Министерство образования Республики Беларусь

Учреждение образования
«Полоцкий государственный университет»

Д. Г. РУГОЛЬ

НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ

Учебно-методический комплекс
для студентов специальностей 1-40 02 01 «Вычислительные машины, системы и сети», 1-40 01 01 «Программное обеспечение информационных технологий», 1-36 04 02 «Промышленная электроника»

Новополоцк
ПГУ
2008

УДК 004.42(075.8)
ББК 32.973-01я73
Р82

Рекомендовано к изданию методической комиссией
радиотехнического факультета в качестве
учебно-методического комплекса (протокол № 9 от 17.12.2007)

РЕЦЕНЗЕНТЫ:

инженер по АСУП 2 категории ОАО «Нафтан» О. Ю. ШУБИН;
канд. техн. наук, доц. О. Е. ШЕСТОПАЛОВА

Руголь, Д. Г.

Р82 Низкоуровневое программирование : учеб.-метод комплекс для студентов спец. 1-40 02 01 «Вычислительные машины, системы и сети», 1-40 01 01 «Программное обеспечение информационных технологий», 1-36 04 02 «Промышленная электроника» / Д. Г. Руголь. – Новополоцк : ПГУ, 2008. – 128 с.

ISBN 978-985-418-764-8.

Рассмотрены программная модель микропроцессоров семейства x86, принципы сегментной организации памяти и распределения адресного пространства, структура системы прерываний; изложены принципы использования языка программирования Ассемблера и программирования различных функциональных блоков ПЭВМ.

УДК 004.42(075.8)
ББК 32.973-01я73

ISBN 978-985-418-764-8

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Основная цель курса «Низкоуровневое программирование» – изучение основ и принципов низкоуровневого программирования на языке Ассемблера для платформы PC для микропроцессоров семейства x8086.

Основными задачами курса являются:

- приобретение знаний о программной модели микропроцессоров семейства x8086;
- изучение принципов программирования различных функциональных блоков ПЭВМ;
- наработка практического опыта по разработке программного обеспечения на языке Ассемблера.

В результате изучения курса студенты должны:

- иметь представление об архитектуре микропроцессоров семейства x8086;
- знать программную модель микропроцессора i80486;
- понимать принципы сегментной организации памяти и распределения адресного пространства;
- знать регистры и флаги состояния процессора;
- знать типы данных языка Ассемблера;
- знать форматы команд и директив языка Ассемблера;
- знать способы адресации операндов в Ассемблере;
- знать структуры программных модулей типа COM и EXE;
- знать структуру системы прерываний, порядка обслуживания прерываний;
- уметь использовать функции системы ввода – вывода данных посредством функций DOS и BIOS;
- наработать практический опыт по разработке программного обеспечения на языке Ассемблера.

Согласно учебному плану курс «Низкоуровневое программирование» изучается студентами на 1 курсе (2 семестр), рассчитан на 48 аудиторных часов и включает в себя следующие виды занятий:

- 32 часа лекций;
- 16 часов лабораторных работ.

РАБОЧАЯ ПРОГРАММА

ЛЕКЦИОННЫЙ КУРС

Наименования разделов и тем лекций и их содержание	**Количество часов**
Введение в курс «Низкоуровневое программирование»	
Содержание дисциплины и ее взаимосвязь с другими дисциплинами	2
Раздел I. Введение в структурную и функциональную организация ЭВМ	
Архитектура ЭВМ. Общие понятия. Структурная схема машины фон Неймана. Свойства и принципы работы машины фон Неймана. Архитектура IA-32. Структурная схема ЭВМ типа PC	2
Программная модель микропроцессора i80486. Сегментная организация памяти	2
Раздел II. Загрузка и выполнение программ в DOS	
Форматы исполняемых файлов. Загрузка и завершение программ	2
Раздел III. Язык Ассемблера. Общие сведения	
Введение в язык Ассемблера. Идентификаторы, переменные, метки, имена, ключевые слова. Простые типы данных. Предложения	2
Операнды. Выражения	2
Директивы определения данных. Виды адресации операндов в памяти	2
Структура программы на языке Ассемблера. Стандартные директивы сегментации	2
Структура программы на языке Ассемблера. Упрощенные директивы сегментации. Директива INCLUDE	2
Структура программы на языке Ассемблера. Процедуры. Структура EXE- и COM- программ	2
Раздел IV. Команды микропроцессора i80486	
Команды пересылки данных. Арифметические команды	2
Команды побитовой обработки данных	2
Команды сравнения и передачи управления. Цепочечные команды	2
Раздел V. Система прерываний микропроцессора i80486	
Классификация прерываний. Таблица векторов прерываний. Порядок обслуживания прерываний. Установка обработчиков прерываний	2
Раздел VI. Сложные типы данных	
Массивы. Способы описания массивов в программе. Доступ к элементам массива. Двухмерные массивы	2
Структуры. Описание шаблона структуры. Определение данных с типом структуры. Методы работы со структурой	2
ВСЕГО:	**32**

ЛАБОРАТОРНЫЕ ЗАНЯТИЯ

Наименование лабораторной работы	**Количество часов**
1. Компиляция программ на Ассемблере. Изучение структуры текстового видеобуфера	4
2. Программирование текстового видеобуфера. Использование переменных. Организация циклов	4
3. Изучение функций прерывания 21h	4
4. Программирование устройства ввода типа «мышь»	4
ВСЕГО:	**16**

Оценка знаний студентов

Для оценки работы и знаний студентов в результате изучения курса «Низкоуровневое программирование» используется накопительная система. Результирующая оценка выставляется по сумме баллов, которые студент набирает в течение всего учебного семестра, а также в результате выходного итогового контроля – экзамена.

Для получения аттестации необходимо получить отметку о выполнении всех лабораторных работ.

РАСПРЕДЕЛЕНИЕ БАЛЛОВ ПО ВИДАМ ЗАНЯТИЙ

Вид занятий	Форма оценки активности студента	Максимальное количество баллов по каждой форме оценки	Максимальное количество баллов по каждому виду занятий
Лабораторные занятия	Защита работы в течение отведенного на нее занятия	+5	25·4 = 100
	Защита работы в течение семестра	20	
Экзамен	Качество ответов на экзаменационные вопросы	100	100

Дополнительные баллы предусматриваются за выполнение задач повышенной сложности (до 100 баллов). Для получения аттестации студент должен сдать все предшествующие лабораторные работы.

Итоговая оценка выставляется по шкале:

Оценка	1	2	3	4	5
Сумма баллов	0 – 79	80 – 99	100 – 119	120 – 139	140 – 159

Окончание

Оценка	6	7	8	9	10
Сумма баллов	160 – 179	180 – 189	190 – 199	200 – 219	220 и более

Для получения минимальной положительной оценки (4 балла) студент должен набрать 120 баллов, для чего требуется:

- выполнить все лабораторные работы – минимум 80 баллов;
- получить 40 баллов при сдаче экзамена.

Для получения высшей оценки (10 баллов) студент должен будет проявить способность самостоятельно и творчески решать задачи повышенной сложности.

МОДУЛЬ 1. ВВЕДЕНИЕ В СТРУКТУРНУЮ И ФУНКЦИОНАЛЬНУЮ ОРГАНИЗАЦИЮ ЭВМ

Цель модуля – приобретение студентами общих понятий о функциональной организации ЭВМ типа PC, изучение программной модели микропроцессора и организации памяти.

В результате изучения модуля студенты должны:

- иметь представление об архитектуре современных ЭВМ;
- знать структурную схему ЭВМ типа PC;
- знать классификацию регистров процессора i80486;
- знать сегментную организацию памяти процессора i80486 в реальном режиме работы;
- знать различные типы сегментов;
- знать особенности сегментации памяти в процессоре i80486.

Содержание модуля

1.1. Архитектура ЭВМ. Общие понятия

Однозначно определить понятие архитектуры ЭВМ довольно трудно, потому что при желании в него можно включить все, что связано с компьютерами вообще и какой-то конкретной моделью компьютера в частности. Однако попытаемся все же его формализовать.

Архитектура ЭВМ – это абстрактное представление ЭВМ, которое отражает ее структурную, схемотехническую и логическую организацию. Понятие архитектуры ЭВМ является комплексным, в него входят:

- структурная схема ЭВМ;
- средства и способы доступа к элементам структурной схемы ЭВМ;
- организация и разрядность интерфейсов ЭВМ;
- набор и доступность регистров;
- организация и способы адресации памяти;
- способы представления и форматы данных ЭВМ;
- набор машинных команд ЭВМ;
- форматы машинных команд;
- правила обработки нештатных ситуаций (прерываний).

Таким образом, описание архитектуры включает в себя практически всю необходимую для программиста информацию о компьютере. Понятие архитектуры ЭВМ – иерархическое. Допустимо вести речь как об архитектуре компьютера в целом, так и об архитектуре отдельных его составляющих, например, архитектуре процессора или архитектуре подсистемы ввода – вывода.

Все современные компьютеры обладают некоторыми общими и индивидуальными архитектурными свойствами. Индивидуальные свойства присущи только конкретной модели компьютера. Общие архитектурные свойства присущи некоторой, часто довольно большой группе компьютеров. На сегодняшний день общие архитектурные свойства большинства современных компьютеров подпадают под понятие *фон-неймановской архитектуры*, которая получила свое название от имени ученого фон Неймана. Когда фон Нейман начал заниматься компьютерами, программирование последних осуществлялось способом коммутирования. В первых ЭВМ для генерации нужных сигналов необходимо было с помощью переключателей выполнить ручное программирование всех логических схем. Фон Нейман предложил схему ЭВМ с программой в памяти и двоичной логикой вместо десятичной. Логически машину фон Неймана составляли пять блоков (рис. 1.1): оперативная память, арифметико-логическое устройство (АЛУ) с аккумулятором, блок управления, устройства ввода и вывода. Особо следует выделить роль аккумулятора. Физически он представляет собой регистр АЛУ. Для процессоров

Intel, в которых большинство команд – двухоперандные, его роль не столь очевидна. Но существовали и существуют процессорные среды с однооперандными машинными командами. В них наличие аккумулятора играет ключевую роль, так как большинство команд используют его содержимое в качестве либо второго, либо единственного операнда команды.

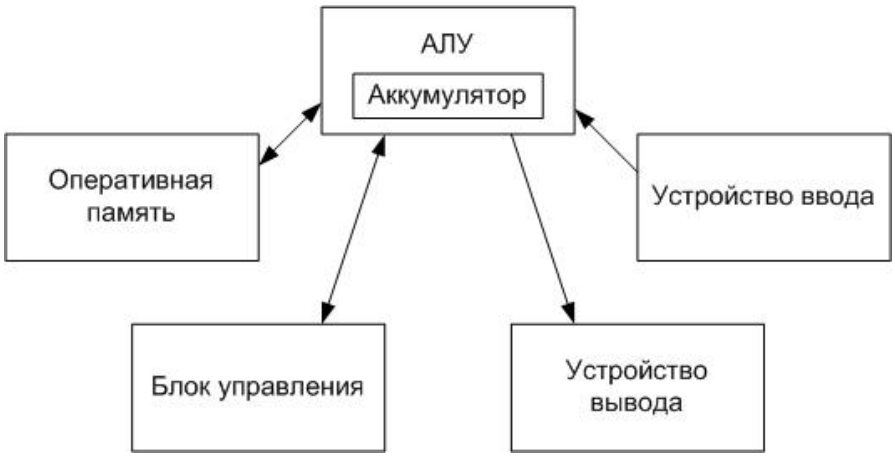

Рис. 1.1 Структурная схема машины фон Неймана

Рассмотрим принципы работы машины фон Неймана:

Ø *линейное пространство памяти*. Для оперативного хранения информации компьютер имеет совокупность ячеек с последовательной нумерацией (адресами) 0, 1, 2,... Данная совокупность ячеек называется *оперативной памятью;*

Ø *принцип хранимой программы*. Согласно этому принципу код программы и ее данные находятся в одном и том же адресном пространстве оперативной памяти;

Ø *принцип микропрограммирования*. Суть этого принципа заключается в том, что машинный язык еще не является той конечной субстанцией, которая физически приводит в действие процессы в машине. В состав процессора входит *устройство микропрограммного управления*, поддерживающее набор действий-сигналов, которые нужно сгенерировать для физического выполнения каждой машинной команды;

Ø *последовательное выполнение программ*. Процессор выбирает из памяти команды строго последовательно. Для изменения прямолинейного хо-

да выполнения программы или осуществления ветвления необходимо использовать специальные команды. Они называются командами *условного* и *безусловного* переходов;

Ø *отсутствие разницы между данными и командами в памяти.* *С* точки зрения процессора, нет принципиальной разницы между данными и командами. Данные и машинные команды находятся в одном пространстве памяти в виде последовательности нулей и единиц. Это свойство связано с предыдущим. Процессор, поочередно обрабатывая некоторые ячейки памяти, всегда пытается трактовать содержимое ячеек как коды машинных команд, а если это не так, то происходит аварийное завершение программы. Поэтому важно всегда четко разделять в программе пространства данных и команд;

Ø *безразличие к назначению данных*. Машине все равно, какую логическую нагрузку несут обрабатываемые ею данные.

Учебно-методический комплекс посвящен вопросам программирования процессоров фирмы Intel и совместимых процессоров других фирм, поэтому в качестве *индивидуальных* архитектурных свойств компьютера далее будут рассматриваться свойства именно этих процессоров. Их полное рассмотрение не является целью этого модуля, поэтому внимание будет уделено лишь тем их них, которые наиболее характерны и понадобятся для дальнейшего изучения материала.

Согласно материалам фирмы Intel индивидуальные архитектурные свойства и принципы работы всех ее процессоров, начиная с i8086 и заканчивая Pentium IV, выстроены в рамках единой архитектуры, которая позже получила название IA-32 (32-bit Intel Architecture). Эта архитектура не является закостенелым образованием. В процессе эволюции процессоров Intel она постоянно изменяется и развивается. Каждый из процессоров вносил в IA-32 одно или несколько архитектурных новшеств. Несмотря на то, что датой рождения архитектуры IA-32 нужно считать дату появления на свет процессора i80386, предыдущие модели процессоров также внесли существенный вклад в представление ее принципов и свойств. Так, например, благодаря процессорам 18086/88 в IA-32 существует сегментация памяти, i80286 ввел защищенный режим работы и т.д.

На рис. 1.2 представлена структурная схема персонального компьютера, построенного на базе процессора с архитектурой IA-32.

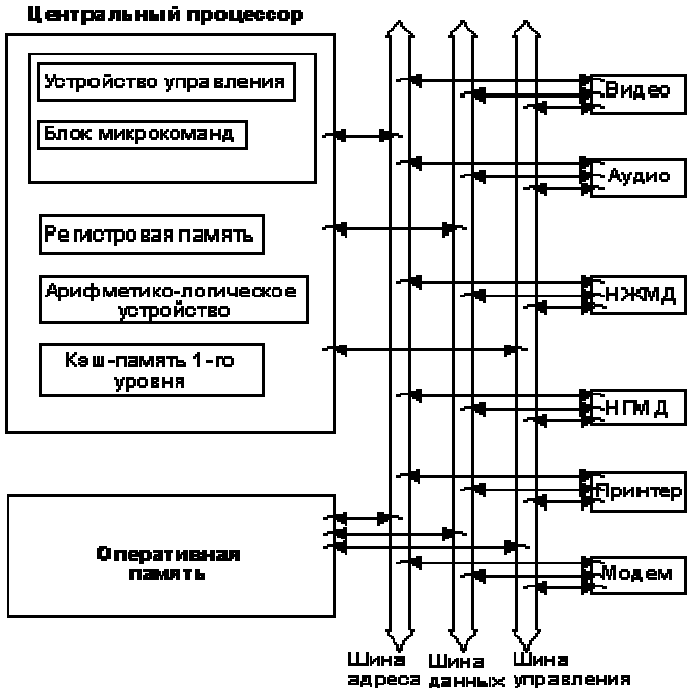

Рис. 1.2. Структурная схема ЭВМ типа PC

1.2. Программная модель микропроцессора i80486

К изучению машинного языка любого компьютера имеет смысл приступать только после выяснения того, какая часть компьютера оставлена видимой и доступной для программирования на этом языке. Это так называемая программная модель компьютера, частью которой является программная модель микропроцессора. В данном учебно-методическом комплексе мы будем рассматривать вопросы программирования процессора i80486, который содержит 32 регистра, в той или иной мере доступных для использования программистом. Данные регистры можно разделить на две большие группы:

- 16 пользовательских регистров, которые пользователь может свободно использовать в своих программах для реализации поставленной задачи;
- 16 системных регистров, предназначенных для поддержки различных режимов работы, сервисных функций.

Регистрами называются области высокоскоростной памяти, расположенные внутри процессора в непосредственной близости от его исполнительного ядра. Доступ к ним осуществляется несравнимо быстрее, чем к ячейкам

оперативной памяти. Соответственно, машинные команды с операндами в регистрах выполняются максимально быстро.

К пользовательским регистрам относятся:

- восемь 32-битных регистров, которые могут использоваться программистами для хранения данных и адресов. Их называют регистрами общего назначения (РОН):
  - EAX/AX/AH/AL;
  - EBX/BX/BH/BL;
  - EDX/DX/DH/DL;
  - ECX/CX/CH/CL;
  - ESI/SI;
  - EDI/DI;
  - ESP/SP;
  - EBP/BP;
- шесть сегментных регистров:
  - CS;
  - DS;
  - SS;
  - ES;
  - FS;
  - GS;
- регистры состояния и управления:
  - регистр флагов EFlags/Flags;
  - регистр указателя команды EIP/IP.

Многие из имен регистров приведены с наклонной разделительной чертой. Следует заметить, что это не разные регистры – это части одного большого 32-разрядного регистра. Их можно использовать в программе как отдельные объекты. Это сделано для обеспечения работоспособности программ, написанных для младших 16-разрядных моделей микропроцессоров фирмы Intel, начиная с i8086. Микропроцессоры i486 и Pentium имеют в основном 32-разрядные регистры (рис. 1.3). Их количество, за исключением сегментных регистров, такое же, как и у i8086, но размерность больше, что и отражено в их обозначениях – они имеют приставку E (Extended).

Рассмотрим состав и назначение пользовательских регистров.

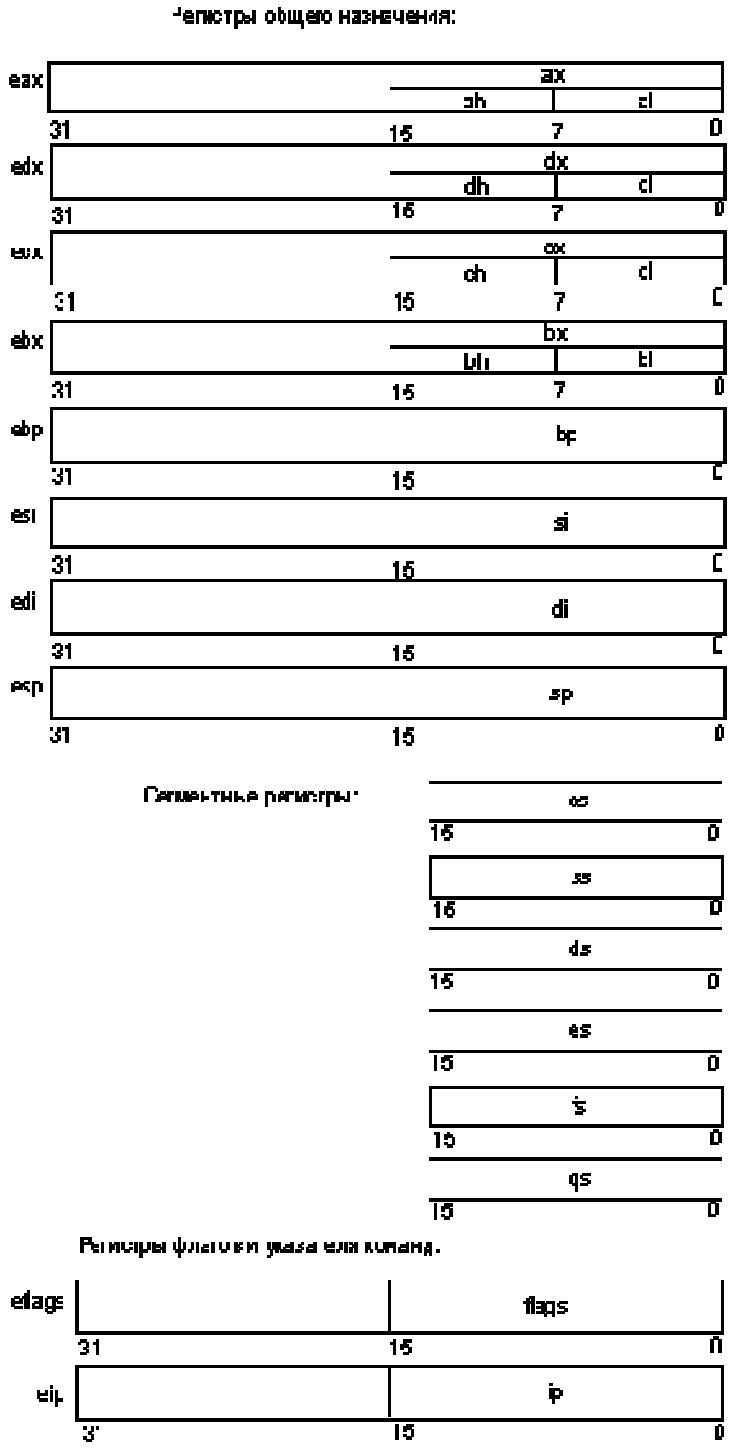

Рис. 1.3. Пользовательские регистры микропроцессора i486

1.2.1. Регистры общего назначения

Все регистры этой группы позволяют обращаться к своим «младшим» частям (см. рис. 1.3). Заметим, что как самостоятельные объекты можно использовать только младшие 16- и 8-битные части этих регистров. Старшие 16 бит этих регистров как самостоятельные объекты недоступны. Это сделано, как было отмечено выше, для совместимости с младшими 16-разрядными моделями микропроцессоров фирмы Intel.

Перечислим более подробно регистры, относящиеся к группе регистров общего назначения. Так как эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства (АЛУ), то их часто называют регистрами АЛУ:

- EAX/AX/AH/AL (Accumulator register) – *аккумулятор*. Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно;
- EBX/BX/BH/BL (Base register) – *базовый* регистр. Применяется для хранения базового адреса некоторого объекта в памяти;
- ECX/CX/CH/CL (Counter register) – *регистр-счетчик*. Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды. К примеру, команда организации цикла **`loop`** кроме передачи управления команде, находящейся по некоторому адресу, уменьшает на единицу и анализирует значение регистра ECX/CX;
- EDX/DX/DH/DL (Data register) – регистр *данных*. Так же, как и регистр EAX/AX/AH/AL, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно (например, умножение и деление).

Следующие два регистра используются для поддержки так называемых цепочечных операций, то есть операций, производящих последовательную обработку цепочек элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

- ESI/SI (Source Index register) – *индекс источника*. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;

Ø EDI/DI (Destination Index register) – *индекс приемника* (получателя). Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

В архитектуре микропроцессора на программно-аппаратном уровне поддерживается такая структура данных, как ***стек***.

Стек – это область памяти, специально выделяемая для временного хранения данных программы. Работу со стеком микропроцессор организует по следующему принципу: *последний занесенный в эту область элемент извлекается первым*.

Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

Ø ESP/SP (Stack Pointer register) – регистр *указателя стека*. Содержит указатель вершины стека в текущем сегменте стека;

Ø EBP/BP (Base Pointer register) – регистр *указателя базы кадра стека*. Предназначен для организации произвольного доступа к данным внутри стека.

Более подробно особенности использования стека рассматриваются в модуле 4 «Команды микропроцессора i80486», пп. 4.1.5 «Команды работы со стеком».

На самом деле функциональное назначение регистров АЛУ на является жестким. Большинство из регистров могут использоваться при программировании для хранения операндов практически в любых сочетаниях. Но, как было отмечено выше, некоторые команды используют фиксированные регистры для выполнения своих действий.

1.2.2. Сегментные регистры

В программной модели микропроцессора имеется шесть сегментных регистров: CS, SS, DS, ES, FS, GS. Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Она заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых *сегментами*. Соответственно, такая организация памяти называется *сегментной*.

Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены *сегментные регистры*. Фактически, с небольшой поправкой, как мы увидим далее, в этих регистрах содержатся адреса памяти, с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Более подробно типы сегментов и соответствующих им регистров рассмотрены в п. 1.3 «Сегментная организация памяти».

1.2.3. Регистры состояния и управления

В микропроцессор включены два регистра, которые постоянно содержат информацию о состоянии как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер:

- 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 (Flags register). Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру Flags для i8086. На рис. 1.4 показано 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. В табл. 1.1 приведены основные флаги состояния и указано их назначение;
- *1 флаг управления*. Обозначается DF (Direction 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`** (снять 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;

Ø *5 системных флагов*, управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086. В прикладных программах не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы. В табл. 1.2 перечислены системные флаги и их назначение.

