**Московский государственный университет путей сообщения (МИИТ)**

**Институт управления, телекоммуникаций и электрификации**

---

**Кафедра «Автоматика и телемеханика на железнодорожном транспорте»**

**П.Е. МАЩЕНКО,**
**А.М. РОМАНЧИКОВ**

# АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ МИКРОПРОЦЕССОРА INTEL 8086

## УЧЕБНОЕ ПОСОБИЕ

**Для студентов 3 – го курса специальности**

**«Автоматика, телемеханика и связь на железнодорожном транспорте».**

**Москва – 2011**

**Московский государственный университет путей сообщения (МИИТ)**

**Институт управления, телекоммуникаций и электрификации**

---

**Кафедра «Автоматика и телемеханика на железнодорожном транспорте»**

**П.Е. МАЩЕНКО,**
**А.М. РОМАНЧИКОВ**

# АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ МИКРОПРОЦЕССОРА INTEL 8086

## УЧЕБНОЕ ПОСОБИЕ

**Для студентов 3 – го курса специальности**

**«Автоматика, телемеханика и связь на железнодорожном транспорте».**

**Москва – 2011**

**П.Е. Мащенко, А.М. Романчиков**
Учебное пособие. – М.: МИИТ, 2011. – 128 с.

В учебном пособии рассмотрены принципы программирования микропроцессоров серии Intel 8086.

Учебное пособие предназначено для изучения разделов курса специальной дисциплины «Прикладное программирование и основы микропроцессорной техники».

**© Московский государственный университет путей сообщения (МИИТ), 2011**

# Содержание

Стр.

Введение………………………………………………… 5

1. Краткое описание микропроцессора Intel 8086……………. 7

2. Архитектура микропроцессора Intel 8086 ………………... 10

3. Системы счисления и операции с ними, применяемые при программировании микропроцессоров серии Intel 8086…….. 17

4. Программирование линейных алгоритмов на базе микропроцессора Intel 8086…………………………………… 25

5. Программирование ветвящихся алгоритмов на базе микропроцессора Intel 8086…………………………………… 32

6. Обработка структур данных (массивов), программирование циклических алгоритмов, операции с памятью на базе микропроцессора Intel 8086………………… 58

7. Обработка символьной информации на базе микропроцессора Intel 8086…………………………………… 76

8. Варианты индивидуальных заданий……………………... 94

9. Вопросы для самопроверки…………………………………. 111

10. Рекомендуемая литература ………………………………... 113

Приложение 1. Работа с отладчиком Insight, компилятором TASM и компоновщиком TLINK……………………………. 115

Приложение 2. Система команд микропроцессора Intel 8086.. 121

# Введение

Целью изучения дисциплины “Прикладное программирование и основы микропроцессорной техники” является формирование у студентов знаний общей методологии, а также конкретных методов проектирования основных разновидностей современных микропроцессорных средств.

Специализированная микропроцессорная техника на железной дороге применяется в системах обеспечения безопасности движения поездов: сигнализации, автоблокировке (АБ), автоматической локомотивной сигнализации (АЛСН), системах диспетчерского контроля и диспетчерской централизации (ДК/ДЦ), системах автоматического управления тормозами (САУТ), системах контроля состояния пути (вагоны-путеизмерители и дефектоскопы), системах контроля подвижного состава на ходу поезда (ДИСК), комплексных устройствах локомотивной безопасности (КЛУБ) и т. д. Микропроцессорная техника постепенно вытесняет предшествующую ей релейную технику с аналогичным, однако, значительно меньшим, набором функций.

Необходимо учитывать, что современное техническое оборудование предполагает обязательное использование микропроцессоров в профессиональной деятельности инженера, конструктора или технолога. Уникальность применения микропроцессорных средств состоит, прежде всего, в том, что, не изменяя как таковое физическое их устройство, можно заставить микропроцессор выполнять самые различные функции, превращая

его либо в систему автоматического проектирования сложных устройств, либо в обучающее устройство и т.д.

В настоящем пособии описывается выполнение практических работ по дисциплине «Прикладное программирование и основы микропроцессорной техники». Подчеркиванием выделены конкретные требования и указания по выполнению работы. Обычным шрифтом дается необходимая информация для выполнения работы. Полужирным выделена часть текста, на которую следует обратить особое внимание.

# 1. КРАТКОЕ ОПИСАНИЕ МИКРОПРОЦЕССОРА INTEL 8086

В 1978 г. рынок 8-битных микропроцессоров был переполнен, и вместо того, чтобы продолжать борьбу на нём, фирма Intel сделала качественный шаг вперёд и выпустила первый в мире 16-битный микропроцессор.

Технические характеристики микропроцессора Intel 8086:

- Тактовая частота (МГц): 5 (модель 8086), 8 (модель 8086-2), 10 (модель 8086-1);
- Разрядность регистров: 16 бит;
- Разрядность шины данных: 16 бит;
- Разрядность шины адреса: 20 бит;
- Объём адресуемой памяти: 1 Мбайт;
- Количество транзисторов: 29 000;
- Площадь кристалла (кв. мм): 16 мм²;
- Максимальное тепловыделение: 1,75 Вт;
- Напряжение питания: +5 В;
- Разъём: нет (микросхема припаивалась к плате).

16-битный микропроцессор Intel 8086 (рис. 1.1), содержит на кристалле размером 5,5х5,5 мм около 29000 транзисторов и производится по высококачественной nМОП-технологии. Производительность процессора 8086 значительно превышает производительность его 8-битного предшественника – микропроцессора 8080 – и составляет от 330 до 750 тыс. операций в секунду. Хотя и имеется определённая совместимость микропроцессора 8086 с архитектурой микропроцессора 8080, разработчики не ставили перед собой цели достичь её полностью.

Число линий адреса увеличено с 16 до 20, что позволяет адресовать память 1 Мбайт вместо 64 Кбайт. Увеличение ёмкости памяти упрощает переход к мультипрограммированию, поэтому в микропроцессоре 8086 предусмотрено несколько мультипрограммных возможностей. Кроме того, в микропроцессор 8086 встроены некоторые средства, упрощающие реализацию мультипроцессорных систем, что позволяет применять его с другими процессорами, например с процессором числовых данных 8087.

Рис. 1.1

То обстоятельство, что 16 из линий адреса микропроцессора используются и как линии данных, приводит к тому, что на системную шину нельзя одновременно выдавать адреса и данные. Мультиплексирование адресов и данных во времени сокращает число контактов корпуса до 40, но и замедляет скорость передачи данных. Однако благодаря тщательно разработанной временной диаграмме работы скорость передачи уменьшается не столь значительно, как этого следовало бы ожидать. Микропроцессор имеет 16 линий управления. Он рассчитан на одно напряжение питания +5 В и однофазную синхронизацию, частота которой достигает 5 МГц..

Система команд микропроцессора 8086 состоит из 98 команд: 19 команд передачи данных, 38 команд их обработки, 24 команд перехода и 17 команд управления процессором. Каждая команда состоит из кода операции, идентифицирующего её, и операндов, несущих требуемую для операции информацию. Команды могут содержать несколько операндов, но чем больше операндов и чем они длиннее, тем больше места занимает команда в памяти и тем больше времени требуется для передачи её в ЦП. Коды операций и операнды могут иметь произвольную длину и не обязаны быть непрерывными; в то же время общая длина команды должна выражаться целым числом байт. Чтобы минимизировать общее число бит в команде, большинство команд процессора 8086 имеют не более двух операндов, причём минимум одним из операндов в двухоперандной команде является регистр, так как адреса памяти и ввода-вывода требуют сравнительно много бит (8…20), а из-за ограниченного числа регистров для определения регистра требуется всего несколько бит. Ограничение двумя операндами, конечно, уменьшает гибкость многих операций, но в действительности излишняя гибкость и не нужна. Например, команда сложения, в которой необходимо указывать два слагаемых и результат, приводится к двум операндам посредством загрузки суммы на место одного из слагаемых. При этом оно теряется, но обычно это не играет роли. Если же слагаемое потребуется в дальнейшем, его приходится дублировать (запоминать где-то ещё) до выполнения сложения.

## 2. АРХИТЕКТУРА МИКРОПРОЦЕССОРА INTEL 8086

В микропроцессоре Intel 8086 есть возможность адресации 1 Мбайта памяти, обращения к 65536 устройствам ввода и такому же количеству устройств вывода информации. В i8086 имеется возможность изменения внутренней аппаратной конфигурации с помощью специального управляющего сигнала. В более простом режиме 8086 ориентирован на использование в простых вычислительных и управляющих устройствах. При этом микропроцессор сам вырабатывает сигналы управления шиной и обеспечивает прямой доступ к ней посредством контроллера Intel 8257. В режиме полной конфигурации обеспечивается работа с контроллером шины 8288, который декодирует три сигнала состояния процессора и в зависимости от них выдает семь сигналов управления шиной. Такой режим используется в мультипроцессорных системах и в сложных вычислительных устройствах, в частности, в компьютере IBM PC/XT.

Интересно организована память: хранение 16-разрядных слов осуществляется в виде отдельных байтов, причем байты, передающиеся по восьми младшим линиям шины данных (D7-D0), собраны в банк 0, а передаваемые по восьми старшим линиям — в банк 1. Объем каждого банка составляет 512 Кбайт. Таким образом, нечетные байты хранятся в банке 1, а четные — в банке 0. Выбор банка осуществляется с помощью младшего адреса и сигнала управления старшими разрядами шины данных.

Еще одна важная особенность — возможность обработки 256 типов прерываний (от 0 до 255), в том числе есть прерывания, определяемые пользователем, и пошаговые прерывания.

Микропроцессор Intel 8086 приспособлен для работы с несколькими процессорами в одной системе, причем возможно использование, как независимых процессоров, так и сопроцессоров. Отличие заключается в том, что независимый процессор выполняет свою собственную последовательность команд, а сопроцессор следит за потоком команд центрального процессора и выделяет из него "свои" команды, расширяя набор команд основного процессора и улучшая таким образом характеристики системы. Для поддержки этих режимов используются команды ESC, LOCK и XCHG, а также специальные управляющие сигналы, позволяющие разрешать конфликты доступа к общим ресурсам.

Внешние шины адреса и данных в i8086 объединены, и поэтому наличие на шине в данный момент времени информации или адреса определяется порядковым номером такта внутри цикла. Процессор ориентирован на параллельное выполнение команды и выборки следующей команды. Команда выбирается из памяти и принимается микропроцессором в свободный регистр очереди команд, причем в то же самое время выполняется предыдущая команда. Конвейеризация команд позволяет значительно повысить быстродействие системы. При выполнении команд проверяются состояния входов запросов прерываний и захвата шины, и при необходимости выполняются соответствующие действия.

Микропроцессор i8086 (рис. 2.1) состоит из трех основных частей: устройства сопряжения шины, устройства обработки и устройства управления и синхронизации.

Устройство сопряжения шины состоит из шести 8-разрядных регистров очереди команд, четырех 16-разрядных регистров адреса команды, 16-разрядного регистра команды и 16-разрядного сумматора адреса (см. рис. 2.1). Оно выполняет следующие функции: выбирает команды из памяти и записывает их в регистр очереди команд, вычисляет и формирует физический адрес, читает операнды из памяти или из регистров и записывает результат выполнения команд в память или в регистры.

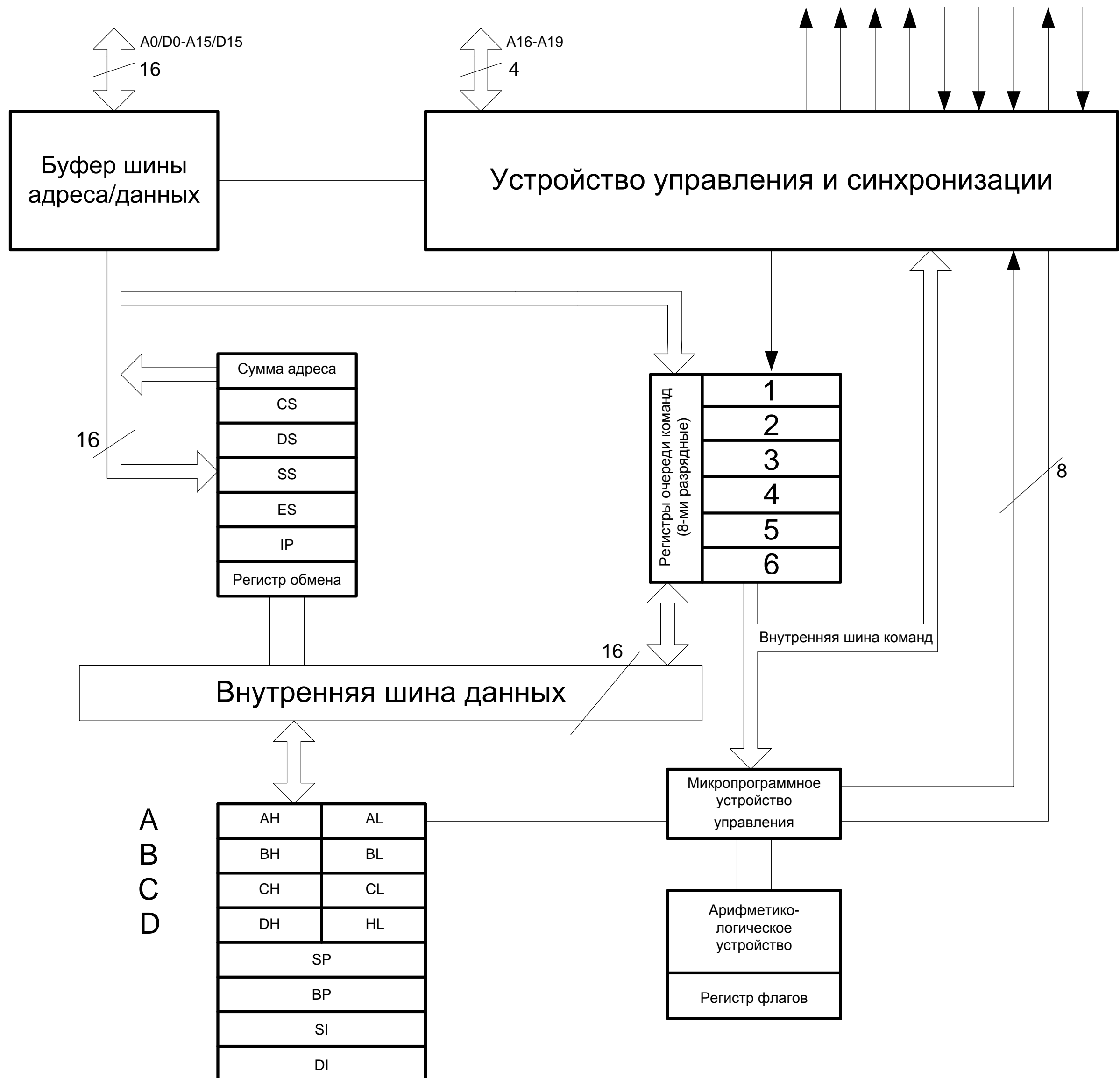



Рис. 2.1. Внутренняя структура микропроцессора 8086

Устройство обработки преобразует данные. Команда из очереди команд по запросу устройства обработки поступает на внутреннюю шину команд, а с нее на микропрограммное устройство управления, декодирующее ее и генерирующее соответствующие последовательности микрокоманд, необходимые для выполнения текущей операции. В отличие от первых

