долговременных носителях путем поддержки файловой системы.

Архитектура - совокупность технических средств и их конфигураций, с помощью которых реализована ЭВМ. ЭВМ 5 поколения, имеет, как правило, шинную архитектуру, что означает подключение всех устройств к одной электрической магистрали, называемой шиной. Если устройство выставило сигнал на шину, другие могут его считать. Это свойство используется для организации обмена данными. С этой целью шина разделена на 3 адреса - шина адреса, шина данных и шина управляющего сигнала. Все современные ЭВМ также включают устройство, называемое арбитром шины, которое определяет очередность занятия ресурсов шины разными устройствами. В PC распространены шины ISA, EISA, PCI, VLB. Состав и функции основных блоков вычислительной системы: процессора, оперативной памяти, устройства управления, внешних устройств.

Процессор (ЦП) - устройство, выполняющее вычислительные операции и управляющее работой машины содержит устройство управления и арифметико-логическое устройство. Работа всех электронных устройств машины координируется сигналами, вырабатываемыми ЦП.

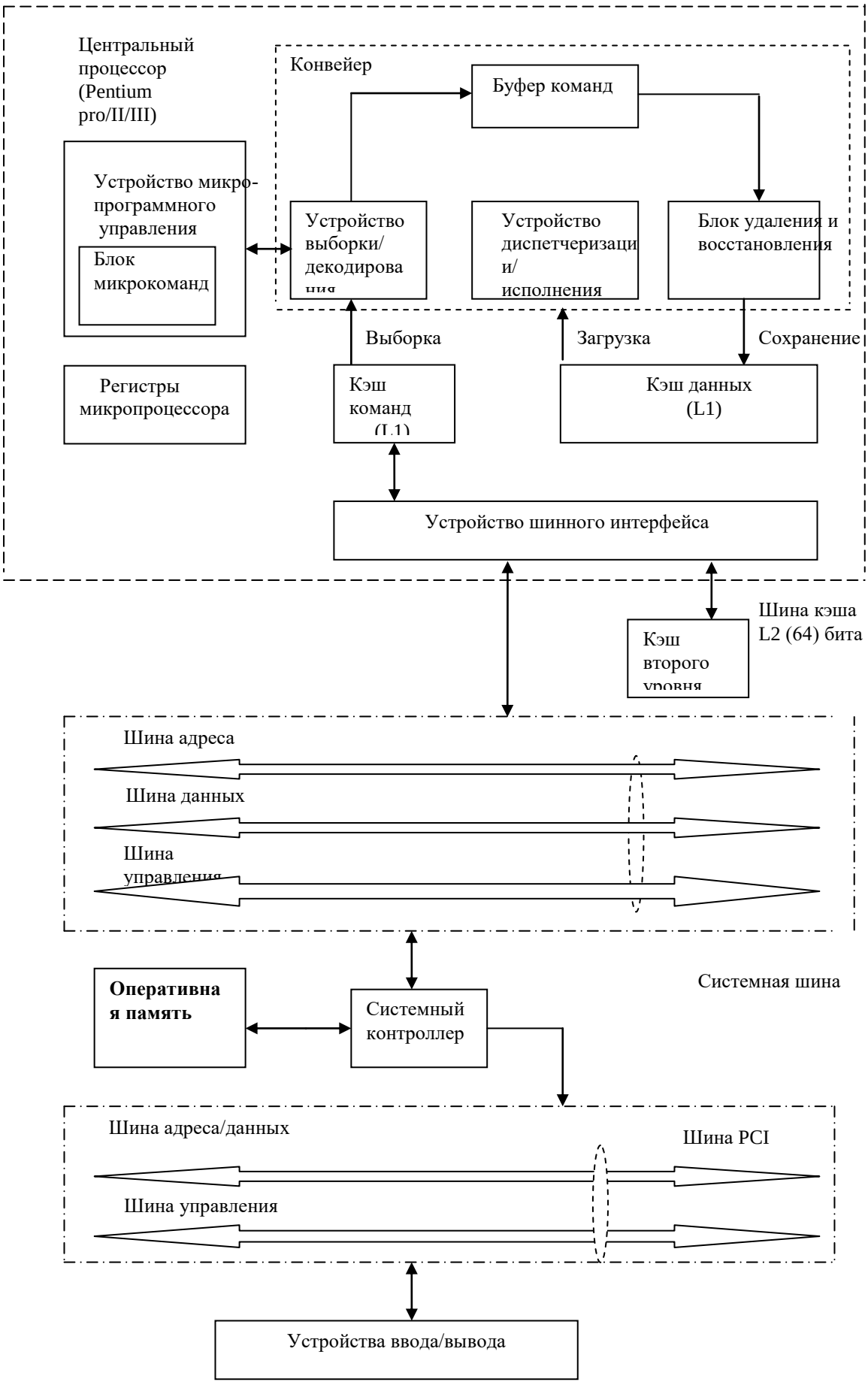

Рисунок 1- Структурная схема персонального компьютера

Распределение адресного пространства

В зависимости от модификации персонального компьютера и состава его периферийного оборудования распределение адресного пространства может несколько различаться. Тем не менее размещение основных компонентов системы довольно строго унифицировано. Типичная схема использования адресного пространств компьютера приведена в таблице 1. Значение адресов на этом рисунке, как и повсюду далее в книге, даны в 16-ричной системе счисления. Признаком 16-ричного числа служит буква **h**, стоящая после числа.

Таблица 1 - Типичное распределение адресного пространства

1 Кбайт	Векторы прерываний	00000h	
256 байтов	Область данных BIOS	00400h	
512 байтов	Область данных DOS	00500h	
	IO.SYS и MS DOS.SYS		
	Загружаемые драйверы		> Стандартная память (640Кбайт)
	COMAND.COM (резидентная часть)		Дополнительная память (EMS)
	Свободная память для загружаемых прикладных и системных программ		
64 Кбайт	Графический буфер EGIA	A0000h =	
32 Кбайт	UMB	B0000h	
32 Кбайт	Текстовой буфер EGIA	B8000h	
64 Кбайт	ПЗУ - расширения BIOS	C0000h	> Верхняя память (384 Кбайт)
64 Кбайт	UMB	D0000h	
128 Кбайт	ПЗУ BIOS	E0000h	
64 Кбайт	HMA	100000h	

DOS 15 Мбайт (80286) До 4 Гбайт (80386/486	XMS	10FFF0h —	>Расширенная память

Первые 640 Кбайт адресного пространства с адресами от 00000h до 9FFFFh отводятся под основную оперативную память, которую еще называют стандартной (conventional). Начальный килобайт оперативной памяти занят векторами прерываний(256 векторов по 4байта). Вслед за векторами прерываний располагается область данных BIOS, которая занимает адреса от 00400h до 004FFh. В этой области хранятся разнообразные данные, используемыми программами BIOS в процессе управлением периферийным оборудованием, так здесь размещаются:

- входной буфер клавиатуры с системой указателей;
- адреса параллельных и последовательных портов;
- данные, характеризующие настройку видео системы (форма курсора и его текущее местоположение на экране, текущий видеорежим, ширина экрана и прочее);
- ячейки для отсчета текущего времени;
- область межзадачных связей и т. д.

Область данных BIOS заполняется информацией в процессе начальной загрузки компьютера и динамически модифицируется системой по мере необходимости; многие прикладные программы обращаются к этой области с целью чтения или модификации содержащейся в ней информации.

В области памяти, начиная с адреса 500h содержится некоторые системные данные DOS. Вслед за областью данных DOS располагаются собственно операционная система, загружаемая из файлов IO. SYS и MSDOS.SYS (IBMBIO.COM и IBMDOS.COM для системы PC-DOS). Система обычно занимает несколько десятков Кбайт.

Перечисленные выше компоненты операционной системе занимают обычно 60-90 Кбайт. Вся оставшаяся память до границы 640 Кбайт (называемая иногда транзитной областью) свободна для загрузки любых систем или прикладных программ. Оставшиеся 384 Кбайт адресного пространства, называемого верхней (upper) памятью, первоначально были предназначены для размещения постоянных запоминающих устройств (ПЗУ). Практически под ПЗУ занята только часть адресов. В самом конце адресного пространства, в области F0000h…FFFFFh (или E0000h… FFFFFh) располагается основное постоянное запоминающее устройство BIOS, а начиная с адреса C0000h-так называемое ПЗУ расширений BIOS для обслуживания графических адаптеров и дисков. Часть адресного пространства верхней памяти отводится для адресации к видео буферам графического адаптера. Приведенное на рисунке расположение видео буферов характерно для адаптера EGA; для других адаптеров оно может быть иным,

(например, видеобуфер простейшего монохромного адаптера MDA занимает всего 4 Кбайт и располагается, начиная с адреса B0000h).

В состав компьютеров PC/AT наряду со стандартной памятью (640Кбайт) может входить расширенная (extended) память, максимальный объем которой зависит от ширины адресной шины процессора при использовании процессора 80386/486 - 4Гбайт. Эта память располагается за пределами первого мегабайта адресного пространства и начиная с адреса 100000h. Поскольку функционирование расширенной памяти подчиняется "Спецификации расширенной памяти" (Extended Memory Specification, сокращенно XMS), то и саму память часто называют XMS-памятью. Как уже отмечалось выше, доступ к расширенной памяти осуществляется в защищенном режиме, поэтому для MS DOS,работающей только в реальном режиме, расширенная память недоступна. Первые 64Кбайт расширенной памяти, точнее, 64Кбайт- 16 байт с адресами от 100000h до 10FFEFh, носят специальное название область старшей памяти (Higih Memory Area, HMA). Эта область замечательна тем, что, хотя она находится за пределами первого мегабайта, к ней можно обратиться в реальном режиме работы микропроцессора.

Лекция №2. Ассемблер как язык программирования и связь его конструкций с архитектурой микропроцессора.

Цель лекции: изучение внутренней архитектуры микропроцессора.
Содержание лекции: свойства ЭВМ, сегментная адресация, режимы работы.

ЭВМ являются преобразователями информации. В них исходные данные задачи преобразуются в результат ее решения. В соответствии с используемой формой представления информации машины делятся на два класса: непрерывного действия - аналоговые и дискретного действия - цифровые. В силу универсальности цифровой формы представления информации цифровые электронные вычислительные машины представляют собой наиболее универсальный тип устройства обработки информации. Основные свойства ЭВМ - автоматизация вычислительного процесса на основе программного управления, огромная скорость выполнения арифметических и логических операций, возможность хранений большого количества различных данных, возможность решения широкого круга математических задач и задач обработки данных. Особое значение ЭВМ состоит в том, что впервые с их появлением человек получил орудие для автоматизации процессов обработки информации. Управляющие ЭВМ – предназначены для управления объектом или производственным процессом. Для связи с объектом их снабжают датчиками. Непрерывные значения сигналов с датчиков преобразуются с помощью аналогово - цифровых преобразователей в цифровые сигналы, которые вводятся в ЭВМ в соответствии с алгоритмом управления. После анализа сигналов формируются управляющие воздействия, которые с помощью цифро - аналоговых преобразователей преобразуются в аналоговые сигналы. Через

исполнительные механизмы изменяется состояние объекта.

Универсальные ЭВМ – предназначены для решения большого круга задач, состав которых при разработке ЭВМ не конкретизируется. Пример современных архитектурных линий ЭВМ: персональные ЭВМ (IBM PC и Apple Macintosh – совместимые машины), машины для обработки специфической информации (графические станции Targa, Silicon Graphics), большие ЭВМ (мэйнфреймы IBM, Cray, ЕС ЭВМ).

Важнейшей характеристикой любого микропроцессора является разрядность его внутренних регистров, а также внешних шин адресов и данных. Микропроцессор i8086 имеет 16-разрядную внутреннюю архитектуру и такой же разрядности шину данных. Таким образом, максимальное целое число (данное или адрес), с которым может работать микропроцессор, составляет 2^{16}-1=65535 (64К-1). Однако адресная шина микропроцессора i8086 содержит 20 линий, что соответствует адресному пространству 2^{20}=1 Мбайт. Для того, чтобы с помощью 16-разрядных адресов можно было обращаться в любую точку 20-разрядного адресного пространства, в микропроцессоре предусмотрена сегментная адресация памяти, реализуемая с помощью четырех сегментных регистров.

Суть сегментной адресации заключается в следующем: исполнительный 20–разрядный адрес любой ячейки памяти вычисляется процессором путем сложения начального адреса сегмента памяти, в котором располагается эта ячейка, со смещением к ней (в байтах) от начала сегмента, которое обычно называют относительным адресом Сегментный адрес без четырех младших бит, т. е деленный на 16, хранится в одном из сегментных регистров. При вычислении исполнительного адреса процессор умножает содержимое сегментного регистра на 16 (путем сдвига влево на 4 двоичных разряда) и прибавляет к полученному 20-разрядному адресу относительный адрес. Умножение базового адреса на 16 увеличивает диапазон адресуемых ячеек до величины 64 Кбайт*16 =1Мбайт

Микропроцессор i80286, используемый как центральный процессор компьютеров IBM PC/AT, является усовершенствованным вариантом i8086, дополненным схемами управления памятью и ее защиты. Микропроцессор i80286 работает с 16-разрядными операндами, но имеет 24-разрядную адресную шину, что соответствует адресному пространству 2^{24}=16 Мбайт. Однако описанный выше способ сегментной адресации памяти не позволяет выйти за пределы 1 Мбайт. Для преодоления этого ограничения в микропроцессоре i80286 (так же, как и в микропроцессоре i80386) используются два режима работы: реального адреса и виртуального защищенного адреса, или просто защищенный режим. В реальном режиме микропроцессор i80286 функционирует фактически также, как микропроцессор i8086 с повышенным быстродействием и может обращаться лишь к 1 Мбайт адресного пространства. Оставшиеся 15 Мбайт памяти, даже если они установлены в компьютере, использоваться не могут.

В защищенном режиме по-прежнему используются сегменты и смещения в них, однако начальные адреса не вычисляются путем умножения на 16 содержимого сегментных регистров, а извлекаются из таблиц сегментных дескрипторов, индексируемых теми же сегментными регистрами. Каждый сегментный дескриптор занимает 6 байтов, из которых 3 байта (24 двоичных разряда) отводятся подсегментный адрес. Тем самым обеспечивая полное использование 24-разрядного адресного пространства.

В каждом сегментном регистре под индекс таблицы сегментных дескрипторов отводится 14 двоичных разрядов. Полный логический адрес адресуемой ячейки состоит из 14- разрядного индекса номера сегмента и 16-разрядного относительного адреса. Это позволяет каждой программе использовать до 2^{30}=1 Гбайт логического, или виртуального пространства, которое таким образом, в 64 раза превышает максимально возможный объем физической памяти. Операционная система виртуальной памяти хранит все сегменты выполняемых программ в большом дисковом пространстве, автоматически загружая в оперативную память те или иные сегменты по мере необходимости.

Микропроцессоры i80386 и i80486 являются высокопроизводительными процессорами с 32-разрядными шинами данных и адресов и 32-разрядной внутренней архитектурой. Последнее означает, что внутренние регистры этих процессоров, в отличии от процессоров ранних моделей, имеют длину 31-бита. Поэтому максимальное целое число, с которым может работать микропроцессор, составляет 2^{32}-1 = 4294967296 (4Гбайт). Во многих случаях использование 32-битовых операндов позволяет существенно упростить и ускорить вычисления. Помимо этого, в микропроцессорах i80386 и i80486 расширен состав регистров, что также предоставляет программисту значительные удобства. Наконец, в новых моделях процессоров имеются встроенные средства поддержки многозадачного режима, а также мультипроцессорных систем. Естественно, что эти процессоры, как и микропроцессор i80286, могут работать в реальном и защищенном режимах. В последнем случае микропроцессор позволяет адресовать до 2^{32}=4 Гбайта физической памяти и 2^{46}=64 Гбайт виртуальной. При этом следует подчеркнуть, что разработчиками обеспечена полная совместимость новых моделей процессоров со старым, в том смысле, что программы, написанные для процессоров i8086 - i80286, т.е с использованием 16-битовых операндов, выполняются на новых процессорах без всяких исправлений.

Лекция №3. Жизненный цикл программы на ассемблере. Структура программы на ассемблере.

Цель лекции: изучить этапы разработки программы на ассемблере.
Содержание лекции: ввод, трансляция, редактирование и выполнение программы.

Язык ассемблер относится к классу машинно-ориентированных языков. Программа на этом языке управляет процессом, уровень абстракции которого соответствует архитектуре ЭВМ, с которой он работает:

- принципом управления процессом обработки информации, формой представления программы и правилами ее исполнения, множеством операций, выполняемых ЭВМ;

- формой представления данных, способами адресации данных в программах;

- организацией обмена информацией между ЭВМ и внешней средой,

способами синхронизации процессов управления работой ЭВМ.

При работе с программами на языке ассемблер, необходимо знать

процессы ассемблирования, компоновки и выполнения программы.

Ввод исходного текста программы

Для ввода исходного текста программы используется любой стандартный текстовый редактор, при этом исходный модуль должен иметь расширение **asm.** Например, можно использовать NORTON EDITOR.

Shift+F4

Имя - программы. asm

Программа может набираться строчными или прописными буквами. Между полями операторов необходимо оставлять хотя бы один пробел. Для удобства чтения листинга программы необходимо записывать метки, команды, операнды и комментарии, выровненными в колонки. Использование комментариев в программе улучшает ее ясность. Комментарий всегда начинается на любой строке исходного модуля с символа «точка с запятой» (;). Комментарий может занимать всю строку или следовать за командой на той или иной строке.

Трансляция программы

Для ввода исходной программы необходимо проделать два основных шага, прежде чем программу можно будет выполнить. Сначала необходимо ассемблировать программу, а затем выполнить компоновку. Шаг ассемблирования включает в себя трансляцию исходного модуля в машинный объектный код и генерацию OBJ-модуля. На этом шаге формулируется объектный модуль, который включают в себя представление исходной программы в машинных кодах, а также информацию ,необходимую для отладки и компоновки его с другими модулями.