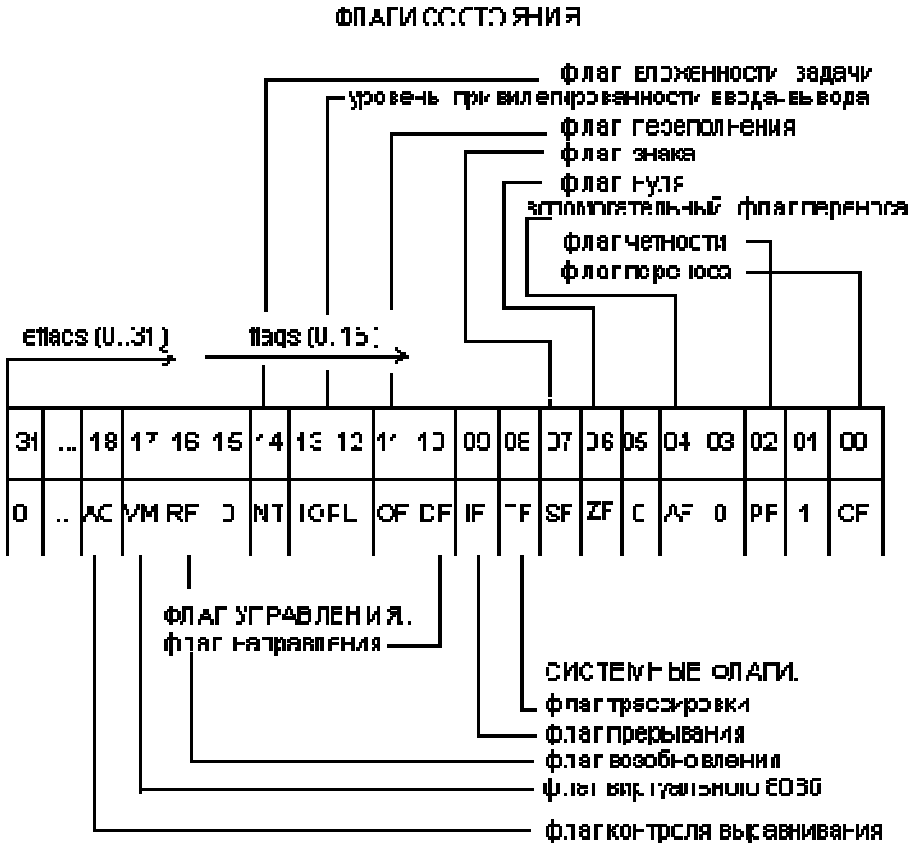

Рис. 1.4. Содержимое регистра EFlags

Таблица 1.1

Основные флаги состояния

Мнемоника флага	**Флаг**	**Номер бита в EFlags**	**Содержание и назначение**
cf	Флаг переноса (Carry Flag)	0	1 – арифметическая операция произвела перенос из старшего бита результата. Старшим является 7-й, 15-й или 31-й бит в зависимости от размерности операнда; 0 – переноса не было
pf	Флаг паритета (Parity Flag)	2	1 – 8 младших разрядов (этот флаг – только для 8-и младших разрядов операнда любого размера) результата содержат четное число единиц; 0 – 8 младших разрядов результата содержат нечетное число единиц
zf	Флаг нуля (Zero Flag)	6	1 – результат нулевой; 0 – результат ненулевой
sf	Флаг знака (Sign Flag)	7	Отражает состояние старшего бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов соответственно): 1 – старший бит результата равен 1; 0 – старший бит результата равен 0
of	Флаг переполнения (Overflow Flag)	11	Флаг of используется для фиксирования факта потери значащего бита при арифметических операциях: 1 – в результате операции происходит перенос (заем) в(из) старшего, знакового бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов соответственно); 0 – в результате операции не происходит переноса (заема) в(из) старшего, знакового бита результата

Таблица 1.2

Системные флаги

Мнемоника флага	Флаг	Номер бита в EFlags	Содержание и назначение
tf	Флаг трассировки (Trace Flag)	8	Предназначен для организации пошаговой работы микропроцессора: 1 – микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ, в частности отладчиками; 0 – обычная работа
if	Флаг прерывания (Interrupt enable Flag)	9	Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR): 1 – аппаратные прерывания разрешены; 0 – аппаратные прерывания запрещены
rf	Флаг возобновления (Resume Flag)	16	Используется при обработке прерываний от регистров отладки
vm	Флаг виртуального режима (Virtual 8086 Mode)	17	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)	18	Предназначен для разрешения контроля выравнивания при обращениях к памяти

EIP/IP (Instruction 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 не доступен 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.

1.3. Сегментная организация памяти

1.3.1. Базовый адрес, смещение и формирование линейных адресов

В реальном режиме работы процессоров семейства x86 используется 20-разрядная адресная шина, соединяющая процессор и память, что позволяет обращаться к 2^{20} различным ячейкам или к 1Мб ОЗУ. Числа, устанавливаемые процессором на адресной шине, являются адресами, то есть номерами ячеек оперативной памяти (ОЗУ). Размер ячейки ОЗУ составляет 8 разрядов (битов), то есть 1 байт. Поскольку для адресации памяти в реальном режиме работы процессор использует 16-разрядные адресные регистры, то это обеспечивает ему доступ к 65536 (FFFFh) байт или 64К (1К = 1024 байт) основной памяти, т.к. максимальное число, которое возможно занести в 16-разрядный регистр, равно FFFFh. Блок непосредственно адресуемой памяти размером 64К называется *сегментом*. Такой объем адресуемой памяти явно недостаточен, т.к. сама по себе адресная шина позволяет адресовать 1Мб. Рассмотрим способ обращения к ячейкам памяти в пределах всего адресного пространства.

В процессорах семейства x86 используется принцип сегментации памяти. Любой адрес формируется из двух составляющих:

- адреса сегмента, который всегда кратен 16, то есть начинается с границы параграфа. Эта часть адреса называется *сегментной составляющей* или *базовым адресом сегмента*;
- адреса ячейки относительно начала сегмента (эта часть адреса называется *смещением*).

Фактически *сегментная составляющая* представляет собой 20-битовый адрес начала сегмента памяти, у которого 4 младших бита всегда равны 0. Поэтому достаточно 16-разрядного сегментного регистра для хранения только 16 старших разрядов адреса.

Рассмотрим процедуру пересчета логических адресов в физические, называемую вычислением *абсолютного* адреса, на следующем примере.

Когда процессор выбирает очередную команду на исполнение, в качестве ее смещения относительно начала сегмента кода используется содержимое регистра IP. Этот адрес называется *исполнительным*. Поскольку регистр IP шестнадцатиразрядный, исполнительный адрес тоже содержит 16 двоич-

ных разрядов. Сегментные регистры также имеют размер в 16 разрядов, а содержащиеся в этих регистрах 16-битовые значения представляют собой адреса начала сегментов в памяти (*базовые адреса сегментов*). Микропроцессор объединяет 16-битовый исполнительный адрес и 16-битовый базовый адрес следующим образом: он расширяет содержимое сегментного регистра (*базовый адрес*) четырьмя нулевыми битами в младших разрядах (как было сказано выше, в сегментном регистре хранятся 16 старших бит 20-битового адреса, у которого младшие 4 бита всегда равны 0), делая его 20-битовым. В результате получается абсолютный адрес начала сегмента памяти (полный адрес сегмента). Далее процессор прибавляет смещение (*исполнительный адрес*). При этом 20-битовый результат является *физическим* или *абсолютным* адресом ячейки памяти (рис. 1.5).

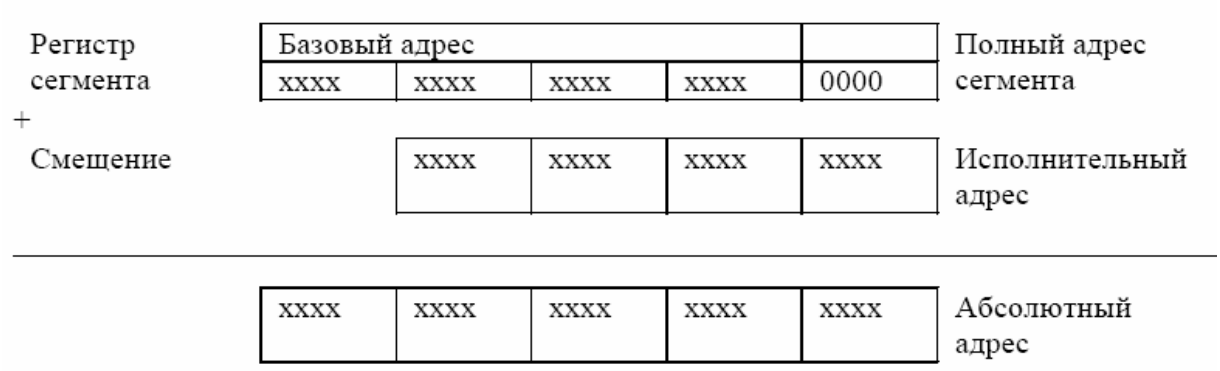

Рис. 1.5. Принцип получения абсолютного адреса

Существуют три основных типа сегментов:

- сегмент кода – содержит машинные команды, адресуется регистром CS;
- сегмент данных – содержит данные, то есть константы и рабочие области, необходимые программе, адресуется регистром DS;
- сегмент стека – содержит временные данные, адресуется регистром SS.

При записи команд на языке Ассемблера принято указывать адреса с помощью следующей конструкции:

```
<адрес сегмента>:<смещение>
```

или

```
<сегментный регистр>:<адресное выражение>
```

1.3.2. Сегменты кода, данных и стека[1]

Микропроцессор на аппаратном уровне поддерживает следующие типы сегментов:

1. *Сегмент кода*. Содержит команды программы. Для доступа к этому сегменту служит регистр CS (Code Segment register) – *сегментный регистр кода*. Он содержит адрес сегмента с машинными командами, к которому имеет доступ микропроцессор. Адрес следующей исполняемой машинной команды в сегменте содержится в регистре IP.

2. *Сегмент данных*. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр DS (Data Segment register) – *сегментный регистр данных*, который хранит адрес сегмента данных текущей программы.

3. *Сегмент стека*. Этот сегмент представляет собой область памяти, называемую *стеком*. Работу со стеком микропроцессор организует по следующему принципу: *последний записанный в эту область элемент выбирается первым*. Для доступа к этому сегменту служит регистр SS (Stack Segment register) – *сегментный регистр стека*, содержащий адрес сегмента стека. Указатель на вершину стека в сегменте стека находится в регистре SP (Stack Pointer register)

4. *Дополнительный сегмент данных*. Неявно процессор предполагает, что обрабатываемые им данные расположены в сегменте данных, адрес которого находится в сегментном регистре DS. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре DS, при использовании дополнительных сегментов данных их требуется указывать явно с помощью специального *оператора переопределения сегмента* в команде[2]. Адреса дополнительных сегментов данных должны содержаться в регистрах ES, FS, GS.

---

[1] Особенности работы с сегментами каждого типа рассматриваются в модуле 3, п. 3.9 «Структура программы на языке Ассемблер».

[2] Особенности использования оператора переопределения сегмента рассмотрены в модуле 3, п. 3.6 «Выражения».

1.3.3. Особенности сегментации памяти в процессоре i80486

Особенности сегментации памяти в процессоре i80486:

1. Сегменты памяти определяются только сегментными регистрами.
2. Начальный адрес сегмента связан с физическим адресом параграфа.
3. Никаких средств проверки корректности использования сегментов нет.
4. Размещение сегментов в памяти достаточно произвольно. Ограничение – только выравнивание на границе параграфа.
5. Сегменты могут частично или полностью перекрываться или не иметь общих частей.
6. Программа может обращаться к любому сегменту как для считывания, так и для записи данных и команд.
7. Система не делает различий между сегментами данных, кода и стека.
8. Нет никаких препятствий для обращения к физически не существующей памяти (при суммировании сегментной составляющей адреса и смещения возможен выход за пределы 1Мб, например, FFF0h + FFF0h = = 10FFEFh). Результат при этом непредсказуем.

1.4. Вопросы и задания для самопроверки

1. Дайте определение понятию «Архитектура ЭВМ».
2. Изобразите структурную схему машины фон-Неймана.
3. Перечислите основные принципы работы машины фон Неймана.
4. Что такое программная модель микропроцессора?
5. Сформулируйте определение регистра процессора.
6. Приведите классификацию пользовательских регистров.
7. Для чего используется регистр IP?
8. Какой объем памяти позволяет адресовать процессор семейства x86 в реальном режиме работы?
9. Из каких составляющих формируется абсолютный адрес ячейки памяти?
10. Какие типы сегментов вы знаете?
11. Можно ли в реальном режиме работы процессора i80486 обратиться к области памяти за пределами 1Мб?
12. Имеются ли какие-либо средства проверки корректности использования сегментов?

МОДУЛЬ 2. ЗАГРУЗКА И ВЫПОЛНЕНИЕ ПРОГРАММ В DOS

Цель модуля – изучение студентами механизмов загрузки, выполнения и завершения программ в DOS.

В результате изучения модуля студенты должны:

- знать распределение сегментов памяти программы после ее загрузки;
- иметь представление о назначении префикса программного сегмента (PSP);
- знать различия между форматами исполняемых файлов;
- знать особенности загрузки EXE и COM программ;
- знать способы завершения программ.

Содержание модуля

При загрузке программ в оперативную память DOS (дисковая операционная система) инициализирует как минимум три сегментных регистра: CS, DS и SS. При этом совокупности байтов, представляющих команды процессора (код программы), и данные помещаются из файла на диске в оперативную память, а адреса этих сегментов записываются в CS и DS соответственно. Сегмент стека либо выделяется в области, указанной в программе, либо совпадает (если он явно в программе не описан) с самым первым сегментом программы. Адрес сегмента стека помещается в регистр SS.

Программа может иметь несколько кодовых сегментов и сегментов данных и в процессе ее выполнения специальными командами осуществлять переключения между ними. Для того чтобы адресовать одновременно два сегмента данных, например, при выполнении операции пересылки из одной области памяти в другую, можно использовать регистр дополнительного сегмента ES.

Кодовый сегмент и сегмент стека всегда определяются содержимым своих регистров (CS и SS), и поэтому в каждый момент выполнения программы всегда используется какой-то один кодовый сегмент и один сегмент стека. Причем, если переключение кодового сегмента – довольно простая

операция, то переключать сегмент стека можно только при условии четкого представления логики работы программы со стеком, иначе это может привести к зависанию системы.

Все сегменты могут использовать различные области памяти, а могут частично или полностью перекрываться (рис. 2.1).

Кодовый сегмент должен обязательно описываться в программе, все остальные сегменты могут отсутствовать. В этом случае DOS при загрузке программы в оперативную память инициирует регистры DS и ES значением адреса *префикса программного сегмента* PSP (Program Segment Prefics) – специальной области оперативной памяти размером 256 (100h) байт. PSP может использоваться в программе для определения имен файлов и параметров из командной строки, введенной при запуске программы на выполнение, объема доступной памяти, переменных окружения системы и так далее. Регистр SS при этом инициализируется значением сегмента, находящегося сразу за PSP, то есть первого сегмента программы. При этом необходимо учитывать, что стек «растет вниз» (при помещении в стек содержимое регистра SP, указывающего на вершину стека, уменьшается, а при считывании из стека – увеличивается). Таким образом, при помещении в стек каких-либо значений они могут «затереть» PSP и программы, находящиеся в младших адресах памяти, что может привести к непредсказуемым последствиям. Поэтому рекомендуется всегда явно описывать сегмент стека в тексте программы, задавая ему размер, достаточный для нормальной работы.

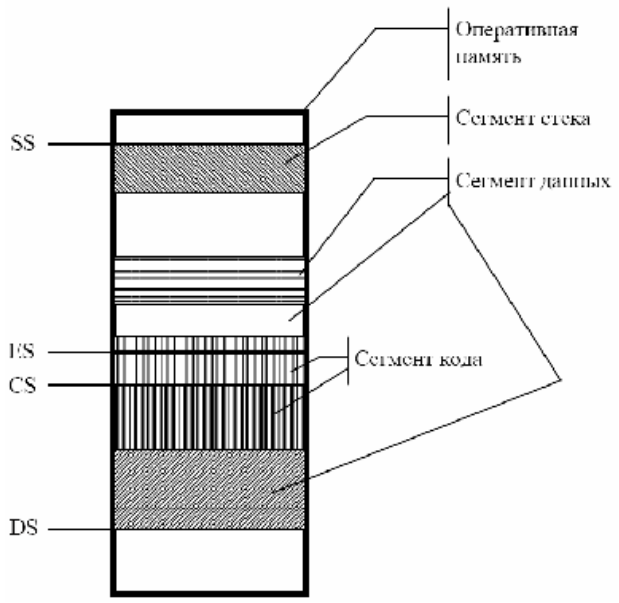

Рис. 2.1. Пример размещения сегментов программы в памяти

Рассмотрим распределение памяти на примере простейшей программы.

```
;Сегмент данных программы
DATA SEGMENT
MSG DB ‘Текст$’
DATA ENDS
;Сегмент кода программы
CODE SEGMENT
ASSUME CS:CODE,DS:DATA
START:
MOV AX,DATA
MOV DS,AX
MOV AH,09H ;Вывод сообщения
MOV DX,OFFSET MSG
INT 21H
MOV AH,4CH ;Завершение работы
INT 21H
CODE ENDS
END START
```

В этой программе явно описаны два сегмента – кода с именем CODE и данных с именем DATA. Директива ASSUME связывает имена этих сегментов, которые в общем случае могут быть произвольными, с сегментными регистрами CS и DS соответственно. Распределение памяти при загрузке программы на исполнение показано на рис. 2.2.

Как видно из рис. 2.2, сегмент стека в данном случае установлен на PSP, что при его интенсивном использовании может привести к неожиданным результатам. После инициализации в регистре IP находится смещение первой команды программы относительно начала кодового сегмента, адрес которого помещен в регистр CS. Процессор, считывая эту команду, начинает выполнение программы, постоянно изменяя содержимое регистра IP и при необходимости CS для получения кодов очередных команд до тех пор, пока не встретит команду завершения программы. DS после загрузки программы установлен на начало PSP, поэтому для его использования в первых двух командах программы выполняется загрузка DS значением сегмента данных.

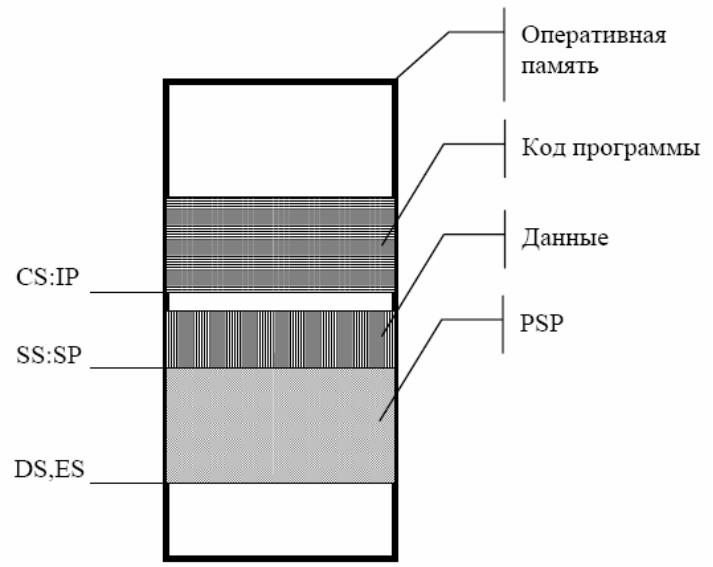

Рис. 2.2. Распределение памяти простейшей программы после ее загрузки

2.1. Форматы исполняемых файлов

DOS может загружать и выполнять программные файлы двух типов: COM и EXE.

Ввиду сегментации адресного пространства процессора 8086 и того факта, что переходы (JMP) и вызовы (CALL) используют относительную адресацию, оба типа программ могут выполняться в любом месте памяти. Программы никогда не пишутся в предположении, что они будут загружаться с определенного адреса (за исключением некоторых самозагружающихся, защищенных от копирования программ).

Файл COM-формата – это двоичный образ кода и данных программы. Такой файл должен занимать менее 64K и не содержать перемещаемых адресов сегментов.

Файл EXE-формата содержит специальный заголовок, при помощи которого загрузчик выполняет настройку ссылок на сегменты в загруженном модуле.

Перед загрузкой COM- или EXE-программы DOS определяет сегментный адрес, называемый префиксом программного сегмента (PSP), как базовый для программы, а затем выполняет следующие шаги:

- создает копию текущего окружения DOS (область памяти, содержащая ряд строк в формате ASCIIZ, которые могут использоваться приложениями для получения некоторой системной информации и для передачи данных между программами) для программы;

Ø помещает путь к загружаемой программе в конец окружения;

Ø заполняет поля PSP информацией, полезной для загружаемой программы (количество памяти, доступное программе, сегментный адрес окружения DOS, текущие векторы прерываний INT 22H INT 23H и INT 24H и т.д.).

2.1.1. Программы формата EXE

EXE-программы содержат несколько программных сегментов, включая сегмент кода, данных и стека. EXE-файл загружается начиная с адреса PSP:0100h. В процессе загрузки считывается информация заголовка EXE в начале файла и выполняется перемещение адресов сегментов. Это означает, что ссылки типа

```
mov ax,data_seg
mov ds,ax
call my_far_proc
```

должны быть приведены (пересчитаны), чтобы учесть тот факт, что программа была загружена в произвольно выбранный сегмент. После перемещения управление передается загрузочному модулю посредством инструкции далекого перехода (FAR JMP) к адресу CS:IP, извлеченному из заголовка EXE.

В момент получения управления программой EXE-формата:

Ø DS и ES указывают на начало PSP;

Ø CS, IP, SS и SP инициализированы значениями, указанными в заголовке EXE;

Ø поле PSP MemTop (вершина доступной памяти системы в параграфах) содержит значение, указанное в заголовке EXE.

2.1.2. Программы формата COM

COM-программы содержат единственный сегмент (или, во всяком случае, не содержат явных ссылок на другие сегменты). Образ COM-файла считывается с диска и помещается в память начиная с PSP:0100h. В общем случае COM-программа может использовать множественные сегменты, но она должна сама вычислять сегментные адреса, используя PSP как базу. COM-программы предпочтительнее EXE-программ, когда дело касается небольших ассемблерных утилит. Они быстрее загружаются, так как не требуют перемещения сегментов, и занимают меньше места на диске, поскольку заголовок EXE и сегмент стека отсутствуют в загрузочном модуле.

После загрузки двоичного образа COM-программы:

- CS, DS, ES и SS указывают на PSP;
- SP указывает на конец сегмента PSP (обычно 0FFFEH, но может быть и меньше, если полный 64K сегмент недоступен);
- слово по смещению 06H в PSP (доступные байты в программном сегменте) указывает, какая часть программного сегмента доступна;
- IP содержит 100H (первый байт модуля) в результате команды JMP PSP:100H.

2.2. Завершение программ

Завершить программу можно следующими способами:

- через функцию 4CH (EXIT) прерывания 21H в любой момент, независимо от значений регистров;
- через функцию 00H прерывания 21H или отдельное прерывание INT 20H, когда CS указывает на PSP.

Функция DOS 4CH позволяет возвращать родительскому процессу код выхода, который может быть проверен вызывающей программой.

Можно также завершить программу и оставить ее постоянно резидентной (TSR), используя либо INT 27H, либо функцию 31H (KEEP) прерывания 21H. Последний способ имеет те преимущества, что резидентный код может быть длиннее 64K, и в этом случае можно сформировать код выхода для родительского процесса.

2.3. Вопросы и задания для самопроверки

1. Каково размещение сегментов в памяти и инициализация регистров при загрузке программы?
2. Для чего используется PSP?
3. Что такое программа EXE-формата?
4. Что такое программа COM-формата?
5. Каковы основные различия программ COM- и EXE-формата?

МОДУЛЬ 3. ЯЗЫК АССЕМБЛЕРА. ОБЩИЕ СВЕДЕНИЯ

Цель модуля – изучение студентами общих сведений, касающихся синтаксиса языка Ассемблера и структуры программы.

В результате изучения модуля студенты должны:

- Ø четко понимать различия между командами процессора и директивами транслятора;
- Ø знать основные типы предложений языка;
- Ø знать основные типы данных;
- Ø знать способы определения данных;
- Ø знать виды операндов;
- Ø знать классификацию операторов, с помощью которых строятся выражения;
- Ø освоить различные виды адресации операндов в памяти;
- Ø уметь описывать процедуры в тексте программы;
- Ø знать стандартные и упрощенные директивы сегментации;
- Ø знать типовые структуры EXE- и COM-программ.

Содержание модуля

В рамках данного модуля предусмотрено выполнение лабораторной работы № 1 «Компиляция программ на Ассемблере. Изучение структуры текстового видеобуфера» (Методические указания к выполнению лабораторных работ по курсу «Низкоуровневое программирование», сост. Д. Г. Руголь).

3.1. Введение в язык Ассемблера

Язык Ассемблера – то символическое представление машинного языка. Все процессы в машине на самом низком, аппаратном уровне приводятся в действие только командами (инструкциями) машинного языка.

По-настоящему решить проблемы, связанные с аппаратурой (или даже, более того, зависящие от аппаратуры, как, к примеру, повышение быстродействия программы), невозможно без знания Ассемблера. И не случайно практически все компиляторы языков высокого уровня содержат средства связи своих модулей с модулями на Ассемблере либо поддерживают выход на ассемблерный уровень программирования.

Из всего вышесказанного можно сделать вывод, что так как язык Ассемблера для компьютера «родной», то и самая эффективная программа может быть написана только на нем (при условии, что ее пишет квалифицированный программист). Однако следует заметить, что это очень трудоемкий, требующий большого внимания и практического опыта процесс. Поэтому чаще всего на Ассемблере пишут в основном программы, которые должны обеспечить эффективную работу с аппаратной частью. Иногда на Ассемблере пишутся критичные по времени выполнения или расходованию памяти участки программы. Впоследствии они оформляются в виде подпрограмм и совмещаются с кодом на языке высокого уровня.

Программа на Ассемблере представляет собой совокупность блоков памяти, называемых *сегментами памяти*. Программа может состоять из одного или нескольких таких блоков-сегментов. Каждый сегмент содержит совокупность предложений языка, каждое из которых занимает отдельную строку кода программы.