микропроцессоров, устройство обработки в 8086 не связано с внешней шиной, а обменивается с ней информацией через регистр обмена устройства сопряжения шины.

Устройство обработки содержит 16-разрядное арифметико-логическое устройство, восемь 16-разрядных регистров общего назначения и 16-разрядный регистр флагов. Регистры могут использоваться как 16-разрядные или как пары 8-разрядных (при этом их количество удваивается).

Внедрение микропроцессорных устройств задевает все сферы деятельности человека, в том числе и железнодорожный транспорт. В настоящее время на железных дорогах мира преобладают релейные системы автоматики и телемеханики, где в качестве элементной базы используются специализированные реле. В России в течение последних 60 лет развитие систем автоматики и телемеханики происходило по следующим направлениям:

- повышение пропускной способности перегонов и станций;
- типизация схем с целью упрощения проектирования, строительства и обслуживания систем;
- повышение безопасности движения поездов;
- расширение функциональных возможностей;
- увязка с системами верхнего уровня и диагностическими системами.

Автоматизация технологических процессов управления движением поездов на станциях и перегонах оставалась консервативной областью в отношении применения

компьютерных технологий. Следует учитывать, что технические решения и средства для релейной централизации разрабатывались в 1960 – 1980 гг. и к настоящему моменту явно устарели. Реле как элементная база электрической централизации практически себя исчерпали. Попытки получения новых качественных показателей и расширения функций релейной централизации ведут к увеличению числа реле, потребляемой электроэнергии, затрат на техническое обслуживание, объемов проектных и монтажных работ. Поэтому целесообразно использовать в качестве технических средств автоматизации технологических процессов управления движением поездов микропроцессорные системы, успешно эксплуатируемые на зарубежных железных дорогах. Данные системы имеют самодиагностику, легко стыкуются с любыми аппаратно-программными комплексами для создания единой автоматизированной системы управления, позволяют размещать аппаратуру в существующих помещениях, экономить кабель при децентрализованном размещения оборудования путем использования волоконно-оптического кабеля, одновременно решая вопросы по помехозащищенности от источников перенапряжения, а также решают вопросы бесконтактного управления стрелками и сигналами. Минимальное количество релейной аппаратуры позволяет говорить о реальном сокращении, как штата, так и эксплуатационных расходов.

К микропроцессорным или релейно-процессорным системам можно отнести Ebilock-950, ЭЦ-ЕМ, МПЦ-2, МПЦ-И, ЭЦ-МПК, «Диалог-Ц», «Сетунь», «Диалог», «Тракт», ДЦ-МПК «Юг»,

АБТЦ-М, АБТЦ-ЕМ, АБ-ЧКЕ, КЭБ-2, АБ-УЕ, АПК-ДК, АСДК, АДК СЦБ, ГАЦ-МН и другие.

Таким образом, релейные системы практически исчерпали себя для расширения функциональных возможностей, а непрерывно растущие требования к системам управления движением поездов определяют общую тенденцию перехода на микропроцессорную технику и бесконтактное управление.

Для штатных работников железнодорожного транспорта представляет большую сложность переход от обслуживания и тестирования релейных систем к микропроцессорным устройствам. Тем более выпускники ВУЗов, идущие работать в проектные организации, должны представлять себе работу микропроцессорных устройств. Ввиду выше приведённых рассуждений изучение программирования микропроцессорных устройств становится одним из приоритетных направлений обучения в ВУЗе. Далее в учебном пособии будут рассмотрены принципы и тонкости программирования на языке Assembler для наиболее распространённого микропроцессора Intel 8086. Так как микропроцессор работает с шестнадцатиричными числами, то первым шагом в понимании работы микропроцессорных устройств должно быть приобретение навыков по переводу чисел из одних систем счисления в другие и выполнения основных арифметических операций над ними.

# 3. СИСТЕМЫ СЧИСЛЕНИЯ И ОПЕРАЦИИ С НИМИ, ПРИМЕНЯЕМЫЕ ПРИ ПРОГРАММИРОВАНИИ МИКРОПРОЦЕССОРОВ СЕРИИ INTEL 8086

Цель работы – научиться понимать запись чисел в различных системах счисления, переводить числа из одной системы счисления в другую и производить арифметические операции над ними.

Ниже рассмотрены системы счисления и операции с ними с использованием конкретного примера. Для того, чтобы у каждого студента было индивидуальное задание, в качестве примера можно выбрать число, месяц и год рождения студента. Системы счисления и операции с ними, рассмотренные в данном пункте учебного пособия, являются достаточными для освоения курса “Прикладное программирование и основы микропроцессорной техники”. Для более же продвинутых студентов рекомендуется воспользоваться литературой [1, 2].

Выполнение задания состоит из следующих пунктов:

1) Исходные данные переводятся в двоичную и шестнадцатеричную систему счисления. Шестнадцатеричные значения получаются путем последовательного деления исходного числа на 16. Получаемые при этом остатки, прочитанные справа-налево (с последнего остатка) и составят искомое число. Если число меньше 16, то следует воспользоваться таблицей 3.1, приведенной ниже.

Таблица 3.1

| Десятичная | Шестнадцатеричная | Двоичная |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |
| 16 | 10 | 10000 |

Рассмотрим на примере**:** дата рождения студента – 7.09.1977

Число рождения при переводе в шестнадцатеричную и двоичную системы счисления примет вид: 7 – 7h – 0111b (по табл. 3.1). Обозначение h означает шестнадцатеричную систему счисления (hexadecimal), b – двоичную (binary), у десятичного числа обычно не ставится специального обозначения, иногда ставится d (decimal), в случае необходимости указания системы

счисления. Вместо буквенных обозначений (как принято у программистов) в математике принято указывать основание системы счисления в круглых скобках в нижнем индексе: $7_{(10)}$ – $7_{(16)}$ – $0111_{(2)}$. При выполнении задания допускаются оба вида обозначений.

Месяц рождения при переводе в шестнадцатеричную и двоичную системы счисления примет вид: 9 – 9h – 1001b.

Год рождения при переводе в шестнадцатеричную и двоичную системы счисления примет вид: 7B9h. Поясним подробнее.

1977/16 = 123 ост 9

123/16 = 7 ост 11

7/16 = 0 ост 7

Обратите внимания, что остатки больше 9 записываются в виде шестнадцатеричной цифры (по табл. 3.1). Деление также удобно производить в столбик, используя полученное частное снова как делимое и обводя остатки как показано ниже.

```
-1977 | 16
 16   |------
 ---- | 123  | 16
 -37  |-112  |-----
  32  | ---- | (7)
 ---- | (11)
  -57
   48
  ----
   (9)
```

Переводить число из шестнадцатеричной в двоичную систему счисления удобно пользуясь правилом, что **одно шестнадцатеричное число записывается четырьмя двоичными разрядами**. Таким образом, по табл. 3.1 видно, что 7 – 0111, B – 1011, 9 – 1001, искомое число 7B9h – 0111 1011 1001 b. Можно

также получить данное двоичное число путем последовательного деления 1977 на 2, однако этот путь менее эффективен, хотя его также разрешается использовать. Итак, результат первой части задания.

**7 – 7h – 0111b**

**9 – 9h – 1001b**

**1977 – 7B9h – 0111 1011 1001 b**

2) Производится вычисление формулы день·день-год+месяц в шестнадцатеричной системе счисления.

Для рассмотренного примера получим:

$$день \cdot день = 7 \cdot 7 = 49d = 31h$$

В случае, если день даты рождения двухзначное число, умножение необходимо произвести в шестнадцатеричном виде, переводя промежуточные результаты в шестнадцатеричную систему счисления. Например, если день 26 – 1Ah, то умножение стоит производить следующим образом:

$$\begin{array}{r} \times \begin{array}{r} 1A \\ 1A \end{array} \\ \hline + \begin{array}{r} 104 \\ 1A\phantom{0} \end{array} \\ \hline 2A4 \end{array}$$

При этом необходимо выполнить следующие действия (как и при десятичном умножении в столбик): A·A – это 10·10, то есть 100; 100 – это 64h, соответственно 4 записываем 6 переносим в следующий разряд («в уме»), A·1 и +6, перенесенные из предыдущего разряда – это 16, т.е. 10h, таким образом записываем «10». Затем умножаем первый сомножитель на второй разряд

(получаем «1A») и вычисляем сумму поразрядно («в столбик»), как и при обычном десятичном умножении.

Для того, чтобы быстро перевести небольшое число в шестнадцатеричную систему счисления, можно запомнить произведение 16-ти на натуральные числа.

16·1=16 – 10h

16·2=32 – 20h

16·3=48 – 30h

16·4=64 – 40h

16·5=80 – 50h

16·6=96 – 60h

16·7=112 – 70h

16·8=128 – 80h

16·9=144 – 90h

16·10=160 – A0h

Для того, чтобы перевести число в шестнадцатеричную систему счисления, находим минимальное ближайшее к искомому число, и выясняем, сколько нужно к нему прибавить для получения искомого. Например, число 54 это 48+6, значит это 36h; 92 это 80+12, значит это 5Ch и т.п.

Затем от полученного произведения необходимо вычесть год. При этом следует учесть, что результат в любом случае получится отрицательным, и его необходимо представить в **дополнительном коде**.

Для рассмотренного примера получим:

день·день-год=31h-7B9h.

Чтобы произвести вычитание необходимо вычесть из большего меньшее (7B9h-31h=788h) и получить дополнительный код в соответствии с определением, произведя вычитание получившегося числа из 10000h.

```
  ....
  10000
-   788
  -----
   F878
```

При этом, как и при обычном вычитании, производится заем из первого значащего разряда («1»), при этом в младшем разряде вычитание производится из 16 (10h), а в остальных из 15 (Fh); как и в десятичной системе счисления – вычитание бы происходило в младшем разряде из 10, а в остальных из 9. Тот же результат можно получить и произведя непосредственно вычитание из меньшего большее так, как это будет сделано командой процессора Intel 8086, при этом заем надо производить из «несуществующего» старшего разряда (из 11h (17) вычитаем 9, получаем 8; из 12h (18) вычитаем Bh (11), получаем 7, из Fh (15) вычитаем 7, получаем 8).

```
  ....
  10031
-   7B9
  -----
   F878
```

К полученному результату (уже в дополнительном коде) необходимо прибавить месяц.

Для рассмотренного примера получим: день·день-год+месяц=F878h-9h (8+9=17 – 11h, 1 записываем, 1 переносим в следующий разряд («в уме»), 7+1 = 8).

```
  F878
+    9
  ----
  F881
```

Затем необходимо проверить правильность полученного результата. Для этого необходимо узнать значение полученного числа в десятичной системе счисления и сравнить его с подсчитанным на калькуляторе по формуле день·день-год+месяц десятичным значением. Узнать значение можно, получив из дополнительного кода прямой (той же операцией, что и получали дополнительный код – вычитанием из 10000h) и переведя получившееся число в десятичную систему счисления по формуле.