Формат командной строки для запуска **TASM.EXE следующий:**

TASM.EXE имя исходного файла. asm

На экране появится:

1) source filename (.ASM): c:имя файла [ENTER];

2) object filename (filename.OBJ): c: [ENTER];

3) source listing (NUL.LST): c: [ENTER];

4) cross-reference (NUL.CRF): c: [ENTER].

Файл с расширением LST показывает сгенерированный машинный код и номера строк.Программа с расширением CRF-это файл перекрестных ссылок показывающий, какие команды ссылаются на какие поля данных. Кроме того, ассемблер генерирует в LST-файле номера строк, которые используются в CRF файле.

Содержание листинга выдается в виде трех столбцов:

1 столбец – четырехзначное шестнадцатеричное значение смещения адреса от начала каждого сегмента;

2 столбец – содержит объектный код каждой команды. Для сегмента стека и данных – это шестнадцатеричные значения соответствующих констан;.

3 столбец – текст исходной программы.

Компоновка программы

Формат OBJ – модуля уже более приближен к исполнительной форме, но еще не готов к выполнению. Шаг компоновки включает преобразование OBJ – модуля в EXE (исполнимый) модуль.

Формат командной строки для запуска **TLINK.EXE**

TLINK.EXE_имя программы. obj

Запрос компоновщика	Ответ
Object Modules (.OBJ)	C: имя-программы
Run file (имя-прогр.EXE)	C:
List file (NUL.MAP):	CON
Libraries (.LIB)	[ENTER]

Файл с расширением MAP содержит таблицу имен и размеров сегментов, а также ошибки, которые обнаружит TLINK.

Выполнение программы

Чтобы запустить программу на выполнение, достаточно вызвать загрузочный модуль:

имя - программы.exe

Синтаксис ассемблера. Предложения, составляющие программу, могут представлять собой синтаксическую конструкцию, соответствующую команде, макрокоманде, директиве или комментарию. Для того, чтобы транслятор ассемблера мог распознавать их, они должны формироваться по

определенным синтаксическим правилам. Общее описание языка ассемблера. Типы данных. Оформление программ. Формат записи команд.

Ассемблер - машинно-ориентированный язык, имеющий два основных достоинства:

1) Позволяет писать программы на уровне команд процессора.

2) Не требует знания этих команд, каждая из них заменяется удобной для запоминания мнемоникой - сокращением английских слов. Транслятор переводит мнемоники в их числовые эквиваленты. Элементы языка: операторы (команды ассемблера + псевдооператоры макроассемблера), операнды, выражения, константы, метки, комментарии. Собственно команды ассемблера процессора - могут быть без операндов, с одним или двумя операндами, использовать различные типы адресации Псевдооператоры - 5 групп: определение идентификаторов (EQU), данных (DB), внешние ссылки (PUBLIC, EXTRN), определение сегментов и подпрограмм (SEGMENT, PROC), управление трансляцией (END).

Константы - могут быть числовые и литералы (последовательность букв, заключенных в апострофы).

Комментарии - начинаются с символа и предназначены для улучшения читаемости программы.

Метки - предназначены для организации переходов в программе. Могут быть локальные и глобальные. Типы данных языка. Целые типы.

BYTE - байт (однобайтовое целое число, код символа, элемент строки).

WORD - слово (целое число со знаком или без знака).

DWORD - двойное слово, длинное целое.

Указатели Полный 32-битовый указатель или 16-битовое смещение. Вещественные типы (типы мат.сопроцессора) - действительные числа длиной 32, 64, 80 бит. Массивы. В ассемблере возможно объявление массивов чисел.

Перичислимые и составные типы ENUM - набор значений, заним. определенное кол-во бит.

RECORD - запись с битовыми полями, каждое из которых имеет длину опр. количество бит и инициализируется некоторыми значениями.

STRUC - структура, элемент содержащий 1 или более типов данных, называемых членами структуры.

UNION (объединение) - то же самое, что и структура, за исключением того, что все члены объединеия занимают 1 и тот же участок памяти.

Формат команды языка:

[Метка:] мнемокод [операнд] [;комментарий]

По умолчанию заглавные и строчные буквы в языке не различаются.

Оформление программ:

```
; Укажем соответствие сегментных регистров сегментам
assume  CS:code, DS:data

; Опишем сегмент кода
; Откроем сегмент кода
```

```
code segment
begin:
; Настроим  DS на сегмент данных
mov AX,  data
mov DS, AX
; Тут вставляется тело вашей программы
; Завершить программу
; Функция DOS завершения программы
mov AX, 4C00h
; Вызов прерывания DOS
int 21h
code ends
; Опишем сегмент данных
; Откроем сегмент данных
data segment
; Тут добавляются данные
; Закроем сегмент данных
data ends
; Опишем  сегмент стека
; Откроем сегмент стека
Stak segment stack
; Отводим под стек 256 байт
db 256 dup (?)
; Закроем сегмент стека
stak ends
; Конец программы с точкой входа
end begin
```

Константы, метки, условная компиляция. Константы - могут быть числовые (десятичные, двоичные, шестадцатеричные) ten EQU 10, antiten EQU - 10, bitmask EQU 10001001b, video EQU 0A000h, и литералы - символьные, EQU ‘string data'.

Метки - служат для присваивания имени команде языка ассемблера. Предназначены для организации переходов в программе. Метка может содержать следующие символы:

- буквы от A до Z и a до z;
- цифры от 0 до 9;
- специальные символы ?; точка (.); подчеркивание(_); коммерческое эт (@); знак доллара ($).

Глобальные метки действуют во всей программе. Локальные - только внутри подпрограммы. Директивы условной трансляции предназначены для обозначения блока программного кода, который включается в объектный файл только тогда, когда выполняется заданное условие. Синтаксис: Ifxxx операторы, помещаемые в файл при выполнении условия ELSE ;операторы, помещаемые в файл, если условие не выполнено ENDIF.

Лекция №4. Структура и образ памяти программы .EXE и . COM

Цель лекции: изучение структуры программы

Содержание лекции: структура типичной программы типа .EXE и . COM, операнды директивы SEGMENT.

Программы, выполняемые под управлением MS-DOS,могут принадлежать к одному из двух типов: .COM и .EXE. В программах типа .EXE код программы, данных и стек занимают отдельные сегменты, а в .COM – один сегмент.

Таким образом, размер программы типа .COM не может превысить 64Кбайт, а размер программы типа EXE практически неограничен, так как в нее может входить любое число сегментов программы и данных.

При загрузке в программы сегменты размещаются в памяти, как показано в таблице 2.1

Образ программы в памяти начинается с префикса программного сегмента (Program Segment Prefics, PSP), образуемого и заполняемого системой.

PSP всегда имеет размер 256 байтов содержит таблицы и поля данных, используемые системой в процессе выполнения программы. Вслед за PSP располагаются сегменты программы. Сегментные регистры автоматически инициализируются следующим образом: ES и DS указывают на начало PSP, CS- на начало сегмента команд, а SS- на начало сегмента стека. В указатель команд IP загружается относительный адрес точки входа в программу (из операнда директивы END), а указатель стека SP- смещение конца сегмента стека. Таким образом после загрузки программы в память адресуемыми оказываются все сегменты, кроме сегмента данных. Инициализация регистра DS в первых строках программы позволяет сделать адресуемым и этот сегмент

Таблица 2.1 - Образ памяти программы . EXE

ES,DS----->	PSP (256 байтов)	
CS----->	Сегмент команд	<——IP
	Сегмент данных	
SS—>	Сегмент стека	
		<-----SP

Структура и образ памяти программы . COM

Программа типа . COM отличается от программы типа .EXE тем, что содержит лишь один сегмент, включающий все компоненты программы: PSP, программный код (т.е оттранслированные в машинные коды программные строки), данные и стек. Структура типичной программы типа . COM на языке ассемблера выглядит следующим образом: (таблица 2.2)

Таблица 2.2

title	Программа	Типа	.COM
text	segment	'code'	
	assume	CS: text,	DS:tex t
	org	100h	

```
Myproc      proc
            ...                          ;Текст программы
Myproc      Endp
            ...                          ;Определения
                                         данных
text        Ends
            End         Myproc
```

Программа содержит единственный сегмент text, которому присвоен класс 'CODE '. В операторе ASSUME указано, что сегментные регистры CS и DS будут указывать а этот единственный сегмент

Оператор ORG 100h резервирует 256 байтов для PSP. Заполнять PSP будет по- прежнему система, но место под него в начале сегмента должен отвести программист. В программе нет необходимости инициализировать сегментный регистр DS, поскольку его, как и остальные сегментные регистры, инициализирует система. Данные можно разместить после программной процедуры (как это показано на рисунке), или внутри нее, или даже перед ней. Следует только иметь ввиду, что при загрузке программы типа . COM регистр IP всегда инициализируется числом 100 h, поэтому сразу вслед за оператором ORGI 100h должна стоять первая выполнимая строка программы. Если данные желательно расположить в начале программы, перед ними следует поместить оператор перехода на реальную точку входа, например JMP Entry.

Образ памяти программы типа .COM показан в таблице 2.3 После загрузки программы все сегментные регистры указывают на начало единственного сегмента, т.е фактически на начало PSP. Указатель стека автоматически инициализируется числом FFFEh. Таким образом, независимо от фактического размера программы ей выделяется 64 Кбайт адресного пространства, всю нижнюю часть которого занимает стек.

Таблица 2.3 - Образ памяти программы . COM

CS, DS, ES, SS —>	PSP (256 байтов)	
	Программа и данные	<— IP
	Стек	
		<— SP = FFFEh

Синтаксическое описание сегмента на ассемблере представляет собой конструкцию, изображенную на рисунке 2. Важно отметить, что функциональное назначение сегмента несколько шире, чем простое разбиение программы на блоки кода, данных и стека. Сегментация является частью более общего механизма, связанного с концепцией модульного программирования. Она предполагает унификацию оформления объектных модулей, создаваемых компилятором, в том числе с разных языков программирования. Это позволяет объединять программы, написанные на

разных языках. Именно для различных вариантов такого объединения и предназначены операнды в директиве SEGMENT. Рассмотрим их подробнее:

Атрибут выравнивания сегмента (тип выравнивания) сообщает компоновщику о том, что нужно обеспечить размещение начало сегмента на заданной границе. Это важно, поскольку при правильном выравнивания доступ к данным процессорах i80x86 выполняется быстрее. Допустимые значения этого атрибута следующие:

- BYTE- уравнивания не выполняется сегмент может начинаться с любого адреса памяти;
- WORD-сегмент начинается по адресу, кратному двум, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);
- DWORD- сегмент начинается по адресу кратному четырём, то есть два последних (младших) значащих бита равны 0 (выравнивание на границу двойного слова) ;
- PARA-сегмент начинается по адресу кратному 16, то есть последнему шестнадцатеричная цифра адреса должна быть oh (выравнивание на границе параграфа) ;
- PAGE- сегмент начинается по адресу кратному 256, то есть две последние шестнадцатеричные цифры должны быть ooh (выравнивание на границу страницы размерам 256 байт);
- MEMPAGE-- сегмент начинается по адресу кратному 4 Кбайт, то есть три последние цифры должны быть oooh (адрес следующей страницы
- памяти размером 4 Кбайт)

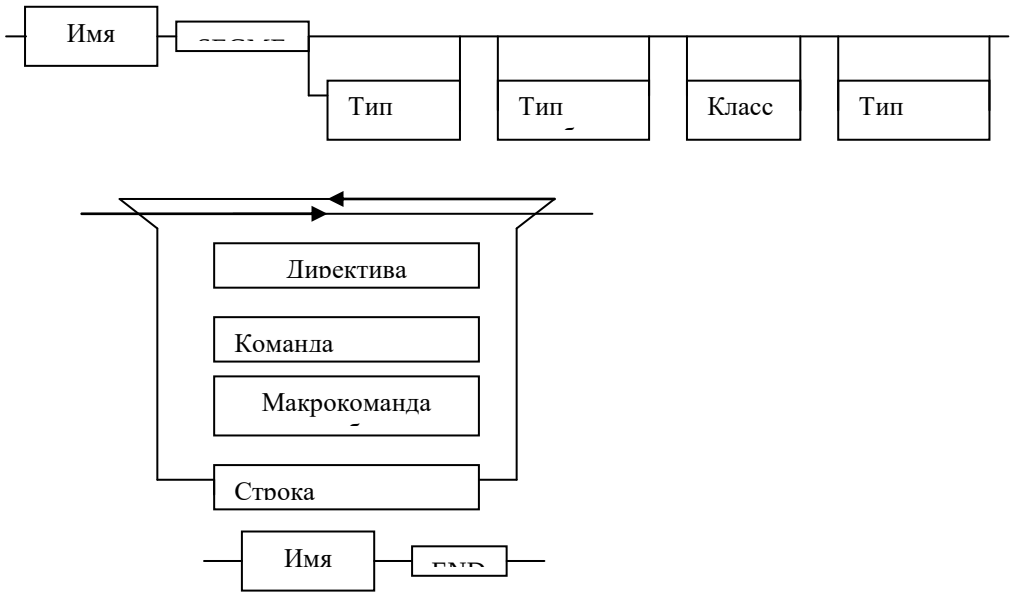

Рисунок 2 - Синтаксис описания сегмента

По умолчанию тип выравнивания имеет значение PARA

Атрибут комбинирования сегментов (комбинаторный тип) сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. По умолчанию атрибут комбинирования принимать значение PRIVATE. Значениями атрибута комбинирования сегмента могут быть:

- PRIVATE-сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;
- PUBLIC- заставлять компоновщик соединить все сегменты с одинаковым именем. Новый объединенной сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть в зависимости от типа сегмента, команды или данные, будут вычисляться относительно начала этого нового сегмента;
- COMMON- располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;
- STACK- определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра ss. Комбинированный тип STACK (стек) аналогичен комбинированному типу PUBLIC, за исключением того, что регистр ss является стандартным сегментным регистром для сегментов стека. Если не указана ни одного сегмента сетка, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан , а комбинированный тип STACK не используется программист должен явно загрузить в регистр ss адрес сегмента (подобно тому, как это делается для регистра ds).

Атрибут класса сегмента (тип класса)-это заключенная в кавычки строка, помогающая компоновщику определить соответствующий порядок следования сегментов при сборке программы из сегментов нескольких модулей . Компоновщик объединяет вместе в памяти все сегменты с одним и тем же именем класса (имя класса, в общем случае, может быть любым, но лучше, если оно будет отражать функциональное назначение сегмента). Типичным примером использования имени класса является объединение в группу всех сегментов кода программы (обычно для этого используется класс << code>>). С помощью механизма типизации класса можно группировать также сегменты инициализированных и неинициализированных данных.

5 Лекция №5. Программно доступные элементы машины

Цель лекции: изучение регистров IBM PC XT

Содержание лекции: регистры общего назначения, сегментные регистры.

В программах на языке ассемблера регистры используются очень интенсивно. Большинство из них имеет определенное функциональное

назначение. Программная модель микропроцессора имеет несколько групп регистров, доступных для использования в программах.

Пользовательские регистры, к которым относятся:

-регистры общего назначения
eax/ax/ah/al,ebx/bx/bh/bl,edx/dx/dh/dl,ecx/cx/ch/cl,ebp/bp,esi/si,edi/di,esp/sp.

Регистры этой группы используются для хранения данных и адресов;

-сегментные регистры cs,ds,ss,es,fs,gs. Регистры этой группы используются для хранения адресов сегментов в памяти;

-регистры сопроцессора st(0),st(1),st(2),st(3),st(4),st(5),st(6),st(7). Регистры этой группы предназначены для написания программ, использующих тип данных с плавающей точкой.

Регистры состояния и управления - это регистры, которые содержат информацию о состоянии микропроцессора, исполняемой программы и позволяют изменить это состояние;

-регистр флагов eflags/flags;

-регистр указатель команды eip/ip;

Системные регистры- это регистры для поддержания различных режимов работы,сервисных функций, а также регистры, спецефичные для определенной модели микропроцессора.

Регистры общего назначения.

Регистры общего назначения используются в программах для хранения:

-операндов логических и арифметических операций;

-компонентов адреса;

-указателей на ячейки памяти.

Регистр eax/ax/ah/al-аккумулятор, применяется для всех операций ввода-вывода, некоторых операций над строками и арифметические операции (сложение, умножение).

Регистр ebx/bx/bh/bl-базовый регистр, применяется для хранения базового адреса некоторого обьекта в памяти.

Регистр ecx/cx/ch/cl-регистр-счетчик, применяется для управления числом повторений циклов и для операций сдвига влево или вправо.Также используется в вычислениях.

Регистр edx/dx/dh/dl-регистр данных, применяется для некоторых операций ввода-вывода,а также используется в командах умножения и деления над большими числами.

Сегментные регистры.

Базовых сегментных регистров – четыре. Все сегментные регистры служат для указания начала соответствующего сегмента. Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel, и заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых *сегментами.* Соответственно, такая организация памяти называется сегментной. В программной модели микропроцессора имеется следующие сегментные регистры: CS, DS, SS,ES.

Регистр CS (Code Segment) – регистр **с**егмента кода, содержит начальный адрес **с**егмента кода. Этот адрес плюс величина смещения в командном указателе (регистр IP) определяют адрес команды, которая должна быть выбрана для выполнения. Для обычных программ нет необходимости делать ссылки на регистр **CS.**

Регистр DS (Data Segment) регистр - сегмента данных, содержит его начальный адрес. Этот адрес плюс величина смещения, определенная в команде, указывают на конкретную ячейку в сегменте данных.

Регистр SS (Stack Segment) – регистр сегмента стека, содержит начальный адрес сегмента стека.

Регистр ES (Extra Segment) – регистр сегмента расширения. Некоторые операции над строками используют дополнительный сегментный регистр ES для управления адресацией памяти. В этом случае регистр ES связан с регистром DI. Обычно эту связь обозначают парой <ES:DI>. Если необходимо использовать регистр ES, ассемблерная программа должна его инициализировать явно.

Регистры указателей и индексов.

eip/ip-указатель команд. Регистр eip/ip- имеет разрядность 32/16 бит и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра eip/ip.

esi/si-индекс источника. Этот регистр в цепочечных операциях содержит текущий адрес в цепочке источника;

edi/di-индекс приемника (получателя). Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-источника;

esp/sp-регистр указателя стека. Содержит указатель вершины стека в текущем сегменте стека.

ebp/bp-регистр указателя базы. Предназначен для организации произвольного доступа к данным внутри стека.