Выделяют два основных типа предложений:

- команды или инструкции, представляющие собой символические аналоги машинных команд. В процессе трансляции инструкции Ассемблера преобразуются в соответствующие команды системы команд микропроцессора;

Ø директивы, являющиеся указанием транслятору Ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении.

Предложения представляют собой комбинацию знаков, входящих в алфавит языка, а также чисел и идентификаторов, которые тоже формируются из знаков алфавита. Допустимыми символами при написании текста программ являются:

Ø все латинские буквы: A – Z, a – z. При этом заглавные и строчные буквы считаются эквивалентными;

Ø цифры от 0 до 9;

Ø знаки ?, @, $, _, &;

Ø разделители , . [ ] ( ) < > { } + / * % ! ' " ? \ = # ^.

3.2. Идентификаторы, переменные, метки, имена, ключевые слова

Конструкции языка Ассемблера формируются из идентификаторов и ограничителей. *Идентификатор* представляет собой набор букв, цифр и символов «_», «.», «?», «$» или «@» (символ «.» может быть только первым символом идентификатора), не начинающийся с цифры. Идентификатор должен полностью размещаться на одной строке и может содержать от 1 до 31 символа (точнее, значимыми являются только первые 31 символ идентификатора, остальные игнорируются). Друг от друга идентификаторы отделяются пробелом или ограничителем, которым считается любой недопустимый в идентификаторе символ. Посредством идентификаторов представляются следующие объекты программы:

Ø переменные;

Ø метки;

Ø имена.

Переменные идентифицируют хранящиеся в памяти данные. Все переменные имеют три атрибута:

1) *сегмент*, соответствующий тому сегменту, который ассемблировался, когда была определена переменная;

2) *смещение*, являющееся смещением данного поля памяти относительно начала сегмента;

3) *тип*, определяющий число байтов, подвергающихся манипуляциям при работе с переменной.

Метка является частным случаем переменной, когда известно, что определяемая ею память содержит машинный код. На нее можно ссылаться посредством переходов или вызовов. Метка имеет два атрибута: *сегмент* и *смещение*.

Именами считаются символы, определенные директивой EQU и имеющие значением символ или число. Значения имен не фиксированы в процессе ассемблирования, но при выполнении программы именам соответствуют константы.

Некоторые идентификаторы, называемые *ключевыми словами*, имеют фиксированный смысл и должны употребляться только в соответствии с этим. Ключевыми словами являются:

- директивы Ассемблера;
- инструкции процессора;
- имена регистров;
- операторы выражений.

В идентификаторах одноименные строчные и заглавные буквы считаются эквивалентными. Например, идентификаторы AbS и abS считаются совпадающими.

3.3. Простые типы данных[3]

Опишем типы и формы представления данных, которые могут быть использованы в выражениях, директивах и инструкциях языка Ассемблера.

Целые числа имеют следующий синтаксис (xxxx – цифры):

```
[[+|-]]xxxx
[[+|-]]xxxxB
[[+|-]]xxxxQ
[[+|-]]xxxxO
[[+|-]]xxxxD
[[+|-]]xxxxH
```

Латинский символ (в конце числа), который может записываться как заглавной, так и строчной буквой, задает основание системы счисления числа: B – двоичное, Q и O – восьмеричное, D – десятичное, H – шестнадцатеричное. Шестнадцатеричные числа не должны начинаться с буквенных цифр (напри-

[3] Кроме целых и символьных типов, рассмотренных в этом разделе, Ассемблер содержит еще ряд типов (массивы, структуры, записи), однако их рассмотрение выходит за рамки данного модуля.

мер, вместо некорректного ABh следует употреблять 0ABh). Разряды шестнадцатеричных цифры от A до F могут записываться в обоих регистрах (строчные и заглавные). Первая форма целого числа ([[+|–]]xxxx) использует основание по умолчанию (обычно десятичное).

Символьные и строковые константы имеют следующий синтаксис:

```
'символы'
"символы"
```

Символьная константа состоит из одного символа алфавита языка. Строковая константа включает в себя два или более символа. В отличие от других компонент языка, строковые константы чувствительны к регистру. Символы «'» и «"» в теле константы должны записываться дважды (как слеш в языке C).

3.4. Предложения

Как было сказано выше, программа на языке Ассемблера представляет последовательность предложений, каждое из которых записывается в отдельной строке:

```
<Предложение>
<Предложение>
...
<Предложение>
```

Предложения определяют структуру и функции программы, они могут начинаться с любой позиции и содержать не более 128 символов. При записи предложений действуют следующие правила расстановки пробелов:

- пробел обязателен между рядом стоящими идентификаторами и/или числами (чтобы отделить их друг от друга);
- внутри идентификаторов и чисел пробелы недопустимы;
- в остальных местах пробелы можно ставить или не ставить;
- там, где допустим один пробел, можно ставить любое число пробелов.

Директивы Ассемблера действуют лишь в период компиляции программы и позволяют устанавливать режимы компиляции, задавать структуру сегментации программы, определять содержимое полей данных, управлять печатью листинга программы, а также обеспечивают условную компиляцию и некоторые другие функции. В результате обработки директив компилятором объектный код не генерируется.

Инструкции процессора представляют собой мнемоническую форму записи машинных команд, непосредственно выполняемых микропроцессором. Все инструкции в соответствии с выполняемыми ими функциями делятся на 5 групп:

1) инструкции пересылки данных;
2) арифметические, логические и операции сдвига;
3) операции со строками;
4) инструкции передачи управления;
5) инструкции управления процессором.

3.5. Операнды

Практически каждое предложение содержит описание объекта, над которым или при помощи которого выполняется некоторое действие. Эти объекты называются операндами. Их можно определить так: **операнды** – это объекты (некоторые значения, регистры или ячейки памяти), на которые действуют инструкции или директивы, либо это объекты, которые определяют или уточняют действие инструкций или директив.

Операнды могут комбинироваться с арифметическими, логическими, побитовыми и атрибутивными операторами для расчета некоторого значения или определения ячейки памяти, на которую будет воздействовать данная команда или директива.

Можно провести следующую классификацию операндов:

- постоянные, или непосредственные операнды;
- адресные операнды;
- перемещаемые операнды;
- счетчик адреса;
- регистровый операнд;
- базовый и индексный операнды.

Рассмотрим подробнее характеристику операндов из приведенной классификации.

Постоянные или непосредственные операнды – число, строка, имя или выражение, имеющие некоторое фиксированное значение. Имя не должно быть перемещаемым, то есть зависеть от адреса загрузки программы в память. Например, оно может быть определено операторами **equ** или **=**.

```
num    equ  5
imd =  num-2
. . .
mov al,num   ;эквивалентно mov al,5
             ;5 здесь непосредственный операнд
mov cl,imd   ;эквивалентно mov cl,3
mov al,5     ;5 - непосредственный операнд
```

В данном фрагменте определяются две константы, которые затем используются в качестве непосредственных операндов в командах пересылки mov.

Адресные операнды – задают физическое расположение операнда в памяти с помощью указания двух составляющих адреса: *сегмента* и *смещения*. Пример:

```
mov ax,0000h
mov ds,ax
mov ax,ds:0000h   ;записать слово в ax из области
                  ;памяти по адресу 0000:0000
```

Здесь третья команда mov имеет адресный операнд.

Перемещаемые операнды – любые символьные имена, представляющие некоторые адреса памяти. Эти адреса могут обозначать местоположение в памяти некоторых инструкций (если операнд – метка) или данных (если операнд – имя области памяти в сегменте данных). Перемещаемые операнды отличаются от адресных тем, что они не привязаны к конкретному адресу физической памяти. Сегментная составляющая адреса перемещаемого операнда неизвестна и будет определена только после загрузки программы в память для выполнения. Пример:

```
data segment
mas_w   dw      25 dup (0)
. . .
code    segment
. . .
lea     si,mas_w        ;mas_w - перемещаемый операнд
```

В этом фрагменте mas_w – символьное имя, значением которого является начальный адрес области памяти размером 25 слов. Полный физический адрес этой области памяти будет известен только после загрузки программы в память для выполнения.

Счетчик адреса – специфический вид операнда. Он обозначается знаком **$**. Специфика этого операнда заключается в том, что когда транслятор Ассемблера встречает в исходной программе этот символ, то он подставляет вместо него текущее значение счетчика адреса. Значение счетчика адреса или, как его иногда называют, *счетчика размещения*, представляет собой смещение текущей машинной команды относительно начала сегмента кода. В формате листинга счетчику адреса соответствует вторая или третья колонка (в зависимости от того, присутствует или нет в листинге колонка с уровнем вложенности). Если взять в качестве пример любой листинг, то видно, что при обработке транслятором очередной команды ассемблера счетчик адреса увеличивается на длину сформированной машинной команды. Важно правильно понимать этот момент. Например, обработка директив ассемблера не влечет за собой изменения счетчика. Директивы, в отличие от команд ассемблера, – это лишь указания транслятору на выполнение определенных действий по формированию машинного представления программы, и для них транслятором не генерируется никаких конструкций в памяти. В качестве примера использования в команде значения счетчика адреса можно привести следующий:

```
jmp $+3      ;безусловный переход на команду mov
cld          ;длина команды cld составляет 1 байт
mov al,1
```

При использовании подобного выражения для перехода не забывайте о длине самой команды, в которой это выражение используется, так как значение счетчика адреса соответствует смещению в сегменте команд данной, а не следующей за ней команды. В нашем примере команда jmp занимает 2 байта. Необходима осторожность, так как длина команды зависит от того, какие в ней используются операнды. Команда с регистровыми операндами будет короче команды, один из операндов которой расположен в памяти. В большинстве случаев эту информацию можно получить, зная формат машинной команды и анализируя колонку листинга с объектным кодом команды.

Регистровый операнд – это просто имя регистра. В программе на ассемблере можно использовать имена всех регистров общего назначения и большинства системных регистров.

```
mov al,4        ;константу 4 заносим в регистр al
mov dl,pass+4   ;байт по адресу pass+4 в регистр dl
add al,dl       ;команда с регистровыми операндами
```

Базовый и индексный операнды. Этот тип операндов используется для реализации **косвенной базовой**, **косвенной индексной** адресации или их комбинаций и расширений (см. п. 3.8).

Операнды являются элементарными компонентами, из которых формируется часть машинной команды, обозначающая объекты, над которыми выполняется операция. В более общем случае операнды могут входить как составные части в более сложные образования, называемые *выражениями*.

3.6. Выражения

Операнд команды может быть выражением, представляющим собой комбинацию *операндов* (см. выше) и *операторов* ассемблера. *Операторы* выполняют арифметические, логические, побитовые и другие операции над операндами выражений. Транслятор ассемблера рассматривает выражение как единое целое и преобразует его в числовую константу. Логически значением этой константы может быть адрес некоторой ячейки памяти или некоторое абсолютное значение.

Ниже даны описания наиболее часто используемых в выражениях операторов.

Арифметические операторы

```
выражение_1 * выражение_2
выражение_1 / выражение_2
выражение_1 MOD выражение_2
выражение_1 + выражение_2
выражение_1 - выражение_2
+ выражение
- выражение
```

Эти операторы обеспечивают выполнение основных арифметических действий (здесь MOD – остаток от деления выражения_1 на выражение_2, а знаком / обозначается деление нацело). Результатом арифметического оператора является абсолютное значение.

Операторы сдвига

```
выражение SHR счетчик
выражение SHL счетчик
```

Операторы SHR и SHL сдвигают значение выражения соответственно вправо и влево на число разрядов, определяемое счетчиком. Биты, выдвигаемые за пределы выражения, теряются.

Операторы отношений (сравнения)

```
выражение_1 EQ выражение_2
выражение_1 NE выражение_2
выражение_1 LT выражение_2
выражение_1 LE выражение_2
выражение_1 GT выражение_2
выражение_1 GE выражение_2
```

Мнемонические коды отношений расшифровываются следующим образом:

EQ – равно;
NE – не равно;
LT – меньше;
LE – меньше или равно;
GT – больше;
GE – больше или равно.

Операторы отношений формируют значение 0FFFFh при выполнении условия и 0000h в противном случае. Выражения должны иметь абсолютные значения. Операторы отношений обычно используются в директивах условного ассемблирования и инструкциях условного перехода.

Логические операторы

```
NOT выражение
выражение_1 AND выражение-2
выражение_1 OR выражение-2
выражение_1 XOR выражение-2
```

Мнемоники операций расшифровываются следующим образом:

NOT – инверсия;

AND – логическое И;

OR – логическое ИЛИ;

XOR – исключающее логическое ИЛИ (XOR).

Операции выполняются над соответствующими битами выражений. Выражения должны иметь абсолютные значения.

Индексный оператор

```
[[выражение_1]] [выражение_2]
```

Транслятор воспринимает наличие индексного оператора как указание сложить значение **выражение_1** за скобками с **выражение_2**, заключенным в скобки. Наличие двойных квадратных скобок говорит о том, что первое выражение необязательно, при его отсутствии предполагается 0.

Пример:

```
mov ax,mas[si] ;пересылка слова по адресу mas+(si) в регистр ax
```

Оператор переопределения типа PTR

```
тип PTR выражение
```

Применяется для переопределения или уточнения типа метки или переменной, определяемых выражением. Тип может принимать одно из значений, приведенных в табл. 3.1.

Таблица 3.1

Типы оператора PTR

Имя типа	Значение
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFFFh
FAR	0FFFEh

Если PTR не используется, Ассемблер подразумевает умалчиваемый тип ссылки. Чаще всего оператор PTR используется для организации доступа к объекту, который при другом способе вызвал бы генерацию сообщения об ошибке (например, для доступа к старшему байту переменной размера Word).

Пример:

```
d_wrd  dd 0aabbccddh0
...
mov al,byte ptr d_wrd+1    ;пересылка второго байта из
                           ;двойного слова
```

Поясним этот фрагмент программы. Переменная d_wrd имеет тип двойного слова. Если попытаться обратиться не ко всей переменной, а только к одному из входящих в нее байтов (например, ко второму) командой mov al,d_wrd+1, то транслятор выдаст сообщение о несовпадении типов операндов. Оператор **ptr** позволяет непосредственно в команде переопределить тип и выполнить команду.

Оператор переопределения сегмента : (двоеточие) заставляет вычислять физический адрес относительно конкретно задаваемой сегментной составляющей. Его назначение состоит в том, чтобы указать микропроцессору (а по сути, микропрограмме) на то, что мы не хотим использовать сегмент по умолчанию. Возможности для подобного переопределения, конечно, ограничены. Сегмент команд переопределить нельзя, адрес очередной исполняемой команды однозначно определяется парой cs:ip. А вот сегменты стека и данных – можно. Для этого и предназначен оператор «**:**». Транслятор ассемблера,

обрабатывая этот оператор, формирует соответствующий однобайтовый префикс замены сегмента. Например:

```
.code
...
jmp  met1      ;обход обязателен, иначе поле ind будет
               ;трактоваться как очередная команда
ind  db    5   ;описание поля данных в сегменте команд
met1:
...
mov al,cs:ind  ;переопределение сегмента позволяет
работать с данными, определенными внутри сегмента кода
```

Операторы HIGH и LOW

```
HIGH выражение
LOW выражение
```

Операторы HIGH и LOW вычисляют соответственно старшие и младшие 8 бит значения выражения. Выражение может иметь любое значение.

Оператор SEG (получения сегментной составляющей)

```
SEG выражение
```

Этот оператор вычисляет значение сегментной составляющей адреса выражения. Выражение может быть меткой, переменной, именем сегмента, именем группы или другим символом.

Оператор OFFSET(получения смещения)

```
OFFSET выражение
```

Этот оператор вычисляет значение смещения адреса выражения. Выражение может быть меткой, переменной, именем сегмента или другим символом. Для имени сегмента вычисляется смещение от начала этого сегмента до последнего сгенерированного в этом сегменте байта.

Оператор SIZE

```
SIZE переменная
```

Оператор SIZE определяет число байтов памяти, выделенных переменной.

Приоритеты операций

При вычислении значения выражения операции выполняются в соответствии со следующим списком приоритетов (в порядке убывания):

1) LENGTH, SIZE, WIDTH, MASK, (), [], <>
2) Оператор имени поля структуры (.)
3) Оператор переключения сегмента (:)
4) PTR, OFFSET, SEG, TYPE, THIS
5) HIGH, LOW
6) Унарные + и –
7) *, /, MOD, SHR, SHL
8) Бинарные + и –
9) EQ, NE, LT, LE, GT, GE
10) NOT
11) AND
12) OR, XOR
13) SHORT, .TYPE

3.7. Директивы определения данных

Директивы определения данных служат для задания размеров, содержимого и местоположения полей данных, используемых в программе на языке Ассемблера. Директивы определения данных могут задавать:

- скалярные данные, представляющие собой единичное значение или набор единичных значений;
- записи, позволяющие манипулировать с данными на уровне бит;
- структуры, отражающие некоторую логическую структуру данных.

В рамках данного модуля будут рассмотрены только директивы для определения скалярных данных, так как они используются наиболее часто. Директивы определения скалярных данных приведены в табл. 3.2.

Таблица 3.2.

Директивы определения скалярных данных

Формат	**Функция**
[[имя]] DB значение,...	определение байтов
[[имя]] DW значение,...	определение слов
[[имя]] DD значение,...	определение двойных слов
[[имя]] DQ значение,...	определение квадрослов
[[имя]] DT значение,...	определение 10 байтов

Директива DB обеспечивает распределение и инициализацию 1-го байта памяти для каждого из указанных значений. В качестве значения может кодироваться целое число, строковая константа, оператор DUP (см. ниже), абсолютное выражение или знак «?». Знак «?» обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми. Если директива имеет имя, создается переменная типа Byte с соответствующим данному значению указателя позиции смещением. Если в одной директиве определения памяти заданы несколько значений, им распределяются последовательные байты памяти. В этом случае имя, указанное в начале директивы, именует только первый из этих байтов (фактически имя определяет смещение этого байта относительно начала сегмента), остальные остаются безымянными. Для ссылок на них используется выражение вида имя+k, где k – целое число.

Строковая константа может содержать столько символов, сколько помещается на одной строке. Символы строки хранятся в памяти в порядке их следования, то есть 1-й символ имеет самый младший адрес, последний – самый старший.

Во всех директивах определения памяти в качестве одного из значений может быть задан оператор DUP. Он имеет следующий формат:

```
счетчик DUP (значение,...)
```

Указанный в скобках список значений повторяется многократно в соответствии со значением счетчика. Каждое значение в скобках может быть любым выражением (целое число, символьная константа или другой оператор DUP) (допускается до 17 уровней вложенности операторов DUP). Значения, если их несколько, должны разделяться запятыми.

Следует заметить, что оператор DUP может использоваться не только при определении памяти, но и в других директивах.

Синтаксис директив DW, DD, DQ и DT идентичен синтаксису директивы DB.

Примеры директив определения скалярных данных:

```
integer1 DB 16 ;1 байт, имеющий начальное значение 16
string1 DB 'abCDf'              ;строка из 5 символов
empty1 DB ?       ;не инициализированная переменная
contan2 DW 4*3    ;2-х байтная переменная, равная 12
high4 DQ 18446744073709551615 ; 8-байтная переменная
db6 DB 5 DUP(5)   ; массив из 5 байт, каждый равен 5
```

3.8. Виды адресации операндов в памяти

Перечислим и затем рассмотрим особенности основных видов адресации операндов в памяти:

- прямая адресация;
- косвенная базовая (регистровая) адресация;
- косвенная базовая (регистровая) адресация со смещением;
- косвенная индексная адресация со смещением;
- косвенная базовая индексная адресация;
- косвенная базовая индексная адресация со смещением.

Прямая адресация

Это простейший вид адресации операнда в памяти, так как эффективный адрес содержится в самой команде и для его формирования не используется никаких дополнительных источников или регистров. Эффективный адрес берется непосредственно из поля смещения машинной команды, которое может иметь размер 8, 16, 32 бит. Это значение однозначно определяет байт, слово или двойное слово, расположенные в сегменте данных.

Прямая адресация может быть двух типов:

1) ***относительная прямая адресация***. Используется для команд условных переходов, для указания относительного адреса перехода. Относительность такого перехода заключается в том, что в поле смещения машинной команды содержится 8-, 16- или 32-битное значение, которое в результате работы команды будет складываться с содержимым регистра указателя команд ip/eip. В результате такого сложения получается адрес, по которому и осуществляется переход. Например:

```
jc m1    ;переход на метку m1, если флаг cf = 1
mov al,2
...
m1:
```

Несмотря на то, что в команде указана некоторая метка в программе, ассемблер вычисляет смещение этой метки относительно следующей команды (в нашем случае это mov al,2) и подставляет его в формируемую машинную команду jc;

2) ***абсолютная прямая адресация***. В этом случае эффективный адрес является частью машинной команды, но формируется этот адрес только из

значения поля смещения в команде. Для формирования физического адреса операнда в памяти микропроцессор складывает это поле со сдвинутым на 4 бит значением сегментного регистра. В команде ассемблера можно использовать несколько форм такой адресации. Например:

```
mov ax,dword ptr [0000]  ;записать слово по адресу
;ds:0000 в регистр ax
```

Но такая адресация применяется редко – обычно используемым ячейкам в программе присваиваются символические имена. В процессе трансляции ассемблер вычисляет и подставляет значения смещений этих имен в формируемую им машинную команду в поле смещения в команде. В итоге получается так, что машинная команда прямо адресует свой операнд, имея, фактически, в одном из своих полей значение эффективного адреса. Например:

```
data segment
per1 dw 5
...
data ends
code segment
mov  ax,data
mov  ds,ax
...
mov  ax,per1;записать слово per1 (адрес ds:0000) в ax
```

Мы получим тот же результат, что и при использовании команды mov ax,dword ptr [0000]. Заметим, что смещение переменной per1 равно 0, т.к. она располагается первой в самом начале сегмента данных.

Остальные виды адресации относятся к косвенным. Слово «косвенный» в названии этих видов адресации означает то, что в самой команде может находиться лишь часть эффективного адреса, а остальные его компоненты находятся в регистрах.

Косвенная базовая (регистровая) адресация

При такой адресации эффективный адрес операнда может находиться в любом из регистров общего назначения, кроме sp/esp и bp/ebp (это специфические регистры для работы с сегментом стека). Синтаксически в команде этот режим адресации выражается заключением имени регистра в квадратные

скобки [ ]. Например, команда **mov ax,[ecx]** помещает в регистр **ax** содержимое слова по адресу из сегмента данных со смещением, хранящимся в регистре **ecx**. Так как содержимое регистра легко изменить в ходе работы программы, данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это свойство используется, например, для организации циклических вычислений и для работы с различными структурами данных типа таблиц или массивов.

Косвенная базовая (регистровая) адресация со смещением

Этот вид адресации является дополнением предыдущего и предназначен для доступа к данным с известным смещением относительно некоторого базового адреса. Этот вид адресации удобно использовать для доступа к элементам структур данных, когда смещение элементов известно заранее, на стадии разработки программы, а базовый (начальный) адрес структуры должен вычисляться динамически на стадии выполнения программы. Модификация содержимого базового регистра позволяет обратиться к одноименным элементам различных экземпляров однотипных структур данных. Например, команда `mov ax,[edx+3h]` пересылает в регистр `ax` слова из области памяти по адресу: содержимое `edx + 3h`.

Команда `mov ax,mas[dx]` пересылает в регистр `ax` слово по адресу: содержимое `dx` плюс значение идентификатора `mas` (не забывайте, что транслятор присваивает каждому идентификатору значение, равное смещению этого идентификатора относительно начала сегмента данных).

Косвенная индексная адресация со смещением

Этот вид адресации очень похож на косвенную базовую адресацию со смещением. Здесь также для формирования эффективного адреса используется один из регистров общего назначения. Но индексная адресация обладает одной интересной особенностью, которая очень удобна для работы с массивами. Она связана с возможностью так называемого масштабирования содержимого индексного регистра. Например, в команде `mov ax,mas[si*2]` значение эффективного адреса второго операнда вычисляется выражением `mas+(si)*2`. В связи с тем, что в Ассемблере нет средств для организации индексации массивов, то программисту приходится ее организовывать своими силами. Наличие возможности масштабирования существенно помогает в решении этой проблемы, но при условии, что размер элементов массива составляет 1, 2, 4 или 8 байт.

Косвенная базовая индексная адресация

При этом виде адресации эффективный адрес формируется как сумма содержимого двух регистров общего назначения: базового и индексного. В качестве этих регистров могут применяться любые регистры общего назначения, при этом часто используется масштабирование содержимого индексного регистра. Например в команде `mov eax,[esi][edx]` эффективный адрес второго операнда формируется из двух компонентов (esi)+(edx).

Косвенная базовая индексная адресация со смещением

Этот вид адресации является дополнением косвенной индексной адресации. Эффективный адрес формируется как сумма трех составляющих: содержимого базового регистра, содержимого индексного регистра и значения поля смещения в команде. Например, команда `mov eax,[esi+5][edx]` пересылает в регистр eax двойное слово по адресу: (esi) + 5 + (edx). Команда `add ax,array[esi][ebx]` производит сложение содержимого регистра `ax` с содержимым слова по адресу: значение идентификатора `array + (esi) + (ebx).`

3.9. Структура программы на языке Ассемблера

Исходный программный модуль – это последовательность предложений. Как было сказано выше, различают два основных типа предложений: *инструкции* процессора и *директивы* Ассемблера. *Инструкции* управляют работой процессора, а *директивы* указывают Ассемблеру и редактору связей, каким образом следует объединять инструкции для создания модуля, который и станет работающей программой.