Для рассмотренного примера получим.

$$\begin{array}{r} \dot{1}\dot{0}\dot{0}\dot{0}0 \\ - \quad F881 \\ \hline 77F \end{array}$$

Операцию получения дополнительного (и обратно прямого) кода можно выполнить и в двоичной системе счисления, инвертируя двоичные разряды (заменяя «0» на «1» и наоборот) и прибавив к полученному результату 1. Для рассмотренного примера получим:

F881h = 1111 1000 1000 0001b

0000 0111 0111 1110b

+1b

0000 0111 0111 1111b = 77Fh

Шестнадцатеричное число 77Fh переводим в десятичную систему счисления по формуле $X=X0\cdot p^0+X1\cdot p^1+\ldots+X(N\text{-}1)\cdot p^{(n-1)}$, где p – основание системы счисления (16), n – количество разрядов (цифр) в числе. На практике это обычно делается следующим образом – над каждым разрядом, начиная с правого, расставляется его «вес», начиная с 0; этот вес и представляет собой степень, в

которую необходимо возвести основание системы счисления для получения веса разряда:

Для рассмотренного примера получим:

$$\overset{2}{7}\overset{1}{7}\overset{0}{F}$$

Далее в соответствии с приведённой выше формулой составляется сумма.

$$77Fh = F\cdot 16^0+7\cdot 16^1+7\cdot 16^2= 15 + 7\cdot 16 + 7\cdot 256 =15+112+1792=1919$$

Следовательно, число F881h – это представленное в **дополнительном коде** число -1919.

Теперь проверим правильность вычислений.

$$7\cdot 7 - 1977 + 9 = 49 - 1977 + 9 = -1928 + 9 = -1919.$$

Результаты совпали, соответственно произведенные вычисления верны.

Набор несложных арифметических действий с двоичными и шестнадцатеричными числами, рассмотренный в данном пункте пособия способствует повышению понимания студентами того, как микропроцессор обрабатывает информацию, запрограммированную в него пользователем.

Так как программирование в языке низкого уровня Assembler представляется для студентов достаточно трудным, то следующий этап в его изучении будет заключаться в программировании микропроцессора на выполнение всех тех несложных операций, рассмотренных выше для конкретного примера.

# 4. ПРОГРАММИРОВАНИЕ ЛИНЕЙНЫХ АЛГОРИТМОВ НА БАЗЕ МИКРОПРОЦЕССОРА INTEL 8086

Цель работы: получение первоначальных навыков низкоуровневого программирования, работа с отладчиком Insight.

Линейным принято называть вычислительный процесс, в котором операции выполняются последовательно, в порядке их записи. Каждая операция является самостоятельной, независимой от каких-либо условий. На схеме блоки, отображающие эти операции, располагаются в линейной последовательности.

Если языки высокого уровня отличаются наглядностью программ, то про Assembler так сказать нельзя. Программист сам должен выбрать для себя: в каком виде, где он хочет разместить исходные данные, результаты и промежуточные действия. В некоторых случаях без программиста, написавшего программу, понять, что делает программа, достаточно сложно. В учебном пособии программы на языке Assembler программируется при помощи отладчика Insight. Он является достаточно профессиональным инструментом, наглядно отображающим все действия программы. Изучение отладчика в данном пособии будет происходить по мере усложнения алгоритмов программирования. Основные комбинации клавиш при работе с отладчиком и внешний вид окна при его открытии показаны в приложении 1. На данном этапе для написания простейшей программы понадобиться только основное поле программы.

Выполнение задания состоит из написания простейшей программы на языке ассемблера (последовательность команд),

реализующей расчеты, которые были проведены студентами в процессе выполнения пункта 3 пособия. Она состоит всего из нескольких команд, выполняемых последовательно.

Перед выполнением задания следует понять: как можно размещать информацию в языке Assembler. Для этого существуют регистры общего назначения [3, 6, 9]. Среди них,16-битные регистры: AX (аккумулятор), BX (база), CX (счетчик), DX (регистр данных), которые могут использоваться без ограничений для любых целей – временного хранения данных, аргументов или результатов различных операций. Названия этих регистров происходят от того, что некоторые команды применяют их специальным образом: так, аккумулятор часто используется для хранения результата действий, выполняемых над двумя операндами, регистр данных в этих случаях получает старшую часть результата, если он не умещается в аккумулятор, регистр-счетчик используется как счетчик в циклах и строковых операциях, а регистр-база используется при так называемой адресации по базе. Отдельные байты в 16-битных регистрах AX – DX тоже имеют свои имена и могут использоваться как 8-битные регистры. Старшие байты этих регистров называются AH, BH, CH, DH, а младшие – AL, BL, CL, DL (рис. 4.1).

BX
BH
CX
CH
DX
DH
16 15 8 7
SI
DI
BP
SP

Рис. 4.1. Регистры общего назначения

Другие четыре регистра общего назначения – SI (индекс источника), DI (индекс приемника), BP (указатель базы), SP (указатель стека) – имеют более конкретное назначение и могут применяться для хранения всевозможных временных переменных, только когда они не используются по назначению. Регистры SI и DI используются в строковых операциях, BP и SP используются при работе со стеком (будут рассмотрены далее).

Для того, чтобы запрограммировать микропроцессор Intel 8086 на выполнение вычисления выражения день·день-год+месяц потребуются следующие команды.

– MOV op1,op2 – присваивание (копирование). op1 присваивается значение op2, значение последнего при этом не

изменяется. op1 может представлять собой регистр или ячейку памяти. В настоящей работе используются только регистры общего назначения AX, BX, CX, DX. op2 может быть также регистром, ячейкой памяти (при условии, что первый операнд не является ячейкой памяти) или непосредственно указанным числом. Разрядности операндов (8 или 16 бит, байт или слово соответственно) должны совпадать.

– ADD op1,op2 – сложение (addition). op1 присваивается значение op1+ op2, значение последнего при этом не изменяется. Требования к операндам такие же, как и в команде MOV.

– SUB op1,op2 – вычитание (subtraction). op1 присваивается значение op1- op2, значение последнего при этом не изменяется. Требования к операндам такие же, как и в команде MOV.

– MUL op1 – умножение (multiplication). Если операнд 16-разрядный (как в данной работе) – то *двойному слову*, старшая часть которого расположена в регистре DX, а младшая в AX (принято записывать DX:AX) присваивается значение произведения AX·op1.

*В случае, если op1 восьмиразрядный (AL,AH,BL,BH,CL,CH,DL,DH), то регистру AX присваивается значение произведения AL·op1.*

*Следует запомнить, что в результате 16-разрядного умножения значение регистра DX уничтожается, поэтому не следует располагать в нём данные перед умножением!*

*Команда IMUL op1 идентична команде MUL за исключением того, что множители воспринимаются как числа со знаком (в отличие от команд сложения и умножения это имеет значение*

*при выполнении команды процессором). Для выполнения задания, рассматриваемого в этом пункте пособия, можно использовать любую команду, поскольку множители являются положительными числами.*

Рассмотрим пример пункта 3 пособия день·день-год+месяц=7h·7h-7b9+9h. Перед программированием этой последовательности арифметических операций необходимо выбрать регистры для загрузки исходных данных. Поскольку день необходимо умножать, логично расположить его в регистре AX, остальные данные – в регистрах BX и CX (не в DX, поскольку он будет уничтожен умножением). Итак, первая часть программы – загрузка данных (данные из примера пункта 3 пособия, соответственно каждый студент использует свои данные).

```
Mov ax,7
Mov bx,9
Mov cx,7B9
```

Все числа при вводе программы в отладчике вводятся в **шестнадцатеричной системе счисления**, при этом обозначение данной системы счисления не указывается. Отрицательные числа могут быть введены как непосредственно в дополнительном коде (например, F881 для числа -77F), так и в прямом, указав знак «-»(-77F), в последнем случае программа автоматически преобразует введенное число в дополнительный код.

Затем необходимо вычислить произведение день·день. Это выполняется следующей операцией:

```
Mul ax
```

Максимально возможное число, которое может быть размещено в регистрах общего назначения составляет 32767. Результат данного произведения располагается в регистрах DX:AX, однако очевидно, что результат будет в любом случае меньше 32767, то есть уместится в регистре AX, а значение DX обнулится. Поэтому считаем, что результат расположен в AX.

Затем необходимо вычислить произведение день·день-год. Это выполняется следующей операцией:

sub ax,cx

Полученная разность располагается в регистре AX в дополнительном коде, так как число получится отрицательным.

Затем необходимо вычислить произведение день·день-год+месяц. Это выполняется следующей операцией:

Add ax,bx

Результат вычислений расположен в AX.

Таким образом, общий вид программы для рассмотренного примера имеет вид.

Mov ax,7

Mov bx,9

Mov cx,7B9

Mul ax

Sub ax,cx

Add ax,bx

Используя вместо 7h, 9h и 7B9h соответственно свои данные, студенты могут получить результат для своего индивидуального случая, который будет располагаться в регистре

AX. В отладчике результат будет находиться в окне регистров (рис. П.1.2).

Для набора программы следует перевести отладчик в режим редактирования при помощи команд (**F10, E, A**). Набор каждой команды заканчивается нажатием клавиши **Enter**. По окончанию набора программы, нужно перевести отладчик обратно в режим выполнения (**ESC**), после чего есть возможность выполнить программу покомандно с помощью клавиши **F7**, при этом можно наблюдать за изменениями значений в регистрах (окно справа вверху – окно регистров на рис. П.1.2), сравнивая их с промежуточными результатами. Выявив конечный результат выполнения программы, следует сравнить его с результатом расчётов, выполненных в пункте 3 пособия. Если результаты совпадают, то всё выполнено правильно.

Следует иметь ввиду, что, если при вводе команды допущена ошибка – программу необходимо набрать заново, поскольку длина в байтах машинных кодов команд разная, а машинный код интерпретируется последовательно по адресам памяти.

В этом пункте пособия было показано: как запрограммировать микропроцессор Intel 8086 на выполнение простейшего линейного алгоритма, что не представляет большой трудности. В следующем пункте будут рассмотрены более сложные программы, включающие в себя переходы по какому-либо условию (ветвящиеся алгоритмы).

# 5. ПРОГРАММИРОВАНИЕ ВЕТВЯЩИХСЯ АЛГОРИТМОВ НА БАЗЕ МИКРОПРОЦЕССОРА INTEL 8086

Цель работы – научиться разработке простейших алгоритмов ветвящихся процессов и их реализации на языке Ассемблера.

Ветвящимся алгоритмом называется алгоритм, в котором из блока может выходить два потока данных. Таким блоком является блок условия, внутри которого записывается логическое выражение. Логическое выражение может иметь два значения: истина или ложь.

Исходными данными для студентов служит условная функция в соответствии с номером варианта, приведённым в пункте 8 данного пособия. Рассмотрим на конкретном примере программирование алгоритма с ответвлениями.

$$F = \left\{ \begin{array}{c} A - B,\ \text{если}\ A < B < C \\ (A + B)/C,\ \text{если}\ A < B \geq C \\ A \circ B - C,\ \text{иначе} \end{array} \right\}$$

Выполнение задания состоит из следующих пунктов:

1) Выполняется формализация задачи (для таких простых алгоритмов не является обязательным этапом и может быть опущена). Результат представляется в виде HIPO-диаграммы (рис. 5.1). На рисунке показываются исходные данные и результат, внутри блока могут быть отображены действия, приведшие к результату.

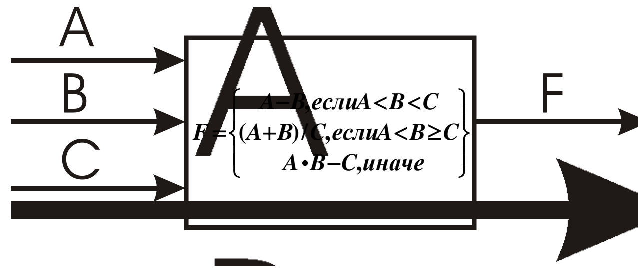



Рис. 5.1

2) <u>Выполняется разработка алгоритма. При разработке алгоритма первым выполняется операция сравнения, которая одинакова в двух строчках из трёх (в нашем примере это $A<B$). Только такое решение позволит использовать минимум сравнений (2), а оптимальность алгоритма (скорость его работы) определяется именно количеством сравнений. В противном случае придется проверять два раза одно из условий, что конечно не приведет к ошибке в работе программы, однако не является оптимальным. Алгоритм представляется в виде блок-схемы, допускается так же текстовое описание.</u>

Напомним значения элементов блок-схем наиболее частых в употреблении (в соответствии с теоремой «О структурном программировании», достаточных *для составления любого алгоритма*), рис. 5.2.

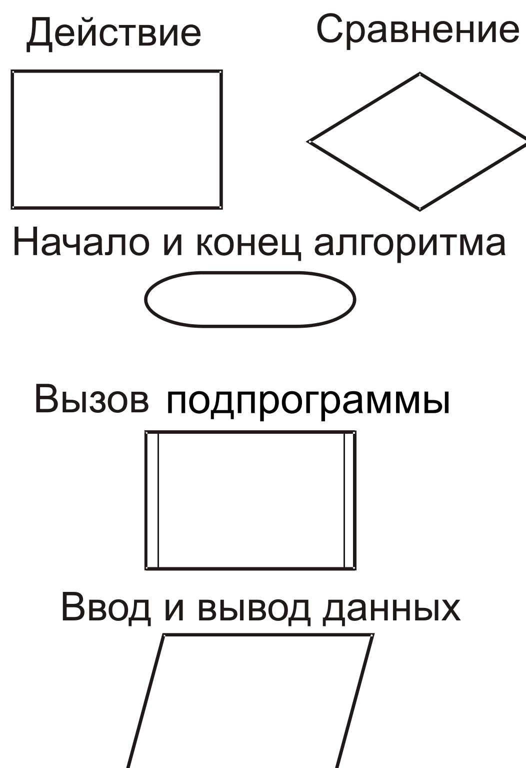



Рис. 5.2

Для рассмотренного примера сначала составим текстовое описание алгоритма, которое фактически повторяет мыслительный процесс, сопровождающий создание алгоритма.