Регистры состояния и управления.

В микропроцессор включены несколько регистров (рис.3), которые постоянно содержат информацию о состоянии как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер. К этим регистрам относятся:

- регистр флагов **eflags/flags**;
- регистр указателя команды **eip/ip**.

Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора. Рассмотрим подробнее назначение и содержимое этих регистров:

eflags/flags (flag register) — регистр флагов. Разрядность eflags/flags — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру flags для i8086. На рисунке 5 показано содержимое регистра eflags.

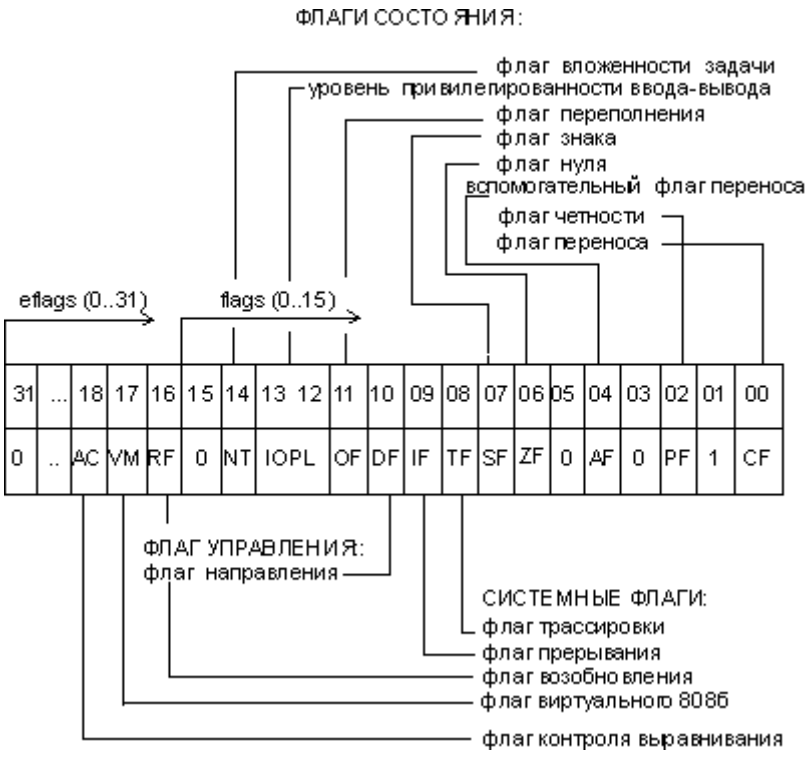

Рисунок 3 - Содержимое регистра eflags

Исходя из особенностей использования, флаги регистра *eflags/flags* можно разделить на три группы:

- *8 флагов состояния*. Эти флаги могут изменяться после выполнения машинных команд.

Флаги состояния регистра eflags отражают особенности результата

исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм.

- *1 флаг управления*. Обозначается df (Directory Flag). Он находится в 10-м бите регистра eflags и используется цепочечными командами. Значение флага df определяет направление поэлементной обработки в этих операциях: от начала строки к концу (df = 0) либо наоборот, от конца строки к ее началу (df = 1). Для работы с флагом df существуют специальные команды: cld (обнулить флаг df) и std (установить флаг *df*). Применение этих команд позволяет привести флаг *df* в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;
- *5 системных флагов*, управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086.

Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы.

Флаги состояния

CF – Флаг переноса (Carry Flag). Устанавливается в 1, если в результате арифметической операции произошел перенос из старшего бита результата. Старшим является 7, 15 или 31-й бит в зависимости от размерности операнда. Если переноса не было, то CF = 0.

PF - Флаг паритета (Parity Flag). Устанавливается в 1, если младший байт результата содержат четное число единиц, иначе – 0.

AF – Вспомогательный флаг переноса (Auxiliary carry Flag). Устанавливается в 1, если в результате предыдущей операции произошел перенос или заем между 3 и 4 разрядами, иначе – 0.

ZF - Флаг нуля (Zero Flag). Устанавливается в 1, если получен нулевой результат, иначе – 0.

SF – Флаг знака (Sign Flag). Он всегда равен старшему биту результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно).

OF – Флаг переполнения (Overflow Flag). Устанавливается в 1, если результат предыдущей арифметической операции над числами со знаком выходит за допустимые для них пределы.

IOPL – Уровень привилегий ввода-вывода (Input/Output Privilege Level). Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода-вывода в зависимости от привилегированности задачи.

NT – флажок вложенности задачи (Nested Task).

Системные флаги

TF - Флаг трассировки (Trace Flag). Предназначен для организации пошаговой работы микропроцессора. Если TF=1 — микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ, в частности отладчиками. А если TF=0 — обычная работа.

IF - Флаг прерывания (Interrupt enable Flag). Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR). 1—аппаратные прерывания разрешены; 0—аппаратныепрерывания запрещены.

RF – Флаг возобновления (Resume Flag). Используется при обработке прерываний от регистров отладки.

VM – Флаг виртуального режима (Virtual 8086 Mode). Признак работы микропроцессора в режиме виртуального 8086. 1 — процессор работает в режиме виртуального 8086;
0 — процессор работает в реальном или защищенном режиме.

AC - Флаг контроля выравнивания (Alignment Check). Предназначен для разрешения контроля выравнивания при обращениях к памяти. Используется совместно с битом AM в системном регистре CR0. К примеру, Pentium разрешает размещать команды и данные с любого адреса. 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 возбуждать исключительную ситуацию

eip/ip (Instraction Pointer register) — регистр-*указатель команд*. Регистр *eip/ip* имеет разрядность 32/16 бит и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра *eip/ip*.

Лекция №6 Системные регистры микропроцессора

Цель лекции: изучение регистров IBM PC XT

Содержание лекции: регистры управления, системных адресов, отладки.

Использование системных регистров жестко регламентировано. Именно они обеспечивают работу защищенного режима. Их также можно рассматривать как часть архитектуры микропроцессора, которая намеренно оставлена видимой для того, чтобы квалифицированный системный программист мог выполнить самые низкоуровневые операции.

Системные регистры можно разделить на три группы:

- четыре регистра управления;
- четыре регистра системных адресов;
- восемь регистра отладки.

Регистры управления

В группу регистров управления входят 4 регистра: CR0, CR1, CR2, CR3. Эти регистры предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0. Хотя микропроцессор имеет четыре регистра управления, доступными являются только три из них — исключается CR1, функции которого пока не определены (он зарезервирован для будущего использования).

Регистр CR0 содержит системные флаги, управляющие режимами работы микропроцессора и отражающие его состояние глобально, независимо от конкретных выполняющихся задач.
Назначение системных флагов:

- pe (Protect Enable), бит 0 — разрешение защищенного режима работы. Состояние этого флага показывает, в каком из двух режимов — реальном (pe=0) или защищенном (pe=1) — работает микропроцессор в данный момент времени.
- mp (Math Present), бит 1 — наличие сопроцессора. Всегда 1.
- ts (Task Switched), бит 3 — переключение задач. Процессор автоматически устанавливает этот бит при переключении на выполнение другой задачи.
- *am* (Aligment Mask), бит 18 — *маска выравнивания*. Этот бит разрешает (am = 1) или запрещает (am = 0) контроль выравнивания.
- *cd* (Cache Disable), бит 30, — *запрещение кэш-памяти*. С помощью этого бита можно запретить (cd = 1) или разрешить (cd = 0) использование внутренней кэш-памяти (кэш-памяти первого уровня).
- *pg* (PaGing), бит 31, — *разрешение (pg = 1) или запрещение (pg = 0) страничного преобразования*. Флаг используется при страничной модели организации памяти.

Регистр CR2 используется при страничной организации оперативной памяти для регистрации ситуации, когда текущая команда обратилась по адресу, содержащемуся в странице памяти, отсутствующей в данный момент времени в памяти. В такой ситуации в микропроцессоре возникает исключительная ситуация с номером 14, и линейный 32-битный адрес команды, вызвавшей это исключение, записывается в регистр CR2. Имея эту информацию, обработчик исключения 14 определяет нужную страницу, осуществляет ее подкачку в память и возобновляет нормальную работу программы.

Регистр CR3 также используется при страничной организации памяти. Это так называемый регистр каталога страниц первого уровня. Он содержит 20-битный физический базовый адрес каталога страниц текущей задачи. Этот каталог содержит 1024 32-битных дескриптора, каждый из которых содержит адрес таблицы страниц второго уровня. В свою очередь каждая из таблиц страниц второго уровня содержит 1024 32-битных дескриптора, адресующих страничные кадры в памяти. Размер страничного кадра — 4 Кбайт.

Регистры системных адресов

Эти регистры еще называют *регистрами управления памятью*. Они предназначены для защиты программ и данных в мультизадачном режиме работы микропроцессора.

При работе в защищенном режиме микропроцессора адресное пространство делится на:

- *глобальное* — общее для всех задач;
- *локальное* — отдельное для каждой задачи.

Этим разделением и объясняется присутствие в архитектуре микропроцессора следующих системных регистров:

- *регистра таблицы глобальных дескрипторов* GDTR (Global Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый (биты 16—47) базовый адрес глобальной дескрипторной таблицы GDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы GDT;
- *регистра таблицы локальных дескрипторов* LDTR (Local Descriptor Table Register) имеющего размер 16 бит и содержащего так называемый селектор дескриптора локальной дескрипторной таблицы LDT. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT;
- *регистра таблицы дескрипторов прерываний* IDTR (Interrupt Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый (биты 16–47) базовый адрес дескрипторной таблицы прерываний IDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы IDT;
- *16-битового регистра задачи* TR (Task Register), который подобно регистру ldtr, содержит селектор, то есть указатель на дескриптор в таблице GDT. Этот дескриптор описывает *текущий сегмент состояния задачи* (TSS — Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит контекст (текущее состояние) задачи. Основное назначение сегментов TSS — сохранять текущее состояние задачи в момент переключения на другую задачу.

Регистры отладки

Это очень интересная группа регистров, предназначенных для аппаратной отладки. Средства аппаратной отладки впервые появились в микропроцессоре i486. Аппаратно микропроцессор содержит восемь регистров отладки, но реально из них используются только 6.

Регистры DR0, DR1, DR2, DR3 имеют разрядность 32 бит и предназначены для задания линейных адресов четырех точек прерывания. Используемый при этом механизм следующий: любой формируемый текущей программой адрес сравнивается с адресами в регистрах DR0...DR3, и при совпадении генерируется исключение отладки с номером 1. Регистр DR6

называется регистром состояния отладки. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали возникновение последнего исключения с номером 1.

Перечислим эти биты и их назначение:

- b0 — если этот бит установлен в 1, то последнее исключение (прерывание) возникло в результате достижения контрольной точки, определенной в регистре DR0;
- b1 — аналогично b0, но для контрольной точки в регистре DR1;
- b2 — аналогично b0, но для контрольной точки в регистре DR2;
- b3 — аналогично b0, но для контрольной точки в регистре DR3;
- bd (бит 13) — служит для защиты регистров отладки;
- bs (бит 14) — устанавливается в 1, если исключение 1 было вызвано состоянием флага tf = 1 в регистре eflags;
- bt (бит 15) устанавливается в 1, если исключение 1 было вызвано переключением на задачу с установленным битом ловушки в TSS t = 1.

Все остальные биты в этом регистре заполняются нулями. Обработчик исключения 1 по содержимому DR6 должен определить причину, по которой произошло исключение, и выполнить необходимые действия.

Регистр DR7 называется регистром управления отладкой. В нем для каждого из четырех регистров контрольных точек отладки имеются поля, с помощью которых можно уточнить следующие условия, при которых следует сгенерировать прерывание:

место регистрации контрольной точки — только в текущей задаче или в любой задаче. Эти биты занимают младшие восемь бит регистра DR7 (по два бита на каждую контрольную точку (фактически точку прерывания), задаваемую регистрами DR0, DR1, DR2, DR3 соответственно). Первый бит из каждой пары — это так называемое локальное разрешение; его установка говорит о том, что точка прерывания действует если она находится в пределах адресного пространства текущей задачи.

Второй бит в каждой паре определяет глобальное разрешение, которое говорит о том, что данная контрольная точка действует в пределах адресных пространств всех задач, находящихся в системе;

тип доступа, по которому инициируется прерывание: только при выборке команды, при записи или при записи/чтении данных. Биты, определяющие подобную природу возникновения прерывания, локализуются в старшей части данного регистра.

Большинство из системных регистров программно доступны.

Лекция №7. Способы задания операндов команды. Команды пересылки данных

Цель лекции: изучение режимов адресации для составления программ.

Содержание лекции: режимы адресации: регистровый, непосредственный, косвенный, прямой, по базе, прямой с индексированием, по базе с индексированием. Команды пересылки общего назначения.

Операнды могут находится в регистрах, памяти, в самих командах. Способ нахождения операнда определяется режимом адресации. В командах ассемблера используются следующие режимы адресации: регистровый, непосредственный, косвенный, прямой, по базе, прямой с индексированием, по базе с индексированием.

1. Регистровый режим адресации

Операнд находится в регистре:

MOV AX, BX

2. Непосредственный режим адресации

Операнд находится в команде:

а) MOV AX, 10. ; 10 – непосредственный операнд;

б) MOV BX, OFFSET A; OFFSET A – непосредственный операнд;

в) MOV AX, K; K – непосредственный операнд, если K определен с помощью псевдооператоров EQU или =; т.е.

K EQU 10.

3. Косвенный режим адресации

Операнд находится в памяти, например: MOV AX, [BX]; в BX находится адрес операнда. Квадратные скобки указывают на то, что в регистре находится адрес. Для указания адреса операнда могут использоваться регистры BX, BP, SP, SI, DI

4. Прямой режим адресации

Операнд находится в памяти. В команде указывается адрес операнда: MOV AX, A ; A – адрес операнда, который описан в сегменте данных в виде оператора (строки):

A DW 10, 20, 30

MOV AX, A+2; AX = 20

5. Режим адресации по базе

Операнд находится в памяти, адрес операнда задается с помощью одного из базовых регистров BX или BP:

MOV AL, [BX]

MOV DX, [BP]

MOV DX, [BX+2]

6. Прямой режим адресации с индексированием

Операнд находится в памяти. Адрес операнда вычисляется сложением сдвига и содержимого одного из индексных регистров SI или DI:

MOV AL, A[SI]

MOV B[DI], BX

7. Режим адресации по базе и индексированием

Операнд находится в памяти. Адрес операнда вычисляется сложением сдвига, содержимым одного индексного и одного базового регистров:

MOV AX, TAB[BX][SI]

MOV A[BP][DI], AX

Для перечисленных режимов адресации операнды находятся в сегменте данных, исключением является использование при адресации регистра BP, когда операнд находится в сегменте стека.
Адреса операндов загружаются в регистры с помощью команд:
MOV BX, OFFSET TAB или LEA BX, TAB

Команды пересылки данных.

Команды пересылки используются для пересылки данных, адресов, флагов.

Команды пересылки общего назначения.

MOV - команда пересылки. Формат команды: MOV приемник, источник

Команда MOV пересылает операнд источник (байт, слово или двойное слово) на место операнда приемника, где в качестве приемника может использоваться регистр или ячейка памяти, а в качестве источника - регистр, ячейка памяти или константа.

С помощью команды MOV нельзя переслать:
а) содержимое одной ячейки памяти в другую;
б) содержимое одной ячейки памяти в сегментный регистр и наоборот;
в) содержимое одного сегментного регистра в другой сегментный регистр.

Эти пересылки можно выполнить через промежуточный регистр. В качестве промежуточного регистра используются регистры общего назначения, кроме SP.

В команде MOV разрешены пересылки: регистр - регистр, регистр - память, память - регистр, регистр - непосредственное значение, память - непосредственное значение.Например:

```
mov   AX, DX
mov   AX, FLDA[SI]
mov   FLDA, AX
mov   AL, 22h
mov   FLDA[BP][SI], 33h
```

PUSH - команда записи в стек. Формат команды: PUSH источник

Команда PUSH записывает в стек слово или двойное слово. В качестве источника используется регистр или ячейка памяти. Например:

```
push  CX
push  TABL
```

POP - Команда чтения из стека. Формат команды: POP приемник

Команда POP читает из стека слово или двойное слово. В качестве приемника используется регистр или ячейка памяти. Например:

```
pop   BX
pop   TABL
```

XCHG - команда обмена. Формат команды: XCHG операнд1, операнд2

Команда меняет местами байты, слова или двойные слова. В качестве операндов могут использоваться регистры и ячейки памяти в сочетаниях: регистр - регистр, регистр - память.

Например: xchg AX, DX

```
xchg  BX, A[SI]
```

XLAT - Команда перекодировки. Формат команды: XLAT таблица_источник

Команда вычисляет адрес, равный ds:bx+(al), а затем выполняет замену байта в регистре al байтом из памяти по вычисленному адресу.

При использовании команды XLAT номер элемента, который нужно перекодировать заносится в регистр AL, а адрес таблицы_источника заносится в регистр BX. После выполнения команды XLAT результат заносится в регистр AL. Например:

```
; в сегмент данных
    ASCII       DB  '0123456789'
; в сегмент кода
    mov   BX, offset ASCII
    mov   AL, 5
    xlat                        ; AL=35h
```

Команды ввода-вывода

IN - Ввод операнда из порта. Формат команды: IN аккумулятор, порт
Команда IN пересылает байт, слово или двойное слово из порта в микропроцессор.

OUT - Вывод операнда в порт. Формат команды: OUT порт, аккумулятор
Команда пересылает байт, слово или двойное слово из микропроцессора в порт.

В качестве аккумулятора используются регистры AL (для пересылки байтов), AX (для пересылки слов) и EAX (для пересылки двойных слов), в качестве оператора порт используются номера портов от 0 до 255 или регистр DX. Например:

```
in      AL, 60h
out     20h, AL
in      AX, DX
out     DX, AX
```

Команды пересылки адреса.