Инструкция процессора на языке Ассемблера состоит не более чем из четырех полей и имеет следующий формат:

```
[[метка:]] мнемоника [[операнды]] [[;комментарии]]
```

Единственное обязательное поле – поле кода операции (мнемоника), определяющее инструкцию, которую должен выполнить микропроцессор.

Поле операндов определяется кодом операции и содержит дополнительную информацию о команде. Каждому коду операции соответствует определенное число операндов.

Метка служит для обозначения какого-то определенного места в памяти, то есть содержит в символическом виде адрес, по которому хранится инструкция. Преобразование символических имен в действительные адреса осуществляется программой Ассемблера.

Часть строки исходного текста после символа «;» (если он не является элементом знаковой константы или строки знаков) считается комментарием и Ассемблером игнорируется. Комментарии вносятся в программу как поясняющий текст и могут содержать любые знаки до ближайшего символа конца строки. Для создания комментариев, занимающих несколько строк, может быть использована директива COMMENT. Например,

```
Метка Код операции   Операнды     ;Комментарий
MET:     MOVE        AX, BX       ;Пересылка
```

Структура директивы аналогична структуре инструкции. Второе поле – код псевдооперации – определяет смысловое содержание директивы. Как и у инструкции, у директивы есть операнды, причем их может быть один или несколько и они отделяются друг от друга запятыми. Допустимое число операндов в директиве определяется кодом псевдооперации. Например,

```
ARRAY DB 0, 0, 0, 0, 0
END START
```

Директива может быть помечена символическим именем и содержать поле комментария. Символическое имя, стоящее в начале директивы распределения памяти, называется переменной. В отличие от метки команды, после символического имени директивы двоеточие не ставится. Комментарий записывается так же, как в случае команды, и может занимать целую строку.

3.9.1. Стандартные директивы сегментации

Программа на языке Ассемблера может состоять из программных модулей, содержащихся в различных файлах. Каждый модуль, в свою очередь, состоит из инструкций процессора или директив Ассемблера и заканчивается директивой END. Метка, стоящая после кода псевдооперации END, определяет адрес, с которого должно начаться выполнение программы, и называется точкой входа в программу.

Директивы SEGMENT и ENDS

Каждый модуль может разбиваться на отдельные части директивами сегментации, определяющими начало и конец сегмента. Каждый сегмент начинается директивой начала сегмента – SEGMENT и заканчивается директивой конца сегмента – ENDS. В начале директив ставится имя сегмента. Директива SEGMENT может указывать выравнивание сегмента в памяти, способ, которым он объединяется с другими сегментами, а также имя класса. Существует два типа выравнивания сегмента: тип PARA, когда сегмент будет расположен начиная с границы параграфа, и тип BYTE, когда сегмент расположен начиная с любого адреса. Различные виды взаимной связи сегментов определяют параметры сборки, например, при модульном программировании. Объявление PUBLIC вызывает объединение всех сегментов с одним и тем же именем в виде одного большого сегмента. Объявление AT с адресным выражением располагает сегмент по заданному абсолютному адресу.

Каждый сегмент может быть также разбит на части. В общем случае информационные сегменты (адреса которых хранятся в сегментных регистрах SS, ES и DS) состоят из определений данных, а программный сегмент (адрес хранится в сегментном регистре CS) – из инструкций и директив, группирующих инструкции в блоки. Программный сегмент может разбиваться на части (процедуры) директивами определения процедур. Как и для определения сегмента, имеются две директивы определения процедуры (подпрограммы) – директива начала PROC и директива конца ENDP. Процедура имеет имя, которое должно включаться в обе директивы, PROC и ENDP. В сегменте процедуры могут располагаться последовательно одна за другой или могут быть вложенными одна в другую.

Директива ASSUME

С помощью директивы ASSUME Ассемблеру сообщается информация о соответствии между сегментными регистрами и программными сегментами. Директива имеет следующий формат:

```
ASSUME пара [[, <пара>]]
ASSUME NOTHING
```

где пара – это `<сегментный регистр> : <имя сегмента>`

либо `<сегментный регистр> : NOTHING`

Например, директива

```
ASSUME ES:A, DS:B, CS:C
```

сообщает Ассемблеру, что для адреса сегмента A выбирается регистр ES, для адреса сегмента B – регистр DS, а для адреса сегмента C – регистр CS.

В качестве особенностей директивы прежде всего следует отметить, что директива ASSUME не загружает в сегментные регистры начальные адреса сегментов. Этой директивой автор программы лишь сообщает, что в программе будет сделана такая загрузка. Директиву ASSUME можно размещать в любом месте программы, но обычно ее помещают в начале сегмента команд, так как информация из нее нужна только при трансляции инструкций. При этом в директиве обязательно должно быть указано соответствие между регистром CS и именем сегмента кода, иначе при появлении первой же метки Ассемблер зафиксирует ошибку.

Если в директиве ASSUME указано несколько пар с одним и тем же сегментным регистром, то последняя из них «отменяет» предыдущие, так как каждому сегментному регистру можно поставить в соответствие только один сегмент. В то же время на один и тот же сегмент могут указывать разные сегментные регистры. Если в директиве ASSUME в качестве второго элемента пары задано служебное слово NOTHING (ничего), например, ASSUME ES: NOTHING, то это означает, что с данного момента сегментный регистр не указывает ни на какой сегмент, что Ассемблер не должен использовать этот регистр при трансляции команд.

В связи с тем, что директивой ASSUME автор программы лишь сообщает, что все имена из таких-то программных сегментов должны сегментироваться по таким-то сегментным регистрам (в начале выполнения программы в этих регистрах ничего нет), то выполнение программы необходимо начинать с команд, которые загружают в сегментные регистры адреса соответствующих сегментов памяти.

Загрузка производится следующим образом. Пусть в регистр DS необходимо загрузить адрес сегмента B. Для загрузки регистра необходимо выполнить присваивание вида DS:=B. Однако сделать это командой MOV DS,B нельзя, поскольку имя сегмента – это константное выражение, то есть непосредственный операнд, а по команде MOV запрещена пересылка непосредственного операнда в сегментный регистр (см. пп. 4.1.1). Поэтому такую пересылку следует делать через другой, несегментный регистр, например, через AX:

```
MOV AX,B
MOV DS,AX ;DS:=B
```

Аналогичным образом загружается и регистр ES.

Регистр CS загружать нет необходимости, так как к началу выполнения программы этот регистр уже будет указывать на начало сегмента кода.

Такую загрузку выполняет операционная система, прежде чем передает управление программе.

Загрузить регистр SS (адрес сегмента стека) можно двояко. Во-первых, его можно загрузить в самой программе так же, как DS или ES. Во-вторых, такую загрузку можно поручить операционной системе. Для этого в директиве SEGMENT, открывающей описание сегмента стека, надо указать специальный параметр STACK, например,

```
S SEGMENT STACK
...
S ENDS
```

В таком случае загрузка S в регистр SS будет выполнена автоматически до начала выполнения программы.

3.9.2. Упрощенные директивы сегментации

Для простых программ, содержащих по одному сегменту для кода, данных и стека, хотелось бы упростить описание. Для этого в трансляторы MASM и TASM ввели возможность использования упрощенных директив сегментации. Но здесь возникла проблема, связанная с тем, что необходимо было как-то компенсировать невозможность напрямую управлять размещением и комбинированием сегментов. Для этого совместно с упрощенными директивами сегментации стали использовать директиву указания модели памяти MODEL, которая частично стала управлять размещением сегментов и выполнять функции директивы ASSUME (поэтому при использовании упрощенных директив сегментации директиву ASSUME можно не использовать). Эта директива связывает сегменты, которые в случае использования упрощенных директив сегментации имеют предопределенные имена, с сегментными регистрами (хотя явно инициализировать ds все равно придется).

Ниже приведен пример программы с использованием упрощенных директив сегментации.

```
    model small             ;модель памяти
 .data                      ;сегмент данных
message db 'Введите две шестнадцатеричные цифры,$'
.stack                  ;сегмент стека
db 256 dup ('?')
.code                   ;сегмент кода
main     proc           ;начало процедуры main
    mov ax,@data        ;в ax - адрес сегмента данных
    mov ds,ax           ;ax в ds
    . . .               ;текст программы
    mov ax,4c00h        ;Завершение работы программы
    int 21h
main     endp           ;конец процедуры main
end      main           ;точкой входа в программу main
```

Обязательным параметром директивы MODEL является модель памяти. Этот параметр определяет модель сегментации памяти для программного модуля. Предполагается, что программный модуль может иметь только определенные типы сегментов, которые определяются упомянутыми нами ранее упрощенными директивами описания сегментов. Эти директивы приведены в табл. 3.3.

Таблица 3.3

Упрощенные директивы определения сегмента

Формат директивы (режим MASM)	**Формат директивы (режим IDEAL)**	**Назначение**
.CODE [имя]	CODESEG[имя]	Начало или продолжение сегмента кода
.DATA	DATASEG	Начало или продолжение сегмента инициализированных данных. Также используется для определения данных типа near
.CONST	CONST	Начало или продолжение сегмента постоянных данных (констант) модуля
.STACK [размер]	STACK [размер]	Начало или продолжение сегмента стека модуля. Параметр [размер] задает размер стека

Наличие в некоторых директивах параметра **[имя]** говорит о том, что возможно определение нескольких сегментов этого типа.

При использовании директивы MODEL транслятор делает доступными несколько идентификаторов, к которым можно обращаться во время работы программы с тем, чтобы получить информацию о тех или иных характеристиках данной модели памяти (табл. 3.4).

Таблица 3.4

Идентификаторы, создаваемые директивой MODEL

Имя идентификатора	Значение переменной
@code	Физический адрес сегмента кода
@data	Физический адрес сегмента данных типа near
@fardata	Физический адрес сегмента данных типа far
@fardata?	Физический адрес сегмента неинициализированных данных типа far
@stack	Физический адрес сегмента стека

Если вы посмотрите на текст приведенного выше листинга, то увидите пример использования одного из этих идентификаторов. Это @data, с его помощью мы получили значение физического адреса сегмента данных нашей программы.

Теперь можно закончить обсуждение директивы MODEL. Операнд директивы MODEL используют для задания модели памяти, которая определяет набор сегментов программы, размеры сегментов данных и кода, способ связывания сегментов и сегментных регистров. В табл. 3.5 приведены некоторые значения параметра *модель памяти* директивы MODEL.

Таблица 3.5

Модели памяти

Модель	Тип кода	Тип данных	Назначение модели
TINY	near	near	Код и данные объединены в одну группу с именем DGROUP. Используется для создания программ формата .com.
SMALL	near	near	Код занимает один сегмент, данные объединены в одну группу с именем DGROUP. Эту модель обычно используют для большинства программ на Ассемблере

Окончание табл. 3.5

Модель	Тип кода	Тип данных	Назначение модели
MEDIUM	far	near	Код занимает несколько сегментов, по одному на каждый объединяемый программный модуль. Все ссылки на передачу управления – типа far. Данные объединены в одной группе; все ссылки на них – типа near
COMPACT	near	far	Код в одном сегменте; ссылка на данные – типа far
LARGE	far	far	Код в нескольких сегментах, по одному на каждый объединяемый программный модуль

Рассмотренные нами стандартные и упрощенные директивы сегментации не исключают друг друга. Стандартные директивы используются, когда программист желает получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей.

Упрощенные директивы целесообразно использовать для простых программ и программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня. Это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления.

3.9.3. Директива INCLUDE

Встретив директиву INCLUDE, Ассемблер весь текст, хранящийся в указанном файле, вставит в программу вместо этой директивы. В общем случае обращение к директиве INCLUDE (включить) имеет следующий вид:

```
INCLUDE имя_файла
```

Директиву INCLUDE можно указывать любое число раз и в любых местах программы. В ней можно указать любой файл, причем название файла записывается по правилам операционной системы. Директива полезна, когда в разных программах используется один и тот же фрагмент текста; чтобы не выписывать этот фрагмент в каждой программе заново, его записывают в какой-то файл, а затем подключают к программам с помощью данной директивы.

3.9.4. Процедуры

Процедура, часто называемая также подпрограммой, – это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку. В простейшем случае программа может состоять из одной процедуры. Другими словами, процедуру можно определить как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

Для описания последовательности команд в виде процедуры в языке Ассемблера используются две директивы: **PROC** и **ENDP**.

Синтаксис описания процедуры следующий:

```
имя_процедуры PROC [расстояние]
. . .
тело процедуры
 . . .
RET      ;команда возврата из процедуры
имя_процедуры ENDP
```

Из листинга видно, что в заголовке процедуры (директиве **PROC**) обязательным является только задание имени процедуры. Среди большого количества операндов директивы **PROC** следует особо выделить **[расстояние]**. Этот атрибут может принимать значения **near** или **far** и характеризует возможность обращения к процедуре из другого сегмента кода. По умолчанию атрибут **[расстояние]** принимает значение **near** (обращение к процедуре возможно только из того сегмента кода, в котором она описана).

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то микропроцессор будет воспринимать команды процедуры как часть этого потока и, соответственно, будет осуществлять выполнение команд процедуры. Дойдя до команды **RET**, микропроцессор извлечет из стека адрес возврата, который не является таковым, так как он не был помещен в стек, как при корректном вызове процедуры. Далее процессор передаст управление на указанный адрес, что приведет к непредсказуемым последствиям. Подробнее о механизме вызова процедур и возврата из них будет рассказано далее.

3.9.5. Структура EXE- и COM-программ

Следует отметить, что какой-либо фиксированной структуры программы на языке Ассемблера нет, но для небольших EXE-программ с трехсегментной организацией (сегменты кода, данных и стека) типична следующая структура:

```
;Определение сегмента стека
STAK SEGMENT STACK
DB 256 DUP (?)
STAK ENDS
;Определение сегмента данных
DATA SEGMENT
SYMB DB '#'      ;Описание переменной с именем SYMB
                 ;типа Byte и со значением '#'
. . .            ;Определение других переменных
DATA ENDS
;Определение сегмента кода
CODE SEGMENT
ASSUME CS:CODE,DS:DATA,SS:STAK
40
;Определение подпрограммы
PROC1 PROC
. . .            ;Текст подпрограммы
RET              ;Возврат из подпрограммы
PROC1 ENDP
START:           ;Точка входа в программу START
XOR AX,AX
MOV BX,data      ;Обязательная инициализация
MOV DS,BX        ;регистра DS в начале программы
CALL PROC1       ;Пример вызова подпрограммы
. . .            ;Текст программы
MOV AH,4CH       ;Операторы завершения программы
INT 21H
CODE ENDS
END START        ;указание точки входа в программу
```

В общем случае взаимное расположение сегментов программы может быть любым, но рекомендуется сегмент команд размещать в конце текста программы.

Сегмент стека в приведенной структуре описан с параметром **STACK**, поэтому в самой программе нет необходимости загружать сегментный регистр SS. Сегментный регистр CS тоже нет необходимости загружать, как уже отмечалось ранее. В связи с этим в начале программы загружается лишь регистр DS.

Относительно сегмента стека нужно сделать следующее замечание. Даже если сама программа не использует стек, описывать в программе сегмент стека все равно надо. Дело в том, что стек программы используется операционной системой при обработке прерываний и при вызове подпрограмм.

Необходимо также заметить, что все предложения, по которым Ассемблер заносит что-либо в формируемую программу (инструкции, директивы определения данных и т.д.) обязательно должны входить в какой либо программный сегмент, размещать их вне программных сегментов нельзя. Исключение составляют директивы информационного характера, например, директивы EQU, директивы описания типов структур и записей. Кроме того, не рекомендуется размещать в сегменте данных инструкции, а в сегменте кода – описание переменных из-за возникающих в этом случае проблем с сегментированием.

Типичная структура COM-программы аналогична структуре EXE-программы, с той лишь разницей, что, как уже отмечалось выше, COM-программа содержит лишь один сегмент – сегмент кода, который включает в себя инструкции процессора, директивы и описания переменных.

```
;Определение сегмента кода
CODE SEGMENT
ASSUME CS:CODE,DS:CODE,SS:CODE
ORG 100H          ;Начало, необходимое для COM-программы
;Определение подпрограммы
PROC1 PROC
. . .             ;Текст подпрограммы
PROC1 ENDP
START:
. . .             ;Текст программы
```

```
MOV AH,4CH        ;Операторы завершения программы
INT 21H
;===== Data =====
BUF DB 6          ;Определение переменной типа Byte
. . .             ;Определение других переменных
CODE ENDS
END START         указание точки входа в программу
```

3.10. Вопросы и задания для самопроверки

1. Что в языке Ассемблера называется переменными, метками и именами?
2. Может ли идентификатор начинаться с цифры?
3. Чувствителен ли Ассемблер к регистру идентификаторов?
4. Какие типы данных языка Ассемблера вам известны?
5. Корректна ли запись шестнадцатеричного числа B8c2h?
6. Что представляют собой предложения в языке Ассемблера?
7. Какие вы знаете основные операторы языка Ассемблера?
8. Какие виды операндов вам известны?
9. Перечислите виды адресации операндов в памяти.
10. Какие существую директивы определения скалярных данных?
11. Опишите особенности использования директивы ASSUME в ассемблерных программах.
12. Каково назначение директивы INCLUDE?
13. Опишите структуру EXE-программы на языке Ассемблера.
14. Опишите структуру COM-программы на языке Ассемблера.
15. В каких случаях используются упрощенные директивы сегментации?

МОДУЛЬ 4. КОМАНДЫ МИКРОПРОЦЕССОРА I80486

Цель модуля – изучение основных команд микропроцессора i80486.

В результате изучения модуля студенты должны:

- знать общую классификацию команд микропроцессора;
- знать классификацию команд пересылки данных и алгоритмы их работы;
- знать набор основных арифметических команд, особенности их работы, а также способы обработки чисел большой размерности;
- знать классификацию команд побитовой обработки данных, иметь представление об алгоритме работы каждой команды;
- знать набор команд, предназначенных для сравнения и передачи управления, и алгоритмы их работы;
- знать способы вызова процедур и прерываний;
- знать различные способы организации циклов;
- знать группы цепочечных команд и алгоритмы их работы.

Содержание модуля

В рамках данного модуля предусмотрено выполнение лабораторной работы № 2 «Программирование текстового видеобуфера. Использование переменных. Организация циклов» (Методические указания к выполнению лабораторных работ по курсу «Низкоуровневое программирование», сост. Д. Г. Руголь)

4.1. Команды пересылки данных

4.1.1. Команда MOV

Команда MOV – основная команда пересылки данных, которая пересылает один байт или слово данных из памяти в регистр, из регистра в память или из регистра в регистр. Команда MOV может также занести число (непосредственный операнд) в регистр или в память. В действительности команда MOV – это целое семейство машинных команд микропроцессора. На рис. 4.1 представлены различные способы, которыми в микропроцессоре можно пересылать данные из одного места в другое. Каждый прямоугольник означает здесь регистр или ячейку памяти. Стрелки показывают пути пересылки данных, которые допускает микропроцессор. Необходимо помнить, что все команды микропроцессора могут указывать только один операнд памяти.

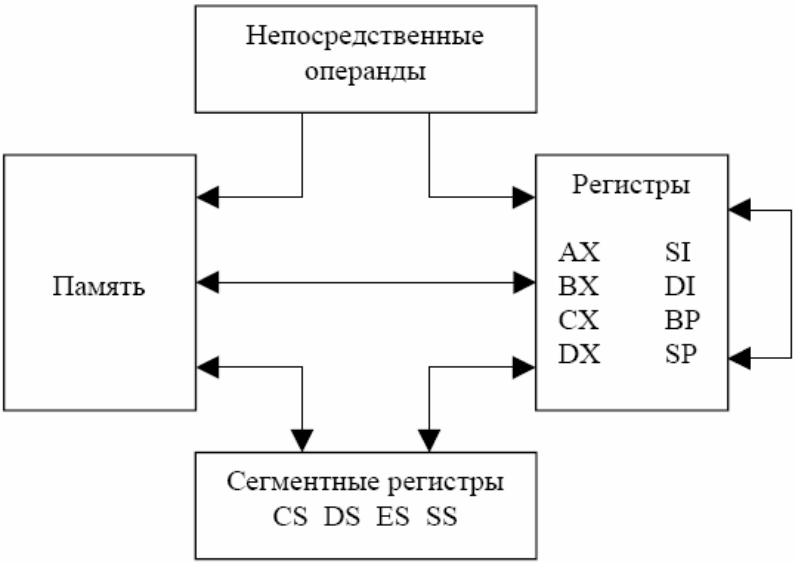

Рис. 4.1. Комбинации операндов команды пересылки MOV

Из рисунка видно, что при использовании команды MOV следует учитывать следующие ограничения:

- командой **mov** нельзя осуществить пересылку из одной области памяти в другую. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения. Например, рассмотрим фрагмент программы для пересылки байта из ячейки fls в ячейку fld:

```
masm
model small
.data
fls db 5
fld db ?
.code
start:
...
mov al,fls
mov fld,al
...
end start
```

- нельзя загрузить в сегментный регистр непосредственное значение. Поэтому для выполнения такой загрузки нужно использовать промежуточный объект. Это может быть регистр общего назначения или стек;
- нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных все те же регистры общего назначения. Вот пример инициализации регистра es значением из регистра ds:

```
mov ax,ds
mov es,ax
```

Но есть и другой, более красивый способ выполнения данной операции: использование стека и команд PUSH и POP (см. ниже):

```
push ds      ;поместить значение регистра ds в стек
pop es       ;извлечь число из стека в регистр es
```

- нельзя использовать сегментный регистр cs в качестве операнда назначения. Причина здесь простая. Дело в том, что в архитектуре микропроцессора пара cs:ip всегда содержит адрес команды, которая должна выполняться следующей. Изменение командой MOV содержимого регистра cs фактически означало бы операцию перехода, а не пересылки, что недопустимо.

4.1.2. Команда обмена данных XCHG

Команда XCHG меняет местами содержимое двух операндов. Порядок следования операндов не имеет значения. В качестве операндов могут выступать регистры (кроме сегментных) и ячейки памяти.

Примеры использования команды XCHG:

```
XCHG BL,BH   ;Обменять содержимое регистров bl и bh
XCHG DH,Char ;Обменять содержимое dh и переменной Char
XCHG AX,BX   ;Обменять содержимое регистров ax и bx
```

4.1.3. Команды работы с адресами и указателями памяти

При написании программ на Ассемблере производится интенсивная работа с адресами операндов, находящимися в памяти. Для поддержки такого рода операций есть специальная группа команд, в которую входят следующие команды:

- `lea` *назначение*,*источник* – загрузка эффективного адреса;
- `lds` *назначение*,*источник* – загрузка указателя в регистр сегмента данных ds;
- `les` *назначение*,*источник* – загрузка указателя в регистр дополнительного сегмента данных es;
- `lgs` *назначение*,*источник* – загрузка указателя в регистр дополнительного сегмента данных gs;
- `lfs` *назначение*,*источник* – загрузка указателя в регистр дополнительного сегмента данных fs;
- `lss` *назначение*,*источник* – загрузка указателя в регистр сегмента стека ss.

Команда **`lea`** похожа на команду **`mov`** тем, что она также производит пересылку. Однако обратите внимание, команда **`lea`** производит пересылку не данных, а эффективного адреса данных (то есть смещения данных относительно начала сегмента данных) в регистр, указанный операндом назначения.

Часто для выполнения некоторых действий в программе недостаточно знать значение одного лишь эффективного адреса данных, а необходимо иметь полный указатель на данные (вспомним, что полный указатель состоит из сегментной составляющей и смещения).

Все остальные команды этой группы позволяют получить в паре регистров полный указатель на операнд в памяти. При этом имя сегментного регистра, в который помещается сегментная составляющая адреса, определяется кодом операции. Соответственно, смещение помещается в регистр общего назначения, указанный операндом назначения.

Но не все так просто с операндом-источником. На самом деле, в команде в качестве источника нельзя указывать непосредственно имя операнда в памяти, на который мы бы хотели получить указатель.

Предварительно необходимо получить само значение полного указателя в некоторой области памяти и указать в команде получения полного адреса имя этой области. Для выполнения этого действия необходимо вспомнить

директивы резервирования и инициализации памяти. При применении этих директив возможен частный случай, когда в поле операндов указывается имя другой директивы определения данных (фактически, имя переменной). В этом случае в памяти формируется адрес этой переменной. Какой адрес будет сформирован (эффективный или полный), зависит от применяемой директивы. Если это **dw**, то в памяти формируется только 16-битное значение эффективного адреса, если же **dd** – в память записывается полный адрес. Размещение этого адреса в памяти следующее: в младшем слове находится смещение, в старшем – 16-битная сегментная составляющая адреса.

Приведем пример, в котором производится копирование строки байт str_1 в строку байт str_2:

```
<1>;---------Prg_.asm--------------
<2>masm
<3>model   small
<4>.data
<5>...
<6>str_1 db ‘Ассемблер — базовый язык компьютера’
<7>str_2   db      50 dup (‘ ‘)
<8>full_pnt        dd str_1
<9>...
<10>.code
<11>start:
<12>...
<13>lea si,str_1
<14>lea di,str_2
<15>les bx,full_pnt ;полный указатель на str1 в es:bx
<16>m1:
<17>mov al,[si]
<18>mov [di],al
<19>inc si
<20>inc di
<21>;цикл на метку m1 до пересылки всех символов
<22>...
<23>end start
```

В строках 13 и 14 в регистры si и di загружаются значения эффективных адресов переменных str_1 и str_2. В строках 17 и 18 производится пересылка очередного байта из одной строки в другую. Указатели на позиции байтов в строках определяются содержимым регистров si и di. Для пересылки очередного байта необходимо увеличить на единицу регистры si и di, что и делается командами инкрементирования **inc** (строки 19, 20). После этого программу необходимо зациклить до обработки всех символов строки.