Итак, текстовое описание алгоритма с комментариями (выделены курсивом) приведено ниже.

*Функция принимает значение по одному из трех выражений в зависимости от значений входных данных. Следовательно, кратко алгоритм можно свести к следующему –* ввод данных, сравнение и расчет, вывод данных.

Ввод A,B,C.

*Первое сравнение, как показано выше, должно быть то, которое одинаково в двух выражениях.*

Если A<B, то *необходимо выполнить второе сравнение, чтобы определить, по какому именно выражению – первому или второму – должен быть вычислен результат.*

> Если B<C, то *необходимо вычислить функцию по первому выражению:* F:=A-B,
>
> Иначе (то есть, если B не меньше C) *необходимо вычислить функцию по второму выражению (так как если B не меньше C, то оно больше или равно ему, а то, что A<B выяснилось в процессе первого сравнении)*: F:=(A+B)/C,

Иначе (если A не меньше B) *необходимо вычислить функцию по третьему выражению, так как, если условие «A меньше B» не выполняется, то нет смысла проверять соотношение B и C – в любом случае ни одно из первых двух двойных неравенств не выполнится*: F:=A·B-C.

Вывод результата (F).

Составить по данному описанию блок-схему не составляет труда – ввод и вывод изображаются параллелограммами; слово «если» – ромбом (символом ветвления), действие (расчет значения F) – прямоугольником (рис. 5.3).

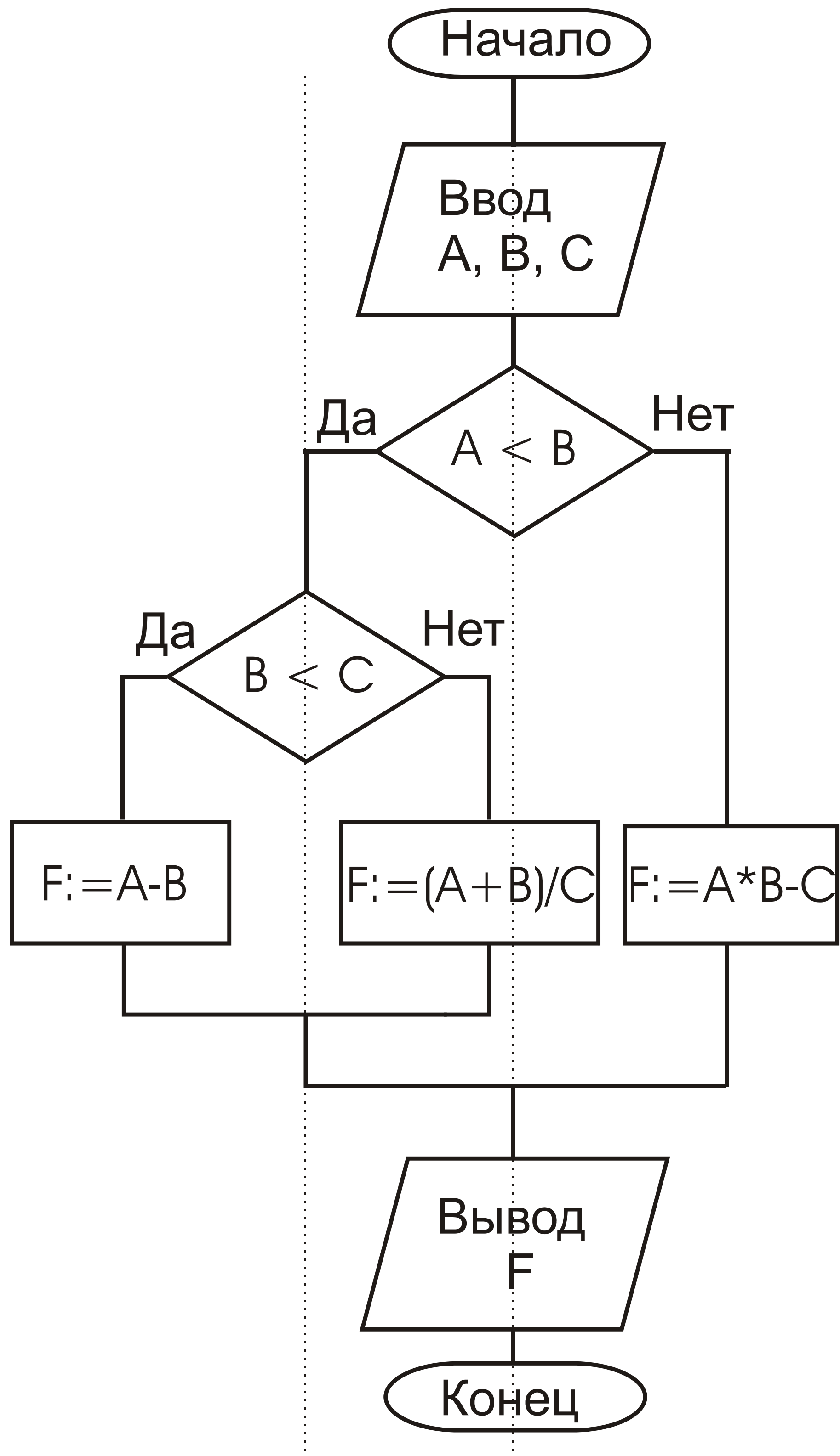



Рис. 5.3

При изображении блок-схем желательно придерживаться следующих принципов.

1. Весь алгоритм размещать на одной странице, если это невозможно, то часть алгоритма отображается одним блоком, который затем детализируется на следующей странице.

2. Элементы ветвлений располагать относительно оси симметрии (которых, соответственно, будет столько, сколько ветвлений в алгоритме). На рис. 5.3 оси симметрии показаны пунктиром.

3. Ветвь, соответствующую выполнению условия, вести налево от символа сравнения. Желательно также указывать словами «Да» и «Нет», или символами «+» и «-», «1» и «0» выполнение/невыполнение условия соответственно. Рядом с блоками допускается делать текстовые пометки (комментарии), отделяя от блок-схемы прерывистой линией и квадратной скобкой (рис. 5.4).

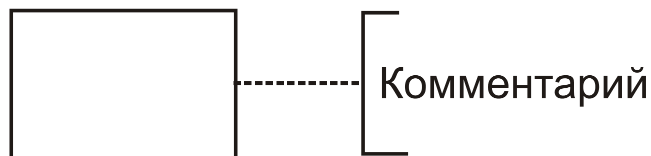



Рис. 5.4

4. В дальнейшем, при составлении более сложных алгоритмов, рекомендуется нумеровать блоки на схеме, указывая в комментариях к программе соответствующий блок.

3) Кодирование алгоритма. В соответствии с блок-схемой составляется программа на языке программирования (в случае данной работы – на Ассемблере 8086). При этом необходимы

команды передачи данных и арифметические команды, изученные при программировании линейных алгоритмов, а также команды:

XCHG op1,op2 – меняет местами содержимое операндов op1 и op2. Требования к операндам аналогичны команде MOV, за исключением того, что в качестве операндов не может быть использовано непосредственное значение.

DIV op1 – деление (division). Если операнд 16-разрядный (как в данной работе) – то *двойное слово*, старшая часть которого расположена в регистре DX, а младшая в AX (принято записывать DX:AX) делится на op1. Регистру **AX** присваивается значение **частного** от деления, регистру **DX – остатка**.

*В случае, если op1 восьмиразрядный (AL,AH,BL,BH,CL,CH,DL,DH), то делится слово (**регистр AX**). Регистру **AL** присваивается значение **частного** от деления, **AH – остатка**.*

*Команда IDIV op1 идентична команде DIV за исключением того, что числа воспринимаются как числа со знаком (в отличие от команд сложения и умножения это имеет значение при выполнении команды процессором).*

CWD – команда расширения слова до двойного слова (**c**onvert **w**ord to **d**ouble word). Расширяет число, содержащееся в регистре AX, до двойного слова, старшая часть которого расположена в регистре DX (записывается как AX->DX:AX). Данная команда необходима в основном для того, чтобы поделить два числа одинаковой разрядности (16 бит) со знаком. Логика работы этой команды такова: знаковым (старшим, крайним левым) разрядом регистра AX заполняются все разряды регистра DX.

Таким образом, если число в AX положительно, то DX:=0; если отрицательное, то DX:=FFFFh (дополнительный код, в отличие от прямого, расширяется слева не нулями, а единицами.)

Эту команду необходимо использовать перед командой IDIV, помня, что она уничтожит содержимое регистра DX.

Например,

```
Cwd
Idiv cx
```

**Почему отсутствие CWD может привести к ошибке?**

Потому что команда деления использует в качестве делимого двойное слово, и, если его содержимое не определено, то деление выполнится ошибочно.

Например, пусть нужно поделить число 10h на число 3. Программа будет выглядеть следующим образом.

```
Mov ax,10h
Mov cx,3
Idiv cx
```

При этом деление может быть выполнено ошибочно, поскольку команда деления выполнит деление DX:AX / CX, соответственно делится будет не 10, а XXXX0010h, где XXXX – содержимое регистра DX. Таким образом, делимое может составлять 00050010, FFFF0010 и т.д., то есть отличаться от того, что нужно. После же выполнения команды CWD содержимое регистра DX станет 0, и делимое примет необходимый вид DX:AX = 0000 0010.

Если числа, которые нужно делить, рассматриваются как числа без знака, то в данном случае вместо CWD необходимо

использовать mov dx,0. То есть, правильными вариантами программы будут:

| | | |
|---|---|---|
| Mov ax,10h | | Mov ax,10h |
| Mov cx,3 | или | Mov cx,3 |
| cwd | | Mov dx,0 |
| Idiv cx | | div cx |

CBW – команда расширение байта до слова (**c**onvert **b**yte to **w**ord). Алгоритм работы и использование аналогичны команде CWD, только вместо преобразования AX->DX:AX выполняется AL->AX. Соответственно данная команда используется перед командами деления с восьмиразрядным операндом (IDIV BL и т.п.).

Команды CWD и CBW не надо использовать, если делимое уже имеет необходимую (в два раза большую) разрядность, чем делитель; например, получено в результате предшествующего умножения (AX·BX)/CX. Фрагмент программы выглядит следующим образом.

```
imul bx  ; результат уже DX:AX
idiv cx
```

В данном случае использование CWD перед делением может привести к ошибке, если старшая часть полученного произведения (DX) отлична от 0 (или FFFF при отрицательном результате).

Вообще, логику команд умножения и деления просто запомнить, используя тот факт, что в качестве первого операнда (делимого или первого сомножителя) **всегда** используется

аккумулятор, как и **всегда** он используется для хранения результатов. При этом, если необходимо двойное слово (32 бита), то старшие 16 бит берутся из регистра DX (как бы из продолжения аккумулятора), а если необходим байт (8 бит), то он берётся из половинки аккумулятора (AL).

В случае, если результата два (команда деления), то частное и остаток всегда имеют размерность делителя и частное располагается в младших разрядах (правее) (см. 5.5). Также в младших разрядах располагается первый сомножитель (см. рис. 5.6).