LEA - Команда загрузки эффективного адреса. Формат команды: LEA приемник, источник

При выполнении команды в регистр приемник загружается 16-битное, либо 32-битное значение смещения операнда источник. В качестве операнда приемник используются регистры общего назначения. Операнд источник должен быть определен через директиву DW или DD. Например:

```
TAB  DW   10h, 20h,30h
lea     BX, TAB
```

Данная команда является альтернативой оператору ассемблера offset. В отличие от offset команда lea допускает индексацию операнда, что позволяет более гибко организовать адресацию операндов. Например:

```
; загрузить в регистр bx адрес пятого элемента массива mas
.data
 mas db 10 dup (0)
.code
        ...
 mov            di, 4
 lea    bx, mas[di]
;или
 lea   bx, mas[4]
;или
 lea    bx, mas+4
```

LDS/LES/LFS/LGS/LSS Загрузка сегментного регистра ds/es/fs/gs/ss указателем из памяти

Формат команды: LDS приемник,источник

Команда загружает младшее слово в регистр, указанный в команде, а старшее слово - в сегментный регистр DS. Например:

```
main  DD    10000100h
lds     BX, main
```

Команда LDS заменяет 3 команды MOV:

```
mov   BX, offset main
mov   AX, seg main
mov   DS, AX
```

Команды LES, LFS, LGS, LSS выполняются аналогична **LDS**, но адрес сегмента загружается в регистры ES, FS, GS, SS соответственно.

Команды пересылки флагов.

LAHF - Загрузка флагов в регистр AH

Команда LAHF загружает младший байт регистра флагов в регистр AH.

SAHF - Установка флагов из регистра AH

Команда SAHF переписывает разряды 0, 2, 4, 6, 7 регистра AH в младший байт регистра флагов.

PUSHF - Занесение флагов в стек

Команда PUSHF сохраняет содержимое регистра флагов в стеке.

POPF - Извлечение флагов из стека

Команда POPF читает слово из стека и записывает его в регистр флагов.

Лекция №8. Команды передачи управления

Цель лекции: изучение команд для программирования нелинейных алгоритмов.

Содержание лекции: команды безусловной, условной передачи управления, команды управления циклом.

По принципу действия команды микропроцессора, обеспечивающие организацию переходов в программе, можно разделить на три группы:

1. Команды безусловной передачи управления:

- JMP метка - команды безусловного перехода;

- CALL имя процедуры и **RET-** вызов процедуры и возврат из процедуры;

- INT номер прерывания - вызов программных прерываний и возврат из программных прерываний.

2. Команды передачи управления:

- Jxx метка_ перехода, где xx-определяет конкретное условие, анализируемое командой;

- CMP *операнд 1, операнд **2*** **-**команды перехода по результату команды сравнения;

- команды перехода по состоянию определенного флага:

JC, (JNC); JP, (JNP); JZ, (JNZ); JS, (JNS); JO, (JNO);

- команды перехода по содержимому регистра CX: JCXZ < метка – перехода >

3. Команды управления циклом:

- LOOP метка_перехода-повторить цикл;

- команда организации цикла со счетчиком.

В программах на языке ассемблер для передачи управления применяются метки. Метка- это символическое имя, обозначающее определенную ячейку памяти, предназначенное для использования в качестве операнда в командах передачи управления.

Команды безусловной передачи управления.

Jmp - команда безусловного перехода. Формат команды: jmp [модификатор] адрес_перехода

Адрес_перехода представляет собой адрес в виде метки либо адрес области памяти, в которой находится указатель перехода. Имеется несколько кодов машинных команд безусловного перехода **jmp**. Адрес_перехода может находиться в текущем сегменте кода или в некотором другом сегменте. В первом случае переход называется *внутрисегментным*, или *близким*, во втором — *межсегментным*, или *дальним*. При внутрисегментном переходе изменяется только содержимое регистра **eip/ip**. Можно выделить три варианта внутрисегментного использования команды jmp: прямой короткий; прямой; косвенный.

Модификатор может принимать следующие значения:

near ptr — прямой переход на метку внутри текущего сегмента кода. Модифицируется только регистр eip/ip (use16 или use32) на основе указанного в команде адреса (метки);

far ptr — прямой переход на метку в другом сегменте кода. Адрес перехода задается в виде непосредственного операнда или адреса (метки) и состоит из 16-битного селектора и 16/32-битного смещения, которые загружаются, соответственно, в регистры cs и ip/eip;
word ptr — косвенный переход на метку внутри текущего сегмента кода. Модифицируется только eip/ip. Размер смещения 16 или 32 бит;
dword ptr — косвенный переход на метку в другом сегменте кода. Модифицируются оба регистра, cs и eip/ip. Первое слово/двойное слово этого адреса представляет смещение и загружается в ip/eip; второе/третье слово загружается в cs.
CALL - вызов процедуры. Формат команды: CALL имя_процедуры
RET - возврат из процедуры.

Процедура представляет собой совокупность команд, которая написана один раз, но может быть исполнена по мере необходимости в любом месте программы.

Команда CALL осуществляет функции запоминания адреса возврата и передачи управления процедуре. Она помещает в стек смещение адреса возврата, если процедура определена с артибутом NEAR, и содержимое регистра сегмента команд CS, а затем смещение адреса, если она определена с атрибутом FAR. Процедуры с атрибутом NEAR могут быть вызваны только из того сегмента, в котором они находятся, а с FAR могут быть вызваны и из другого сегмента.

После сохранения адреса возврата команда CALL загружает смещение адреса метки "имя_процедуры" в указатель команд IP (EIP). Если процедура имеет атрибут FAR, то команда CALL загружает также сегментный адрес метки "имя_процедуры" в регистр CS.

Команда RET извлекает из стека адрес возврата. Если процедура имеет атрибут NEAR, то команда RET извлекает из стека одно слово (двойное слово) и загружает его в указатель команд IP (EIP). Если процедура имеет атрибут FAR, то команда RET извлекает из стека два (три) слова: сначала смещение адреса для загрузки в указатель команд IP (EIP), а затем адрес сегмента для загрузки в регистр CS.

Команды условной передачи управления.

Команды условной передачи управления имеют следующий общий формат:
Jx метка_перехода,

где x - модификатор, состоящий их одной или нескольких букв, может принимать следующие значения:

E (equal) – равно,
N (not) – не,
G (greater) – больше,
L (less) – меньше,
A (above) – выше,
B (below) – ниже.

Модификаторы E, N используются для любых типов операндов, G и L – для чисел со знаком, A и B - для чисел со знаком.

Команды условного перехода удобно применять для проверки различных условий, возникающих в ходе выполнения программы. Многие команды формируют признаки результатов своей работы в регистре flags (eflags). Это обстоятельство используется командами условного перехода для работы. Ниже в таблице 3 приведены перечень команд условного перехода, анализируемые ими флаги и соответствующие им логические условия перехода.

Таблица 3 – Перечень команд условного перехода

Команда	Состояние проверяемых флагов	Условие перехода
JA	CF = 0 и ZF = 0	если выше
JAE	CF = 0	если выше или равно
JB	CF = 1	если ниже
JBE	CF = 1 или ZF = 1	если ниже или равно
JC	CF = 1	если перенос
JE	ZF = 1	если равно
JZ	ZF = 1	если 0
JG	ZF = 0 и SF = OF	если больше
JGE	SF = OF	если больше или равно
JL	SF <> OF	если меньше
JLE	ZF=1 или SF <> OF	если меньше или равно
JNA	CF = 1 и ZF = 1	если не выше
JNAE	CF = 1	если не выше или равно
JNB	CF = 0	если не ниже
JNBE	CF=0 и ZF=0	если не ниже или равно
JNC	CF = 0	если нет переноса
JNE	ZF = 0	если не равно
JNG	ZF = 1 или SF <> OF	если не больше
JNGE	SF <> OF	если не больше или равно
JNL	SF = OF	если не меньше
JNLE	ZF=0 и SF=OF	если не меньше или равно
JNO	OF=0	если нет переполнения
JNP	PF = 0	если количество единичных битов результата нечетно (нечетный паритет)
JNS	SF = 0	если знак плюс (знаковый (старший) бит результата равен 0)
JNZ	ZF = 0	если нет нуля
JO	OF = 1	если переполнение
JP	PF = 1	если количество единичных

		битов результата четно (четный паритет)
JPE	PF = 1	то же, что и JP, то есть четный паритет
JPO	PF = 0	то же, что и JNP
JS	SF = 1	если знак минус (знаковый (старший) бит результата равен 1)
JZ	ZF = 1	если ноль
JCXZ	не влияет	если регистр CX=0
JECXZ	не влияет	если регистр ECX=0

Логические условия "больше" и "меньше" относятся к сравнениям целочисленных значений со знаком, а "выше и "ниже" — к сравнениям целочисленных значений без знака. С целью удобства ассемблер допускает несколько различных мнемонических обозначений одной и той же машинной команды условного перехода. Изначально в микропроцессоре i8086 команды условного перехода могли осуществлять только короткие переходы в пределах -128...+127 байт, считая от следующей команды. Начиная с микропроцессора i386, эти команды уже могли выполнять любые переходы в пределах текущего сегмента команд. Для реализации межсегментных переходов необходимо комбинировать команды условного перехода и команду безусловного перехода jmp.

Командам условной передачи управления могут предшествовать любые команды, изменяющие состояние флагов, но обычно они используются совместно с командой сравнения CMP.

Команды управления циклами.

Команды управления циклами обеспечивают условные передачи управления при организации циклов. Каждая команда уменьшает содержимое регистра счетчика CX на 1, а затем использует его новое значение для принятия решения о выполнении или невыполнении перехода.

LOOP - повторять цикл до конца счетчика. Формат команды: **LOOP метка**

Команды уменьшает содержимое регистра счетчика CX на 1 и передает управление оператору метка, если содержимое регистра CX не равно нулю. Команда LOOP завершает выполнение цикла только, если содержимое регистра CX станет равным нулю. Однако во многих приложениях требуются такие циклы, которые должны завершаться при выполнении определенных условий до того, как содержимое регистра CX достигает нуля. Такое альтернативное завершение цикла обеспечивается командами LOOPE (повторить цикл, если равно) и LOOPNE (повторить цикл, если не равно).

LOOPE (LOOPZ) - повторять цикл, пока равно (нуль). Команда уменьшает содержимое регистра CX на 1, а затем осуществляет переход, если содержимое регистра CX не равно 0 и флаг нуля ZF равен единице. Таким

образом повторение цикла завершается, если либо содержимое регистра CX равно нулю, либо флаг нуля ZF равен нулю, либо оба они равны нулю. Обычно команда LOOPE используется для поиска первого ненулевого результата в серии операций.

LOOPNE (LOOPNZ) - повторять цикл, пока не равно (нуль). Команда уменьшает содержимое регистра CX на 1, затем осуществляет переход, если содержимое регистра CX не равно нулю и флаг нуля ZF равен нулю. Таким образом, повторение цикла завершается, если либо содержимое регистра CX равно нулю, либо флаг нуля ZF равен единице, либо будет выполнено и то и другое. Обычно команда LOOPNE используется для поиска первого нулевого результата в серии операций.

Команды прерывания.

Подобно вызову процедуры, прерывание заставляет микропроцессор сохранить в стеке информацию для последующего возврата, а затем исполнить программу обработки прерывания.

Прерывание всегда вызывает косвенный переход к своей программе обработки за счет получения ее адреса из 32-битового вектора прерывания. Сохраняя в стеке адрес прерывания, сохраняют еще и флаги. Прерывания могут быть инициированы внешним устройством системы или специальной командой прерывания из программы. Есть три различные команды прерывания - две команды вызова и одна команда возврата.

INT - Команда прерывания. Формат команды: **INT тип_прерывания**
Тип прерывания это номер, идентифицирующий один из 256 различных векторов, находящихся в памяти.

При исполнении команды INT микропроцессор производит следующие действия:
1. Помещает в стек значение регистра флагов;
2. Обнуляет флаг трассировки OF и флаг включения/выключения прерываний IF;
3. Помещает в стек значение регистра CS;
4. Вычисляет адрес вектора прерывания, умножая тип прерывания на 4;
5. Загружает второе слово вектора прерываний в регистр CS;
6. Помещает в стек значение регистра указателя команд IP;
7. Загружает в регистр указателя команд IP первое слово вектора прерывания.

После команды INT в стеке окажутся значения регистра флагов и регистров CS и IP. Флаги трассировки TF и прерывания IF будут равны нулю. Пара регистров CS и IP будет указывать на начальный адрес программы обработки прерывания. Затем микропроцессор начнет исполнять эту программу.

INTO - команда прерывания по переполнению. Она представляет собой команду условного прерывания. Команда инициирует прерывание лишь тогда, когда флаг переполнения OF равен единице.

IRET - команда возврата после прерывания. Действие команды то же, что и у команды RET для процедуры. Поэтому она исполняется последней при исполнении микропроцессором программы обработки прерывания. Команда IRET извлекает из стека три 16-битовых значения и загружает их в регистр указателя команд IP, регистр CS и регистр флагов.

Команды управления микропроцессором.

Эти команды позволяют управлять работой микропроцессора из программы. Делятся на три группы.

Команды управления флагами.

У микропроцессора есть семь команд, которые позволяют изменять флаг переноса CF, флаг направления DF и флаг прерывания IF.

STC - Команда установки флага переноса, т.е. флаг CF будет равен единице.

CLC - Команда обнуления флага переноса, т.е. флаг CF будет равен нулю.

Они полезны для установки нужного состояния флага CF перед исполнением команд циклического сдвига с флагом переноса RCL и RCR.

CMC - Команда инвертирования флага переноса. Переводит флаг CF в нуль, если он имел состояние единица и наоборот.

STD - Команда установки флага направления, т.е. флаг DF будет равен единице.

CLD – Команда обнуления флага направления, т.е. флаг DF будет равен нулю.

Эти команды используются для указания направления обработки строк. Если флаг направления DF равен нулю, то после каждой операции над строкой значения индексных регистров SI и DI увеличиваются, если флаг DF равен единице, то они уменьшаются.

CLI - Команда обнуления флага прерывания. Обнуляет флаг прерывания IF, что заставляет микропроцессор игнорировать маскируемые прерывания, инициируемые внешними устройствами системы. Однако немаскируемые прерывания обрабатываются. Это прерывание для выдачи сообщений об ошибке в ячейке памяти.

STI - Команда установки флага прерываний. Переводит флаг прерывания IF в состояние единица, что разрешает микропроцессору реагировать на прерывания, инициируемые внешними устройствами.

Команды внешней синхронизации.

Эти команды используются в основном для синхронизации действий микропроцессора с внешними событиями.

HLT - Команда останова. Переводит микропроцессор в состояние останова.

WAIT - Команда ожидания. Переводит микропроцессор на холостой ход.

ESC - Команда "убежать". Заставляет микропроцессор извлечь содержимое указанного в ней операнда и передать его на шину данных, тем

самым она обеспечивает другим микропроцессорам системы возможность получения своих команд из потока команд микропроцессора.

Формат команды: **ESC внешний_код,источник**

Внешний_код это 6-битовый непосредственный операнд, а источник - регистр или переменная.

Команда холостого хода.

NOP - нет операции. Она не действует ни на флаг, ни на регистры, ни на ячейки памяти, а только увеличивает значение указателя команд IP.

Команда NOP удобна при тестировании последовательности команд. Ее можно сделать последней в тестируемой программе и тем самым получить удобное место для остановки трассировки.

9 Лекция №9. Арифметические команды

Цель лекции: изучение команд для выполнения арифметических операций.

Содержание лекции: арифметические команды, команды коррекции.

Команда сложения ADD

ADD ***операнд_1, операнд_2***

операнд_1= операнд_1+ операнд_2

```
ADD   AL, M, BYTE ;содержание регистра + ячейка памяти;
ADD  A, BX ;слово из ячейки памяти (A) + содержание регистра BX;
 ADD  BX,  10 ;содержание регистра BX+10;
ADD  A, 200 ;слово из ячейки памяти (A)+200;
```

ADC ***операнд_1, операнд_2***-команда сложения с учетом флага переноса CF: операнд_1= операнд_1+ операнд_2+значение CF

Команда коррекции результатов сложения

AAA: Коррекция ASCII – формата для сложения.

Корректирует сумму двух ASCII - байтов в регистре AL. Если правые четыре байта регистра AL имеют значения больше 9 или флаг установлен в 1 (AF=1), то команда AAA прибавляет к регистру AH единицу и устанавливает флаги AF и CF. Команда всегда очищает четыре левых бита в регистре AL.

Пример:
```
ADD AL,  BL
    AAA
```

INC операнд - операция инкремента, команда увеличения значения операнда на 1;

DAA – десятичная коррекция для сложения.

Корректирует результат сложения двух BCD десятичных упакованных элементов в регистре AL , т.е. преобразует результат сложения в 2 правильные десятичные цифры в регистре AL.

Если четыре правых байта имеют значение больше 9 или флаг AF = 1, то команда DAA прибавляет 6 к регистру AL и устанавливает флаг AF.

Если AL > 9F или CF= 1, то команда прибавляет 60 H к регистру AL

```
ADD   AL,  DL
DAA
```

Команда вычитания SUB

SUB *операнд_1, операнд_2*

операнд_1= операнд_1- операнд_2

SBB *операнд_1, операнд_2*– команда вычитания с учетом заема

Пример:

```
SUB      BX, DX ; CF = 1
SBB      AX, CX ;результат в AX BX
```

Команда воздействует на флаги

CF, PF , AF , ZF , SF , OF

Команды коррекции результатов вычитания

AAS: Коррекция ASCII – формата для вычитания. Корректирует разность двух ASCII – байтов в регистре AL. Если первые четыре байта имеют значение больше 9 или флаг CF установлен в 1, то команда AAS вычитает 6 из регистра AL и 1 из регистра AH.

Флаги AF и CF при этом устанавливается в 1. Команда всегда очищает левые четыре бита в регистре AL.

DAS: Десятичная коррекция для вычитания.

Корректирует результат вычитания двух BCD (десятичных упакованных) чисел в регистре AL.