В строке 8 в двойном слове full_pnt формируются сегментная часть адреса и смещение для переменной str_1. При этом 2 байта смещения занимают младшее слово full_pnt, а значение сегментной составляющей адреса – старшее слово full_pnt. В строке 15 командой **les** эти компоненты адреса помещаются в регистры BX (смещение) и ES (сегмент).

4.1.4. Команда перекодировки XLAT

Команда **xlat** заменяет содержимое регистра AL байтом из таблицы перекодировки (максимальная длина – 256 байт), начальный адрес которой относительно сегмента DS находится в регистре BX.

Алгоритм выполнения команды **xlat** состоит из двух этапов:

- содержимое регистра AL прибавляется к содержимому регистра BX;
- полученный результат рассматривается как смещение относительно регистра DS. По данному адресу выбирается байт и помещается в регистр AL.

Команда **xlat** всегда использует в качестве смещения начала таблицы содержимое регистра BX, поэтому перед ее выполнением необходимо поместить в BX смещение таблицы. Пример использования команды **xlat**:

```
MOV BX,OFFSET Talbe
MOV AL,2
XLAT          ;в al будет помещен код символа "c"
...
Table DB 'abcde'
```

4.1.5. Команды работы со стеком

Эта группа представляет собой набор специализированных команд, ориентированных на организацию гибкой и эффективной работы со стеком.

Стек – это область памяти, специально выделяемая для временного хранения данных программы. Важность стека определяется тем, что для него

в структуре программы предусмотрен отдельный сегмент. На тот случай, если программист забыл описать сегмент стека в своей программе, компоновщик tlink выдаст предупреждающее сообщение.

Размер стека зависит от режима работы микропроцессора и ограничивается 64 Кбайт (или 4 Гбайт в защищенном режиме). В каждый момент времени доступен только один стек, адрес сегмента которого содержится в регистре SS. Этот стек называется текущим. Для того чтобы обратиться к другому стеку («переключить стек»), необходимо загрузить в регистр SS другой адрес. Регистр SS автоматически используется процессором для выполнения всех команд, работающих со стеком.

Перечислим еще некоторые особенности работы со стеком:

- запись и чтение данных в стеке осуществляется в соответствии с принципом LIFO (Last In First Out – «последним пришел, первым ушел»);
- по мере записи данных в стек последний растет в сторону младших адресов. Эта особенность заложена в алгоритм команд работы со стеком;
- при использовании регистров SP и BP для адресации памяти Ассемблер автоматически считает, что содержащиеся в нем значения представляют собой смещения относительно сегментного регистра SS.

В общем случае стек организован так, как показано на рис. 4.2.

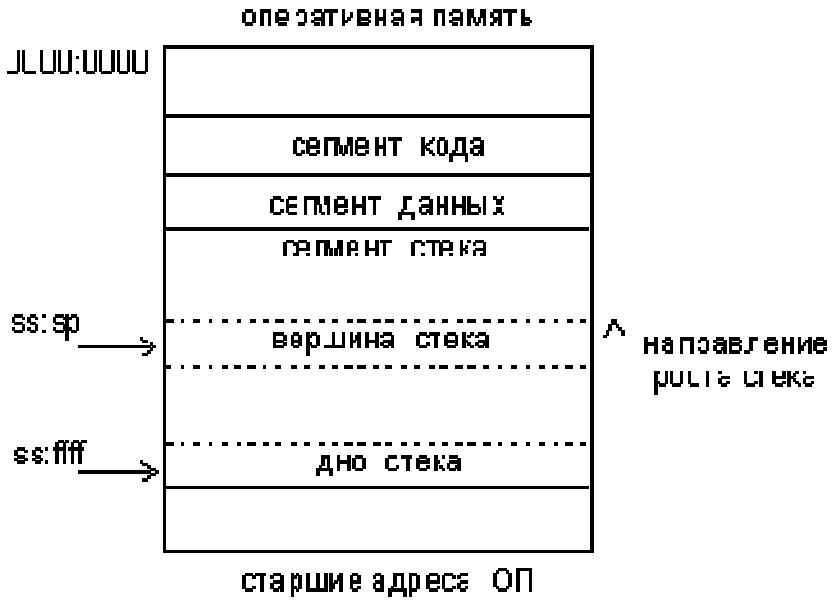

Рис. 4.2. Организация стека

Вспомним, что для работы со стеком предназначены три регистра:

- SS – сегментный регистр стека;
- SP – регистр указателя стека;
- BP – регистр указателя базы кадра стека.

Эти регистры используются комплексно, и каждый из них имеет свое функциональное назначение. Регистр SP всегда указывает на вершину стека, то есть содержит смещение, по которому в стек был занесен последний элемент. Команды работы со стеком неявно изменяют этот регистр так, чтобы он указывал всегда на последний записанный в стек элемент. Если стек пуст, то значение SP равно адресу последнего байта сегмента, выделенного под стек.

При занесении элемента в стек процессор уменьшает значение регистра SP, а затем записывает элемент по адресу новой вершины.

При извлечении данных из стека процессор копирует элемент, расположенный по адресу вершины, а затем увеличивает значение регистра указателя стека SP.

Таким образом, получается, что стек растет вниз, в сторону уменьшения адресов.

Для произвольного доступа к данным не на вершине, а внутри стека применяют регистр BP – регистр указателя базы кадра стека. Например, типичным приемом при входе в подпрограмму является передача нужных параметров путем записи их в стек. Если подпрограмма тоже активно работает со стеком, то доступ к этим параметрам становится проблематичным. Выход в том, чтобы после записи нужных данных в стек сохранить адрес вершины стека в указателе кадра (базы) стека – регистре BP. Значение в регистре BP в дальнейшем можно использовать для доступа к переданным параметрам.

Начало стека расположено в старших адресах памяти. На рис. 4.2 этот адрес обозначен парой SS:0FFFFh. Смещение 0FFFFh приведено здесь условно. Реально это значение определяется величиной, которую программист задает при описании сегмента стека в своей программе. Например, для стека размером 256 байт началу стека будет соответствовать пара SS:0100h. Адресная пара SS: 0FFFFh – это максимальное для реального режима работы процессора значение адреса начала стека, так как размер сегмента в нем ограничен величиной 64 Кбайт (0FFFF).

Для организации работы со стеком существуют специальные команды записи и чтения.

push источник – запись значения **источник** в вершину стека. Алгоритм работы этой команды включает следующие действия:

- (SP) = (SP) – 2; значение SP уменьшается на 2;
- значение из источника записывается по адресу, указываемому парой SS:SP;

pop назначение – запись значения из вершины стека по месту, указанному операндом **назначение**. Значение при этом «снимается» с вершины стека. Алгоритм работы команды pop обратен алгоритму команды push:

- запись содержимого вершины стека по месту, указанному операндом **назначение**;
- (SP) = (SP) + 2 – увеличение значения SP;

pusha – команда групповой записи в стек. По этой команде в стек последовательно записываются регистры ax, cx, dx, bx, sp, bp, si, di. Заметим, что записывается оригинальное содержимое sp, то есть то, которое было до выдачи команды **pusha**;

popa – команда группового чтения из стека. Алгоритм работы обратен алгоритму команды **pusha**;

pushf – сохраняет регистр флагов в стеке;

popf – извлекает значение из стека и помещает его в регистр флагов.

Отметим основные виды операций, в которых интенсивно используется стек:

- вызов подпрограмм;
- временное сохранение значений регистров;
- определение локальных переменных.

4.1.6. Команды ввода – вывода в порт

Все устройства ЭВМ принято делить на внутренние (центральный процессор ЦП, оперативная память ОП) и внешние (внешняя память, клавиатура, дисплей и т.д.). Под вводом – выводом понимается обмен информацией между ЦП и любым внешним устройством. В ЭВМ передача информации между ЦП и внешним устройством, как правило, осуществляется через порты. ***Порт*** – некоторый регистр, находящийся вне ЦП (в адресном пространстве ввода – вывода). Обращение к портам происходит по номерам. Все порты нумеруются от 0 до 0FFFFh. С каждым внешним устройством связан свой порт или несколько портов, адреса которых заранее известны.

Запись и чтение порта осуществляется при помощи следующих команд:

```
Разрядность порта:    байт          слово
Чтение (ввод):        IN AL, n      IN AX, n
Запись (вывод):       OUT n, AL     OUT n, AX
```

Номер порта n в этих командах может быть задан либо непосредственно, либо регистром DX (IN AX,DX).

Сценарий ввода – вывода через порты существенно зависит от специфики того внешнего устройства, с которым ведется обмен, но обычно ЦП связан с внешним устройством через два порта: первый – порт данных, второй – порт управления, и достаточно типичной является следующая процедура обмена:

- ЦП записывает в порт управления соответствующую команду, а в порт данных – выводимые данные;
- внешнее устройство, считав эту информацию, записывает в порт управления команду «занято» и начинает непосредственно вывод (например, печать);
- ЦП переходит либо в режим ожидания, опрашивая в цикле порт управления, либо занимается другой работой – до тех пор, пока в порте управления не снимется сигнал «занято»;
- внешнее устройство заканчивает вывод и записывает в порт управления сигнал об успешном завершении или об ошибке;
- ЦП анализирует полученную информацию и продолжает свою работу.

4.2. Арифметические команды

Все арифметические команды устанавливают флаги CF, AF, SF, ZF,OF и PF в зависимости от результата операции.

Двоичные числа могут иметь длину 8 и 16 бит. Значение старшего (самого левого бита) задает знак числа: 0 – положительное, 1 – отрицательное. Отрицательные числа представляются в так называемом дополнительном коде, в котором для получения отрицательного числа необходимо инвертировать все биты положительного числа и прибавить к нему единицу. Например,

```
Положительное:    24=18h=  00011000b
Инверсное:                 11100111b
Отрицательное:             11101000b =E8h=-24
Проверка:         24-24=0  00011000b
                           11101000b
                       (1) 00000000b
```

4.2.1. Команды арифметического сложения ADD и ADC

Формат команд:

```
ADD операнд_1,операнд_2
ADD операнд_1,операнд_2
```

Команда ADD выполняет целочисленное сложение двух операндов, представленных в двоичном коде. Результат помещается на место первого операнда, второй операнд не изменяется. Команда корректирует регистр флагов в соответствии с результатом сложения. В частности, она устанавливает флаг CF, если произошел перенос в старшие разряды, и флаг ZF, если в результате получился 0.

Существуют две формы сложения: 8-битовое и 16-битовое. В различных формах сложения принимают участие различные регистры. Компилятор следит за тем, чтобы операнды соответствовали друг другу. На рис. 4.3 иллюстрируются допустимые варианты команды ADD.

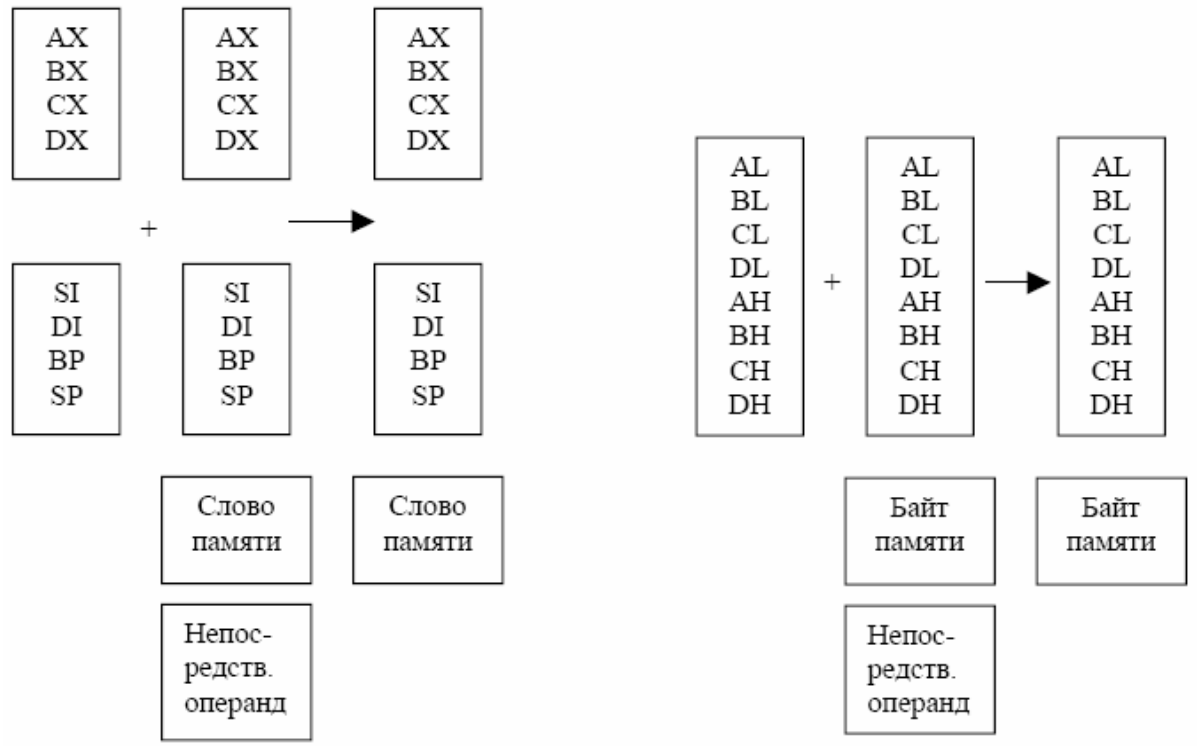

Рис. 4.3. 16-битовое сложение (слева) и 8-битовое (справа)

Команда сложения с переносом ADC – это та же команда ADD, за исключением того, что в сумму включается флаг переноса CF, который прибавляется к младшему биту результата. Для любой формы команды ADD существует аналогичная ей команда ADC. Команда ADC часто используется для сложения чисел большой размерности.

Примеры использования команд ADD и ADC:

```
ADD AL,12h   ;увеличение al на 12h
ADD Count,1  ;увеличение Count на 1
ADC BX,4     ;увеличение bx на 4 + значение CF
ADD AX,BX    ;прибавление к ax содержимого bx
ADC Count,DI ;увеличение Count на содержимое di + CF
```

4.2.2. Команды арифметического вычитания SUB и SBB

Формат команд:

```
SUB операнд_1,операнд_2
SBB операнд_1,операнд_2
```

Команда вычитания SUB идентична команде сложения, за исключением того, что она выполняет вычитание, а не сложение. Для нее верны предыдущие схемы, если в них поменять знак «+» на «–», то есть она из первого операнда вычитает второй и помещает результат на место первого операнда. Команда вычитания также устанавливает флаги состояния в соответствии с результатом операции (флаг переноса здесь трактуется как заем). В частности, она устанавливает флаг CF, если произошел заем из старших разрядов, и флаг ZF, если в результате получился 0.

Команда вычитания с заемом SBB учитывает флаг заема CF, то есть значение заема вычитается из младшего бита результата.

Примеры использования команд SUB и SBB:

```
SUB AL,12h   ;уменьшение al на 12
SUB Count,1  ;уменьшение Count на 1
SBB BX,4     ;уменьшение bx на (4 + CF)
SUB AX,BX    ;ax = ax - bx
SBB Count,DI ;Count = Count - (di + CF)
```

4.2.3. Команда смены знака NEG

Формат команды:

NEG операнд

Команда выполняет инвертирование значения **операнд**. Физически команда выполняет одно действие: операнд = 0 – операнд, то есть вычитает операнд из нуля. Команду **NEG** можно применять:

- для смены знака;
- для выполнения вычитания из константы.

Дело в том, что команды SUB и SBB не позволяют вычесть что-либо из константы, так как константа не может служить операндом-приемником в этих операциях. Поэтому данную операцию можно выполнить с помощью двух команд:

```
neg ax ;смена знака (ax)
...
add ax,340 ;фактически вычитание: (ax)=340-(ax)
```

4.2.4. Команды инкремента INC и декремента DEC

Формат команд:

INC операнд
DEC операнд

Команды инкремента и декремента изменяют значение своего единственного операнда на единицу. Команда INC прибавляет единицу к значению операнда, а команда DEC вычитает единицу из значения операнда.

Обе команды могут работать с байтами или со словами. Следует учитывать тот факт, что на флаги команды влияния не оказывают!

4.2.5. Команды умножения MUL и IMUL

Формат команд:

MUL множитель
IMUL множитель_1

Существуют две формы команды умножения. По команде MUL умножаются два целых числа без знака, при этом результат тоже не имеет знака. По команде IMUL умножаются целые числа со знаком. Обе команды работают с байтами и со словами, но для этих команд диапазон форм представления гораздо уже, чем для команд сложения и вычитания. На рис. 4.4 представлены все варианты команд умножения.

При умножении 8-битовых операндов результат всегда помещается в регистр AX. При умножении 16-битовых данных результат, который может быть длиною до 32 бит, помещается в пару регистров: в регистре DX содержатся старшие 16 бит, а в регистре AX – младшие 16 бит. Умножение не допускает непосредственного задания операнда.

Установкой *флагов* команда умножения отличается от других арифметических команд. Единственно имеющие смысл флаги – это флаг переноса CF и переполнения OF.

Команда MUL устанавливает оба флага, если старшая половина результата не нулевая. Если умножаются два байта, установка флагов переполнения и переноса показывает, что результат умножения больше 255 и не может содержаться в одном байте. В случае умножения слов флаги устанавливаются, если результат больше 65535.

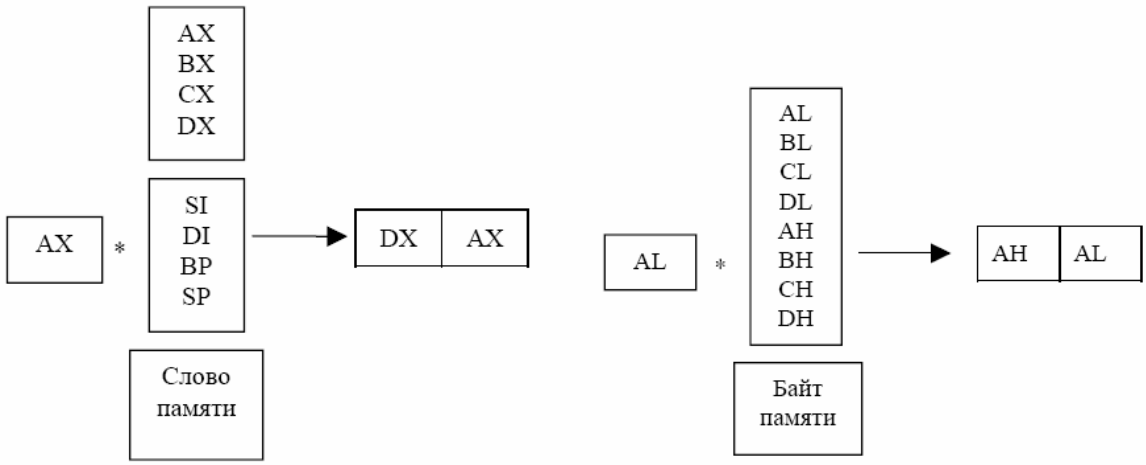

Рис. 4.4. Умножение слов (слева) и байтов (справа)

Команда IMUL устанавливает флаги по тому же принципу, то есть если произведение не может быть представлено в младшей половине результата, но только в том случае, если старшая часть результата не является расшире-

нием знака младшей. Это означает, что если результат положителен, флаг устанавливается как в случае команды MUL. Если результат отрицателен, то флаги устанавливаются в случае, если не все биты, кроме старшего, равны 1. Например, умножение байт с отрицательным результатом устанавливает флаги, если результат меньше 128.

Примеры использования команд умножения:

```
MUL CX          ;Умножение AX на CX, результат в DX:AX
IMUL Width      ;Умножение AX или Al на Width в
                ;зависимости от размера Width
```

4.2.6. Команды деления DIV и IDIV

Формат команд:

```
DIV делитель
IDIV делитель
```

Как и в случае умножения, существуют две формы деления – одна для двоичных чисел без знака DIV, а вторая – для чисел со знаком – IDIV. Любая форма деления может работать с байтами и словами. Один из операндов (делимое) всегда в два раза длиннее обычного операнда (делителя). На рис. 4.5 приведены схемы, иллюстрирующие команды деления.

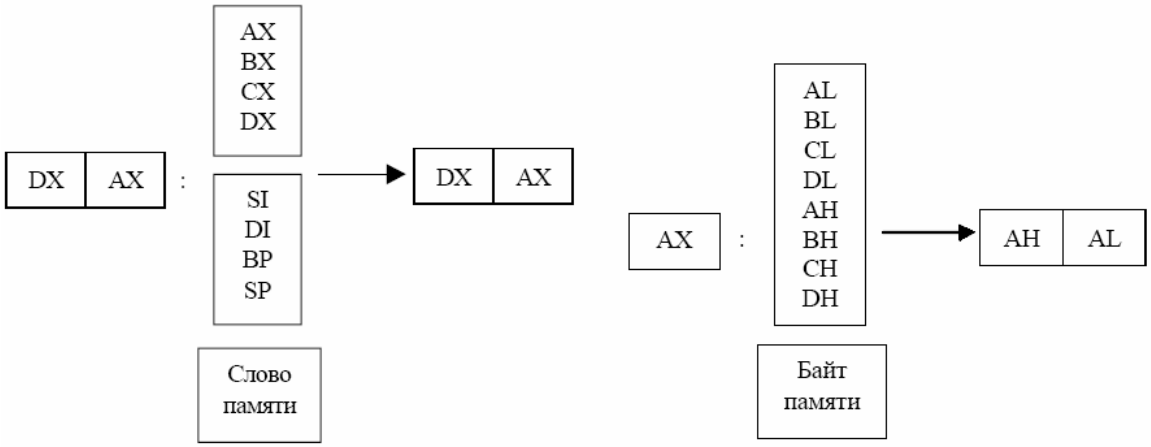

Рис. 4.5. Деление слов (слева) и байтов (справа)

Байтовая команда делит 16-битовое делимое на 8-битовый делитель. В результате деления получается два числа: частное помещается в регистр AL, а остаток – в AH. Команда, работающая со словами, делит 32-битовое делимое на 16-битовый делитель. Делимое находится в паре регистров DX:AX, причем регистр DX содержит старшую значимую часть, а регистр AX – младшую. Команда деления помещает частное в регистр AX, а остаток в DX.

Ни один из флагов состояния не определен после команды деления. Однако, если частное больше того, что может быть помещено в регистр результата (255 для байтового деления и 65535 для деления слов), возникает ошибка значимости и выполняется программное прерывание уровня 0 (о работе системы прерываний будет рассказано ниже).

Примеры использования команд деления:

```
IDIV CX      ;Деление DX:AX на CX, целая часть
             ;результата в AX, остаток в DX
DIV Count    ;Деление DX:AX либо AX на Count, в
             ;зависимости от размера Count
```

Рассмотрим пример программы, использующей большинство из приведенных выше команд.

Пример. Вычислить значение арифметического выражения. Все числа являются 16-битовыми целыми со знаком. Формула вычислений следующая:

$$x = 1 - \frac{A \cdot 2 + B \cdot C}{D - 3}.$$

Эту задачу решает приведенная ниже программа.

```
;Сегмент стека
SSEG SEGMENT STACK
DB 256 DUP(?)
SSEG ENDS
;Сегмент данных
DATA SEGMENT
X DW ? ;Память для переменных
A DW ?
```

```
B DW ?
C DW ?
D DW ?
DATA ENDS
;Сегмент кода
CODE SEGMENT
ASSUME CS:CODE, DS:DATA, SS:SSEG
START:
MOV AX,Data  ;Инициализация DS
MOV DS,AX
;Вычислительная часть
MOV AX,2          ;Загрузка константы
IMUL A       ;dx:ax = a*2
MOV BX,DX
MOV CX,AX    ;bx:cx = a*2
MOV AX,B
IMUL C       ;dx:ax = b*c
ADD AX,CX
ADC DX,BX    ;dx:ax = a*2+b*c
MOV CX,D
SUB CX,3          ;cx = d-3
IDIV CX      ;ax = (a*2+b*c)/(d-3)
NEG AX       ;ax = -ax
INC AX       ;ax = ax+1
MOV X,AX          ;Сохранение результата
MOV AH,4CH   ;завершение программы
INT 21H
CODE ENDS
END START         ;точка входа в программу
```

На первом этапе программа выполняет два умножения. Так как результат умножения всегда помещается в пару регистров DX:AX, то в примере результат первого умножения переносится в пару регистров BX:CX перед выполнением второго умножения. Затем программа выполняет сложение в чис-

лителе. Поскольку умножение дает 32-битовые результаты, в программе требуется сложение повышенной разрядности (с учетом флага переноса). После сложения результат остается в DX:AX (числитель). Знаменатель вычисляется в регистре CX, а затем на него делится числитель. Частное записывается в регистр AX, затем его знак меняется на обратный и к полученному значению прибавляется 1. На последнем этапе программа записывает результат из регистра AX в переменную X. Остаток игнорируется.

4.3. Команды побитовой обработки данных

Наряду со средствами арифметических вычислений система команд микропроцессора имеет также средства логического преобразования данных. Под логическими понимаются такие преобразования данных, в основе которых лежат правила формальной логики.

Формальная логика работает на уровне утверждений «истинно» и «ложно». Для микропроцессора это, как правило, означает 1 и 0 соответственно. Для компьютера язык нулей и единиц является родным, но минимальной единицей данных, с которой работают машинные команды, является байт. Однако на системном уровне часто необходимо иметь возможность работать на предельно низком уровне – на уровне бит. На рис. 4.6 приведена классификация команд побитовой обработки данных.