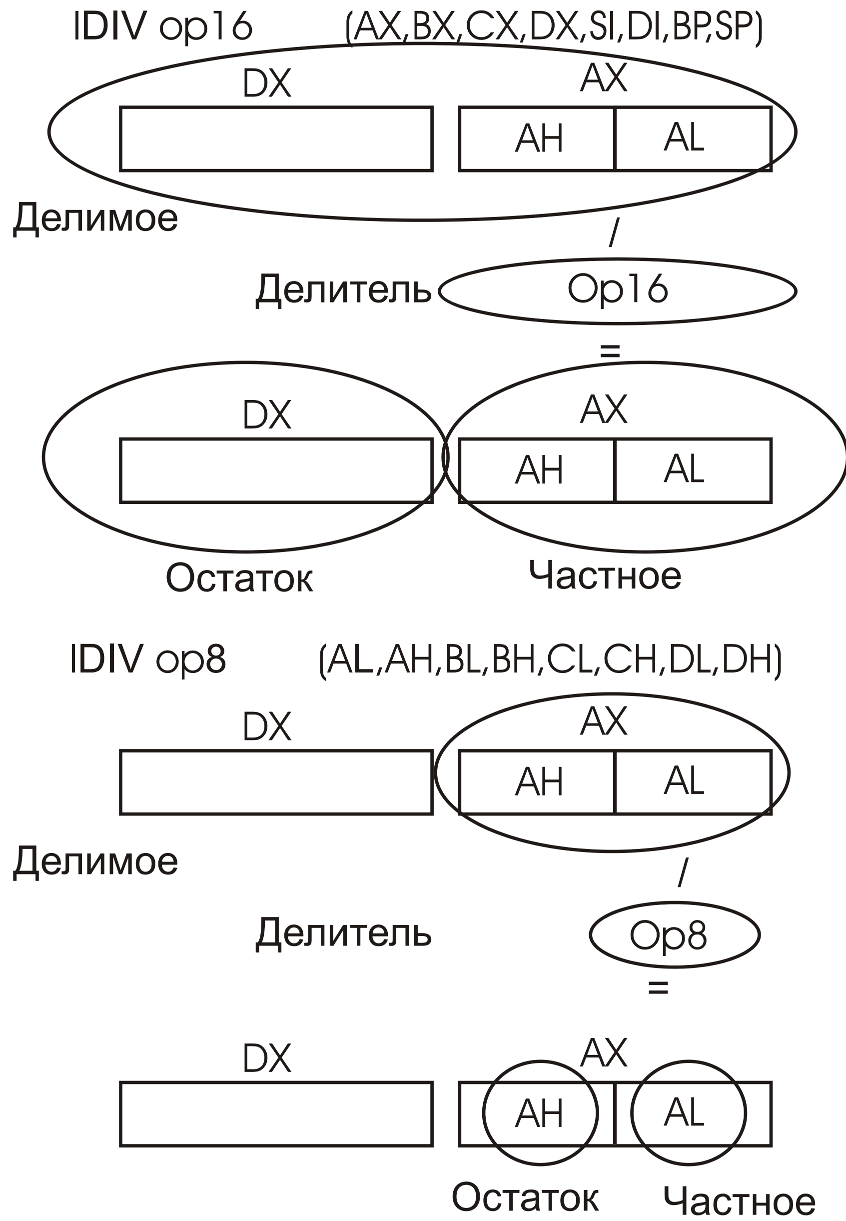



Рис. 5.5

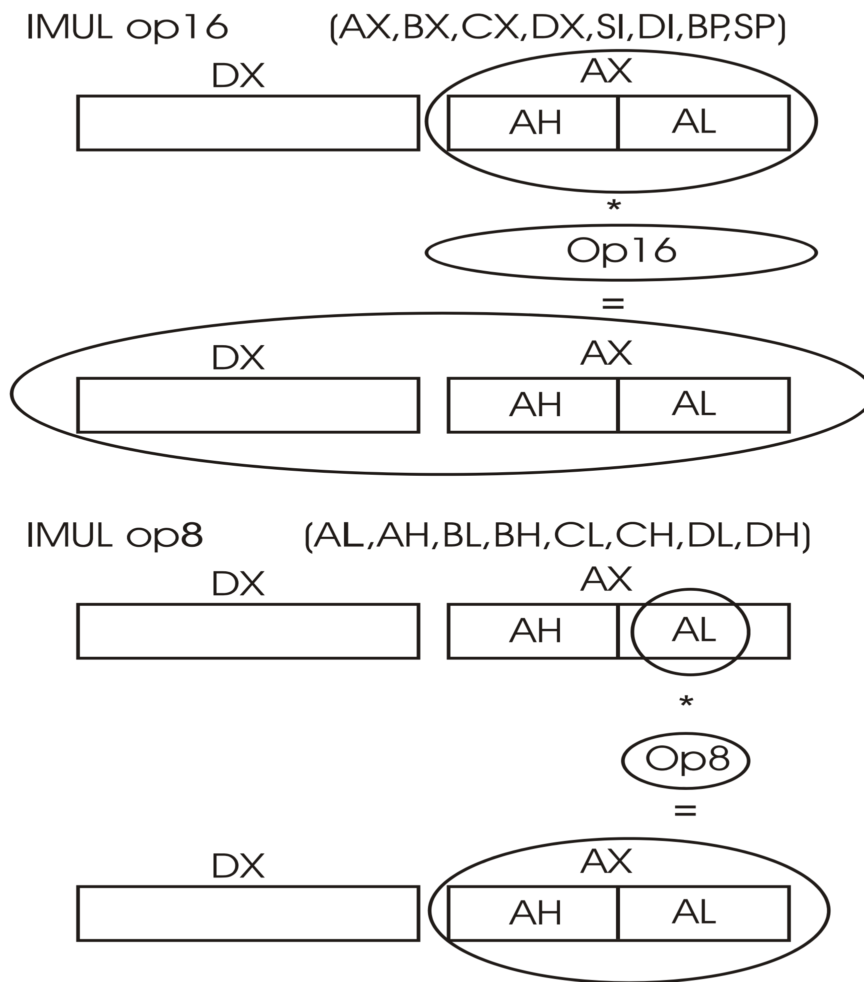



Рис. 5.6

CMP op1,op2 – команда сравнения. Требования к операндам такие же, как и в команде mov. Данная команда осуществляет неразрушающее вычитание, то есть op1-op2, однако в отличие от команды SUB эта команда не изменяет содержимое операндов. Для чего же нужна такая команда? Дело в том, что конструкция сравнения (обозначаемая на блок-схеме геометрической фигурой «ромб») реализуется в языке ассемблера в два этапа. На первом этапе **любая арифметическая или логическая команда** (в том числе и наиболее часто используемая для этих целей CMP)

модифицирует содержание **регистра флагов F** , который затем анализируется командами условного перехода.

Регистр флагов (FLAGS)

| 15 | | | | | | | | | | | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | OF | DF | IF | TF | SF | ZF | | AF | | PF | | CF |

Назначение бит регистра флагов следующие:

OF – переполнение;

DF – направление сканирования;

IF – маска прерываний;

TF – пошаговый режим (трассировка);

SF – знак;

ZF – нуль;

AF – межтетрадный перенос (в байте);

PF – чет/нечет;

CF – перенос.

Команды условного перехода**: J<условие> адрес <метка>.** Условие составляется комбинацией из буквенных мнемоник, следующих в определенном порядке (в том, в каком они перечисляются ниже), обозначающих:

N – отрицание (**n**on), затем;

G > для чисел со знаком (**g**reater), затем;

L < для чисел со знаком (**l**ess), затем;

A > для чисел без знака (**a**bove), затем;

B < для чисел без знака (**b**elow), затем;

E =, затем.

В мнемонике может стоять и одна буква, если это не N. Таким образом, имеются следующие основные команды

(приведенные в одной строке команды эквивалентны, может использоваться и та и другая; в самом деле, если одно число больше другого, то это значит, что оно не меньше и не равно ему).

JA (переход по больше для чисел без знака) и JNBE (переход по не меньше и не равно для чисел без знака);

JNA (переход по не больше для чисел без знака) и JBE (переход по меньше или равно для чисел без знака);

JB (переход по меньше для чисел без знака) и JNAE (переход по не больше и не равно для чисел без знака);

JNB (переход по не меньше для чисел без знака) и JAE (переход по больше или равно для чисел без знака);

JG (переход по больше для чисел со знаком) и JNLE (переход по не меньше и не равно для чисел со знаком);

JNG (переход по не больше для чисел со знаком) и JLE (переход по меньше или равно для чисел со знаком);

JL (переход по меньше для чисел со знаком) и JNGE (переход по не больше и не равно для чисел со знаком);

JNL (переход по не меньше для чисел со знаком) и JGE (переход по больше или равно для чисел со знаком);

JE (переход по равно);

JNE (переход по неравно).

Кроме этих часто употребляемых мнемоник используются также и другие, например, JZ (переход по равенству результата 0, то же что и JE), JNZ (не равно 0), JC (флаг переноса установлен, эквивалентно JB), JNC и некоторые другие.

В качестве метки (адреса) используется:

1. Название метки (идентификатор).

Je m1

…

M1:

…

2. либо используется непосредственно адрес команды, на которую необходимо произвести передачу управления (адрес команды – 16-ти разрядное число /4 шестнадцатеричных цифры/ обозначающее место команды в сегменте памяти; в отладчике адрес команды – самая левая цифра в каждой строке окна кода).

0110: je 012A ; команда условного перехода

……………..

012A: /// ; место, куда будет передано управление.

Логика команды условного перехода заключается в том, что в случае, если результат сравнения удовлетворяет условию – произойдет передача управления по требуемому адресу (на требуемую метку). Иначе выполнение программы продолжится без передачи управления, то есть выполниться следующая написанная в программе (и соответственно, следующая, расположенная в памяти) команда. В принципе, необязательно использовать команду условного перехода сразу за командой сравнения, однако необходимо, чтобы между этими командами не было команд, изменяющих регистр флагов, т.е. арифметических и логических.

JMP <адрес; метка> – команда безусловного перехода. Производит передачу управления в требуемое место программы; это необходимо чаще всего для того, чтобы обойти другую ветвь ветвления (см. дальше в примере).

Пример: требуется закодировать алгоритм на языке Ассемблер, изображенный на рис. 5.3.

Смотрим на блок-схему и кодируем алгоритм. Первым делом определяем места расположения (регистры или ячейки памяти) для всех необходимых данных. Логично выглядит расположение переменной A в регистре AX, B в регистре BX и C в регистре CX, хотя в ряде случаев целесообразно использовать другие регистры, учитывая форматы команд умножения и деления, например, если сомножители – B и C, то можно один из них расположить в AX. Впрочем, содержимое регистров всегда можно поменять в программе командой XCHG.

Итак – первый блок – **ввод данных**:

```
Mov ax,… ; A
Mov bx,… ; B
Mov cx,….; C
```

В программе на бумажном листе пишем «…»; при последующем наборе программы на компьютере подставляем конкретные числовые значения. После знака «;» пишутся комментарии. В программе на ассемблере рекомендуется как можно большее количество комментариев. Рекомендуется писать приблизительно не менее одного комментария на 10 строк кода (в зависимости от алгоритма). Настоящими комментариями обозначаются, каким именно образом располагаются данные в регистрах.

Следующий блок – **сравнение A и B**:

```
Cmp ax,bx
Jl m1
```

Таким образом, если A<B, то произойдет переход на метку m1, иначе выполнится следующая команда (то есть блок F=A·B-C). Следовательно, следующими командами нужно рассчитать функцию A·B-C:

```
Imul bx      ; DX:AX:=ax·bx (A·B)
Sub ax,cx    ; AX:=AX-CX (A·B - C)
```

Замечания:

1. Результат получится в регистре AX. Собственно, там его можно и оставить, поскольку каких-либо дальнейших операций с ним не предусмотрено.

2. Кроме того, при выполнении команды imul результат получится 32-х разрядный и будет занимать 2 регистра – DX:AX. Однако, чтобы не производить 32-разрядных вычислений, можно считать, что наши данные таковы, что произведение A·B будет умещаться в младших шестнадцати разрядах. Вообще, на программы можно накладывать ограничению по составу входных данных, в этом случае они обязательно оговариваются разработчиком в соответствующей документации. Считается «хорошим тоном» также, если программа просто не даст пользователю ввести ошибочные данные и сообщит об ошибке.

3. Можно использовать и команду mul вместо imul, только тогда и в качестве команд условного перехода необходимо использовать jb вместо jl, то есть те, которые предназначены для работы с числами без знака. Также необходимо подобрать (по возможности) такие тестовые данные, чтобы в результате не получалось отрицательных чисел. В последнем случае они будут представлены в дополнительном коде (FFFF = -1, FFFE= -2 и т.д.),

но при беззнаковом представлении числа этот код следует воспринимать как прямой, т.е. FFFF=65535, FFFE=65534. То есть результаты следует считать ошибочными. Кроме того, может возникнуть переполнение при делении с аварийным завершением работы программы.

Далее пишем **блок второго сравнения**, на который происходит передача управлении в случае выполнения первого (метка m1). Однако если написать сразу команду cmp, то возникнет алгоритмическая ошибка – ведь программа в ассемблере выполняется команда за командой, последовательно; а произведя выше расчет по блок-схеме нужно завершить работу программы. Поэтому необходимо поставить команду безусловного перехода на конец программы jmp:

```
  Jmp m3 ; переход на конец
m1:
  Cmp bx,cx
  Jl m2   ; кодируем второе сравнение
```

Замечание: целесообразно делать пометки на блок-схеме соответствующие названиям меток в программе. Это существенно облегчает написание программы и её отладку.

Если условие не выполнилось, то необходимо выполнить блок F=(A+B)/C:

```
add ax,bx    ; ax:=ax+bx (A+B)
cwd          ; ax->dx:ax
idiv cx      ; dx:ax (A+B)/ cx (C)
             ; результат – частное в AX, остаток – в DX
jmp m3       ; обойдем блок F=A-B и перейдем на конец
```

```
m2:
  sub ax,bx     ; ax:=ax-bx результат в AX
m3:
```

В конце блока F=A-B не надо ставить команду безусловного перехода, поскольку следующая за ней команда и так последняя.

В принципе, программу можно на этом закончить, поскольку планируется выполнение её в отладчике. Можно в качестве последней поставить команду **ret** (возврат), которой заканчиваются COM-программы на языке ассемблера, однако выполнять её не следует – это может привести к завершению работы программы и соответственно потере набранного текста. Либо поставить команду **jmp 0100** (переход на первую команду). В этом случае программа будет представлять собой бесконечный цикл, однако, поскольку выполнение планируется в отладчике, данный подход в данном случае удобен – после выполнения программы на одних тестовых данных можно выполнить её на других, не редактируя содержимое регистров.

4) <u>Загрузка и выполнение программы в отладчике</u>

Запустив отладчик “Insight” нужно перевести его в режим редактирования (F10+E+A;  в Windows 98 и XP работает также комбинация Shift-A), после этого можно начинать набор программы. Ввод каждой команды заканчивется клавишей Enter. Вместо меток должны быть проставлены адреса, но поскольку при наборе они ещё неизвестны, то в начале вместо каждого адреса ставится адрес 0100 (адрес первой команды). В качестве значений исходных данных подставляется один из наборов, полученных при разработке тестовых данных (см. пункт 5). Итак, программа

должна выглядеть следующим образом (адреса каждой команды набирать не надо, они формируются автоматически):

0100 mov ax,1

0103 mov bx,3

0106 mov cx,5

0109 cmp ax,bx

010B jl 0100

010D imul bx

010F sub ax,cx

0111 jmp 0100

0113 cmp bx,cx

0115 jl 0100

0117 add ax,bx

0119 cwd

011A idiv cx

011C jmp 0100

011E sub ax,bx

0120 ret

При этом между мнемоникой jmp и адресом перехода автоматически появится указатель short, что означает «короткий». На алгоритм работы он не влияет.