Если четыре правых бита имеют значение больше 9 или флаг AF=1, то DAS вычитает 60 H из регистра AL и устанавливает флаги CF

Операнд находится в регистре AL:

```
1) SUB   AL ,  BL
     AAS
2) SUB   AL ,  DL
    DAS
 3) SUB BX,DX
    SBB AX,CX
```

DEC операнд- операция декремента, команда уменьшения значения операнда на 1; DEC сможет использоваться для уменьшения счетчика при организации цикла и уменьшения индексных регистров (операции над строками).

NEG : изменение знака числа.

```
NEG
ADD     AL ,  100
```

Команда умножение MUL

MUL *сомножитель_1*

В команде указывается всего лишь один *операнд-сомножитель*. Второй *операнд –сомножитель 2* задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Варианты размеров сомножителей и размещения второго операнда и результата приведены в таблице 4.

Умножает без знаковое множимое на без знаковый множитель. Левый единичный бит рассматривается как бит данных. Для 8 – битового умножения множимое должно находиться в регистре AL, а множитель возможен в

регистре или в памяти, например MUL CL. Произведение находится в регистре AX. Для 16 – битового умножения множимое должно находиться в регистре AX, а множитель возможен в регистре или в памяти, например MUL BX

Таблица 4

Сомножитель_1	Сомножитель_2	Результат
Байт	al	16 бит в ax :al –мдадшая часть результата;ah-старшая часть езультата
Слово	ax	32 бит в паре dx:ax: ax – млад.часть р-та; dx –старшая часть р-та
Двойное слово	eax	64 бит в паре edx:eax: eax- младшая часть результата; edx-старшая часть результата

IMUL *операнд_1*– умножение числа со знаком, выполняет умножение на знаковый множитель. Левый единичный бит рассматривается как знак минус для отрицательных чисел.

Команды коррекции результатов умножения

AAM: Коррекция ASCII – формата для умножения

Команда AAM используется для коррекции результата умножения двух неупакованных десятичных чисел, которая делит содержимое регистра AL на 10 записывает частное в регистр AH, а остаток в регистр AL.

Команда деления DIV

DIV делитель-команда деления чисел без знака

IDIV делитель-команда деления чисел со знаком

В командах умножения и деления в качестве источника нельзя использовать непосредственный операнд. Делитель может находиться в памяти или в регистре и иметь размер 8,16 или 32 бит. Местонахождение делимого фиксировано и зависит от размера операндов. Варианты местоположения и размеров операндов операции деления показаны в таблица 5.

Таблица 5

Делимое	Делитель	Частное	Остаток
Слово 16 бит в регистре ax	Байт – регистр или ячейка памяти	Байт в регистре al	Байт в регистре ah
32 бит dx-старшая часть ax-младшая часть	16 бит регистр или ячейка памяти	Слово 16 бит в регистре ax	Слово 16 бит в регистре dx
64 бит edx-старшая часть	Двойное слово 32 бит регистр или ячейка	Двойное слово 32 бит в регистре eax	Двойное слово 32 бит в регистре edx

eax-младшая часть	памяти		

Команды коррекции результатов деления

AAD – коррекция ASCII – формата для деления.

Команда используется перед делением неупакованных десятичных чисел в регистре AX. Эта команда корректирует делимое в двоичное значение в регистре AL для последующего двоичного деления, затем умножает содержимое регистра $(AH)\cdot 10 + AH$ после этого AH=0, т.е. очищает регистр AH. AAD преобразует неупакованное делимое в двоичное значение и загружает его в регистр AL.

Пример:

AAD

DIV BL

Выставляются флаги SF, IF, PF

10 Лекция №10. Описание группы логических команд

Цель лекции: изучение команд для выполнения логических операций.

Содержание лекции: команды обработки битов, команды сдвига и циклического сдвига.

Логические команды действуют по правилам формальной логики. Так как логические команды манипулируют битами операндов, то обычно при записи значений таких операндов используют шестнадцатеричную систему счисления. К логическим командам относятся AND (логическое умножение), OR (логическое сложение), XOR (сложение по модулю два), NOT (логическое отрицание), TEST (логической проверки), рисунок 3.

Операндами команд AND, OR, XOR могут быть байты, слова или двойные слова. В этих командах можно сочетать два регистра, регистр с ячейкой памяти или непосредственное значение с регистром или ячейкой памяти.

AND - логическое "И". Формат команды: **AND приемник, источник**

В каждой позиции бита, где оба операнда содержат единицу, операнд приемник также будет содержать единицу. В тех же позициях, где операнды имеют любую другую комбинацию значений, приемник будет содержать нуль.

Состояние флагов после выполнения команды: OF = CF = 0, SF, ZF и PF изменяют свое значение.

OR - логическое "ИЛИ". Формат команды:: **OR приемник, источник**

Команда устанавливает в единицу те биты операнда приемника, в значениях которых хотя бы один из операндов содержит единицу.

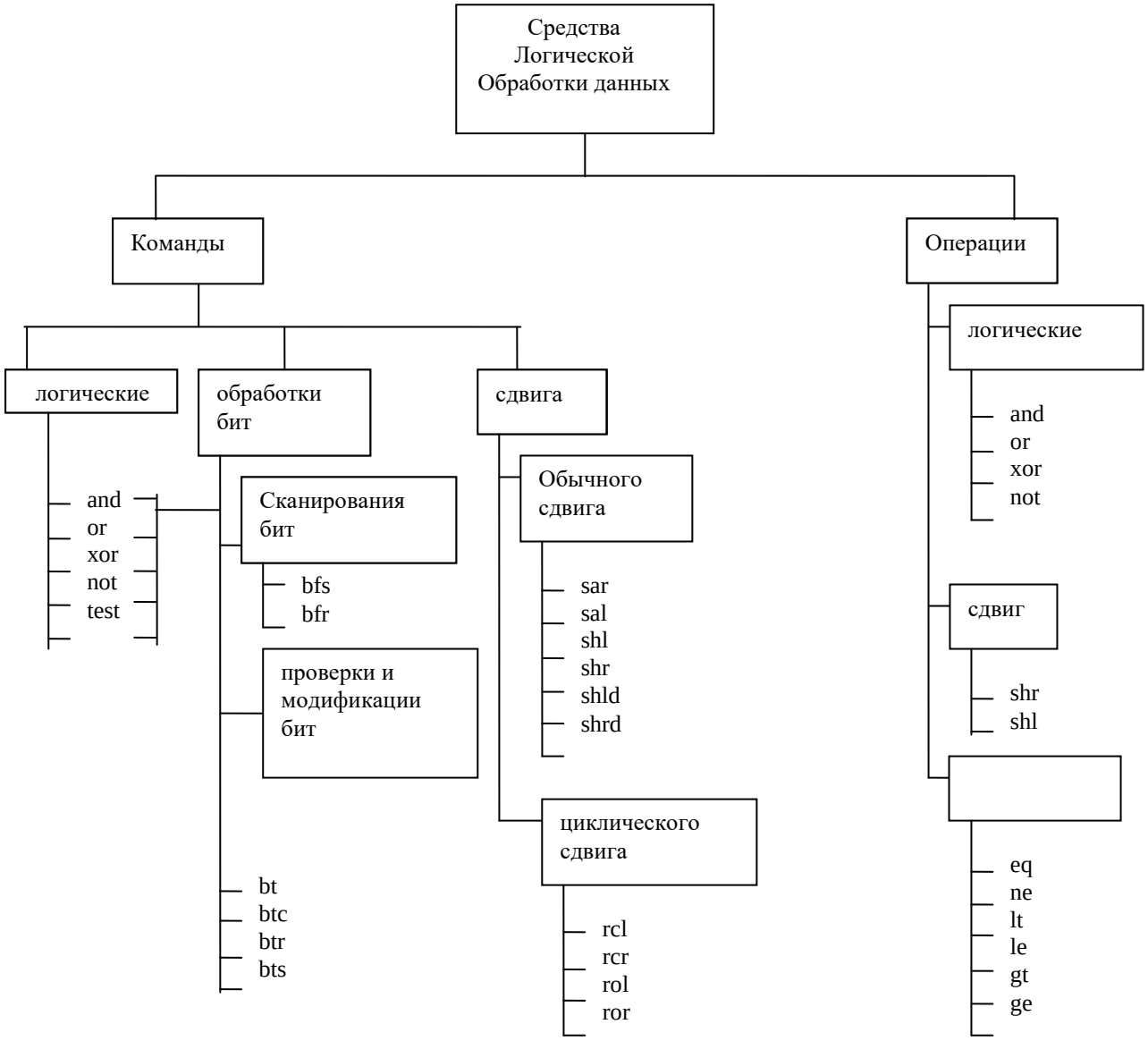

Рисунок 3- Средства микропроцессора для работы с логическими данными

Состояние флагов после выполнения команды: OF = CF = 0, SF, ZF и PF изменяют свое значение.

XOR - логическое "ИСКЛЮЧАЮЩЕЕ ИЛИ". Формат команды:: **XOR приемник, источник**

Команда устанавливает в единицу все те биты приемника, в позициях которых операнды имеют различные значения, т.е. те биты, в позициях которых один из операндов имеет значение нуль, а другой единице. Если оба операнда содержат в данной позиции либо нуль, либо единицу, то команда обнуляет этот бит приемника.

NOT - логическое "НЕ". Формат команды: **NOT операнд**

Команда инвертирует состояние каждого бита регистра или ячейки памяти и ни на какие флаги не воздействует. Таким образом, команда заменяет нуль на единицу, а каждую единицу на нуль.

TEST - команда логической проверки. Формат команды: **TEST приемник, источник**

Команда выполняет операцию AND над операндами, но воздействует только на флаги и не изменяет значения операндов. Состояние флагов после выполнения команды: OF = CF = 0, SF, ZF и PF изменяют свое значение.

Команды сдвига и циклического сдвига.

Команды этой группы обеспечивают манипуляции над отдельными битами операндов и обладают следующими свойствами:

- обрабатывают байт или слово;
- имеют доступ к регистрам или к памяти;
- сдвигают влево или вправо;
- сдвигают логически или динамически.

Линейный сдвиг.

К командам этого типа относятся команды ,осуществляющие сдвиг по следующему алгоритму:

- очередной «выдвигаемый » бит устанавливает флаг cf;
- бит, вводимый в операнд с другого конца, имеет значение 0;
- при сдвиге очередного бита он переходит во флаг cf,при этом значение предыдущего сдвинутого бита *теряется.*

Команды линейного сдвига делятся на два подтипа: команды логического и арифметического линейного сдвига. К командам логического линейного сдвига относятся следующие:

SHL операнд,счетчик_сдвигов – логический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита)вписываются нули;

SHR операнд,счетчик_сдвигов – логический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, пределяемое значением счетчик_сдвигов. Слева (в позицию старшего, знакового бита)вписываются нули.

На рисунке 7 показан принцип работы этих команд.

Команды арифметического линейного сдвига отличаются от команд логического сдвига тем, что они особым образом работают со знаковым разрядом операнда:

SAL операнд,счетчик_сдвигов – арифметический сдвиг влево. Содержимое операнда сдвигается влево на количество битов,определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита)вписываются нули. Команда SAL не сохраняет знака, но устанавливает флаг cf в случае смены знака очередным выдвигаемым битом. В остальном команда SAL полностью аналогична команде SHL.

SAR операнд,счетчик_сдвигов - арифметический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева в операнд вписываются нули. Команда SAR сохраняет знак, восстанавливая его после сдвига каждого очередного бита.

На рисунке 4 показан принцип работы линейного арифметического сдвига.

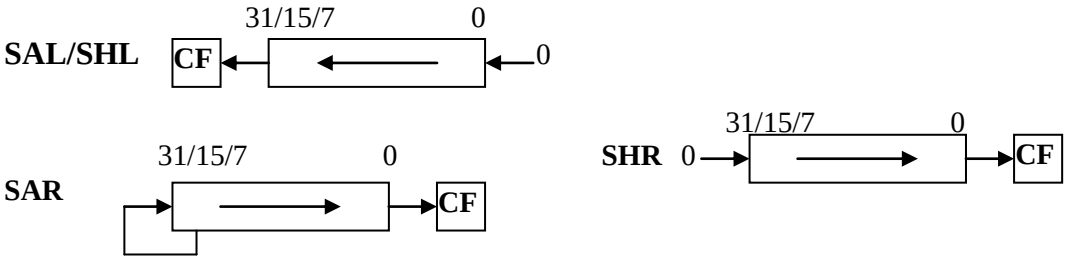

Рисунок 4 - Принцип действия команд

Циклический сдвиг.

К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых бит. Есть два типа команд циклического сдвига:

- команды простого циклического сдвига;

-команды циклического сдвига через флаг переноса cf.

К команды простого циклического сдвига (рисунок 5) относятся:

ROL операнд,счетчик_сдвигов – циклический сдвиг влево. Содержимое операнда сдвигается влево на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые влево биты записываются в тот же операнд справа.

ROR операнд,счетчик_сдвигов - циклический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые вправо биты записываются в тот же операнд слева.

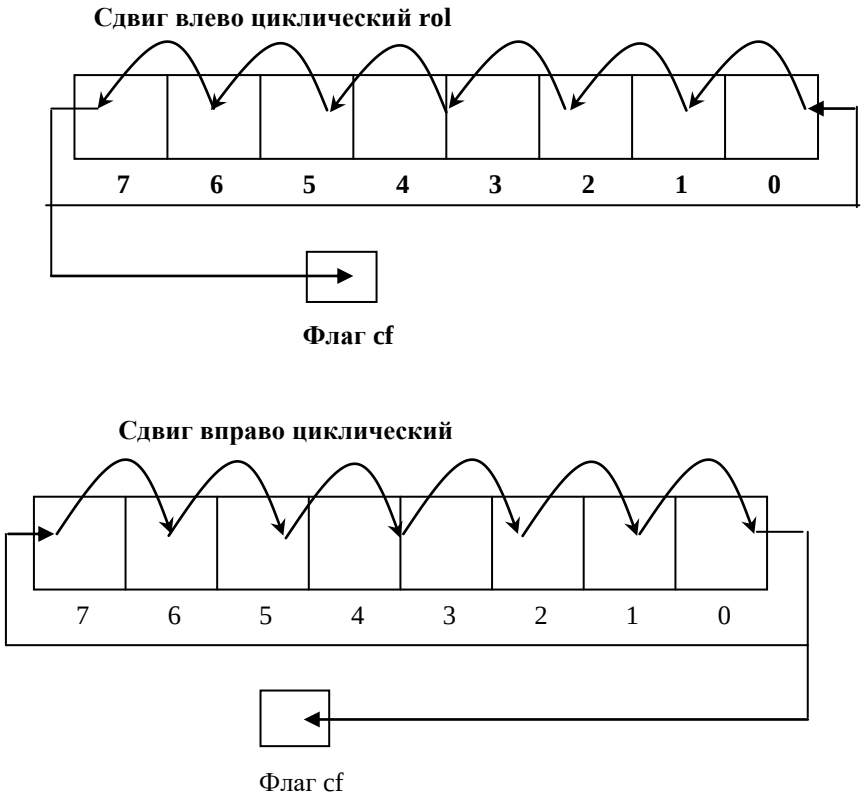

Рисунок 5 - Схема работы команд простого циклического сдвига

К командам циклического сдвига через флаг переноса cf (рисунок 6) относятся:

RCL операнд, счетчик_сдвигов – циклический сдвиг влево через перенос. Содержимое операнда сдвигается влево на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса cf.

RCR операнд,счетчик_сдвигов- циклический сдвиг вправо через перенос. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса cf.

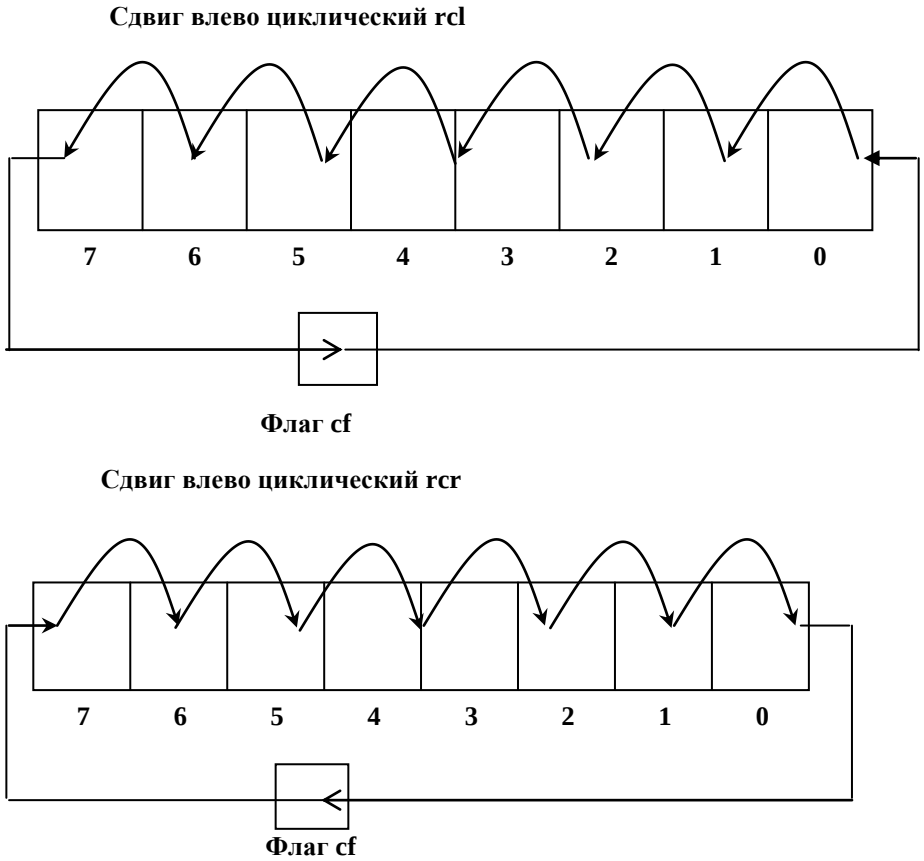

Рисунок 6 - Команды циклического сдвига через флаг переноса cf

11 Лекция №11. Команды обработки строк

Цель лекции: изучение команд, используемых в программе для обработки строк.