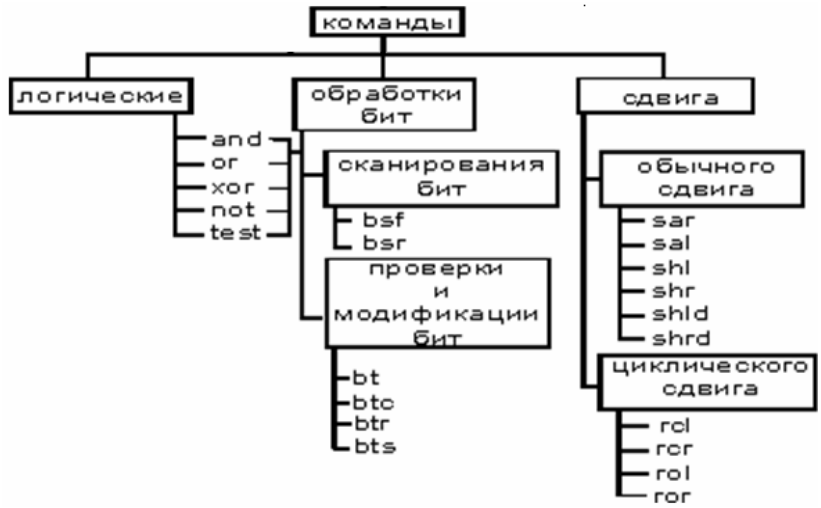

Рис. 4.6. Классификация команд побитовой обработки данных

4.3.1. Логические команды

Над битами информации могут выполняться следующие логические операции:

- Ø отрицание (логическое НЕ);
- Ø логическое сложение (логическое включающее ИЛИ);
- Ø логическое умножение (логическое И);
- Ø логическое исключающее сложение (логическое исключающее ИЛИ).

Отрицание – логическая операция над одним операндом, результатом которой является величина, обратная значению исходного операнда.

Логическое сложение – логическая операция над двумя операндами, результатом которой является «истина» (1), если один или оба операнда имеют значение «истина» (1), и «ложь» (0) – если оба операнда имеют значение «ложь» (0). Эта операция описывается таблицей истинности (табл. 4.1).

Таблица 4.1

Таблица истинности операции логического ИЛИ

Операнд 1	Операнд 2	Результат
0	0	0
0	1	1
1	0	1
1	1	1

Логическое умножение – логическая операция над двумя операндами, результатом которой является «истина» только в том случае, если оба операнда имеют значение «истина». Во всех остальных случаях значение операции «ложь» (табл. 4.2).

Таблица 4.2

Таблица истинности операции логического И

Операнд 1	Операнд 2	Результат
0	0	0
0	1	0
1	0	0
1	1	1

Логическое исключающее сложение – логическая операция над двумя операндами, результатом которой является «истина», если только один из двух операндов имеет значение «истина», и «ложь», если оба операнда имеют значение «ложь» или «истина». Эта операция описывается в табл. 4.3.

Таблица 4.3

Таблица истинности операции логического исключающего ИЛИ

Операнд 1	Операнд 2	Результат
0	0	0
0	1	1
1	0	1
1	1	0

Размерность операндов при выполнении логических операций, естественно, должна быть одинакова. Логическая операция выполняется для каждой пары бит операндов, имеющих одинаковые индексы. Пример реализации операции логического исключающего ИЛИ:

```
Операнд 1:      01011010
Операнд 2:      11110000
Результат:      10101010
```

Система команд микропроцессора содержит пять команд, поддерживающих логические операции:

1. **and операнд_1,операнд_2** – команда выполняет поразрядно логическую операцию И над битами операндов. Результат записывается на место первого операнда.

2. **or операнд_1,операнд_2** – команда выполняет поразрядно логическую операцию ИЛИ над битами операндов. Результат записывается на место первого операнда.

3. **xor операнд_1,операнд_2** – команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов. Результат записывается на место первого операнда.

4. **test операнд_1,операнд_2** – операция «проверить» (способом логического умножения). Команда выполняет поразрядно логическую операцию И над битами операндов. Состояние операндов остается прежним, изменяются только флаги ZF, SF, и PF, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

5. **not операнд** – операция логического отрицания. Команда выполняет поразрядное инвертирование каждого бита операнда. Результат записывается на место операнда.

С помощью логических команд возможно выделение отдельных бит в операнде с целью их установки, сброса, инвертирования или просто проверки на определенное значение. Для организации подобной работы с битами второй операнд обычно играет роль маски. С помощью установленных в 1 бит этой маски и определяются нужные для конкретной операции биты **операнд_1**.

Для установки определенных разрядов (бит) в 1 применяется команда **or.** В этой команде второй операнд, исполняющий роль маски, должен содержать единичные биты на месте тех разрядов, которые должны быть установлены в 1 в первом операнде. Пример:

```
or ax,10b          ;установить 1-й бит в регистре ax
```

Для сброса определенных разрядов (бит) в 0 применяется команда **and.** В этой команде второй операнд, исполняющий роль маски, должен содержать нулевые биты на месте тех разрядов, которые должны быть установлены в 0 в первом операнде. Пример:

```
and  ax,fffeh      ;сбросить в 0 0-й бит в регистре ax
```

Команда **xor операнд_1,операнд_2** применяется: 1) для выяснения того, какие биты в операнд_1 и операнд_2 различаются; 2) для инвертирования состояния заданных бит в первом операнде. Пример:

```
xor ax,10b ;инвертировать 1-й бит в регистре ax
jz mes     ;переход, если 1-й бит в al был единичным
```

Интересующие нас биты маски (операнд_2) при выполнении команды **xor** должны быть единичными, остальные – нулевыми.

Для проверки состояния заданных бит применяется команда **test операнд_1,операнд_2** (проверить операнд_1). Проверяемые биты операнд_1 в маске (операнд_2) должны иметь единичное значение. Алгоритм работы команды **test** подобен алгоритму команды **and**, но он не меняет значения первого операнда. Результатом команды является установка значения флага нуля ZF:

Ø если ZF = 0, то в результате логического умножения получился нулевой результат, то есть ни один единичный бит маски не совпал с соответствующим единичным битом первого операнда;

Ø если ZF = 1, то в результате логического умножения получился ненулевой результат, то есть хотя бы один единичный бит маски совпал с соответствующим единичным битом первого операнда.

Пример:

```
test al,00010000b ;проверить состояние 4-го
jz  m1            ;переход на m1, если 4-й бит равен 1
```

Как видно из примера, для реакции на результат команды **test** целесообразно использовать команду перехода **jnz метка** (Jump if Not Zero) – переход, если флаг нуля ZF ненулевой, или команду с обратным действием – jz метка (Jump if Zero) – переход, если флаг нуля ZF = 0.

4.3.2. Команды поиска и установки бит

Следующие две команды позволяют осуществить поиск первого установленного в 1 бита операнда. Поиск можно произвести как с начала, так и от конца операнда:

Ø bsf операнд_1,операнд_2 (Bit Scan Forward) – сканирование битов вперед. Команда просматривает (сканирует) биты второго операнда от младшего к старшему в поисках первого бита, установленного в 1. Если таковой обнаруживается, в первый операнд заносится номер этого бита в виде целочисленного значения. Если все биты второго операнда равны 0, то флаг нуля ZF устанавливается в 1, в противном случае флаг ZF сбрасывается в 0.

Пример:

```
mov al,02h
bsf bx,al    ;bx=1
jz m1        ;переход, если al=00h
```

Ø bsr операнд_1,операнд_2 (Bit Scaning Reverse) – сканирование битов в обратном порядке. Команда просматривает (сканирует) биты второго операнда от старшего к младшему в поисках первого бита, установленного в 1. Если таковой обнаруживается, в первый операнд заносится номер этого бита в виде целочисленного значения. При этом важно, что позиция первого единичного бита слева отсчитывается все равно относительно бита 0. Если все биты второго операнда равны 0, то флаг нуля ZF устанавливается в 1, в противном случае флаг ZF сбрасывается в 0.

В наборе команд микропроцессора присутствуют также команды, которые позволяют осуществить доступ к одному конкретному биту операнда. Операнд может находиться как в памяти, так и в регистре общего назначения. Положение бита задается его индексом. Положение может задаваться как в виде непосредственного значения, так и содержаться в регистре общего назначения. В качестве индекса возможно использование результатов работы команд bsr и bsf. Все команды присваивают значение выбранного бита флагу CF.

Набор команд для поиска и установки бит:

- **bt операнд,смещение_бита** (Bit Test) – проверка бита. Команда переносит значение бита в флаг CF. Пример:

```
bt ax,5       ;проверить значение бита 5
jnc m1        ;переход, если бит = 0
```

- **bts операнд,смещение_бита** (Bit Test and Set) – проверка и установка бита. Команда переносит значение бита в флаг CF и затем устанавливает проверяемый бит в 1. Пример:

```
mov ax,10
bts pole,ax ;проверить и установить 10-й бит в pole
jc m1       ;переход, если проверяемый бит был равен 1
```

- **btr операнд,смещение_бита** (Bit Test and Reset) – проверка и сброс бита. Команда переносит значение бита в флаг CF и затем устанавливает этот бит в 0.
- **btc операнд,смещение_бита** (Bit Test and Convert) – проверка и инвертирование бита. Команда переносит значение бита в флаг CF и затем инвертирует значение этого бита.

4.3.3. Команды сдвига

Все команды сдвига перемещают биты в поле операнда влево или вправо в зависимости от кода операции. Все команды сдвига имеют одинаковый формат:

имя_команды операнд,счетчик_сдвигов

Количество сдвигаемых разрядов – **счетчик_сдвигов** – может задаваться двумя способами:

- статически, что предполагает задание константного значения;
- динамически, что означает занесение значения счетчика сдвигов в регистр CL перед выполнением команды сдвига.

Из размерности регистра CL следует, что значение счетчика сдвигов может лежать в диапазоне от 0 до 255, однако в целях оптимизации микропроцессор воспринимает только значение пяти младших бит счетчика, то есть значение лежит в диапазоне от 0 до 31.

Все команды сдвига устанавливают флаг переноса CF. По мере сдвига бит за пределы операнда они сначала попадают на флаг переноса, устанавливая его равным значению очередного бита, оказавшегося за пределами операнда. Куда этот бит попадет дальше, зависит от типа команды сдвига и алгоритма программы.

По принципу действия команды сдвига можно разделить на два типа:

- команды линейного сдвига;
- команды циклического сдвига.

Команды линейного сдвига

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

1) очередной «выдвигаемый» бит устанавливает флаг CF;
2) бит, вводимый в операнд с другого конца, имеет значение 0;
3) 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 линейного сдвига;
- команды арифметического линейного сдвига.

К командам **логического линейного сдвига** относятся следующие:

- **`shl операнд,счетчик_сдвигов`** (Shift Logical Left) – логический сдвиг влево. Содержимое операнда сдвигается влево на количество бит, определяемое значением операнда **`счетчик_сдвигов`** . Справа (в позицию младшего бита) вписываются нули;
- **`shr операнд,счетчик_сдвигов`** (Shift Logical Right) – логический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое значением операнда **`счетчик_сдвигов`** . Слева (в позицию старшего, знакового бита) вписываются нули.

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

- **`sal операнд,счетчик_сдвигов`** (Shift Arithmetic Left) – арифметический сдвиг влево. Содержимое операнда сдвигается влево на количест-

во бит, определяемое значением операнда **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** (Shift Arithmetic Right) – арифметический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое значением операнда **счетчик_сдвигов** . Слева в операнд вписываются нули. Команда **sar** сохраняет знак, восстанавливая его после сдвига каждого очередного бита.

Команды циклического сдвига

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 **командам простого циклического** сдвига относятся:

- **rol операнд,счетчик_сдвигов** (Rotate Left) – 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** (Rotate Right) – циклический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом **счетчик_сдвигов**. Сдвигаемые вправо биты записываются в тот же операнд слева.

Команды простого циклического сдвига в процессе своей работы осуществляют одно полезное действие: циклически сдвигаемый бит не только вдвигается в операнд с другого конца, но и одновременно его значение становится значением флага CF.

Команды циклического сдвига через флаг переноса CF отличаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в операнд с другого его конца, а записывается сначала в флаг переноса CF. Лишь следующее исполнение данной команды сдвига (при условии, что она выполняется в цикле) приводит к помещению выдвинутого ранее бита с другого конца операнда (рис. 4.7).

К командам циклического сдвига через флаг переноса CF относятся следующие:

- **`rcl операнд,счетчик_сдвигов`** (Rotate through Carry Left) – 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`** (Rotate through Carry Right) – циклический сдвиг вправо через перенос. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса CF.

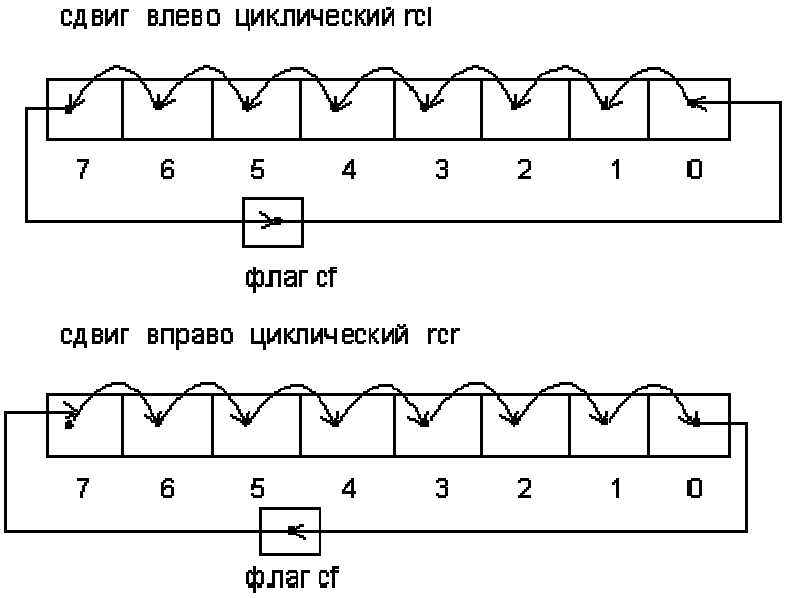

Рис. 4.7. Команды циклического сдвига через флаг переноса CF

4.4. Команды сравнения и передачи управления

Ранее нами были рассмотрены команды, из которых формируются линейные участки программы. Каждая из них в общем случае выполняет некоторые действия по преобразованию или пересылке данных, после чего микропроцессор передает управление следующей команде. Но очень мало программ работает таким последовательным образом. Обычно в программе есть точки, в которых нужно принять решение о том, какая команда будет выполняться следующей.

Это решение может быть:

- безусловным – в данной точке необходимо передать управление не той команде, которая идет следующей, а другой, которая находится на некотором удалении от текущей команды;
- условным – решение о том, какая команда будет выполняться следующей, принимается на основе анализа некоторых условий или данных.

Программа представляет собой последовательность команд и данных, занимающих определенное пространство оперативной памяти. Эта пространство памяти может быть либо непрерывным, либо состоять из нескольких фрагментов. Ранее нами были рассмотрены средства организации фрагментации кода программы и ее данных на сегменты. Адрес команды, которая должна выполняться следующей, определяется содержимым пары регистров CS:IP:

- CS – сегментный регистр кода, в котором находится физический (базовый) адрес текущего сегмента кода;
- IP – регистр указателя команды, в котором находится значение, представляющее собой смещение в памяти следующей команды, подлежащей выполнению, относительно начала текущего сегмента кода.

Таким образом, команды передачи управления изменяют содержимое регистров CS и IP, в результате чего микропроцессор выбирает для выполнения не следующую по порядку команду программы, а команду в некотором другом участке программы.

По принципу действия команды микропроцессора, обеспечивающие организацию переходов в программе, можно разделить на следующие 3 группы:

1. Команды безусловной передачи управления.
2. Команда сравнения.
3. Команды условной передачи управления.

4.4.1. Команды безусловной передачи управления

Команда безусловного перехода **jmp** имеет следующий синтаксис:

```
jmp адрес_перехода
```

Команда выполняет безусловный переход без сохранения информации о точке возврата. Операнд **адрес_перехода** представляет собой адрес в виде метки либо адрес области памяти, в которой находится указатель перехода.

Всего в системе команд микропроцессора есть несколько кодов машинных команд безусловного перехода **jmp**. Их различия определяются дальностью перехода и способом задания целевого адреса.

Дальность перехода определяется местоположением операнда **адрес_перехода**. Этот адрес может находиться в текущем сегменте кода или в некотором другом сегменте. В первом случае переход называется внутрисегментным, или близким (near), во втором – межсегментным, или дальним (far). Внутрисегментный переход предполагает, что изменяется только содержимое регистра IP.

Команда **вызова процедуры** имеет следующий синтаксис:

```
call имя_процедуры
```

Команда помещает в стек адрес возврата (адрес следующей за командой инструкции) и передает управление указанной процедуре.

В конце тела каждой процедуры находится команда возврата из процедуры **ret** (не имеет операндов), которая извлекает из стека адрес возврата и передает управление на него.

Для вызова программного прерывания используется команда

```
int номер_прерывания
```

Как и в случае вызова процедур, команда помещает в стек адрес возврата (адрес следующей за командой инструкции) и передает управление обработчику прерывания.

Каждый обработчик программного прерывания, как и процедура, в конце кода содержит команду **iret** (не имеет операндов), которая извлекает из стека адрес возврата и передает управление на него.

Особенности работы системы прерываний будут рассматриваться ниже.

4.4.2. Команда сравнения CMP

Синтаксис команды:

```
cmp операнд_1,операнд_2
```

Команда **cmp** так же, как и команда **sub**, выполняет вычитание операндов и устанавливает флаги. Единственное, чего она не делает – это запись результата вычитания на место первого операнда, т.е. команда **cmp** сравнивает два операнда и по результатам сравнения устанавливает флаги.

Флаги, устанавливаемые командой **cmp**, можно анализировать специальными командами условного перехода.

4.4.3. Команды условной передачи управления

Условный переход проверяет текущее состояние машины (флагов или регистров), чтобы определить, передать управление или нет. Команды переходов по условию делятся на две группы:

- проверяющие результаты предыдущей арифметической или логической операции **Jcc**;
- управляющие итерациями фрагмента программы (организацией циклов) **LOOPcc**.

Все условные переходы имеют однобайтовое смещение, то есть метка, на которую происходит переход, должна находится в том же кодовом сегменте и на расстоянии, не превышающем –128 +127 байт от первого байта следующей команды. Если условный переход осуществляется на место, находящееся дальше 128 байт, то необходимо использовать промежуточные переходы.

Первая группа команд Jcc (кроме JCXZ/JECXZ) проверяет текущее состояние регистра флагов (не изменяя его) и в случае соблюдения условия осуществляет переход на смещение, указанное в качестве операнда. Флаги, проверяемые командой, кодируются в ее мнемонике, например: JC – переход, если установлен CF. В табл. 4.4 приведены значения аббревиатур команды, а в табл. 4.5 приведены команды условного перехода и проверяемые ими флаги.

Таблица 4.4

Значение аббревиатур в команде jcc

Мнемоническое обозначение	Английский	Русский	Тип операндов
E e	equal	Равно	Любые
N n	not	Не	Любые
G g	greater	Больше	Числа со знаком
L l	less	Меньше	Числа со знаком
A a	above	Выше, в смысле «больше»	Числа без знака
B b	below	Ниже, в смысле «меньше»	Числа без знака

Таблица 4.5

Команды условного перехода и проверяемые ими флаги

Мнемоника	Флаги					Комментарии
	OF	CF	ZF	PF	SF	
Проверка флагов						
JE/JZ	x	x	1	x	x	
JP/JPE	x	x	x	1	x	
JO	1	x	x	x	x	
JS	x	x	x	x	1	
JNE/JNZ	x	x	0	x	x	
JNP/JPO	x	x	x	0	x	
JNO	0	x	x	x	x	
JNS	x	x	x	x	0	
Арифметика со знаком (используется команда CMP)						
JL/JNGE	a	x	x	x	b	Если a<>b(OF<>SF), то операнд_1 < операнд_2
JLE	a	x	1	x	b	Если Z или a не равно b (Z or (OF<>SF)), то операнд_1 <= операнд_2
JNL/JGE	a	x	x	x	b	Если a<>b(OF<>SF), операнд_1 => операнд_2
JNLE/JG	a	x	0	x	b	Если не Z и a не равно b (Z &(OF<>SF)), то операнд_1 > операнд_2
Арифметика без знака (используется команда CMP)						
JB/JNAE	x	1	x	x	x	операнд_1 < операнд_2
JBE/JNA	x	1	1	x	x	операнд_1 <= операнд_2
JNB/JAE	x	0	x	x	x	операнд_1 => операнд_2
JNBE/JA	x	0	0	x	x	операнд_1 > операнд_2

Буква x в любой позиции означает, что команда не проверяет флаг. Цифра 0 означает, что флаг должен быть сброшен, а цифра 1 означает, что флаг должен быть установлен, чтобы условие было выполнено (переход произошел).

Команды условного перехода можно разделить на три подгруппы:

1) непосредственно проверяющие один из флагов на равенство нулю или единице;

2) арифметические сравнения со знаком. Существуют четыре условия, которые могут быть проверены: меньше (JL), меньше или равно (JLE), больше (JG), больше или равно (JGE). Эти команды проверяют одновременно три флага: знака, переполнения и нуля;

3) арифметические без знака. Здесь также существует четыре возможных соотношения между операндами. Учитываются только два флага: флаг переноса показывает, какое из двух чисел больше, флаг нуля определяет равенство.

Ниже приведен фрагмент программы, иллюстрирующей использование команд сравнения и перехода.

```
CSEG SEGMENT
ASSUME CS:CSEG, DS:DSEG, SS:SSEG
START:
...
MOV BH,X          ;Загрузка в BH значения X
MOV BL,Y          ;Загрузка в BL значения Y
CMP BH,BL         ;Сравнение BH и BL
JE MET1           ;Если BH=BL, то переход на MET1
JMP MET2          ;Иначе переход на MET2
MET1:
...
JMP MET3
MET2:
...
MET3:
MOV AH,4Ch
INT 21H
CSEG ENDS
END START
```

Команда **JCXZ** отличается от других команд условного перехода тем, что она проверяет содержимое регистра CX, а не флагов. Эту команду лучше всего применять в начале условного цикла, чтобы предотвратить вхождение в цикл, если CX=0.

Вторая группа команд условного перехода **LOOPcc** служит для организации циклов в программах. Все команды цикла используют регистр CX в качестве счетчика цикла. Простейшая из них – команда **LOOP**. Она уменьшает содержимое CX на 1 и передает управление на указанную метку, если содержимое CX не равно 0. Если вычитание единицы из CX привело к нулевому результату, выполняется команда, следующая за LOOP.

Команда **LOOPNE** (цикл пока не равно) осуществляет выход из цикла, если установлен флаг нуля или если регистр CX достиг нуля.

Команда **LOOPE** (цикл пока равно) выполняет обратную к описанной выше проверку флага нуля: в этом случае цикл завершается, если регистр CX достиг нуля или если не установлен флаг нуля.

Приведенный ниже фрагмент программы иллюстрирует использование команд организации циклов. Программа посимвольно выводит на экран содержимое строки **BUF**.

```
DSEG SEGMENT
BUF DB ”0123406789”
DSEG ENDS
CSEG SEGMENT
ASSUME CS:CSEG,DS:DSEG,SS:SSEG
START:
...
MOV BX,OFFSET BUF     ;В BX – начало буфера
MOV CX,10             ;В CX – длина буфера
MOV SI,0
M1: MOV DL,[BX+SI]    ;В DL – символ из буфера
MOV AH,2              ;в AH номер функции-вывода
INT 21H               ;Вывод на экран
INC SI                ;Увеличение индекса на 1
LOOP M1               ;Оператор первого цикла
...
CSEG ENDS
END START
```

4.5. Цепочечные команды

Цепочечные команды позволяют проводить действия над блоками памяти, представляющими собой последовательности элементов следующего размера:

- 8 бит, то есть байт;
- 16 бит, то есть слово;
- 32 бита, то есть двойное слово.

Содержимое этих блоков для процессора не имеет никакого значения. Это могут быть символы, числа и все что угодно. Главное, чтобы размеры элементов соответствовали одному из перечисленных ранее вариантов и эти элементы находились в соседних ячейках памяти.

Всего в системе команд процессора поддерживаются семь операций-примитивов обработки цепочек. Каждая из них реализуется в процессоре тремя командами, в свою очередь, каждая из этих команд работает с соответствующим размером элемента – байтом, словом или двойным словом. *Особенность всех цепочечных команд заключается в том, что они, кроме обработки текущего элемента цепочки, корректируют содержимое определенных регистров с тем, чтобы автоматически продвинуться к следующему элементу цепочки!*

Перечислим операции-примитивы цепочечной обработки.

- Пересылка цепочки:

```
movs адрес_приемника,адрес_источника;
```

- Сравнение цепочек:

```
cmps адрес_приемника,адрес_источника;
```

- Сканирование цепочки:

```
scas адрес_приемника;
```

- Загрузка элемента из цепочки:

```
lods адрес_источника;
```

- Сохранение элемента в цепочке:

```
stos адрес_приемника;
```

Реализующие их команды Ассемблера для элементов размерностью 8, 16 и 32 бита образуются путем добавления к операции-примитиву соответствующего суффикса (b – BYTE, w – Word, D – Double Word). При этом следует заметить, что команды не имеют операндов. Подробнее об этом – далее.

Логически к этим командам нужно отнести и так называемые префиксы повторения. Они предназначены для использования цепочечными командами. Префиксы повторения имеют свои мнемонические обозначения:

rep;
repe, или **repz**;
repne, или **repnz**.