Затем необходимо расставить адреса, сверяясь с написанной на бумаге программой – итак первый переход jl 0100 заменяется на jl 0113 (адрес команды второго сравнения). Чтобы заменить команду, необходимо подвинуть к ней курсор, ввести новое значение адреса и нажать на Enter. При этом следует обратить внимание, что **заменить одну команду** без последствий для

остального кода можно только той, **машинный код который имеет такую же длину в байтах**, иначе все нижеследующие команды изменятся, поскольку процессор дешифрует команды в памяти последовательно!

У всех команд безусловного перехода **jmp** ставим адрес конца программы – 0120. У команды перехода после второго сравнения ставим адрес 011E (вычисления блока F=A-B):

0100 mov ax,1

0103 mov bx,3

0106 mov cx,5

0109 cmp ax,bx

010B jl **0113**

010D imul bx

010F sub ax,cx

0111 jmp ***0120***

**0113** cmp bx,cx

0115 jl **011E**

0117 add ax,bx

0119 cwd

011A idiv cx

011C jmp ***0120***

**011E** sub ax,bx

***0120*** ret

После этого программа готова к работе.

Возвращаем отладчик в режим исполнения (**ESC**), при этом на первую команду указывает указатель выполняемой команды (символ из двух треугольников сразу после адреса команды и

перед её машинным кодом) и начинаем выполнять программу покомандно (в режиме отладки) с помощью клавиши **F7.** Конечно, существует возможность выполнить программу целиком (F9), но поскольку в ней не предусмотрено операций ввода/вывода на периферийные устройства (на языке ассемблера это достаточно большой код), то невозможно будет увидеть результат. После каждого нажатия на F7 контролируем в окне справа состояние регистров и правильность выполнения команды условных переходов. Если в процессе выполнения команды содержимое регистра изменилось, то его значение выделяется светло-голубым.

Останавливаемся тогда, когда дойдем до команды ret и проверяем в окне регистров результат. После запуска (прогона) программы на первом наборе тестовых данных необходимо запустить программу на остальных наборах.

Для этого:

1) Переводим отладчик в режим редактирования команд.

2) Изменяем первые три команды mov, записывая новые данные.

3) Переводим отладчик в режим исполнения (ESC).

4) Переводим отладчик в режим редактирования регистров (**F10+E+R** или **shift-R**).

5) Устанавливаем значение регистра IP равным 0100. После этого указатель выполняемой команды вновь станет указывать на первую команду.

6) Переводим отладчик в режим исполнения (ESC).

Вновь выполняем программу покомандно (F7), учитывая изменения исходных данных и проверяя таким образом программу для следующего тестового набора данных.

(если в качестве последней команды программы написана не **ret** а **jmp 0100,** то выполнять действия 4-6 не нужно.

Если все три результата тестовых данных совпадают – задание выполнено верно.

Примечание: в ряде случаев после первого выполнения программы обнаруживается небольшая ошибка отладчика, проявляющаяся в том, что первая команда записывается неправильно:

00FF add […..],….

0103 mov bx,3

Это происходит от того, что отладчик дизассемблирует программу, начиная с нулевого адреса. Если появляется такая ошибка, необходимо заменить первую команду (войдя в режим редактирования команд) на **nop** – (пустая команда, «нет операции»), нажать Enter и исходный вид программы восстановится:

00FF nop

0100 mov ax,1

0103 mov bx,3

Если ошибочная команда начинается с адреса 00FE, то необходимо вставить 2 команды nop, чтобы выровнить адреса:

| | |
|---|---|
| 00FE add […],… | 00FE nop |
| 0103 mov bx,3 | 00FF nop |
| | 0100 mov ax,1 |
| | 0103 mov bx,3 |

Не следует начинать набирать программу заново с адреса 00FE, это приведет к необходимости изменения всего программного кода.

5) После написания программы её требуется протестировать и отладить. **Тестирование** представляет собой запуск программы на различных наборах данных с заранее известным результатом, с целью определения правильности её работы. **Тестовые данные** должны быть подобраны таким образом, чтобы «пройтись» по всем ветвям алгоритма. В случае наличия ошибки (а иногда и просто для уверенности) выполняется **отладка** (debug) , которая представляет собой пошаговое выполнение программы с контролем значений различных переменных после выполнения каждого оператора с целью определить место в программе, где возможна (или имеет место быть) ошибка.

Пример: итак, чтобы произвести тестирование написанной программы необходимо подобрать комплект тестовых данных, таких, чтобы выполнились все ветви алгоритма. При этом составляется таблица, в которой столбцы представляют собой исходные данные, а строки – различные ветви алгоритма (табл. 5.1).

Таблица 5.1

| A | B | C | Ветвь алгоритма и рассчитанное значение F |
| --- | --- | --- | --- |
| | | | |
| | | | |
| | | | |

Затем, глядя на блок-схему, подбираются требуемые числа. Для рассмотренного примера удобно взять натуральные числа с шагом 2 (табл. 5.2).

Таблица 5.2

| A | B | C | Ветвь алгоритма и рассчитанное значение F |
| --- | --- | --- | --- |
| 1 | 3 | 5 | A<B<C F=A-B= -2 |
| | | | |
| | | | |

Таким образом подбираются тестовые данные для самой левой ветви алгоритма. Затем следующая ветвь – A<B>=C: значения A и B остаются такими же, а C должно быть меньше или равно B (табл. 5.3).

Таблица 5.3

| A | B | C | Ветвь алгоритма и рассчитанное значение F |
|---|---|---|---|
| 1 | 3 | 5 | A<B<C F:=A-B= -2 |
| 1 | 3 | 2 | A<B>=C F:=(A+B)/C=(1+3)/2= 2 (частное 2, остаток 0) |
| | | | |

И наконец, последний комплект данных для случая A>=B (табл. 5.4).

Таблица 5.4

| A | B | C | Ветвь алгоритма и рассчитанное значение F |
|---|---|---|---|
| 1 | 3 | 5 | A<B<C F:=A-B= -2 |
| 1 | 3 | 2 | A<B>=C F:=(A+B)/C=(1+3)/2= 2 (частное 2, остаток 0) |
| 3 | 1 | 2 | Иначе F:=A*B-C=3*1-2=1 |

В этом пункте было рассмотрено, как в Ассемблере программируются ветвящиеся алгоритмы. При этом в качестве данных использовались конкретные числа, никак неструктурированные. Но в языках программирования существуют также и специфические типы данных, например, массивы. Методы программирования таких типов данных рассмотрены в следующем пункте.

# 6. ОБРАБОТКА СТРУКТУР ДАННЫХ (МАССИВОВ), ПРОГРАММИРОВАНИЕ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ, ОПЕРАЦИИ С ПАМЯТЬЮ НА БАЗЕ МИКРОПРОЦЕССОРА INTEL 8086

Цель работы – научиться разработке простейших алгоритмов обработки массивов и их реализации на языке Ассемблера.

В практике программиста значительно чаще приходится иметь дело не с одиночными, а с организованными определенным образом данными. Простейшая и наиболее часто встречающаяся структура – массив (array). Массив – индексированная последовательность однотипных данных. Соответственно, сколько типов данных в языке программирования – столько и типов массивов. В ассемблере таких типов четыре:

– восьмиразрядные (однобайтные) (загружаются в «половинки» регистров – AL, AH, BL, BH, CL, CH, DL, DH):

· byte (без знака), 00h,..,FFh (0,..,255);

·short integer (со знаком) 80h,..,FFh,00h,01h,..,7Fh (-128,…,-1,-,1,…,127);

– шестнадцатиразрядные (загружаются в регистры AX, BX, CX, DX, SI, DI, SP, BP):

·word (без знака), 0000h,..,FFFFh (0,..,65535);

·integer (со знаком), 8000h,…,FFFFh, 0000h, 0001h,…,7FFFh (-32768,…,-1, 0, 1,…,32767).

Эти типы данных соответствуют аналогичным типам в языках высокого уровня, например, в Турбо-Паскале 7.0.

Массив представляет собой последовательность элементов, каждый из которых имеет порядковый индекс. Аналог массива в математике – вектор.

Кроме того, массивы могут быть 2-х мерными (матрица), трех- и т.д. мерными, при этом в них имеются соответственно 2, 3 и более индексов; в программировании наиболее часто используются одномерные массивы – для представления списка определенных данных (значений). В низкоуровневом программировании (машинно-ориентированном, на языке ассемблера или в машинном коде) наиболее часто принято начинать отсчет адресов с «0», это связанно с особенностями аппаратно-программной реализации.

Массивы могут обозначаться в памяти любыми доступными для наименования именами переменных (идентификаторами). В программе на ассемблере используются идентификаторы, состоящие из последовательности латинских букв, цифр и символа «_» длиной не более 127 символов, начинающиеся с буквы. В небольших программах для простоты обычно используются обозначения латинскими буквами (A, B и т.д.). Массив имеет свою размерность (то есть количество элементов в нем). Например, A[5] – массив из пяти элементов. Размерность массива определяется программистом при его описании.

Массивы в памяти ЭВМ располагаются линейно в порядке увеличения адресов, т.е. элементу с индексом. Собственно сама оперативная память представляет собой большой массив, роль индекса в котором выполняет адрес.

Процессор I8086 физически адресует память 20-ю разрядами (1Мбайт), то есть **физический адрес** памяти образуется из 2-х шестнадцатиразрядных значений. 32-х разрядный адрес, записываемый в двух регистрах, называется **логическим адресом**. Старшие 16 разрядов называются сегментным адресом, младшие – смещением (эффективным адресом, offset). Физический адрес образуется из логического следующим образом – физический адрес (ФА) = сегментный адрес·16+смещение. Умножение на 16 соответствует «дописыванию» нуля справа. Одному физическому адресу может соответствовать множество логических адресов.

Логический адрес 0123h:4567h соответствует физическому 01230h+4567h = 05797h.

Такому же физическому адресу 05790h+7h=0597h соответствует, например, логический адрес 0579h:0007h.

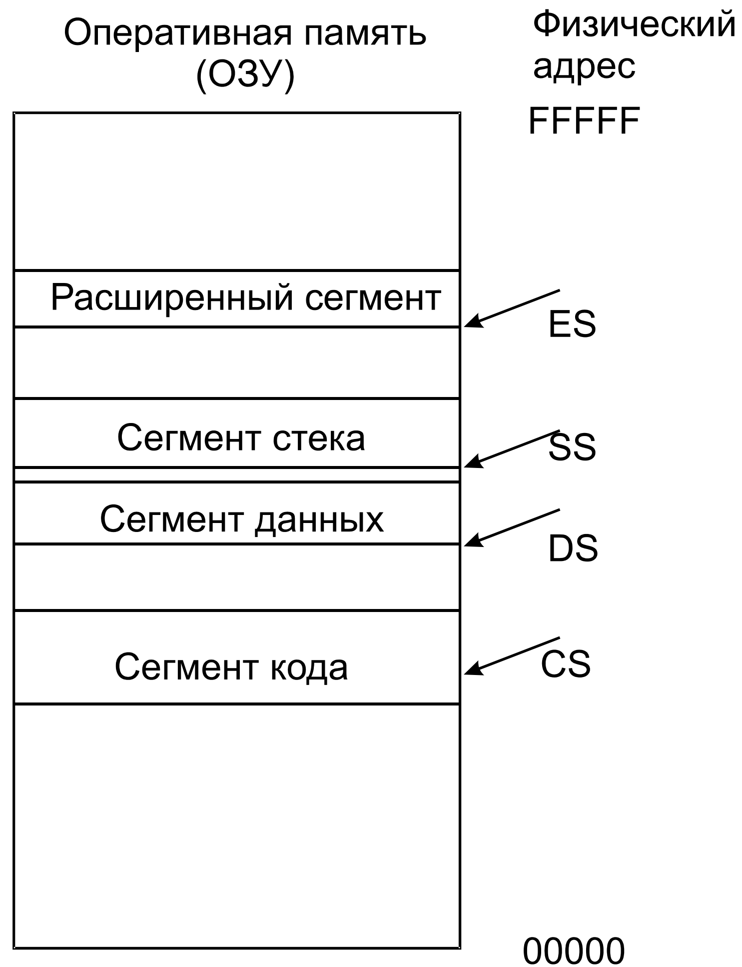



Рис. 6.1

Для хранения сегментных адресов используются сегментные регистры процессора (рис. 6.1):

CS (Code segment) – указатель сегмента кода;

DS (Data segment) – указатель сегмента данных;

SS (Stack segment) – указатель сегмента стека;

ES (Extended segment) – указатель расширенного сегмента (сегмента дополнительных данных).

Указатели сегментов хранят как раз такие указатели на начало сегментов памяти. Чтобы обратиться к байту или слову

внутри сегмента, нужно добавить к адресу началу сегмента 16-ти разрядное смещение. В процессоре этот механизм выполняется с помощью 20-разрядного сумматора, первое слагаемое (сегментный адрес) может поступать от одного из сегментных регистров, а смещение формируется в процессе дешифрации команды и может, например, считываться из некоторых 16-разрядных РОН, или же из слова, находящегося в памяти, и некоторыми другими способами. Логический адрес сегмент + смещение принято записывать через двоеточие: XXXXh:XXXXh.