Содержание лекции: понятие префиксы повторения, команды пересылки, сравнения, сканирования, загрузки, сохранения строки.

Команды обработки строк выполняют действия над блоками байтов или слов памяти. Длина блока или строки может достигать 64 кб

Команда может за 1 цикл обработать только 1 элемент (байт или слово).

Микропроцессор предполагает, что стр-прм. находится в дополнительном сегменте ES, а стр-ист. в сегменте данных DS. стр-прм. адресуется с помощью регистра DI, стр-ист. с помощью регистра SI. Флаг DF указывает

направление обработки строки. Если DF=0, обработка строки идет слева направо, в сторону увеличения адресов. Если DF=1, обработка строки идет справа налево, адреса уменьшаются.
Установить флаг направления DF можно с помощью команд:
CLD (DF=0); STD (DF=1)

Префиксы повторения.

Префиксы повторения заставляют микропроцессор повторять цепочечную команду. Число повторений извлекается из регистра CX (ECX).
Префикс **REP** - дает указание повторять, пока не обнаружится конец строки, т.е. пока значение регистра CX не станет равным нулю.

Остальные префиксы повторения используют при решении о продолжении или прекращении повторений флаг нуля ZF.

Следовательно, они используются с командами сравнения строк и поиска значения в строке, которые воздействуют на флаг нуля ZF.

Префикс REPE (повторять, пока равно), имеющий синоним REPZ (повторять, пока нуль), повторяет команду, пока флаг нуля ZF равен единице и значение регистра CX не равно нулю. Префикс REPNE (повторять, пока не равно), имющий синоним REPNZ (повторять, пока не нуль), обеспечивает повторение, пока флаг нуля ZF равен нулю и значение регистра CX не равно нулю.

Команды пересылки строки

MOVS ***адрес - приемника, адрес - источника***

MOVSB/MOVSW/MOVSD

Для области, принимающей строку, сегментным регистром является регистр ES, а регистр DI содержит относительный адрес. Для области, передающей строку, сегментным регистром является регистр DS, а регистр SI содержит относительный адрес.
Групповая пересылка состоит из следующих 5 шагов

1).Установить флаг направления DF командами CLD или STD в зависимости от того, будет ли пересылка осуществляться от младших адресов к старшим и наоборот;

2). Загрузить смещение адреса строки-источника в регистр SI;

3). Загрузить смещение адреса строки-приемника в регистр DI;

4). Загрузить счетчик элементов в регистр CX

5). Выполнить команду MOVS с префиксом REP.

Примеры:

1) CLD; DF=0:направление слева направо LEA SI, SOURCE
   LEA DI, ES:DEST
   MOV CX, 100; цикл на 100
   REP MOVS DEST, SOURCE
2) CLD
   LEA SI, SOURCE
   LEA DI, ES:DEST
   MOV CX, 100

REP MOVSB ;групповая пересылка байтов

Команда сравнения строк

CMPS адрес_приемника, адрес_источника

CMPSB/CMPSW/CMPSD

Сравнивают строки любой длинны. Перед командами сравнения строк обычно ставится префикс REPE (повтор строковой операции).

Например: REPE CMPSB DEST, SUORCE.

При использовании префикса REP в регистре CX должны находится значение длины сравниваемых полей.

Команды CMPS сравнивает содержимое одной области памяти, адресуемой регистрами DS:SI с содержимым другой области, адресуемой регистрами ES:DI.

Если флаг DF=0, то сравнение происходит слева направо, адреса в регистрах SI и DI при этом увеличивается после каждого сравнения.

Если флаг DF=1, сравнение происходит справа налево, а адреса в регистрах SI и DI при этом уменьшаются.

Пример: Сравнение, пока не найдется пара несовпадающих элементов.

```
 CLD
 MOV   CX, 100
 REPE  CMPS DEST, SOURCE
JNE    M1; обнаружено несовпадение => M1.
  ....
M1:
```

Команда сканирования строки

SCAS *адрес_приемника*

SCASB/SCASW/SCASD

Выполняют после определенного байта или слова в строке. Для команды SCASB необходимое значение загружается в регистре AL, а для SCASW в регистр AX. Регистровая пара ES:DI указывает на строку в памяти, которая должна быть сканирована. Данные команды обычно используются с префиксом REPE и REPNE. Если флаг DF=0, то операция сканирует память слева направо и увеличивает адрес в регистре DI. Если DF=1, то сканирует справа налево и уменьшает адрес регистра DI.

Пример: Поиск элемента, отличного от ‘_’.

```
CLD
LEA  DI, ES: B_STRING
MOV AL, ‘_’
MOV  CX, 100
REP   SCAS B_STRING
```

Команда SCAS особенно полезна в текстовых редакторах, где программа должна сканировать строки, выполняя поиск знаков пунктуации: точек, запятых и пробелов.

Команда загрузки строки

LODS *адрес_источника*

LODSB/LODSW/LODSD

Загружается из памяти один байт в регистр AL или одно слово в регистр AX.

Адрес памяти определяется регистрами DS:SI. Регистровая пара DS:SI адресуется в памяти байт (для LODSBW).

В зависимости от значения флага DF происходит увеличение или уменьшение значения в регистре SI.

Пример: искать до несовпадения элементов, несовпадение обнаружено – считать элемент в регистре AL.

```
 CLD
LEA    DI, ES; DEST
LEA    SI, SOURSE
MOV   CX, 500
REPE   CMPSB
JNE M1        ; совпадение обнаружено
DEC SI                ; да, подправить регистр SI
LODS  SOURCE    ; считать элемент в AL
...
 M1: , несовпадений нет.
```

Команда сохранения строки

STOS *адрес_источника*

STOSB/STOSW/STOSD

Команда сохраняет байт или слово в памяти. Адрес памяти всегда представляется регистрами ES:DI. При использовании префикса REP операция дублирует значение байта или слова определенное количество раз.

Для STOSB – байт загружается в регистр AL

Для STOSW – слово загружается в регистр AX

Команду удобно использовать, когда необходимо заполнить строку заданным значением.

Пример: сканировать строку DEST длиной в 200 слов в поисках 1-го ненулевого элемента, если такой обнаружен, то он и следующие за ними 5 слов обнуляются.

```
CLD
LEA  DI,ES:DEST
MOV  CX, 200
MOV  AX, 0 ; искомое значение 0
REPNE  SCASW
JNE  M1 ; найдено ненулевое слово
SUB  DI, 2  ; да, подправить регистр DI
MOV  CX, 6
REP      STOS W_STRING  ; 6 слов заполнить 0
...
M1:      нет
```

Лекция №12. Директивы ассемблера

Цель лекции: изучение директив ассемблера и применение их в программах.

Содержание лекции: директива определения сегмента, данных, процедуры, листинга, идентификаторов.

Программа на языке ассемблер состоит из операторов. В качестве операторов могут использоваться команды, макрокоманды и директивы. Директивы управляют процессом ассемблирования, и не генерируют машинных кодов.

Директива определения сегмента.

Программа оформляется в виде отдельных сегментов и может включать сегменты данных, стека, кода и дополнительный. При описании сегмента начало задается директивой SEGNENT, а конец - директивой ENDS. Сегмент описывается по следующему формату:

Имя_сегмента SEGMENT [тип_выравнивания] [тип_объединения] [класс] [тип_размера_сегмента]

...

Имя_сегмента ENDS

Тип_выравнивания сегментов определяет границу начала сегмента и может принимать следующие значения:

BYTE - сегмент начинается с любого адреса

WORD - сегмент начинается с адреса, кратного двум (xxx0b)

DWORD сегмент начинается с адреса, кратного четырем (xx00b)

PAPA - сегмент начинается на границе параграфа с адреса, кратного 16 (xxx0h) - используется по умолчанию

PAGE - сегмент начинается на границе 256-байтовой страницы с адреса, кратного 256 (xx00h).

MEMPAGE — сегмент начинается по адресу, кратному 4 Кбайт (x000h)

Тип_объединения сегментов сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя.

Используются следующие типы объединения:

PRIVATE — сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля – используется по умолчанию.

PUBLIC – соединяет все сегменты с одинаковыми именами, длина объединенного сегмента равна сумме длин объединяемых сегментов,

COMMON - располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Длина объединенного сегмента равна наибольшей из длин объединяемых сегментов.

STACK - определение сегмента стека, заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра SS. Длина сегмента равна сумме объединяемых сегментов. Если сегмент стека создан, а комбинированный тип STACK не используется, программист должен явно загрузить в регистр ss адрес сегмента (подобно тому, как это делается для регистра DS).

MEMORY - вызывает размещение сегмента данных после сегмента кода. Размер сегмента определяется аналогично объединению COMMON.

АТ-ПАРАГРАФ. Данный операнд обеспечивает определение меток и переменных по фиксированным адресам.

Например, для определения адреса дисплейного видеобуфера используется

VIDEO-RAM SEGMENT AT 0B800H

Класс. Данный операнд заключенный в апостофы, используется для группирования сегментов при компоновке. В качестве операнда "класс" можно использовать имена 'STACK', 'CODE', 'DATA'.

Тип размера сегмента. Для процессоров i80386 и выше сегменты могут быть 16 или 32-разрядными. Это влияет, прежде всего, на размер сегмента и порядок формирования физического адреса внутри него. Используются следующие типы размеров сегмента:

USE16 — это означает, что сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных;

USE32 — сегмент будет 32-разрядным. При формирования физического адреса может использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных.

Упрощенные директивы определения сегмента

.CODE [имя] Начало или продолжение сегмента кода

.DATA Начало или продолжение сегмента инициализированных данных. Также используеться для определения данных типа near

.CONST Начало или продолжение сегмента постоянных данных (констант) модуля

.DATA? Начало или продолжение сегмента неинициализированных данных. Также используеться для определения данных типа near

.STACK[размер] Начало или продолжение сегмента стека модуля. Параметр [размер] задает размер стека

.FARDATA[имя] Начало или продолжение сегмента инициализированных данных типа far

.FARDATA[имя] Начало или продолжение сегмента неинициализированных данных типа far

Директива определения процедуры.

Программа может включать в себя одну или несколько процедур. При описании процедуры начало задается директивой PROC, а конец - директивой ENDP. Процедура описывается по следующему формату:

Имя_процедуры PROC [атрибут_дистанции]

. . .

RET

Имя_процедуры ENDP

В качестве атрибута дистанции используются операнды FAR (дальний) и NEAR (близкий).

При использовании атрибута дистанции FAR обращение к процедуре можно осуществлять из другого программного сегмента, а при использовании NEAR - только из сегмента, где оно описывается.

Команда RET выполняет возврат из процедуры. Процедуры могут быть вложенными. Степень вложенности ограничивается только размером сегмента стека.

Директива ASSUME

Директива ASSUME сообщает Ассемблеру какие программные сегменты относятся к программе и загружает адреса для сегментов кода и стека. Загрузку сегментных адресов для дополнительного сегмента и сегмента данных обеспечивает программист.

Директива ASSUME описывается в сегменте кода следующим образом:

ASSUME SS:имя_стек, DS:имя_дан, CS:имя_код, ES:имя_доп.дан

Если программа не использует какой-либо сегмент, то его можно опустить или указать NOTHING.

Например:

ASSUME ss:sseg,ds:dseg,cs:cseg или

ASSUME ss:sseg,ds:dseg,cs:cseg,es:nothing

Кроме того с помощью NOTHING можно отменить предыдущее назначение сегментного регистра.

Директива завершения программы END

Директива END сообщает Ассемблеру где завершить трансляцию и описывается следующим образом:

END [метка_входа]

Метка_входа должна присутствовать только в главном программном модуле, в остальных не обязательна. В качестве метки_входа может использоваться имя основной процедуры, либо метка начала программы.

Директивы определения данных.

Для определения данных используются следующие директивы:

DB - определить байт,

DW - определить слово,

DD - определить двойное слово,

DQ - определить восемь байт или четыре слова,

DT - определить десять байт.

Формат директивы определения данных имеет вид:

[имя] Dn выражение

В качестве выражения могут использоваться:

- константа;
- таблица, массив или строка;
- символьная строка.

Константа.

С помощью директив определения данных можно:

- задать переменную в виде константы: **A DB 10** ;
- занять место в памяти, не указывая начального значения: **A DB ?**

Если переменная описанна с помощью директивы DD

A DD 10203040H ,

а нам нужен определенный байт, то к нему можно обратиться следующим образом:

```
mov   AL, byte ptr A          ; AL=40h
mov   AL, byte prt A+2  ; AL=20h
```

для обращения к определенному слову:

```
mov   AX ,word prt A          ; AX=3040h
mov   AX, word prt A+2 ; AX=1020h
```

С помощью директивы DW в памяти можно сохраниь адрес смещения (указатель) какой-либо метки или процедуры:

```
adr_near     DW   main
```

Полный адрес метки или процедуры (вектор) можно записать в память с помощью директивы DD:

```
adr_ far              DD    main
```

Таблица.

При описании таблицы элементы таблицы записываются через запятую:

```
tab     DB    10, 20, 30, 40, 50
```

Можно просто занять место в памяти для таблицы:

```
tab     DB    5 dup (?)
```

Здесь каждый элемент таблицы имеет длину 1 байт.

```
tab     DW    5 dup (?)
```

В этом случае для таблицы резервируется 10 байт, для каждого элемента 2 байта или слово. Оператор DUP обозначает дублировать. Следует помнить, что независимо от размера элемента обращение к элементам таблицы идет по младшему байту элемента.

Например:

```
mas   DB     1122h, 3344h, 5566h, 7788h ;
```

здесь адрес нулевого слова (счет начинается с нуля) определяется по адресу байта со значением 22h

```
mas     - адрес байта со значением 22h
mas+1 - адрес байта со значением 11h
mas+2 - адрес первого слова массива
mas+4 - адрес второго слова массива
```

Символьная строка.

С помощью директивы DB в памяти можно хранить любой текст. Любые символы, заключенные в апострофы, являются символьной строкой. В память каждый символ будет записан в ASCII-коде.

Например: STR DB 'символьная строка'.

Директивы определения идентификаторов.

К директивам определения идентификаторов относятся директивы **EQU** и **=.**

Эти директивы присваивают имена данным и в памяти места не занимают. Используется следующий формат:

ИМЯ EQU выражение

ИМЯ = числовое_выражение

Отличие директивы EQU от = в том, что с помощью директивы = можно поменять числовое_выражение.

С помощью директивы EQU можно присвоить имя константе, регистру, комбинации адресов, определить синоним.

Директива внешних ссылок.

К директивам внешних ссылок относятся директивы PUBLIC, EXTRN и INCLUDE. Директива PUBLIC указывает, что заданный идентификатор доступен из других программных модулей. В качестве идентификатора может использоватся переменная, метка, константа. Директива имеет формат: **PUBLIC идентификатор [,..]**

Например

```
publicA
 dseg  segment
A      DW    1020h
dseg   ends
```

Директива **EXTRN** обьявляет, что в данном программном модуле используются имена, описанные в других программных модулях. Формат директивы: **EXTRN имя : тип [,..]**

В качестве имени используются переменные, метки, константы. Тип элемента зависит от того как определяется имя.

Если имя является переменной и описывается в сегменте данных, то тип принимает значение WORD, DWORD, BYTE.

Если имя является меткой, то тип принимает значение FAR или NEAR.

Если имя является константой и описывается с помощью директив EQU или = ,то тип принимает значение ABS.

Директива **INCLUDE** вставляет во время трансляции в программу текст файла, указанного в директиве. Формат директивы: **INCLUDE имя_файла**

Директивы управления листингом

К ним относятся директивы PAGE, TITLE и SUBTTL. Директива PAGE задает размер страницы листинга. Формат директивы PAGE имеет вид:

PAGE [число_строк] [,число_столбцов]

Количество строк может изменяться в пределах от 10 до 255.

Столбец указывает количество символов в строке и изменяется в пределах от 59 до 255. По умолчанию устанавливаются размеры страницы 66 строк по 80 символов в строке.

Директива PAGE без операндов делает прогон листа.

Директива TITLE вверху на каждой странице печатает заголовок программы. Формат директивы: TITLE текст

Максимальная длина текста - 60 символов.

Директива SUBTTL печатает подзаголовок на следующей строке после заголовка, имеет формат:

SUBTTL текст

Лекция №13. Организация программ. Макроопределения

Цель лекции: изучение макросредств языка ассемблер.

Содержание лекции: основные понятия, макроопределение и макрокоманда.

Макросредства языка Ассемблера IBM PC являются одним из самых мощных средств языка Ассемблера, который еще называют макроассемблером за способность оперировать с набором команд как с одной командой. Таким образом можно обобщить и сохранить некоторые процедуры несколько в другом виде - в виде макросов.

Использование макросредств позволяет:

- сделать программу более понятной - за счет применения макрокоманд с параметрами; .
- упростить и сократить исходный текст программы;
- уменьшить число возможных ошибок кодирования - за счет использования уже отлаженных макрокоманд;
- динамически изменять программу за счет изменения входных параметров;
- использовать директивы условного ассемблирования и управления листингом;
- сделать свою библиотеку макроопределений;
- существенно ускорить выполнение программы за счет отсутствия накладных расходов на вызов процедур - правда, при этом может возрасти длина программы;
- воспользоваться уже разработанными библиотеками макроопределений (часто они снабжаются расширением имени файла **inc** — include) и вставлять их в свою программу, что позволяет ускорить программирование на Ассемблере.

Использование макросов имеет и свои недостатки:

- макросы хранятся в виде исходных файлов и при подключении к программе нуждаются в ассемблировании и если их много, то может существенно увеличиться время ассемблирования;
- макросы могут легко скрывать результаты работы команд.

С макросами широко работает язык С на уровне описания функций с помощью препроцессорной директивы #define и язык С++ - на уровне inline - функций. Но эти языки накладывают на макросы достаточно существенные ограничения, чего, конечно, нет в Ассемблере. **Макросредства языка Ассемблера IBM PC включают в себя три составляющие:**

1.Макроопределение (макрос). Это - набор команд, содержащий описание какого-то действия или алгоритма. Любую ассемблерную процедуру в принципе можно сделать макросом. ***Макроопределение должно находиться в самом начале программы, до определения сегментов.*** Макрос имеет следующую структуру:

ИмяМакроса MACRO [ФормальныеПараметры]
; тело макроса

Endm

2.Макрокоманда - краткая ссылка на макроопределение (вызов макроса):

ИмяМакроса [ФактическиеПараметры]

3.Макрорасширение (макроподстановка, макровставка) - вставка вместо макрокоманды макроса с заменой формальных параметров фактическими, если они есть.