Эти префиксы повторения указываются перед нужной цепочечной командой в поле метки. Цепочечная команда без префикса выполняется один раз. Размещение префикса перед цепочечной командой заставляет ее выполняться в цикле. Различия приведенных префиксов – в основании, по которому принимается решение о циклическом выполнении цепочечной команды: по состоянию регистра ECX/CX или по флагу нуля ZF:

- префикс повторения **rep** (REPeat) используется с командами, реализующими операции-примитивы пересылки и сохранения элементов цепочек, соответственно, **movs** и **stos**. Префикс **rep** заставляет данные команды выполняться, пока содержимое в ECX/CX не станет равным 0. При этом цепочечная команда, перед которой стоит префикс, автоматически уменьшает содержимое ECX/CX на единицу в каждой итерации. Та же команда, но без префикса, этого не делает;
- префиксы повторения **repe** (REPeat while Equal) и **repz** (REPeat while Zero) являются абсолютными синонимами. Они заставляют цепочечную команду выполняться до тех пор, пока содержимое ECX/CX не станет равным 0 или флаг ZF – равным 1. Как только одно из этих условий нарушается, управление передается следующей команде программы. Благодаря возможности анализа флага ZF наиболее эффективно эти префиксы можно использовать с командами **cmps** и **scas** для поиска различающихся элементов цепочек;
- префиксы повторения **repne** (REPeat while Not Equal) и **repnz** (REPeat whileNot Zero) также являются абсолютными синонимами. Их действие на цепочечную команду несколько отличается от действий префиксов **repe**/**repz**. Префиксы **repne**/**repnz** заставляют цепочечную команду циклически выполняться до тех пор, пока содержимое ECX/CX не станет равным

нулю или флаг ZF – равным нулю. При нарушении одного из этих условий работа команды прекращается. Данные префиксы также можно использовать с командами CMPS и SCAS, но для поиска совпадающих элементов цепочек.

Следующий важный момент, связанный с цепочечными командами, заключается в особенностях формирования физического адреса операндов **адрес_источника** и **адрес_приемника**. Цепочка-источник, адресуемая операндом **адрес_источника**, может находиться в текущем сегменте данных, определяемом регистром DS. Цепочка-приемник, адресуемая операндом **адрес_приемника**, должна быть в дополнительном сегменте данных, адресуемом сегментным регистром ES. Важно отметить, что допускается замена (с помощью префикса замены сегмента) только регистра DS, регистр ES подменять нельзя. Вторые части адресов (смещения цепочек) также находятся в строго определенных местах. Для цепочки-источника это регистр SI (Source Index register – индексный регистр источника). Для цепочки-получателя это регистр DI (Destination Index register – индексный регистр приемника).

Таким образом, полные физические адреса для операндов цепочечных команд следующие:

- **адрес_источника** – пара DS:SI;
- **адрес_приемника** – пара ES:DI.

Все семь групп команд, реализующих цепочечные операции-примитивы, имеют схожую структуру. В каждой из групп присутствуют одна операция-примитив с явным указанием операндов и три команды, не имеющие операндов. На самом деле набор команд процессора имеет соответствующие машинные команды только для цепочечных команд Ассемблера без операндов. Команды с операндами транслятор Ассемблера задействует только для определения типов операндов. После того как выяснен тип элементов цепочек по их описанию в памяти, генерируется одна из трех машинных команд для каждой из цепочечных операций. По этой причине все регистры, содержащие адреса цепочек, должны быть инициализированы заранее, в том числе и для команд, допускающих явное указание операндов. В силу того, что цепочки адресуются однозначно, нет особого смысла применять команды с операндами. Главное, что необходимо запомнить, – правильная загрузка регистров указателями обязательно требуется до выдачи любой цепочечной команды.

Последний важный момент, касающийся всех цепочечных команд, – это направление обработки цепочки: от начала цепочки к ее концу, то есть в направлении возрастания адресов, или от конца цепочки к началу, то есть в направлении убывания адресов.

Как было сказано выше, цепочечные команды сами выполняют модификацию регистров, адресующих операнды, обеспечивая тем самым автоматическое продвижение по цепочке. Количество байтов, на которые эта модификация осуществляется, определяется размером элемента цепочки (например, если это цепочка слов, то содержимое индексных регистров DS и SI изменяется на 2). Знак этой модификации определяется значением флага направления DF (Direction Flag) в регистре FLAGS:

- если DF = 0, то значения индексных регистров SI и DI будут автоматически увеличиваться (операция инкремента) цепочечными командами, то есть обработка будет осуществляться в направлении возрастания адресов;
- в если DF = 1, то значения индексных регистров SI и DI будут автоматически уменьшаться (операция декремента) цепочечными командами, то есть обработка будет идти в направлении убывания адресов.

Состоянием флага DF можно управлять с помощью двух команд, не имеющих операндов:

- **`cld`** (Clear Direction Flag) – очистить флаг направления (команда сбрасывает флаг направления DF в 0);
- **`std`** (Set Direction Flag) – установить флаг направления (команда устанавливает флаг направления DF в 1).

Рассмотрим более подробно каждую группу команд.

4.5.1. Пересылка цепочек

Команды, реализующие операцию-примитив пересылки цепочек, производят копирование элементов из одной области памяти (цепочки) в другую. Размер элемента определяется применяемой командой. Ассемблер предоставляет в распоряжение программиста четыре команды, работающие с разными размерами элементов цепочки:

- **`movs адрес_приемника,адрес_источника`** – переслать цепочку (MOVe String);
- **`movsb`** – переслать цепочку байтов (MOVe String Byte);

- **movsw** – переслать цепочку слов (MOVe String Word);
- **movsd** – переслать цепочку двойных слов (MOVe String Double word).

Синтаксис команды MOVS:

```
movs адрес_приемника,адрес_источника
```

Команда копирует байт, слово или двойное слово из цепочки, адресуемой операндом **адрес_источника**, в цепочку, адресуемую операндом **адрес_приемника**. Размер пересылаемых элементов Ассемблер определяет, исходя из атрибутов идентификаторов, указывающих на области памяти приемника и источника. К примеру, если эти идентификаторы были определены директивой DB, то пересылаться будут байты, если идентификаторы были определены с помощью директивы DD, то пересылке подлежат 32-разрядные элементы, то есть двойные слова. Ранее уже было отмечено, что для цепочечных команд с операндами, к которым относится и команда пересылки **movs адрес_приемника,адрес_источника**, не существует машинного аналога. При трансляции в зависимости от типа операндов транслятор преобразует ее в одну из трех машинных команд: MOVSB, MOVSW или MOVSD.

Сама по себе команда MOVS пересылает только один элемент, исходя из его типа, и модифицирует значения регистров ESI/SI и EDI/DI. Если перед командой поместить префикс REP, то одной командой можно переслать такое число элементов, которое было предварительно загружено в счетчик – регистр CX. Последовательность действий, которые нужно выполнить в программе для того, чтобы переслать цепочку элементов из одной области памяти в другую с помощью команды MOVS, выглядит следующим образом:

1. Установить значение флага DF в зависимости от того, в каком направлении будут обрабатываться элементы цепочки – в направлении возрастания или убывания адресов.
2. Загрузить указатели на адреса цепочек в памяти в пары регистров DS:SI и ES:DI.
3. Загрузить в регистр CX количество обрабатываемых элементов.
4. Выдать команду MOVS с префиксом REP.

В общем случае эту последовательность можно рассматривать как типовую для выполнения любой цепочечной команды.

В примере ниже производится пересылка символов из одной строки в другую. Строки находятся в одном сегменте памяти. Для пересылки используется команда-примитив **movs** с префиксом повторения **rep**.

```
masm
model small
stack 256
.data
source db "Тестируемая с т р о к а " , ' $ '
dest db 19 dup (" ")
.code
main:                   ;точка входа в программу
mov ax,@data            ;загрузка сегментных регистров
mov ds,ax               ;настройка регистров DS и ES
mov es,ax               ;на адрес сегмента данных
cld                     ;сброс флага DF – обработка
                        ;строки от начала к концу
lea si,source           ;загрузка в si смещения
                        ;строки-источника
lea di,dest             ;загрузка в DS смещения
;строки-приемника
mov cx,20               ;счетчик для префикса rep
rep movs dest,source ;пересылка строки
lea dx,dest
mov ah,09h              ;вывод на экран строки-приемника
int 21h
exit:
mov ax,4c00h            ;завершение программы
int 21h
end main
```

Пересылка байтов, слов и двойных слов

Пересылка байтов, слов и двойных слов производится командами **movsb**, **movsw** и **movsd**. Единственной отличительной особенностью этих команд от команды movs является то, что последняя может работать с элементами цепочек любого размера – 8, 16 или 32 бита. При трансляции команда MOVS преобразуется в одну из трех команд: **movsb**, **movsw** или **movsd**. Ранее было показано, что решение о том, в какую конкретно команду будет произведено преобразование, принимается транслятором исходя из размеров элементов цепочек, адреса которых указаны в качестве операндов команды MOVS. Что касается адресов цепочек, то для любой из команд они должны формироваться программой явно и заранее в регистрах SI и DI.

В примере ниже показано использование команды **movsb** для реализации предыдущей задачи:

```
.data
source db "Пересылаемая строка$" ;строка-источник
dest db 20 DUP (?) ;строка-приемник
.code
main:
mov ax,@data          ;загрузка сегментных регистров
mov ds,ax             ;настройка регистров DS и ES
mov es,ax             ;на адрес сегмента данных
cld                   ;сброс флага DF – просмотр
                      ;строки от начала к концу
lea si,source         ;загрузка в ES строки-источника
lea di,dest           ;загрузка в DS строки-приемника
mov cx,20             ;для префикса rep - длина строки
rep movsb             ;пересылка строки
```

Далее для каждой группы команд мы будем рассматривать только операцию-примитив.

4.5.2. Сравнение цепочек

Команды, реализующие операцию-примитив сравнения цепочек, производят сравнение элементов цепочки-источника с элементами цепочки-приемника.

Здесь ситуация с набором команд и методами работы с ними аналогична операции-примитиву пересылки цепочек. TASM предоставляет программисту четыре команды сравнения цепочек, работающие с разными размерами элементов цепочки:

- **cmps адрес_приемника,адрес_источника** – сравнить строки (CoMPare String);
- **cmpsb** – сравнить строку байтов (CoMPare String Byte);
- **cmpsw** – сравнить строку слов (CoMPare String Word);
- **cmpsd** – сравнить строку двойных слов (CoMPare String Double word).

Рассмотрим работу этих команд на примере команды **scas.**

Здесь: **адрес_источника** определяет адрес цепочки-источника в сегменте данных, заранее загружаемый в пару регистров DS: SI; **адрес_приемника** определяет адрес цепочки-приемника, которая должна находиться в дополнительном сегменте, заранее загружаемый в пару регистров ES:DI.

Алгоритм работы команды **cmps** заключается в последовательном выполнении вычитания (элемент цепочки-источника минус элемент цепочки-получателя) над очередными элементами обеих цепочек. Принцип выполнения вычитания командой **cmps** такой же, как у команды сравнения **cmp**. Она так же, как и **cmp**, производит вычитание элементов, не записывая при этом результата, и устанавливает флаги ZF, SF и OF. После вычитания очередных элементов цепочек командой **cmps** индексные регистры SI и DI автоматически изменяются в соответствии со значением флага DF на значение, равное размеру элемента сравниваемых цепочек. Чтобы заставить команду **cmps** выполняться несколько раз, то есть произвести последовательное сравнение элементов цепочек, необходимо перед командой **cmps** поместить префикс повторения. С командой **cmps** можно использовать префиксы повторения **rep**, **repe/repz** или **repne/repnz**:

- **rep** – сравнивать элементы цепочек, пока ECX/CX>0;

Ø **repe/repz** – сравнивать элементы цепочек до выполнения одного из двух условий:

ü содержимое ECX/CX равно нулю;

ü в цепочках встретились разные элементы (флаг ZF стал равен нулю);

Ø **repne/repnz** – сравнивать элементы цепочек до выполнения одного из двух условий:

ü содержимое ECX/CX равно нулю;

ü в цепочках встретились одинаковые элементы (флаг ZF стал равен единице).

Таким образом, выбор подходящего префикса позволяет организовать гибкий поиск одинаковых или различающихся элементов цепочек командой CMPS.

Для определения местоположения очередных совпавших или несовпавших элементов в цепочках вспомним, что после каждой итерации цепочечная команда автоматически осуществляет инкремент – декремент значения адреса в соответствующих индексных регистрах. Поэтому после выхода из цикла в этих регистрах будут находиться адреса элементов, находящихся в цепочке до или после элементов, послуживших причиной выхода из цикла. Для получения истинного адреса этих элементов необходимо скорректировать содержимое индексных регистров, увеличив либо уменьшив значение в них на длину элемента цепочки.

4.5.3. Сканирование цепочек

Команды, реализующие операцию-примитив сканирования цепочек, производят поиск некоторого значения в области памяти. Логически эта область памяти рассматривается как последовательность (цепочка) элементов фиксированной длины размером 8, 16 или 32 бита. Искомое значение предварительно должно быть помещено в один из регистров AL/AX/EAX. Выбор конкретного регистра из этих трех должен быть согласован с размером элементов цепочки, в которой осуществляется поиск.

Группа команд сканирования:

scas адрес_приемника – сканировать цепочку (SCAning String);

scasb – сканировать цепочку байтов (SCAning String Byte);

scasw – сканировать цепочку слов (SCAning String Word);

scasd – сканировать цепочку двойных слов (SCAning String Double Word).

Рассмотрим работу этих команд на примере команды **scas.**

Команда имеет один операнд, обозначающий местонахождение цепочки в дополнительном сегменте (адрес цепочки должен быть заранее сформирован в регистрах ES:DI). Транслятор анализирует тип идентификатора **адрес_приемника**, который обозначает цепочку в сегменте данных, и формирует одну из трех машинных команд: **scasb**, **scasw** или **scasd**. Условие поиска для каждой из этих трех команд задано в строго определенном месте. Так, если цепочка описана с помощью директивы DB, то искомый элемент должен быть байтом, находиться в регистре AL, и сканирование цепочки осуществляется командой **scasb**. Если цепочка описана с помощью директивы DW, то искомый элемент – слово в регистре AX, и поиск ведется командой **scasw**. Если цепочка описана с помощью директивы DD, то искомый элемент – двойное слово в EAX, и поиск ведется командой **scasd**. Принцип поиска тот же, что и в команде сравнения **cmps**, то есть последовательное выполнение вычитания (содержимое регистра аккумулятора минус содержимое очередного элемента цепочки). В зависимости от результатов вычитания производится установка флагов, при этом сами операнды не изменяются. Так же как и в случае команды **cmps**, с командой **scas** удобно использовать префиксы **repe/repz** или **repne/repnz**:

- REPE или REPZ – если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий:
  - будет достигнут конец цепочки (содержимое ECX/CX равно 0);
  - в цепочке встретится элемент, отличный от элемента в регистре AL/AX/EAX;
- REPNE или REPNZ – если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий:
  - будет достигнут конец цепочки (содержимое ECX/CX равно 0);
  - в цепочке встретится элемент, совпадающий с элементом в регистре AL/AX/EAX.

Таким образом, команда **scas** с префиксом **repe/repz** позволяет найти элемент цепочки, отличающийся по значению от заданного в аккумуляторе, а с префиксом **repne/repnz** – найти элемент цепочки, совпадающий по значению с элементом в аккумуляторе.

4.5.4. Загрузка элемента цепочки в аккумулятор

Операция-примитив загрузки элемента цепочки в аккумулятор позволяет извлечь элемент цепочки и поместить его в регистр-аккумулятор AL, AX или EAX. Эту операцию удобно использовать вместе с поиском (сканированием) с тем, чтобы, найдя нужный элемент, извлечь его (например, для изменения). Возможный размер извлекаемого элемента определяется применяемой командой. Программист может использовать четыре команды загрузки элемента цепочки в аккумулятор, работающие с элементами разного размера:

- **`lods адрес_источника`** – загрузить элемент из цепочки (LOaD String) в регистр-аккумулятор AL/AX/EAX;
- **`lodsb`** – загрузить байт из цепочки (LOaD String Byte) в регистр AL;
- **`lodsw`** – загрузить слово из цепочки (LOaD String Word) в регистр AX;
- **`lodsd`** – загрузить двойное слово (LOaD String Double Word) из цепочки в регистр EAX.

Рассмотрим работу этих команд на примере команды **`lods`**.

Команда имеет один операнд, обозначающий строку в основном сегменте данных. Работа команды заключается в том, чтобы извлечь элемент из цепочки по адресу, соответствующему содержимому пары регистров DS: SI, и поместить его в регистр EAX/AX/AL. При этом содержимое SI подвергается инкременту или декременту (в зависимости от состояния флага DF) на величину, равную размеру элемента. Эту команду удобно использовать после команды **`scas`**, локализующей местоположение искомого элемента в цепочке. Префикс повторения в этой команде может и не понадобиться – все зависит от логики программы.

4.5.5. Загрузка элемента из аккумулятора в цепочку

Операция-примитив загрузки элемента из аккумулятора в цепочку позволяет произвести действие, обратное действию команды LODS, то есть сохранить значение из регистра-аккумулятора в элементе цепочки. Эту операцию удобно использовать вместе с операциями поиска (сканирования) **`scans`** и загрузки **`lods`** с тем, чтобы, найдя нужный элемент, извлечь его в регистр и записать на его место новое значение. Также использование команды удобно для заполнения одинаковыми значениями некоторой области памяти. Команды, поддерживающие эту операцию-примитив, могут работать с элементами размером 8, 16 или 32 бита:

- **stos адрес_приемника** – сохранить в цепочке элемент (STOre String) из регистра-аккумулятора AL/AX/EAX;
- **stosb** – сохранить в цепочке байт (STOre String Byte) из регистра AL;
- **stosw** – сохранить в цепочке слово (STOre String Word) из регистра AX;
- **stosd** – сохранить в цепочке двойное слово (STOre Siring Double Word) из регистра EAX.

Рассмотрим работу этих команд на примере команды **lods**.

Команда имеет один операнд **адрес_приемника**, адресующий цепочку в дополнительном сегменте данных. Команда пересылает элемент из аккумулятора (регистра EAX/AX/AL) в элемент цепочки по адресу, соответствующему содержимому пары регистров ES:DI. При этом содержимое DI подвергается инкременту или декременту (в зависимости от состояния флага DF) на величину, равную размеру элемента цепочки.

Префикс повторения в этой команде может и не понадобиться – все зависит от логики программы. Например, если использовать префикс повторения **rep**, то можно применить команду для инициализации области памяти некоторым фиксированным значением.

4.6. Вопросы и задания для самопроверки

1. Возможна ли пересылка непосредственных данных в сегментный регистр?
2. Напишите фрагмент программы, осуществляющей обмен между регистрами AX и BX, с использованием команды MOV и без использования последней.
3. Напишите фрагмент программы, осуществляющей перекодировку шестнадцатеричной цифры, содержащейся в регистре CL, в ее символьное представление.
4. Напишите фрагмент программы, осуществляющей обмен данными между регистрами AL и BL, используя стек.
5. Опишите типичный сценарий обмена данными с внешним устройством.
6. Укажите основное различие между командами сложения ADD и ADC.
7. Напишите фрагмент программы, вычисляющей сумму значений регистров общего назначения.

8. Напишите EXE-программу, выполняющую сложение двух 32-разрядных операндов. Результат поместите в ячейку памяти.

9. Напишите фрагмент программы, сбрасывающей в ноль значения четных бит регистра AL

10. Напишите фрагмент программы, устанавливающей в единицу значение бита 0 порта 37Ah (остальные биты порта должны остаться без изменения).

11. Напишите фрагмент программы, выполняющей операцию исключающего ИЛИ между значениями в портах 378h и 278h.

12. Укажите основные виды сдвигов, которые можно выполнить в ассемблерных программах.

13. Напишите фрагмент программы, помещающей во флаг переноса CF значение 5-го бита регистра AX.

14. Какие команды могут использоваться для сравнения операндов?

15. Укажите основные различия между командами условных и безусловных переходов.

16. Каким образом осуществляется переход на метку, отстоящую от команды условного перехода далее чем на 128 байт?

17. Какой из регистров используется в качестве счетчика при организации циклов?

18. Напишите два фрагмента программы, иллюстрирующие организацию цикла с использованием команд типа LOOP и без использования последних.

19. Напишите фрагмент программы, выполняющей сравнение двух строк и выдающей соответствующее сообщение.

20. Напишите фрагмент программы для заполнения некоторой области памяти одинаковыми значениями.

МОДУЛЬ 5. СИСТЕМА ПРЕРЫВАНИЙ МИКРОПРОЦЕССОРА I80486

Цель модуля – изучение особенностей работы системы прерываний микропроцессора i80486 и процедуры установки собственных обработчиков прерываний.

В результате изучения модуля студенты должны:

- Ø четко понимать механизм прерывания программ;
- Ø знать основные функции системы прерываний;
- Ø знать классификацию прерываний;
- Ø знать порядок обслуживания прерываний;
- Ø знать процедуру установки собственных обработчиков прерываний.

Содержание модуля

В рамках данного модуля предусмотрено выполнение лабораторной работы № 3 «Изучение функций прерывания 21h» и лабораторной работы № 4 «Программирование устройства ввода типа «мышь» (Методические указания к выполнению лабораторных работ по курсу «Низкоуровневое программирование», сост. Д. Г. Руголь).

Во время выполнения ЭВМ текущей программы внутри машины и в связанной с ней внешней среде (например, в технологическом процессе, управляемом ЭВМ) могут возникать события, требующие немедленной реакции на них со стороны машины.

Реакция состоит в том, что машина прерывает обработку текущей программы и переходит к выполнению некоторой другой программы, специально предназначенной для данного события. По завершении этой программы ЭВМ возвращается к выполнению прерванной программы.

Рассматриваемый процесс называется прерыванием программ. Принципиально важным является то, что моменты возникновения событий, требую-

щих прерывания программ, заранее не известны и поэтому не могут быть учтены при программировании.

Каждое событие, требующее прерывания, сопровождается сигналом, оповещающим ЭВМ – запросами прерывания. Программу, затребованную запросом прерывания, называют прерывающей программой, противопоставляя ее прерываемой программе, выполнявшейся машиной до появления запроса.

Возможность прерывания программ – важное архитектурное свойство ЭВМ, позволяющее эффективно использовать производительность процессора при наличии нескольких протекающих параллельно во времени процессов, требующих в произвольные моменты времени управления и обслуживания со стороны процессора. В первую очередь это относится к организации параллельной во времени работы процессора и периферийных устройств машины, а также к использованию ЭВМ для управления в реальном времени технологическими процессами.

Чтобы ЭВМ могла, не требуя больших усилий от программиста, реализовывать с высоким быстродействием прерывание программ, машине необходимо придать соответствующие аппаратурные и программные средства, совокупность которых получила название системы прерывания программ.

Основными функциями системы прерывания являются:

1. Запоминание состояния прерываемой программы и осуществление перехода к прерывающей программе.

2. Восстановление состояния прерванной программы и возврат к ней.

Вектором прерывания называется вектор «начального состояния прерывающей программы». Вектор прерывания содержит всю необходимую информацию для перехода к прерывающей программе, в том числе ее начальный адрес. Каждому запросу (номеру) прерывания соответствует свой вектор прерывания, способный инициировать выполнение соответствующей прерывающей программы. Векторы прерывания находятся в специально выделенных фиксированных ячейках памяти – таблице векторов прерываний.

Главное место в процедуре перехода к прерывающей программе занимает процедура передачи из соответствующего регистра (регистров) процессора в память (в частности, в стек) на сохранение текущего вектора состояния прерываемой программы (чтобы можно было вернуться к ее исполнению) и загрузка в регистр (регистры) процессора вектора прерывания прерывающей программы, к которой при этом переходит управление процессором.

5.1. Классификация прерываний

Запросы на прерывания могут возникать внутри самой ЭВМ и в ее внешней среде. К первым относятся, например, запросы при возникновении в ЭВМ таких событий, как появление ошибки в работе ее аппаратуры, переполнение разрядной сетки, попытка деления на 0, выход из установленной для данной программы области памяти, затребование периферийным устройством операции ввода – вывода, завершение операции ввода – вывода периферийным устройством или возникновение при этой операции особой ситуации и др. Хотя некоторые из указанных событий порождаются самой программой, моменты их появления, как правило, невозможно предусмотреть. Запросы во внешней среде могут возникать от других ЭВМ, от аварийных и некоторых других датчиков технологического процесса и т.п.

Семейство микропроцессоров Intel 80x86 поддерживает 256 уровней приоритетных прерываний, вызываемых событиями трех типов:

1. Внутренние аппаратные прерывания.
2. Внешние аппаратные прерывания.
3. Программные прерывания.

Внутренние аппаратные прерывания, иногда называемые отказами (faults), генерируются определенными событиями, возникающими в процессе выполнения программы, например, попыткой деления на 0. Закрепление за такими событиями определенных номеров прерываний «зашито» в процессоре и не может быть изменено.

Внешние аппаратные прерывания инициируются контроллерами периферийного оборудования или сопроцессорами (например, 8087/80287). Источники сигналов прерываний подключаются либо к выводу немаскируемых прерываний процессора (NMI), либо к выводу маскируемых прерываний (INTR). Линия NMI обычно предназначается для прерываний, вызываемых катастрофическими событиями, такими, как ошибки четности памяти или авария питания.

Программные прерывания. Любая программа может инициировать синхронное программное прерывание путем выполнения команды **`int`**. MS-DOS использует для взаимодействия со своими модулями и прикладными программами прерывания от 20H до 3FH (например, доступ к диспетчеру функций MS-DOS осуществляется выполнением команды **`int 21h`**). Про-

граммы BIOS, хранящиеся в ПЗУ, и прикладные программы IBM PC используют другие прерывания, с большими или меньшими номерами. Это распределение номеров прерываний условно и никаким образом не закреплено аппаратно.