Следует заметить, что сегменты могут «наслаиваться» друг на друга, что следует из принципа получения физического адреса. Действительно, один байт физической памяти может быть адресован несколькими способами, а именно, столькими, сколькими можно составить данный адрес как сумму двух чисел – сегментного адреса и смещения. Например, физический адрес 020AF можно логически адресовать как 0000:20AF, так и 0200:00AF, так и 010A:100F и т.п. Для программ типа COM (односегментная модель), выделяется только один сегмент памяти, который является одновременно и сегментом кода, и сегментом данных и сегментом стека.

В программах в рамках учебного пособия следует использовать только односегментную модель (все сегментные регистры указывают на один сегмент), что позволяет не задумываться о сегментной составляющей адреса и о том, какой сегментный регистр используется для адресации в данный момент.

Данные в языке ассемблера описываются с помощью директив db (определить байт) и dw (определить слово).

Существуют также директивы для описание групп данных dd (4 байта), dq (8 байт), dt (10 байт), но они используются реже. Данные описываются при написании программы в текстовом файле (в отладчике работа происходит не с переменными, а с фактическими адресами в памяти; см. далее).

Описание строится следующим образом:

Имя_переменной db значение (список значений)

или

Имя_переменной dw значение (список значений)

Например,

X db 5

Y dw 4

При описании массива перечисляется список значений через запятую.

A db 1,2,3,4,7

B dw 9,-4,3,2

Если значения переменных в начале не известны (они рассчитываются в процессе работы программы), то они заменяются на символ «?».

Например,

X db ?

A db ?,?,?,?,? ; 5 неизвестных элементов размером 1 байт

После символа «;» содержимое строки в ассемблерной программе не анализируется, это комментарий.

Кроме того, можно зарезервировать сразу несколько байт памяти с помощью директивы **dup**.

A db 17 dup (?) ; массив из 17-ти элементов типа байт, неизвестных

B dw 100 dup (0) ; массив из 100 элементов типа word, равных нулю.

Если элемент массива имеет размер в 1 байт, то и адрес каждого последующего элемента отличается от адреса предыдущего на единицу, если же элемент массива имеет размер в 2 байта, то адрес последующего элемента отличается от предыдущего на 2.

Массив типа word и integer .

| A0 | A1 | A2 | A3 | A4 |
|---|---|---|---|---|

*Адрес памяти:*
*0200 0202 0204 0206 0208*

Массив типа byte и short integer .

| A0 | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 |
|---|---|---|---|---|---|---|---|---|---|

*Адрес памяти:*
*0200 0201 0202 0203 0204 0205 0206 0207 0208 0209*

То есть **индекс элемента в массиве, как его порядковый номер, не соответствует адресу элемента в памяти.**

**Младшая часть двухбайтного числа хранится по младшему адресу (то есть число «перевернуто» по сравнению с представлением в регистре) – 0003 – 03 00.**

Для того, чтобы обратится к ячейке памяти, нужно указать её адрес. Указанием на обращение к памяти в командах ассемблера служат квадратные скобки. Способ задания операндов в команде называется **типом адресации.** Он определяет возможные способы задания адреса памяти, с которым будет работать данная команда. Есть следующие типы адресации:

1) Непосредственный. Операнд задается в команде. Например, mov ax,**5**.

2) Регистровый. Операнд задается в регистре. Например, mov ax,**bx**.

3) Относительный. Адрес ячейки указывается непосредственно. Например, mov ax,**[0200]** (mov ax,a[2], mov ax,a). Загрузка фиксированной ячейки памяти.

4) Базовый. Адрес ячейки берется из регистра (могут быть использованы только **BX,SI,DI**) Например, mov ax,**[si]**. Команда будет работать с той ячейкой памяти, адрес которой записан в регистре.

5) Базовый относительный. Адрес берется из суммы значения регистра (**BX,SI,DI,BP**) с фиксированным адресом. Например, mov ax,**[si+0200]** (mov ax,a[SI]).

Иногда необходимо использовать более сложные виды адресации (при обработке многомерных массивов, структур данных и т.д.)

6) Базовый индексный. Адрес берется из суммы одного из базовых регистров (**BX,BP**) с индексным (**SI,DI**). Например, mov ax,**[bx+si]**, mov ax,**[bx+di]**, но не ~~mov ax,[bx+bp]~~.

7) Базовый индексный относительный. Адрес берется из суммы одного из базовых регистров (**BX,BP**) с индексным (**SI,DI**) и с фиксированным адресом. Например, mov ax,**[bx+si+0200]**, mov ax,**[bx+di+0200]** (mov ax,a[bx][si] , mov ax,a[bx+di]).

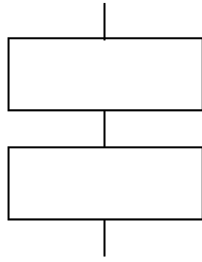
Примечание: при обращении к памяти в качестве сегментной части адреса по умолчанию используется регистр DS, кроме случаев, когда в качестве базового регистра используется BP – в

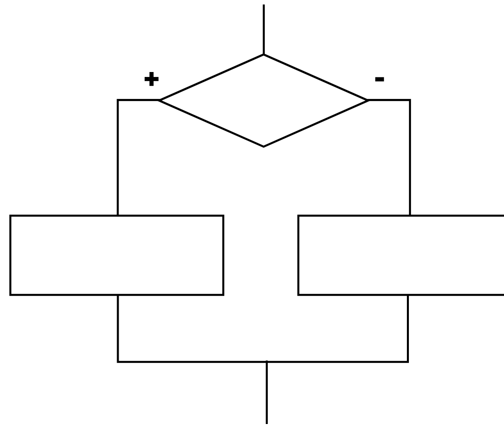
таком случае в качестве сегментного используется регистр SS. Если необходимо изменить значение сегментной части адреса, то можно использовать «префикс сегментного регистра», состоящий из названия сегментного регистра с двоеточием. Например: mov ax,cs:[bx] или mov ax,ss:[si]. Поскольку в учебном пособии рассматривается односегментная модель, то использовать выше приведённые команды не нужно, поскольку все сегментные регистры указывают на один сегмент, однако помнить об этом желательно.

Выполнение задания состоит из следующих пунктов:

1) Разрабатывается алгоритм работы программы. В структурном программировании весь алгоритм представляется с помощью набора базовых управляющих конструкций, с помощью которых можно составить любой возможный алгоритм. Это следование и ветвление, которые были рассмотрены в предыдущих пунктах, а также два вида циклов. **Циклом** называется повтор последовательности действий в алгоритме (команд, операторов). Повтор может осуществляться, пока (while, цикл с предусловием) условие выполняется, либо до тех пор пока (repeat-until), либо определенное количество раз (for, цикл со счетчиком), см. рис. 6.2. Последний случай является частным случаем цикла с предусловием. Действия, выполняемые внутри цикла, называются **телом цикла**.

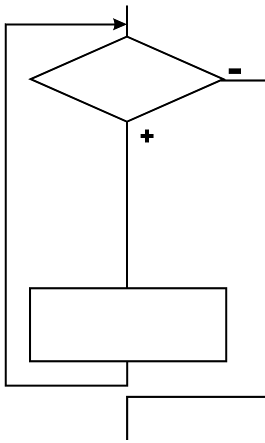
Остальные конструкции (выбор, вызов подпрограммы) являются дополнительными (составными) и могут быть реализованы с помощью описанных выше.

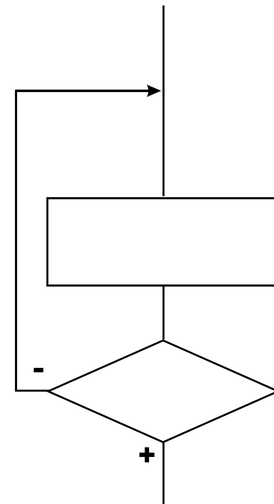




Следование

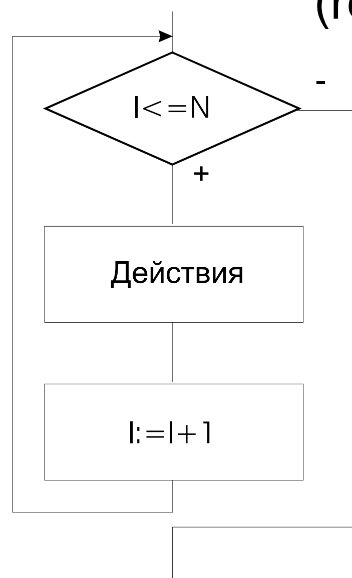
Ветвление





Цикл с предусловием (while-do)

Цикл с постусловием (repeat-until)




Цикл со счетчиком (for)

Рис. 6.2

Большинство программ обработки структур данных (массивов в том числе) имеют в составе алгоритма циклы. Поскольку процессор последовательно выполняет команды, точно также последовательно он и сможет обращаться к отдельным элементам массива. Таким образом, в алгоритме обработки массива данных, так или иначе, присутствует один или несколько циклов «прохода» по массиву, то есть циклов, в которых на каждом последующем проходе нужно обращаться к следующему (или предыдущему) элементу. При этом индекс элемента также является отдельной переменной.

Именно так на самом деле следует действовать при выполнении задач сами на листе бумаги или в уме.

В качестве примера рассмотрим сортировку массива, то есть необходимо расположить его элементы в порядке возрастания или убывания. Как следует действовать, для чтобы расставить, например, числа 2, 0, -3, 1, 7, 4 в порядке возрастания (хотя бы и без компьютера)? Находится минимальный элемент и записывается первым, затем находится следующий элемент, который больше, чем первый, но меньше всех оставшихся и так далее. По сути, здесь уже разработан алгоритм, осталось только записать его в терминах блок-схемы и языка программирования. Есть и другой алгоритм, который будет более эффективен (вообще, эффективность алгоритма в основном определяется по количеству сравнений), так как предложенный алгоритм будет содержать слишком много «лишних» сравнений и кроме того потребует дополнительного места в памяти для ещё одного результирующего массива. Хорошо, когда элементов 5, а если

10 000? Для таких целей разрабатываются другие алгоритмы. Разрабатывая алгоритм, следует представить себе, что нужно решить задачу, обладая возможностями компьютера (взять число, сравнить с другим, сложить, перейти к какому-то шагу). Например, если некуда «выписывать» числа (задействовать дополнительную память), что тогда? Тогда, например, можно менять местами числа в массиве, которые не удовлетворяют условиям сортировки. Сколько раз потребуется это делать? Пока не будет произведено ни одной замены. Такой алгоритм носит название «сортировки методом пузырька» – число как бы постепенно, с каждым новым циклом, «всплывает на поверхность» и постепенно добирается до своего места. То есть, в нашем алгоритме будет два цикла – внешний, который будет повторять менять числа до тех пор, пока не будет сделано ни одной замены, и внутренний, «пробегающий» все элементы массива и при необходимости меняющий их местами.

При создании формализованного алгоритма описываются необходимые переменные. Кроме массива ещё потребуется индекс («счетчик») – переменная, указывающая на то, какой элемент обрабатывается программой на данном шаге («проходе») цикла и переменная, в которой сохраняется информация о том, были ли произведены замены при очередном просмотре массива.

Такой алгоритм выглядит следующим образом для сортировки по возрастанию (рис. 6.3).

Примечание: следует обратить внимание – во внутреннем цикле неравенство строгое (i<n, а не i<=n), так как последний элемент уже не с чем сравнивать.

Для того, чтобы перейти от блок-схемы к программе, необходимо, поставить соответствие между переменными в программе (регистры или ячейки памяти) и на схеме алгоритма. При этом надо учитывать **тип данных** в задании.

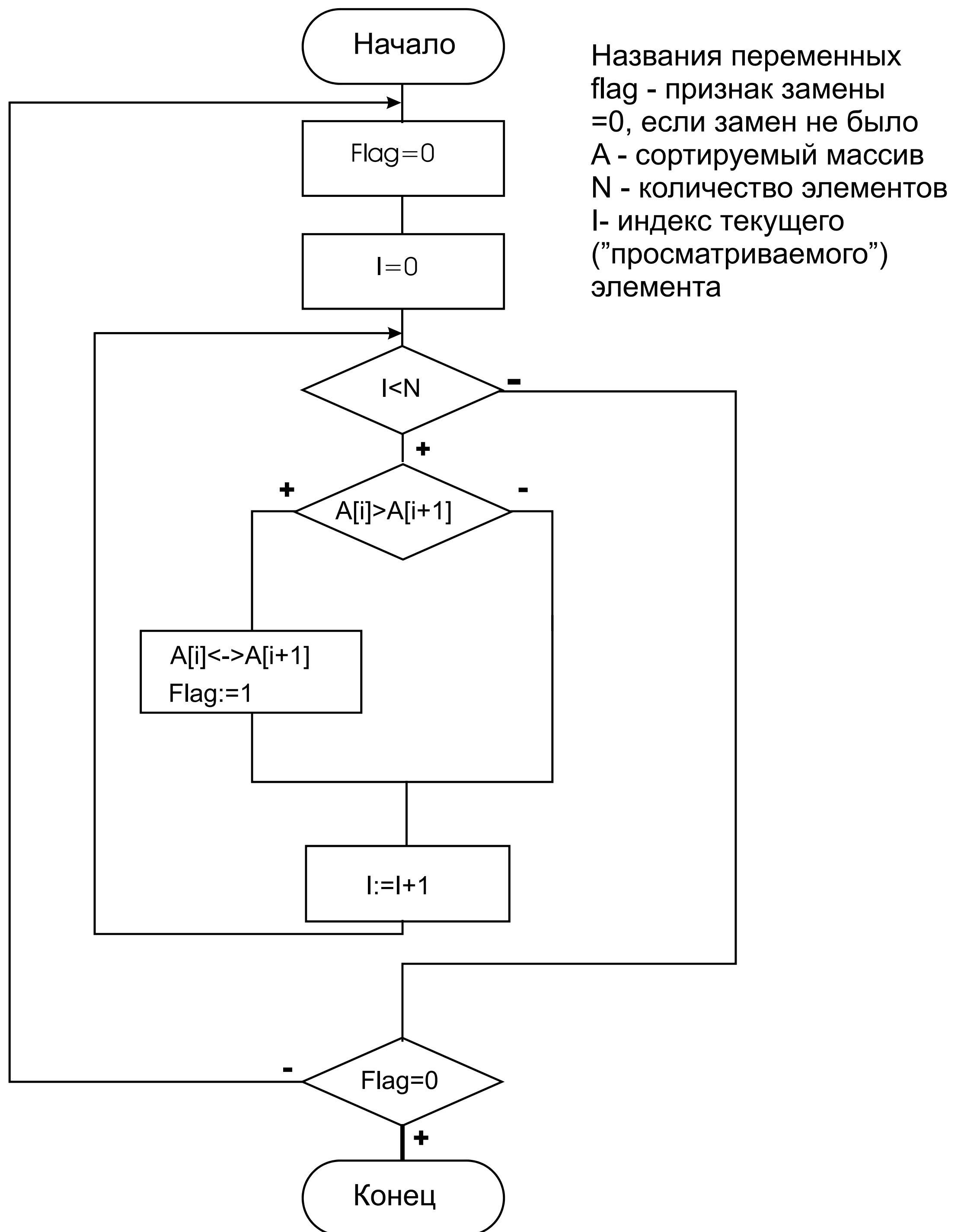



Рис. 6.3

2) Написать программу на языке ассемблера, реализующую разработанный алгоритм, в текстовом файле. Откомпилировать программу в выполняемый модуль (.com )