Макроопределение может быть простое и вложенное, т.е. содержать в себе другое макроопределение. Уровень вложенности макроопределений может быть любым. Из одного макроса можно вызывать другие макросы. Команды в макросе могут быть любые.

Организация многомодульных программ.

При разработке сложных программных комплексов бывает удобно часть процедуры выделить в отдельный исходный модуль, транслируемый самостоятельно.

После трансляции объектные модули этих процедур могут войти в состав объектной библиотеки, подсоединяемой к основной программе на этапе компоновки. Структура такого программного комплекса может выглядеть следующим образом:

```
; MAIN.ASM -файл с главной процедурой
text    segment  public ‘code’`
        extrn    subpr: proc      ; subpr - внешняя ссылка
mymain  proc
        …
        call     subpr            ; Прямой ближний вызов
        …
mymain  endp
text    ends
        end      mymain           ; mymain - точка входа в главную
                                  ; процедуру
;MYSUB.ASM - файл с подпрограммой
text    segment  public  ` code `
        public   subpr            ; subproc - процедура  " общего
пользования"
subpr   proc     near             ; Ближняя процедура
        …
        ret
subpr   endp
text    ends
        end                       ;без точки входа
```

Программные сегменты обоих исходных модулей имеют одно имя **text,** что обеспечивает их слияние компоновщиком в единый сегмент. Тип объединения **PUBLIC** определяет способ слияния - конкатенацию, т.е. подсоединение второго модуля к концу первого (так же действует описатель **STACK**, используемый в сегментах стеков, в то время как тип объединения **COMMON** требует от компоновщика наложения одноименных сегментов друг на друга).

Директива EXTERN в главном модуле определяет, что символьное обозначение subpr является внешней ссылкой и представляет собой имя процедуры. Внешняя ссылка будет разрешена на этапе компоновки.

Процедуры вызываемые из других программных модулей, должны быть определены как процедуры "общего пользования" с помощью директивы PUBLIC. В этом случае их имена записываются транслятором в объектный модуль, что позволяет компоновщику разрешить ссылки на них.

В приведенном примере предполагается, что программные модули после слияния образуют один сегмент команд, т.е. что их суммарный размер (в машинных кодах) не превышает 64 Кбайт. Поэтому процедура-подпрограмма объявлена ближней (near), а в главной процедуре использована команда прямого ближнего вызова.

Модуль с подпрограммой заканчивается директивой завершения трансляции **END** без параметра - в многомодульных программных комплексах только тот модуль, который содержит главную процедуру, оканчивается директивой **END** с указанием входной точки этой процедуры.

Файлы MAIN.ASM и MYSUB.ASM с исходными текстами компонентов комплекса следует порознь оттранслировать, а затем скомпоновать в единый загрузочный модуль:

```
MASM MAIN, MAIN, MAIN;
MASM MYSUB, MYSUB, MYSUB;
LINK MAIN + MYSUB, PROG;
```

В результате трансляции будут получены файлы MAIN.OBJ, MAIN.LST, MYSUB.OBJ и MYSUB.LST; компоновщик объединит модули MAIN.OBJ и MYSUB.OBJ и образует загрузочный (выполнимый модуль) PROG. EXE.

Другой вариант процедуры разработки многомодульного программного комплекса - помещение объектного модуля MYSUB. OBJ в библиотеку объектных модулей с последующим извлечением его оттуда компоновщиком. Библиотекой целесообразно пользоваться при наличии большого числа объектных модулей, используемых в различных программных комплексах. В этом случае после трансляции файла MYSUB.ASM объектный файл MYSUB.OBJ записывается в создаваемую пользователем библиотеку объектных файлов. Включение объектных файлов с подпрограммами во вновь создаваемую объектную библиотеку выполняется следующим образом:

```
LIB MYOBJ.LIB + MYSUB.OBJ, MYOBJ.LST;
```

В этой команде LIB - имя программы библиотекаря, MYSUB.OBJ - имя помещаемого в библиотеку объектного файла, а MYOBJ.LST - имя создаваемого файла с каталогом библиотеки. Знак ‘+’ перед именем модуля обозначает, что этот модуль надо добавить в библиотеку (знак ‘–‘ удаляет данный модуль). В файл MYOBJ.LST будет помещено оглавление библиотеки.

Все расширения, кроме расширения файла с каталогом, можно опустить, так как по умолчанию предполагаются именно указанные в примере расширения.

Если объектная библиотека уже имеется, и мы хотим добавить в нее вновь созданные объектные модули, в строке вызова библиотекаря надо в качестве последнего параметра еще раз указать имя библиотеки:

LIB MYOBJ + NEW1 + NEW2, MYOBJ.LST, MYOBJ

Здесь первый параметр определяет имя исходной библиотеки, а последний - имя создаваемой библиотеки, расширенной за счет добавления новых модулей. Это имя может совпадать с именем старой библиотеки, но может и отличатся от него.

При наличии объектной библиотеки в строке вызова компоновщика в качестве четвертого параметра команды следует указать имя библиотеки:

LINK/ CO MAIN.OBJ, PROG.EXE, PROG.MAP, MYOBJ.LIB

В приведенном примере MAIN.OBJ - объектный файл с основной программой, PROG.EXE - результирующий выполняемый модуль, PROG.MAP - карта выполнимого модуля, а MYOBJ.LIB - объектная библиотека со требуемыми подпрограммами. Как и в предыдущих примерах, все расширения можно опустить, так они предполагаются или назначаются по умолчанию. Карта выполнимого модуля обычно не нужна (в ней мало полезной информации), а имя загрузочного модуля по умолчанию совпадает с именем объектного (при разных расширениях). Поэтому приведенную команду можно упростить:

LINK MAIN, PROG , , MYOBJ.LIB

В результате выполнения этой команды компоновщик создаст выполнимый модуль PROG.EXE

Если суммарный размер главной программы и подпрограмм (в машинных кодах) превышает 64Кбайт, загрузочный модуль может быть многосегментным. Структура такого программного комплекса может выглядеть следующим образом:

```
; MAIN. ASM - файл с главной процедурой
text1   segment  ‘code’
        extern   subpr:proc
text1   ends
text    segment  ‘code’
mymain  proc
        …
        call     far ptr subpr
        …
mymain  ends
text    ends
        end mymain
;MYSUB.ASM - файл с подпрограммой
text1   segment  ‘code’
        public   subpr
subpr   proc     far
```

```
        …
subpr   endp
text1   ends
        end
```

Поскольку сегменты **text (**с главной программой) и **text1** (с подпрограммой) имеют разные имена, они не будут объеденены компоновщиком в один сегмент. В исходном модуле, содержащем вызов подпрограммы, она должна быть описана директивой **EXTRN.** Однако это описание должно находится в сегменте, одноименным с сегментом подпрограммы. Поэтому в модуль **MAIN.ASM** включен пустой пока что сегмент **text1,** содержащий лишь одну строку - объявление вызываемой процедуры **EXTRN SUBPR: PROC.** Далее следует основной сегмент команды **text** с главной программой. В ней использован прямой дальний вызов; процедура-подпрограмма, в свою очередь, атрибутом FAR объявлена дальней. Трансляция и компоновка выполняется так же, как и в случае многомодульной односегментной программы.

Макрокоманды

Программы написанные на языке ассемблера часто содержат повторяющиеся участки текста с одинаковой структурой. Такой участок текста можно оформить в виде макроопределения и списка фактических аргументов, что приводит к генерации всего требуемого текста, называемого макрорасширением. Варьируя фактические аргументы, можно сохраняя неизменной структуру макрорасширения, изменить отдельные его элементы.

Макроопределение должно начинаться строкой с именем макроопределения и директивой MACRO, в поле аргументов которого указывается список формальных аргументов. Заканчивается макроопределение директивой ENDM.

Пусть в программе требуется неоднократно сохранять в стеке содержимое трех регистров, но в каждом конкретном случае номера регистров и их порядок отличаются в виде макроопределения:

```
psh     macro   a, b, c
        push    a
        push    b
        push    c
        endm
```

Появление в исходном тексте программы строки

```
        psh     AX, BX, CX
```

приведет к генерации следующего фрагмента текста

```
        push    AX
        push    BX
        push    CX
```

Если же в исходном тексте имеется строка

```
psh     DX, ES, BP
```

то соответствующее макрорасширение будет иметь вид:

```
        push    DX
        push    ES
        push    BP
```

В качестве фактических аргументов могут выступать любые обозначения ассемблера, допустимые для данной команды. В частности, макровызов

```
psh mem,  [BX], ES:[17]h
```

приведет к следующему макрорасширению:

```
        push    mem
        push    [BX]
        push    ES:[17]
```

Если какие-то строки макроопределения должны быть помещены (например, с целью организации циклов), то обозначения меток должны быть объявлены локальными с помощью оператора LOCAL. В этом случае ассемблер, генерируя макрорасширения, будет создавать собственные расширения меток, не повторяющиеся при повторных вызовах одной и той же макрокоманды:

```
delay   macro
        local   point
        mov     CX, 20000
point:  loop    point
        endm
```

Текст макроопределения можно включить непосредственно в текст программы, однако в тех случаях, когда макрокоманды описывают некоторые стандартные процедуры широкого назначения, например программную задержку или вывод на экран строки текста, целесообразно тексты макроопределений поместить в макробиблиотеку.

Макробиблиотека представляет собой файл с текстами макроопределений. Макроопределения записываются в этот файл точно в таком же виде, как и в текст программы. Ниже приведен текст файла макробиблиотеки с произвольным именем **MYMACRO.MAC**, содержащей две макрокоманды.

```
; Макрокоманда  outprog завершения программы
outprog macro
        mov     AX, 4C00h
        int     21h
        endm
; Макрокоманда delay  небольшой программной задержки
delay   macro
        local   point
        mov     CX, 20000
point:  loop    point
        endm
```

Функционально макроопределения похожи на процедуры. И те, и другие достаточно один раз где-то описать, а затем вызывать их специальным образом. На этом их сходство заканчивается, и начинаются различия, которые в зависимости от целевой установки можно рассматривать и как достоинства и как недостатки:

- в отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с набором фактических параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды. Процедуры в этом отношении объекты менее гибки;
- при каждом вызове макрокоманды ее текст в виде макрорасширения вставляется в программу.
  При вызове процедуры микропроцессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре. Код в этом случае получается более компактным, хотя быстродействие несколько снижается за счет необходимости осуществления переходов.

Макродирективы

С помощью макросредств ассемблера можно не только частично изменять входящие в макроопределение строки, но и модифицировать сам набор этих строк и даже порядок их следования. Сделать это можно с помощью набора макродирективов (далее — просто директив). Их можно разделить на две группы:

- директивы повторения WHILE, REPT, IRP и IRPC. Директивы этой группы предназначены для создания макросов, содержащих несколько идущих подряд одинаковых последовательностей строк. При этом возможна частичная модификация этих строк.

- директивы управления процессом генерации макрорасширения EXITM и GOTO. Они предназначены для управления процессом формирования макрорасширения из набора строк соответствующего макроопределения. С помощью этих директив можно как исключать отдельные строки из макрорасширения, так и вовсе прекращать процесс генерации. Директивы EXITM и GOTO обычно используются вместе с условными директивами компиляции, поэтому они будут рассмотрены вместе с ними.

Директивы WHILE и REPT

Директивы WHILE и REPT применяют для повторения определенное количество раз некоторой последовательности строк. Эти директивы имеют следующий синтаксис:

```
WHILE      константное_выражение
последовательность_строк
ENDM
REPTконстантное_выражение
```

последовательность строк
ENDM

Следующие две директивы, IRP и IRPC, делают этот процесс более гибким, позволяя модифицировать на каждой итерации некоторые элементы в последовательность_строк.

Директива IRP

Директива **IRP** имеет следующий синтаксис:

IRP формальный_аргумент,
<строка_символов_1,...,строка_символов_N>
последовательность_строк
ENDM

Действие данной директивы заключается в том, что она повторяет последовательность_строк N раз, то есть столько раз, сколько *строк_символов* заключено в угловые скобки во втором операнде директивы IRP. Повторение *последовательности_строк* сопровождается заменой в ней *формального_аргумента* строкой символов из второго операнда. Так, при первой генерации *последовательности_строк формальный_аргумент* в них заменяется на *строка_символов_1*.
Если есть *строка_символов_2*, то это приводит к генерации второй копии *последовательности_строк*, в которой *формальный_аргумент* заменяется на *строка_символов_2*. Эти действия продолжаются до *строка_символов_N* включительно.

К примеру, рассмотрим результат определения в программе следующей конструкции:

```
irp     ini,<1,2,3,4,5>
                db      ini
                endm
```

Макрогенератором будет сгенерировано следующее макрорасширение:

```
db      1
        db      2
        db      3
        db      4
        db      5
```

Директива IRPC

Директива **IRPC** имеет следующий синтаксис:

IRPC формальный_аргумент,строка_символов
последовательность строк
ENDM

Действие данной директивы подобно IRP, но отличается тем, что она на каждой очередной итерации заменяет *формальный_аргумент* очередным символом из *строка_символов*.

Понятно, что количество повторений *последовательность_строк* будет определяться количеством символов в *строка_символов*. К примеру:

```
irpc   rg,
       push  rg&x
       endm
```

В процессе макрогенерации эта директива развернется в следующую последовательность строк:

```
push   ax
       push  bx
       push  cx
       push  dx
```

Директивы условной компиляции

Директива **EXITM** не имеет операндов, и ее действие заключается в том, что она немедленно прекращает процесс генерации макрорасширения, начиная с того места, где она встретилась в макроопределении.

Директива **GOTO *имя_метки*** переводит процесс генерации макроопределения в другое место, прекращая тем самым последовательное разворачивание строк макроопределения. *Метка.*

Директивы компиляции по условию

Данные директивы предназначены для организации выборочной трансляции фрагментов программного кода. Такая выборочная компиляция означает, что в макрорасширение включаются не все строки макроопределения, а только те, которые удовлетворяют определенным условиям. То, какие конкретно условия должны быть проверены, определяется типом условной директивы.

Всего имеется 10 типов условных директив компиляции:

- Директивы IF и IFE — условная трансляция *по результату вычисления логического выражения*.

- Директивы IFDEF и IFNDEF — условная трансляция *по факту определения символического имени*.

- Директивы IFB и IFNB — условная трансляция *по факту определения фактического аргумента при вызове макрокоманды*.

- Директивы IFIDN, IFIDNI, IFDIF и IFDIFI — условная трансляция *по результату сравнения строк символов*.

Условные директивы компиляции имеют общий синтаксис и применяются в составе следующей синтаксической конструкции:

```
IFxxx логическое_выражение_или_аргументы
       фрагмент_программы_1
              ELSE
       фрагмент_программы_2
ENDIF
```

Лекция №14. Программирование в среде MS DOS
Цель лекции: изучить основы программирования для MS DOS.
Содержание лекции: функции DOS, функции BIOS, системные средства ввода данных с клавиатуры.

Обращение к системным средствам из прикладной программы

Программа, написанная на ассемблер, так же как и программа, написанная на любом другом языке программирования, выполняется при помощи операционной системы. Операционная система выделяет области памяти для программы, загружает ее, передает ее управление и обеспечивает взаимодействие программы с устройствами ввода-вывода, файловыми системами и другими программами. Обращение к функциям **DOS** и **BIOS** осуществляется с помощью программных прерываний (команда INT).

Система прерываний машин типа IBM PC в принципе не отличается от любой другой системы векторизованных прерываний. Самое начало оперативной памяти от адреса 0000h до03FFh отводится под векторы прерываний - четырехбайтовые области, в которых хранятся адреса программ обработки прерываний (ПОП). В два старшие байта каждого вектора записывается сегментный адрес ПОП, в два младшие - относительный адрес точки входа в ПОП в сегменте. Векторы, как и соответствующие им прерывания, имеют номера, называемые типами, причем вектор с номером 0 (вектор типа 0) располагается начиная с адреса 0, вектор типа 1 - с адреса 4, вектор типа 2 с адреса 8 и т.д. Вектор с номером N занимает, таким образом, байты памяти от **N*4** до **N*4+3.** Всего в выделенные под векторы области памяти помещается 256 векторов.

Обращение из прикладной программы к системным функциям осуществляется единообразно. В регистр AH засылается номер функции (не путать с типом прерывания!), в другие регистры - исходные данные, необходимые для выполнения конкретной системной программы. После этого выполняется команда **INT** с числовым аргументом, указывающим тип (номер) прерывания, например, **INT 21h.**

Большинство функций **DOS** и многие функции **BIOS** возвращают в флаге переноса **CF** код завершения. Если функция выполнилась успешно, **CF=0,** в случае любой ошибки **CF=1.** В последнем случае в одном из регистров (чаще всего в **AX**) возвращается еще и код ошибки. Таким образом, типичная процедура обращения к системным средствам выглядит следующим образом:

```
        mov     AH, func        ; func - номер функции
    ; Заполнение тех или иных регистров (AL, BX, ES, BP и др.)
    ; параметрами, необходимыми для выполнения данной
функции
        …
int     21h             ;Переход в MS-DOS
jc      error           ; Строка выполняется сразу
                        ; после возврата из DOS
```

```
                ; Продолжение программы
                 …
error    cmp     AX,1            ;Анализ кода завершения
         je      err1
         cmp     AX2
         je      err2
         …
```

Аналогично вызываются и функции BIOS.

Системные средства ввода данных с клавиатуры

Операционная система предоставляет несколько способов ввода данных с клавиатуры:

- обращение клавиатуре, как к файлу, с помощью прерывания DOS INT 21h с функцией 3Fh;
- использование группы функции DOS INT 21h из диапазона 1…Ch, обеспечивающих посимвольный ввод с клавиатуры в разных режимах;
- посимвольный ввод путем обращения в обход DOS непосредственно к драйверу BIOS с помощью прерывания INT 16h.