5.2. Таблица векторов прерываний

Для того чтобы связать адрес обработчика прерывания с номером прерывания, используется таблица векторов прерываний (табл. 5.1), занимающая первый килобайт оперативной памяти. Эта таблица находится в диапазоне адресов от 0000:0000 до 0000:03FFh и состоит из 256 элементов – дальних адресов обработчиков прерываний.

Элементы таблицы векторов прерываний называются векторами прерываний. В первом слове элемента таблицы записана компонента смещения, а во втором – сегментная компонента адреса обработчика прерывания.

Вектор прерывания с номером 0 находится по адресу 0000:0000, с номером 1 – по адресу 0000:0004 и т.д. В общем случает адрес вектора прерывания находится путем умножения номера прерывания на 4.

Инициализация таблицы выполняется частично системой базового ввода/вывода BIOS после тестирования аппаратуры и перед началом загрузки операционной системой, частично при загрузке MS-DOS. Операционная система MS-DOS может изменить некоторые векторы прерываний, установленные BIOS.

Таблица 5.1

Таблица векторов прерываний

Номер	Описание
0h	Ошибка деления. Вызывается автоматически после выполнения команд DIV или IDIV, если в результате деления происходит переполнение (например, при делении на 0). Обычно при обработке этого прерывания MS-DOS выводит сообщение об ошибке и останавливает выполнение программы. При этом для процессора i8086 адрес возврата указывает на команду, следующую после команды деления, а для процессора i80286 и более поздних моделей – на первый байт команды, вызвавшей прерывание
1h	Прерывание пошагового режима. Генерируется после выполнения каждой машинной команды, если в слове флагов установлен бит пошаговой трассировки TF. Используется для отладки программ. Это прерывание не генерируется после пересылки данных в сегментные регистры командами MOV и POP

Продолжение табл. 5.1

Номер	Описание
2h	Аппаратное немаскируемое прерывание. Это прерывание может использоваться по-разному в разных машинах. Обычно оно генерируется при ошибке четности в оперативной памяти и при запросе прерывания от сопроцессора
3h	Прерывание для трассировки. Генерируется при выполнении однобайтовой машинной команды с кодом CCh и обычно используется отладчиками для установки точки прерывания
4h	Переполнение. Генерируется машинной командой INTO , если установлен флаг переполнения OF. Если флаг не установлен, команда INTO выполняется как NOP. Это прерывание используется для обработки ошибок при выполнении арифметических операций
5h	Печать копии экрана. Генерируется, если пользователь нажал клавишу <PrtSc>. В программах MS-DOS обычно используется для печати образа экрана. Для процессора i80286 и более старших моделей генерируется при выполнении машинной команды BOUND, если проверяемое значение вышло за пределы заданного диапазона
6h	Неопределенный код операции или длина команды больше 10 байт
7h	Особый случай отсутствия арифметического сопроцессора
8h	IRQ0 – прерывание интервального таймера, возникает 18,2 раза в секунду
9h	IRQ1 – прерывание от клавиатуры. Генерируется, когда пользователь нажимает и отпускает клавиши. Используется для чтения данных с клавиатуры
Ah	IRQ2 – используется для каскадирования аппаратных прерываний
Bh	IRQ3 – прерывание асинхронного порта COM2
Ch	IRQ4 – прерывание асинхронного порта COM1
Dh	IRQ5 – прерывание от контроллера жесткого диска (только для компьютеров IBM PC/XT)
Eh	IRQ6 – прерывание генерируется контроллером НГМД после завершения операции ввода/вывода
Fh	IRQ7 – прерывание от параллельного адаптера. Генерируется, когда подключенный к адаптеру принтер готов к выполнению очередной операции. Обычно не используется
10h	Обслуживание видеоадаптера
11h	Определение конфигурации устройств в системе
12h	Определение размера оперативной памяти
13h	Обслуживание дисковой системы
14h	Работа с асинхронным последовательным адаптером
15h	Расширенный сервис
16h	Обслуживание клавиатуры
17h	Обслуживание принтера
18h	Запуск BASIC в ПЗУ, если он есть
19h	Перезагрузка операционной системы
1Ah	Обслуживание часов

Окончание табл. 5.1

Номер	Описание
1Bh	Обработчик прерывания, возникающего, если пользователь нажал комбинацию клавиш <Ctrl+Break>
1Ch	Программное прерывание, вызывается 18,2 раза в секунду обработчиком аппаратного прерывания таймера
1Dh	Адрес видеотаблицы для контроллера видеоадаптера 6845
1Eh	Указатель на таблицу параметров дискеты
1Fh	Указатель на графическую таблицу для символов с кодами ASCII 128-255
20h – 5Fh	Используется MS-DOS или зарезервировано для MS-DOS
60h – 67h	Прерывания, зарезервированные для программ пользователя
68h – 6Fh	Не используются
70h	IRQ8 – прерывание от часов реального времени
71h	IRQ9 – прерывание от контроллера EGA
72h	IRQ10 – зарезервировано
73h	IRQ11 – зарезервировано
74h	IRQ12 – зарезервировано
75h	IRQ13 – прерывание от арифметического сопроцессора
76h	IRQ14 – прерывание от контроллера жесткого диска
77h	IRQ15 – зарезервировано
78h – 7Fh	Не используются
80h – 85h	Зарезервировано для BASIC
86h – F0h	Используются интерпретатором BASIC
F1h – FFh	Не используются

Прерывания, обозначенные как IRQ0 – IRQ15, являются внешними аппаратными.

5.3. Порядок обслуживания прерываний

ЦП, обнаружив сигнал прерывания, помещает в машинный стек слово состояния программы (определяющее различные флаги ЦП), регистр программного сегмента (CS) и указатель команд (IP) и блокирует систему прерываний. Затем ЦП с помощью 8-разрядного числа (номера прерывания), установленного на системной магистрали прерывающим процессом, извлекает из таблицы векторов адрес обработчика и возобновляет выполнение с этого адреса.

При наличии нескольких источников запросов прерывания должен быть установлен определенный порядок (дисциплина) в обслуживании поступающих запросов. Другими словами, между запросами (и соответствующими прерывающими программами) должны быть установлены приоритетные со-

отношения, определяющие, какой из нескольких поступивших запросов подлежит обработке в первую очередь, и устанавливающие, имеет право или не имеет данный запрос (прерывающая программа) прерывать ту или иную программу. Если наиболее приоритетный из выставленных запросов прерывания не превосходит по уровню приоритета выполняемую процессором программу, то запрос прерывания игнорируется или его обслуживание откладывается до завершения выполнения текущей программы. Каждому прерыванию соответствует определенный номер, который и определяет приоритет. Более приоритетным считается запрос с меньшим номером, т.е. наибольший приоритет имеет запрос прерывания с номером 0, а наименьший – запрос с номером 255.

Состояние системы в момент передачи управления обработчику прерываний совершенно не зависит от того, было ли прерывание возбуждено внешним устройством или явилось результатом выполнения программой команды INT. Это обстоятельство удобно использовать при написании и тестировании обработчиков внешних прерываний, отладку которых можно почти полностью выполнить, возбуждая их простыми программными средствами.

Аргументы передаются обработчикам прерываний через регистры или стек.

5.4. Установка обработчиков прерываний

Для установки корректных обработчиков прерываний таким образом, чтобы они не вступали в конфликт с функциями операционной системы или другими обработчиками прерываний, MS-DOS предоставляет специальные средства в виде функций программного прерывания **`int 21h`**, приведенные в табл. 5.2.

Таблица 5.2

Функции MS-DOS для работы с обработчиками прерываний

Функция	Действие
int 21h, функция 25h	Установить вектор прерывания
int 21h, функция 35h	Получить вектор прерывания
int 21h, функция 31h	Завершить и оставить программу резидентной

Эти функции дают возможность анализировать или модифицировать содержимое таблицы системных векторов прерываний и резервировать память для использования обработчиком, не вступая в конфликты с другими процессами в системе и не нарушая правила использования памяти.

В системе MS-DOS на функционирование обработчиков внешних аппаратных прерываний накладывается ряд весьма жестких ограничений:

- в силу отсутствия свойства повторной входимости у версий MS-DOS обработчик аппаратных прерываний в процессе обработки прерывания не должен вызывать функции MS-DOS;
- как только обработчик получает управление, он должен немедленно разблокировать прерывания, чтобы не нарушать работу других устройств и не снижать точность системных часов.

При составлении программы обработчика прерывания следует иметь в виду следующие правила:

- для модификации вектора прерывания необходимо использовать системные функции Int 21H; а не записывать в таблицу прерываний непосредственно;
- если ваша программа не единственный процесс в системе, использующий данный уровень прерываний, то после выполнения собственной обработки прерывания следует вернуться в предыдущий обработчик, к которому привязан ваш;
- если ваша программа не остается резидентной в памяти, следует получить и сохранить текущее состояние вектора прерываний перед его модификацией, а по завершении программы – восстановить исходное содержимое;
- если ваша программа остается резидентной в памяти, используйте одну из функций завершения и сохранения в памяти (предпочтительно int 21h с функцией 31H), чтобы зарезервировать для вашего обработчика требуемый объем памяти;
- если вы собираетесь обрабатывать аппаратные прерывания, следует, насколько возможно, сокращать время, в течение которого прерывания заблокированы, а также длину программы обработки прерывания.

5.5. Вопросы и задания для самопроверки

1. В чем состоит основное различие обычной подпрограммы и подпрограммы – обработчика прерывания?
2. Назовите три основных типа событий, вызывающих прерывания.
3. Какие отличия между программными и аппаратными прерываниями?
4. Какое из прерываний имеет более высокий приоритет: с номером 0 или с номером 10?
5. Объясните назначение таблицы векторов прерываний.
6. Каким образом можно установить собственный обработчик прерывания?

МОДУЛЬ 6. СЛОЖНЫЕ ТИПЫ ДАННЫХ

Цель модуля – изучение сложных типов данных, в частности, массивов и структур, и методов работы с ними.

В результате изучения модуля студенты должны:

- знать средства для описания массивов в программах;
- знать методы доступа к элементам массива;
- знать особенности работы с двухмерными массивами;
- знать синтаксис описания шаблона структуры;
- уметь определять данные структур;
- знать методы работы со структурой.

Содержание модуля

6.1. Массивы

6.1.1. Способы описания массивов в программе

Массив – структурированный тип данных, состоящий из некоторого числа элементов одного типа.

Специальных средств описания массивов в программах Ассемблера, конечно, нет. При необходимости использовать массив в программе его нужно моделировать одним из следующих способов:

1. Перечислением элементов массива в поле операндов одной из директив описания данных. При перечислении элементы разделяются запятыми, например:

```
mas db 1,2,3,4,5
```

2. Используя оператор повторения **dup**. Пример:

```
mas dw  5 dup (0)    ;массив из 5 нулевых
                     ;элементов (слов)
```

3. Используя директивы **label** и **rept**. Пара этих директив может облегчить описание больших массивов в памяти и повысить наглядность такого описания. Директива **rept** относится к макросредствам языка Ассемблера и вызывает повторение указанное число раз строк, заключенных между директивой и строкой **endm**. Например, определим массив байт в области памяти, обозначенной идентификатором **mas_b**. В данном случае директива **label** определяет символическое имя **mas_b**, аналогично тому, как это делают директивы резервирования и инициализации памяти. Достоинство директивы **label** – в том, что она не резервирует память, а лишь определяет характеристики объекта. В данном случае объект – это ячейка памяти. Используя несколько директив **label**, записанных одна за другой, можно присвоить одной и той же области памяти разные имена и разный тип, что и сделано в следующем фрагменте:

```
mas_b   label   byte
mas_w   label   word
rept 4
dw 0f1f0h
endm
```

В результате в памяти будет создана последовательность из четырех слов f1f0. Эту последовательность можно трактовать как массив байт или слов в зависимости от того, какое имя области мы будем использовать в программе – **mas_b** или **mas_w**.

6.1.2. Доступ к элементам массива

Сами по себе данные не несут никакой информации о своем типе, поэтому при работе с массивами необходимо четко представлять себе, что все элементы массива располагаются в памяти компьютера последовательно. Само по себе такое расположение ничего не говорит о назначении и порядке использования этих элементов. Программист с помощью составленного им алгоритма обработки определяет, как нужно трактовать эту последовательность байт, составляющих массив. Так, одну и ту же область памяти можно трактовать как одномерный массив, и одновременно те же самые данные могут трактоваться как двухмерный массив. Все зависит только от алгоритма обработки этих данных в конкретной программе.

Для того чтобы локализовать определенный элемент массива, к его имени нужно добавить индекс. Так как мы моделируем массив, то должны позаботиться и о моделировании индекса. В языке Ассемблера индексы массивов – это обычные адреса, но с ними работают особым образом. Другими словами, когда при программировании на Ассемблере мы говорим об индексе, то подразумеваем под этим не номер элемента в массиве, а некоторый адрес.

Давайте еще раз обратимся к описанию массива. Например, в программе статически определена последовательность данных:

```
mas dw  0,1,2,3,4,5
```

Пусть эта последовательность чисел трактуется как одномерный массив. Размерность каждого элемента определяется директивой **dw**, то есть она равна 2 байта. Чтобы получить доступ к третьему элементу, нужно к адресу массива прибавить 6. Нумерация элементов массива в Ассемблере начинается с нуля. То есть в нашем случае речь, фактически, идет о 4-м элементе массива – 3, но об этом знает только программист; микропроцессору в данном случае все равно – ему нужен только адрес.

В общем случае для получения адреса элемента в массиве необходимо начальный (базовый) адрес массива сложить с произведением индекса (номер элемента минус единица) этого элемента на размер элемента массива:

```
база + (индекс*размер элемента)
```

Микропроцессор позволяет масштабировать индекс. Это означает, что если указать после имени индексного регистра знак умножения «*» с последующей цифрой 2, 4 или 8, то содержимое индексного регистра будет умножаться на 2, 4 или 8, то есть масштабироваться.

Применение масштабирования облегчает работу с массивами, которые имеют размер элементов, равный 2, 4 или 8 байт, так как микропроцессор сам производит коррекцию индекса для получения адреса очередного элемента массива. Нам нужно лишь загрузить в индексный регистр значение требуемого индекса (считая от 0). Кстати сказать, возможность масштабирования появилась в микропроцессорах Intel начиная с модели i486.

Примеры:

```
mov ax,mas[ebx][ecx*2]
;адрес операнда равен [mas+(ebx)+(ecx)*2]
sub dx,[ebx+8][ecx*4]
;адрес операнда равен [(ebx)+8+(ecx)*4]
```

Если размерность элементов не равна 2, 4 или 8, то организовывать обращение к элементам массива нужно обычным способом.

6.1.3. Двухмерные массивы

Специальных средств для описания такого типа данных в Ассемблере нет. Двухмерный массив нужно моделировать. На описании самих данных это почти никак не отражается – память под массив выделяется с помощью директив резервирования и инициализации памяти.

Непосредственно моделирование обработки массива производится в сегменте кода, где программист, описывая алгоритм обработки Ассемблеру, определяет, что некоторую область памяти необходимо трактовать как двухмерный массив. При этом вы вольны в выборе того, как понимать расположение элементов двухмерного массива в памяти: по строкам или по столбцам.

Если последовательность однотипных элементов в памяти трактуется как двухмерный массив, расположенный по строкам, то адрес элемента (i, j) вычисляется по формуле:

```
адрес = база + i·C·width + j·width
```

Здесь `i` = 0...n – 1 указывает номер строки, а `j` = 0...m – 1 указывает номер столбца, `C` – количество столбцов, `width` – размер элемента в байтах.

Например, пусть имеется массив чисел (размером в 1 байт) mas(i, j) с размерностью 4 на 4 (i = 0...3, j = 0...3):

```
23    04    05    67
05    06    07    99
67    08    09    23
87    09    00    08
```

В памяти элементы этого массива будут расположены в следующей последовательности:

```
23 04 05 67 05 06 07 99 67 08 09 23 87 09 00 08
```

Если мы хотим трактовать эту последовательность как двухмерный массив, приведенный выше, и извлечь, например, элемент mas(2, 3) = 23, то, используя представленную выше формулу, убедимся в правильности наших рассуждений:

```
адрес(mas(2,3)) = mas + 4·1·2 + 3 = mas + 11 = 23
```

В программе это будет выглядеть так:

```
;Фрагмент программы выборки элемента
;массива mas(2,3) и его обнуления
.data
mas db
```

```
23,4,5,67,5,6,7,99,67,8,9,23,87,9,0,8
i=2
j=3
.code
...
  mov   si,4*1*i
  mov   di,j
  mov   al,mas[si][di]   ;в al элемент mas(2,3)
```

6.2. Структуры

Часто в приложениях возникает необходимость рассматривать некоторую совокупность данных разного типа как некоторый единый тип. Это очень актуально, например, для программ баз данных, где необходимо связывать совокупность данных разного типа с одним объектом. Такой объект обычно описывается с помощью специального типа данных – *структуры*. С целью повысить удобство использования языка Ассемблера в него также был введен такой тип данных.

Структура – это тип данных, состоящий из фиксированного числа элементов разного типа.

Для использования структур в программе необходимо выполнить три действия:

1. Задать шаблон структуры. Это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.
2. Определить экземпляр структуры. Этот этап подразумевает инициализацию конкретной переменной заранее определенной (с помощью шаблона) структурой.
3. Организовать обращение к элементам структуры.

Очень важно уяснить, в чем разница между *описанием* структуры в программе и ее *определением*.

Описание структуры в программе означает лишь указание ее схемы или шаблона; память при этом не выделяется. Этот шаблон можно рассматривать лишь как информацию для транслятора о расположении полей и их значении по умолчанию.

Определение структуры – указание транслятору выделить память и присвоить этой области памяти символическое имя. Описать структуру в программе можно только один раз, а определить – любое количество раз.

6.2.1. Описание шаблона структуры

Описание шаблона структуры имеет следующий синтаксис:

```
имя_структуры     STRUC
    <описание полей>
имя_структуры     ENDS
```

Здесь **<описание полей>** представляет собой последовательность директив описания данных **db, dw, dd, dq** и **dt**. Их операнды определяют размер полей и, при необходимости, начальные значения. Этими значениями будут, возможно, инициализироваться соответствующие поля при определении структуры.

Местоположение шаблона в программе может быть произвольным, но, следуя логике работы однопроходного транслятора, он должен быть расположен до того места, где определяется переменная с типом данной структуры. То есть, при описании в сегменте данных переменной с типом некоторой структуры ее шаблон необходимо поместить в начале сегмента данных либо перед ним.

Рассмотрим работу со структурами на примере моделирования базы данных о сотрудниках некоторого отдела. Для простоты, чтобы уйти от проблем преобразования информации при вводе, условимся, что все поля символьные. Определим структуру записи этой базы данных следующим шаблоном:

```
worker   struc                    ;информация о сотруднике
nam db   30  dup (' ')            ;фамилия, имя, отчество
sex db   'м'          ;пол, по умолчанию 'м' – мужской
position db  30 dup (' ')         ;должность
age db   2 dup(‘ ’)               ;возраст
standing db  2 dup(‘ ’)           ;стаж
salary   db  4 dup(‘ ’)           ;оклад в рублях
birthdate    db  8 dup(‘ ’)       ;дата рождения
worker   ends
```

6.2.2. Определение данных с типом структуры

Для использования описанной с помощью шаблона структуры в программе необходимо определить переменную с типом данной структуры. Для этого используется следующая синтаксическая конструкция:

```
[имя переменной] имя_структуры <[список значений]>
```

Здесь:

- `имя переменной` – идентификатор переменной данного структурного типа. Задание имени переменной необязательно. Если его не указать, будет просто выделена область памяти размером в сумму длин всех элементов структуры;
- `список значений` – заключенный в угловые скобки список начальных значений элементов структуры, разделенных запятыми. Его задание также необязательно. Если список указан не полностью, то все поля структуры для данной переменной инициализируются значениями из шаблона, если таковые заданы. Допускается инициализация отдельных полей, но в этом случае пропущенные поля должны отделяться запятыми. Пропущенные поля будут инициализированы значениями из шаблона структуры. Если при определении новой переменной с типом данной структуры мы согласны со всеми значениями полей в ее шаблоне (то есть заданными по умолчанию), то нужно просто использовать угловые скобки.

Для примера определим несколько переменных с типом описанной выше структуры:

```
data segment
sotr1 worker <'Гурко',,'худож.','33',‘15’,‘1800’,'26.01.64'>
sotr2 worker <'Степанов',,'худож.','38','20','1750','01.01.58'>
sotr3 worker<'Юрова','ж','связист','32','2',,'09.01.66'>
sotr4 worker<>   ;здесь все значения по умолчанию
data ends
```

6.2.3. Методы работы со структурой

Для того чтобы сослаться в команде на поле некоторой структуры, используется специальный оператор – символ «.» (точка). Он используется в следующей синтаксической конструкции:

```
адресное_выражение.имя_поля_структуры
```

Здесь:

- **адресное_выражение** – идентификатор переменной некоторого структурного типа или выражение в скобках в соответствии с указанными ниже синтаксическими правилами;
- **имя_поля_структуры** – имя поля из шаблона структуры. Это, на самом деле, тоже адрес, а точнее, смещение поля от начала структуры.

Таким образом, оператор «.» (точка) вычисляет выражение

```
(адресное_выражение) + (имя_поля_структуры)
```

Продемонстрируем на примере определенной нами структуры **worker** некоторые приемы работы со структурами. Например, извлечем в AX значение поля с возрастом. Будьте внимательны, так как из-за принципа хранения данных «младший байт по младшему адресу» старшая цифра возраста будет помещена в AL, а младшая – в AH. Для корректировки достаточно использовать команду **xchg**:

```
mov ax,word ptr sotr1.age ;в al возраст sotr1
xchg ah,al
```

или

```
lea bx,sotr1
mov ax,word ptr [bx].age
xchg ah,al
```

Если сотрудников не четверо, а намного больше, и к тому же их число и информация о них постоянно меняются, то теряется смысл явного определения переменных с типом **worker** для конкретных личностей. Язык Ассемблера разрешает определять не только отдельную переменную с типом структуры, но и массив структур. Например, массив из 10 структур типа **worker** можно определить следующим образом:

```
mas_sotr worker  10 dup (<>)
```

Дальнейшая работа с массивом структур производится так же, как и с одномерным массивом. Однако здесь возникает несколько вопросов: как быть с размером и как организовать индексацию элементов массива?

Аналогично другим идентификаторам, определенным в программе, транслятор назначает имени типа структуры и имени переменной с типом структуры атрибут типа. Значением этого атрибута является размер в байтах, занимаемый полями этой структуры. Извлечь это значение можно с помощью оператора **type**. После того как стал известен размер экземпляра структуры, организовать индексацию в массиве структур не представляет особой сложности. Пример:

```
Worker struc
...
Worker ends
...
mas_sotr worker 10 dup (<>)
...
mov bx,type worker   ;bx=77
lea di,mas_sotr
```

```
;извлечение и вывод на экран пола всех сотрудников:
mov cx,10
cycl:
mov al,[di].sex
...
;вывод на экран содержимого поля sex структуры worker
...
add di,bx          ;к следующей структуре в массиве
loop cycl
```

6.3. Вопросы и задания для самопроверки

1. Какими способами можно описать массивы в программе?
2. Для чего используются директивы **`label`** и **`rept`**?
3. Напишите формулу для вычисления адреса элемента в двухмерном массиве.
4. Дайте определение структуры.
5. Каким образом описывается шаблон структуры?
6. Каким образом осуществляется доступ к полям структуры?

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Юров, В. И. Assembler : учеб. для вузов. – 2-е изд. / В. И. Юров – СПб. : Питер, 2003 – 637 с.
2. Абель, П. Язык Ассемблера для IBM PC и программирования : пер. с англ. / П. Абель – М. : Высш. шк., 1992. – 477 с.
3. Бредли, Д. Программирование на языке ассемблера для персональной ЭВМ фирмы IBM : пер. с англ. / Д. Бредли. – М. : Радио и связь, 1988. – 448 с.
4. Гук, М. Аппаратные средства IBM PC : энциклопедия / М. Гук. – СПб. : Питер, 2001.
5. Использование Turbo Assembler при разработке программ. – Киев : Диалектика, 1994. – 228 с.
6. Кулаков, В. Программирование на аппаратном уровне : спец. справ. / В. Кулаков . – СПб. : Питер, 2003.
7. Майко, Г. В. Ассемблер для IBM PC / Г. В. Майко – М. : Бизнес-Информ, Сирин, 1997. – 212 с.
8. Пильщиков, В. Н. Программирование на языке ассемблера IBM PC / В. Н. Пильщиков . – М. : ДИАЛОГ-МИФИ, 1996. – 288 с.
9. Скэнлон, Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера : пер. с англ. / Л. Скэнлон . – М. : Радио и связь, 1991. – 336 с.

Учебное издание

РУГОЛЬ Дмитрий Геннадьевич

НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ

Учебно-методический комплекс
для студентов специальностей 1-40 02 01 «Вычислительные машины,
системы и сети», 1-40 01 01 «Программное обеспечение информационных технологий»,
1-36 04 02 «Промышленная электроника»

Редактор *Т. В. Булах*

Дизайн обложки *В. А. Виноградовой*

Подписано в печать 17.12.08 Формат 60x84/16 Бумага офсетная Гарнитура Таймс
Ризография Усл.-печ. л. 7,43 Уч.-изд. л. 7,2 Тираж 130 экз. Заказ 2017

Издатель и полиграфическое исполнение –
учреждение образования «Полоцкий государственный университет»

ЛИ № 02330/0133020 от 30.04.04 ЛП № 02330/0133128 от 27.05.04
211440 г. Новополоцк, ул. Блохина, 29