Ввод с клавиатуры средствами файловой системы (INT 21h функция 3Fh) осуществляется точно так же, как и чтение из файла. Обычно используется предопределенный дескриптор 0, закрепленный за стандартным устройством ввода (по умолчанию за клавиатурой). Число вводимых символов указывается в регистре CX, однако ввод завершается лишь после того, как нажата клавиша <Enter>, независимо от того сколько фактически введено символов. Поэтому при вводе строк с клавиатуры нет необходимости заранее задавать их длину, достаточно загрузить в регистр CX максимальную длину строки, например 80 байт. В любом случае в регистре AX возвращается число реально введенных байтов, при этом учитывается так же и два байта (0Ah и 0Dh), поступающие во входной буфер при нажатии клавиши <Enter>.

Второй способ получения данных с клавиатуры в программу, с помощью функции DOS из диапазона 1…Ch обеспечивает более разнообразные возможности. Для ввода с клавиатуры можно использовать 7 функций прерывания INT 21h:

01h – вывод символа с эхом;

06h – прямой ввод – вывод через консоль;

07h – ввод символа без эха и без отработки Ctrl/C;

08h –ввод символа без эха и с отработкой Ctrl/C;

0Ah – буферизированный ввод строки с эхом;

0Bh - проверка состояния стандартного устройства

0Ch – сброс входного буфера и ввод.

Функции 01h, 06h, 07h и 08h при каждом вызове вводят в программу один символ из кольцевого буфера ввода; при необходимости ввести группу символов (строку) функции следует использовать в цикле. Различаются эти функции наличием или отсутствием эха, а так же реакцией на ввод с

клавиатуры сочетания < Ctrl> /C. Функции 01h и 0Ah отображают вводимые символы на экране (эхо); функции 07h и 08h этого не делают, что дает возможность вводить данные тайком от окружающих (например, пароль или ключ). Второе важное различие описываемых функций касается их реакции на ввод сочетания < Ctrl> /C. При выполнении функции 01h и 08h DOS проверяет каждый введенный символ и, обнаружив во входном потоке код < Ctrl> /C (03 h), аварийно завершает программу. Функции же 06h и 07h пропускают код < Ctrl> /C в программу, не инициируя по нему никаких специальных действий. Такой метод ввода используется прикладными программами, если перед завершением в них должны быть выполнены определенные программные действия (сброс буферов на диск, модификация файлов и проч.). Аварийное завершение такой программы средствами DOS по коду < Ctrl> /C могло бы привести к нарушению ее работоспособности.

Функция 0Ah передает в буфер пользователя строку, введенную с клавиатуры; строка должна заканчиваться нажатием клавиши <Enter>. Длина строки может достигать 254 символов. Вводимые символы отображаются на экране; при вводе < Ctrl> /C происходит аварийное завершение программы.

Функция 0Bh позволяет проверить наличие в кольцевом буфере ввода ожидаемых символов. При обнаружении символов программа должна извлечь их из буфера одной из функции ввода; если символов нет, программа может продолжить выполнение.

Функция 0Bh чувствительна к < Ctrl> /C. Функция 0Ch служит для организации ввода с предварительной очисткой кольцевого буфера.

Все функции, кроме 0Ch, вводят в программу наиболее старый из скопившихся в кольцевом буфере ввода символов. Функция 0Ch сначала очищает кольцевой буфер и лишь затем ожидает ввода символа с клавиатуры. В результате коды всех ранее нажатых (по предположению – случайно) клавиш теряются. При этом режим ввода (с эхом или без него и т. д.) определяется тем, какая именно функция ввода (01h, 07h, 08h или 0Ah) реализуется “ внутри “ функции 0Ch.

Функции 01h, 07h, 08h и 0Ah являются синхронными, т.е при отсутствии символа в кольцевом буфере ждут его ввода. Функция 06h позволяет определить состояние кольцевого буфера и при наличии в нем кода извлечь этот код и обработать его, а при отсутствии - продолжить выполнение программы.

Функции 01h, 06h, 07h и 08h позволяет вводить в программу расширенные коды ASC II. Для этого обнаружив, что введенный код ASC II равен нулю, следует выполнить функцию повторно. Это дает возможность управления прикладными программами с помощью функциональных клавиш, а так же сочетаний <Alt>/ цифра, <Alt>/ буква и др.

3.Работа с клавиатурой на уровне BIOS (INT 16h) позволяет считывать двухбайтовые коды, поступающие в кольцевой буфер ввода (код ASC II + скен-код) и анализировать слово флагов клавиатуры(нажатие клавиш <Shift>, <Caps Lock> и др.).

Для ввода используются следующие функции прерывания INT 16h:

00h – чтение двухбайтового кода из входного буфера;

01h – чтение состояния клавиатуры и двухбайтового кода без извлечения его из буфера

02h – чтение флагов клавиатуры.

Функция 00h позволяет в одном действии получить полный двухбайтовый код нажатой клавиши или комбинации клавиш, из которого в частности, можно извлечь скен-код (некоторые программы идентифицируют нажатые клавиши не по кодам ASCII, а по их скен-кодам), а также получить значащую часть расширенного кода ASCII (при нажатии например функциональных клавиш). Функция 00h является синхронной: при выполнении программа останавливается в ожидании нажатия клавиши.

Функция 01h относится к числу асинхронных: определив состояние клавиатуры (точнее буфера ввода), она возвращает управление программе. Состояние буфера возвращается в флаге ZF: если в буфере имеются ожидающие ввода в программу символы ZF=0, если же буфер пуст, ZF=1. При наличии в буфере кода символа его можно проанализировать, так как он возвращается функцией в регистре AX (AH=скен-код, AL=код ASCII). Необходимо, однако, иметь в виду, что функция 01h, копируя двухбайтовый код в регистр AX, не очищает при этом кольцевой буфер. Забрать символ с очисткой буфера можно затем функцией 00h.

Функция 02h – чтение флагов клавиатуры – передает в программу содержимое слова флагов (ячейка 417h). Она может использоваться программами, работающими на уровне скен-кодов, для определения состояния клавиш <Shift>, <Caps Lock> и др.

Лекция №15. Создание Windows - приложений на ассемблере

Цель лекции: изучить особенности разработки Windows - приложений.

Содержание лекции: использование функций API, понятие оконных, консольных приложений, передачи параметров через стек.

Операционные системы MS DOS и Windows поддерживают две совершенно разные идеологии программирования. Windows поддерживает два типа приложений: оконное приложение – строится на базе специального набора функций API и неоконное приложение, также называемое консольным, представляет собой программу, работающую в текстовом режиме. Консольное приложение обеспечивается специальными функциями

Windows. Можно выделить три типа структуры программ, которые условно можно назвать как классическая структура, диалоговая (основное окно - диалоговое), консольная, или безоконная структура.

Программирование в Windows основывается на использовании функций API (Application Program Interfase, т.е. интерфейс программного приложения). Все взаимодействие с внешними устройствами и ресурсами операционной системы будет происходить посредством таких функций.

Главным элементом программы в среде Windows является окно. Для каждого окна определяется своя процедура обработки сообщений. Окно может содержать элементы управления: кнопки, списки, окна редактирования и др. Эти элементы, по сути, также являются окнами, но обладающими особыми свойствами. События, происходящие с этими элементами (и самим окном), приводят к приходу сообщений в процедуру окна.

Операционная система Windows использует линейную модель памяти. Другими словами, всю память можно рассматривать как один сегмент. Это означает, что адрес любой ячейки памяти будет определятся содержимым одного 32-битного регистра, например EBX.

Операционная система Windows является многозадачной средой. Каждая задача имеет свое адресное пространство и свою очередь сообщений.

Начнем с того, как можно вызвать функции API. Выберем любую функцию API, например, MessageBox:

int MessageBox (HWND hwnd, LPCTSTR 1pText, LPCTSTR 1pCaption, 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. Все они в действительности 32-битные целые числа: HWND – 32-битное целое, LPCTSTR – 32-битный указатель на строку, UINT – 32-битное целое.

Часть API-функций получила суффикс “A”. Дело с том, что функции имеют два прототипа: с суффиксом “A” - подденживают ANSI, а с суффиксом “W” - Unicode. Поэтому к имени функции добавляется суффикс «A», кроме того, при использовании MASM необходимо также в конце имени добавить @16. Таким образом, вызов указанной функции будет выглядеть так: CALL MessageBoxA@16. А так же быть с параметрами? Их следует аккуратно поместить в стек. Запомните правило: СЛЕВА НАПРАВО – СНИЗУ ВВЕРХ. Итак, пусть дескриптор окна расположен по адресу HW, строки – по адресам STRl и STR2, а тип окна-сообщения – это константа. Самый простой тип имеет значение 0 и называется MB_OK. Имеем следующее:

```
 MB_OK    equ 0
STR1      DB “Неверный ввод!”, 0
STR2      DB “Сообщение об ошибке”, 0
HW        DWORD ?
    …
   PUSH   MB_OK
   PUSH   OFFSET STR1
   PUSH   OFFSET STR2
   PUSH   HW
CALL      MessageBox@16
```

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

Аналогичным образом в ассемблере легко воспроизвести те или иные Си-структуры. Рассмотрим, например, структуру, определяющую системное сообщение:

```
typedef struct tagMSG { // msq
    HWND hwnd; UINT message;WPARAM wParam;LPARAM lParam;
    DWORD time;POINT pt;}  MSG;
```

На ассемблере эта структура будет иметь вид:

```
MSGSTRUCT STRUC
MSHWND          DD  ?
MSMESSAGE DD  ?
MSWPARAM  DD  ?
MSLPARAM   DD  ?
MSTIME          DD  ?
MSPT        DD  ?
MSGSTRUCT ENDS
```

Теперь обратимся к структуре всей программы. Рассмотрим классическую структуру программы под Windows. В такой программе имеется главное окно, а следовательно, и процедура главного окна. В целом, в коде программы можно выделить следующие секции:

- Регистрация класса окон
- Создание главного окна
- Цикл обработки очереди сообщений
- Процедура главного окна

Конечно, в программе могут быть и другие разделы, но данные разделы образуют основной скелет программы. Разберем эти разделы по порядку.

Регистрация класса окон

Регистрация класса окон осуществляется с помощью функции RegisterClassA, единственным параметром которой является указатель на структуру WNDCLASS, содержащую информацию об окне.

Создание окна

На основе зарегистрированного класса с помощью функции Create WindowExA (или Create WindowA) можно создать экземпляр окна.

Цикл обработки очереди сообщений

Вот как выглядит этот цикл на языке Си:

```
while (GetMessage (&msq, NULL, 0, 0))
{
/ / разрешить использование клавиатуры,
/ / путем трансляции сообщений о виртуальных клавишах
/ / в сообщения о алфавитно-цифровых клавишах
TranslateMessage (&msq);
/ /  вернуть управление Windows и передать сообщение дальше
/ / процедуре окна
```

```
DispatchMessage (&msg);
}
```

Функция GetMessage() «отлавливает» очередное сообщение из ряда сообщений данного приложения и помещает его в структуру MSG.

Что касается функции TranslateMessage, то ее компетенция касается сообщений WM_KEYDOWN и WM_KEYUP, которые транслируются в WM_CHAR и WM_DEDCHAR, а также WM_SYSKEYDOWN и WM_SYSKEYUP, преобразующиеся в WM_SYSCHAR и WM_SYSDEADCHAR. Смысл трансляции заключается не в замене, а в отправке дополнительных сообщений. Так, например, при нажатии и отпускании алфавитно-цифровой клавиши в окно сначала придет сообщение WM_KEYDOWN, затем WM_KEYUP, а затем уже WM_CHAR.

Как можно видеть, выход из цикла ожиданий имеет место только в том случае, если функция GetMessage возвращает 0. Это происходит только при получении сообщения о выходе (сообщение WM_QUIT). Таким образом, цикл ожидания играет двоякую роль: определенным образом преобразуются сообщения, предназначенные для какого-либо окна, и ожидается сообщение о выходе из программы.

Процедура главного окна

Вот прототип функции окна на языке C:

```
LRESULT CALLBACK WindowFunc (HWND hwnd, UINT message, WPARAM wParam,
PARAM lParam)
```

Оставив в стороне тип возвращаемого функцией значения, обратите внимание на передаваемые параметры. Вот смысл этих параметров: hwnd – идентификатор окна, message – идентификатор сообщения, wParam и lParam – параметры, уточняющие смысл сообщения (для каждого сообщения могут играть разные роли или не играть никаких). Все четыре параметра имеют тип DWORD.

А теперь рассмотрим «скелет» этой функции на языке ассемблера.

```
WNDPROC         PROC
PUSH        EBP
MOV EBP, ESP    ; теперь EBP указывает на вершину стека
PUSH  EBX
PUSH  ESI
PUSH  EDI
PUSH  DWORD  PTR  [EBP+14H]          ; LPARAM (lParam)
PUSH  DWORD  PTR  [EBP+10H]          ; WPARAM (wParam)
PUSH  DWORD  PTR  [EBP+0CH]          ; MES (message)
PUSH  DWORD  PTR  [EBP+08H]          ; HWND (hwnd)
CALL DefWindowProcA@16
POP  EDI
POP  ESI
```

```
POP  EBX
POP  EBP
RET  16
WNDPROC        ENDP
```

Прокомментируем фрагмент. RET 16 – выход с освобождением стека от четырех параметров (16 = 4*4).

Доступ к параметрам осуществляется через регистр EBP:

```
DWORD  PTR  [EBP+14H]    ; LPARAM (lParam)
DWORD  PTR  [EBP+10H]    ; WPARAM (wParam)
DWORD  PTR  [EBP+0CH]    ; MES (message) – код сообщения
DWORD  PTR  [EBP+08H]    ; HWND (hwnd) – дескриптор окна
```

Функция DefWindowProc вызывается для тех сообщений, которые не обрабатываются в функции окна

Базовая структура модуля на ассемблере

Структура модуля на ассемблере при программировании Windows – приложения в формате EXE – файла может иметь следующий вид:

```
.386     ;    .386p или  старше
.MODEL Flat, STDCALL
include    win32.inc          ; Windows - макробиблиотека
……..
includelib   import32.lib     ;Windows – библиотека
.DATA
          ;  данные
……
.CONST
          ; константы
……..
.CODE
ТочкаВхода:
          ;   код
…….
End  ТочкаВхода
```

В данной структуре .MODEL Flat, STDCALL – это директива обеспечивает плоскую модель (FLAT), которая используется для платформы Win32. Платформа Win32 использует исключительно тип вызова функций STDCALL, который является гибридом между Паскалем и C: параметры передаются по порядку справа налево, как в языке C/C++, а по окончании работы вызываемая процедура должна сама очистить стек, как в Паскаль.

Рассмотрим на примере некоторые нюансы Windows – программирования в ассемблере. Создать простейшее оконное приложение, которое должно вывести текстовую строку с сообщением ”Я учусь программировать в Windows ”:

```
.386
.model  flat, stdcall
include    win32.inc              ; Windows - макробиблиотека
includelib   import32.lib       ;Windows – библиотека
.data
mesbox _text        db   ‘Я учусь программировать в  Windows!’,0
mesbox_title         db    ‘Заголовок окошка MessageBox’,0

.code
start:
     ; Загрузка в стек параметров (задом наперед)
push 0           ; стиль окна (UNIT  uType) – по умолчанию
push offset  mesbox_title      ; адрес заголовка окна (LPCTSTR lpCaption)
push offset  mesbox_text    ; адрес выводимого текста (LPCTSTR lpText)
push 0  ; HWND hWnd = 0 – владельца нашего окна нет
; Вызов функции  MessageBox
call        MessageBox
push 0  ; завершающий код для всех потоков (UINT   uExitCode)
call       ExitProcess
ends
end start.
```

Создадим теперь EXE – файл оконного приложения и запустим его на выполнение:

```
EXE32win.bat MESSBOX.ASM  MESSBOX.exe
```

Список литературы

1. Пильщиков В.Н. Assembler. Программирование на языке ассемблера IBM PC -М.: Диалог-МИФИ, 2005 г.- 288 с.
2. Галисеев Г.В. Ассемблер для Win 32. Самоучитель - К.:Диалектика, 2007г.- 368 с.
3. Рудольф Марек. Ассемблер на примерах - М.: Наука и техника, 2005 г.- 240 с.
4. Калашников О.А. Ассемблер - это просто. Учимся программировать - Санкт-Петербург: БХВ-Петербург, 2011 г.- 336 с.
5. Джесси Рассел. Ассемблер - М.: Книга по Требованию, 2012 г.- 98 с.
6. Михаэль Хофманн . Микроконтроллеры для начинающих (+ CD-ROM) - Санкт-Петербург: БХВ-Петербург, 2010 г.- 304 с.
7. Иванов В.Б. Программирование микроконтроллеров для начинающих. Визуальное проектирование, язык С, ассемблер (+ CD-ROM) - Санкт-Петербург: Корона-Век, МК-Пресс, 2010 г.- 176 с.
8. Андрей Жуков, Андрей Авдюхин . Самоучитель Ассемблер - Санкт – Петербург:БХВ-Петербург, 2002 г.- 448 с.
9. Голубь Н.Г. Исскуство программирования на Ассемблере – М.: ООО «ДиаСофтЮП»,2005. – 832 с.
10. Пирогов В.Ю. Ассемблер для Windows - Санкт – Петербург: БХВ-Петербург, 2005 г.

Сводный план 2015 г., поз. 310

Айжан Токжумаевна Купарова

ПРОГРАММИРОВАНИЕ В АССЕМБЛЕРЕ

Конспект лекций
для студентов специальности 5В060200 - Информатика

Редактор Л.Т.Сластихина
Специалист по стандартизации Н.К. Молдабекова

Подписано к печати ___.___.____ .
Тираж __ экз.
Объём ____ уч.-изд.л.

Формат 60х84 1/16
Бумага типографская №1
Заказ _____ Цена _____

Копировально-множительное бюро
некоммерческого акционерного общества
«Алматинский университет энергетики и связи»
050013, Алматы, Байтурсынова, 126