Чтобы закодировать разработанный алгоритм, следует сопоставить значения переменных в блок-схеме и в программе (то есть «расположить» переменные) и начать записывать с помощью команд каждый блок алгоритма.

**Например:**

**Flag - DX**

**I – SI**

**N=5, константа, задается непосредственно в команде**

**Массив А – память**

Для рассмотренного **примера** программа выглядит следующим образом.

```
Start:  Mov dx,0
        Mov si,0    ; блок 1
m0:     Cmp si,5
        Jnl m1      ; переход по ветке «–» (условие не соблюдается), при этом удобнее осуществлять переход в обход тела цикла, а не внутрь него (для большей наглядности)
        mov al,a[si] ; поскольку мы не можем сравнить командой cmp два операнда, расположенных в памяти, необходимо один загрузить в регистр
        cmp al,a[si+1]
        jng m2      ; если условие возрастания соблюдается, замену производить не надо
```

xchg al,a[si+1]; меняем местами A[i] и A[i+1]

mov a[si],al ; необходимы две команды, так как команды xchg a[si],a[si+1] не существует. Первой командой меняются местами al, куда поместили a[i] с a[i+1], второй запоминаем a[i+1], загруженное в al в a[i]-м

```
        mov dx,1   ; запоминаем, что сделали замену
m2:     inc si     ; команда inc прибавляет к операнду единицу
        jmp m0     ; перейти к следующему элементу массива
m1:     cmp dx,0   ; были ли сделаны замены?
        jne start
ret
A db 1,5,-6,4,3 ; объявление массива
```

Если массив состоит из шестнадцатиразрядных элементов (**dw**), то необходимо загружать его элементы в 16-разрядный регистры (AX), а также не забывать о том, что **адреса элементов в этом случае будут не совпадать с индексами.** В данном случае в написанной выше программе все **al** заменятся на **ax,** а вместо **a[si+1]** для обращения к следующему элементу необходимо использовать **a[si+2]**. Кроме того, в цикле необходимо увеличить индекс на два (добавить ещё одну команду **inc si** после первой, или заменить её на **add si,2**).

Для организации циклов со счетчиком также используется специальная команда. Команды LOOP metka, LOOPE/LOOPZ metka, LOOPNE/LOOPNZ metka.

Логика работы:

LOOP: (CX)<-(CX-1), Переход на metka, если CX!=0.

LOOPE: (CX)<-(CX-1), переход на metka, если CX!=0 и ZF=1

LOOPNE: (CX)<-(CX-1), переход на metka, если CX!=0 и ZF=0.

Пример:

```
    MOV CX,N ; количество повторений цикла
  Metka:
    /// тело цикла
    LOOP metka
```

Пример 2: суммирование элементов массива

```
MOV CX,N ; количество элементов
MOV AX,0 ; сумма
MOV SI,0
Metka:
  ADD AX,massiv[SI]
  ADD SI,2
 LOOP metka
….
Massiv DW 0,3,4,5,6
N DW 5
```

Отладчик неудобен для написания программ, поскольку не имеет возможности сохранять текст программы и эффективно редактировать его (вставлять новые команды и т.п.). Для создания выполняемого модуля используется **компилятор** ассемблера. Последовательность действий при использовании компилятора и компоновщика приведена в приложении 1.

3) Загрузить выполняемый модуль в отладчик и произвести проверку работы программы.

Выполняя программу по одной команде с помощью клавиши **F7,** следует наблюдать за значениями данных в регистрах и в окне памяти, проверяя правильность работы программы.

Если количество повторений цикла велико, то убедившись в правильности его выполнения на первом шаге, можно выполнить все остальные, выбрав следующую после него команду и нажав клавишу F4. Отладчик выполнит все команды до выбранной сразу.

В представленном примере (сортировка массива A db 1,5,-6,4,3 необходимо проверить, что массив в памяти отсортирован по возрастанию, то есть в окне данных по адресу, соответствующему массиву A (узнается из команды загрузки элемента в программе, загруженной в отладчик) вместо байт 01 05 FA 04 03 записаны FA 01 03 04 05).

Для студентов следует выполнить задачи в соответствии с вариантами индивидуальных заданий, приведённых в пункте 8 данного пособия.

Помимо числового представления данных в Ассемблере существует также возможность представления данных в символьной форме, которая может быть определена как некоторый конечный набор изображающих знаков. Такой набор легко представить себе как совокупность ящиков, на каждом из которых изображен соответствующий знак и в котором лежит множество фишек – копий этого знака. Термин "конечный набор" означает здесь конечное число ящиков, а набор фишек в ящике не ограничен. Конструирование зрительного образа в символьной форме осуществляется путем размещения фишек в определенной плоской клеточной структуре, строке, столбце, клеточном поле,

кроссворде, игровом поле и т.п. В каждой клетке такой структуры может быть размещена только одна фишка набора. Разновидностью такого клеточного поля является и экран компьютера, работающего в режиме ввода смимвольных данных.

# 7. ОБРАБОТКА СИМВОЛЬНОЙ ИНФОРМАЦИИ НА БАЗЕ МИКРОПРОЦЕССОРА INTEL 8086

Цель работы – научиться разработке простейших алгоритмов обработки символьной информации и их реализации на языке Ассемблера.

Вся информация представляется в компьютере в виде двоичных чисел, в том числе графика и текст. Для того, чтобы компьютер мог работать с текстом, необходимо присвоить каждому символу некий двоичный код, который и будет означать этот символ в тексте. В компьютерах на базе 86-х процессоров используется код ASCII – (American Standard Code for Information Interchange: Американская стандартная кодировка для обмена информацией). В этой кодировке каждый символ представляется 1 байтом двоичного кода, т.е. всего им можно закодировать 256 различных символов (2^8). Например, код цифры «0» – 30h, «1» – 31h, «9» – 39h. Код латинской буквы «A» – 41h, «B» – 42h, …, «Z» – 5A, «a» – 61h, … , «z» – 7A. Код пробела – 20h.

В первоначальной американской кодировке отсутствовали буквы кириллицы, так как количество всех возможных символов, как уже говорилось, ограничивалось 256, а кириллицу американцы практически не использовали. Поэтому у нас используются несколько кодировок, основанных на «оригинальной», но включающих буквы русского алфавита. Наиболее часто встречается «Альтернативная» кодировка, при которой «А» – 80, …, «Я» – 9F, «а» – A0, …, «п» – AF, «р» – E0, …, «я» – EF. Как видно, между буквами п и р сделан разрыв. Это сделано потому,

что коды BO – BF использовались в оригинальной версии для символов псевдографики. Например, чёрточки, разделяющие области в отладчике «Insight» выполнены именно этими символами. Есть также основная кодировка, одобренная ГОСТ, где русские символы шли подряд, что приводило к тому, что в англоязычных программах вместо разделительных черт шли буквы «я».

Некоторые символы не есть буквы. Например, 13h – символ «Enter» (символ перевода курсора на одну строку вниз), 10h – LF (символ возврата курсора в начало строки). Такие символы называются управляющими и они предназначаются для операционной системы, считывающей и отображающей текст.

Как компьютер отображает символы и как реализуется кодировка? В некоторой области памяти компьютер хранит изображения 8 символов на 16 точек. В последнее время появились кодировки, которые используют для записи символа два байта, что позволяет занести в одну кодировку символы всех алфавитов, даже иероглифы. Например, “Unicode”. Область сегмента данных в отладчике разделена на две части (Приложение 1): в левой (большей) видно двоичные данные, а в правой – их интерпретацию как ASCII-текста.

Альтернативная кодировка используется программами MS-DOS (например, “Insight”). Программы Windows, работающие с ASCII-кодом (например, Wordpad) используют другую кодировку. Буквы кириллицы представляются в ней сплошным набором А – C0 по я – FF.

Ниже приводятся таблицы (табл. 7.1 и 7.2) кодов символов для различных кодировок.

Таблица 7.1. Таблица кодов символов (ASCII, альтернативная (DOS))

| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | | ☺ | ☻ | ♥ | ♦ | ♣ | ♠ | • | ◘ | ○ | ◙ | ♂ | ♀ | ♪ | ♫ | ☼ |
| 10 | ► | ◄ | ↕ | ‼ | ¶ | § | ▬ | ↨ | ↑ | ↓ | → | ← | ∟ | ↔ | ▲ | ▼ |
| 20 | | ! | “ | # | $ | % | & | ‘ | ( | ) | * | + | , | - | . | / |
| 30 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 40 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 50 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 60 | ` | a | b | c | d | e | f | g | h | I | j | k | l | m | n | O |
| 70 | p | q | r | s | t | u | v | w | x | Y | z | { | \| | } | ~ | ⌂ |
| 80 | А | Б | В | Г | Д | Е | Ж | З | И | Й | К | Л | М | Н | О | П |
| 90 | Р | С | Т | У | Ф | Х | Ц | Ч | Ш | Щ | Ъ | Ы | Ь | Э | Ю | Я |
| A0 | а | б | в | г | д | е | ж | з | и | Й | к | л | м | н | о | п |
| B0 | ░ | ▒ | ▓ | │ | ┤ | ╡ | ╢ | ╖ | ╕ | ╣ | ║ | ╗ | ╝ | ╜ | ╛ | ┐ |
| C0 | └ | ┴ | ┬ | ├ | ─ | ┼ | ╞ | ╟ | ╚ | ╔ | ╩ | ╦ | ╠ | ═ | ╬ | ╧ |
| D0 | ╨ | ╤ | ╥ | ╙ | ╘ | ╒ | ╓ | ╫ | ╪ | ┘ | ┌ | █ | ▄ | ▌ | ▐ | ▀ |
| E0 | р | с | т | у | ф | х | ц | ч | ш | Щ | ъ | ы | ь | э | ю | я |
| F0 | Ё | ё | Є | є | Ї | ї | Ў | ў | ° | · | · | √ | № | ¤ | ■ | я |

Таблица 7.2. Таблица кодов символов (ASCII,windows)

| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | | ☺ | ☻ | ♥ | ♦ | ♣ | ♠ | • | ◘ | ○ | ◙ | ♂ | ♀ | ♪ | ♫ | ☼ |
| 10 | ► | ◄ | ↕ | ‼ | ¶ | § | ▬ | ↨ | ↑ | ↓ | → | ← | ∟ | ↔ | ▲ | ▼ |
| 20 | | ! | “ | # | $ | % | & | ‘ | ( | ) | * | + | , | - | . | / |
| 30 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 40 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 50 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 60 | ` | a | b | c | d | e | f | g | h | I | j | k | l | m | n | O |
| 70 | p | q | r | s | t | u | v | w | x | Y | z | { | \| | } | ~ | ⌂ |
| 80 | Ђ | Ѓ | ‚ | ѓ | „ | … | † | ‡ | € | ‰ | Љ | ‹ | Њ | Ќ | Ћ | Џ |
| 90 | ђ | ‘ | ’ | “ | ” | • | – | — | Ч | ™ | љ | › | њ | ќ | ћ | џ |
| A0 | | Ў | ў | J | ¤ | Ґ | ¦ | § | Ё | © | Є | « | ¬ | - | ® | Ї |
| B0 | ° | ± | I | i | ґ | µ | ¶ | · | ё | № | є | » | j | S | s | ї |
| C0 | А | Б | В | Г | Д | Е | Ж | З | И | Й | К | Л | М | Н | О | П |
| D0 | Р | С | Т | У | Ф | Х | Ц | Ч | Ш | Щ | Ъ | Ы | Ь | Э | Ю | Я |
| E0 | а | б | в | г | д | е | ж | з | и | Й | к | л | м | н | о | п |
| F0 | р | с | т | у | ф | х | ц | ч | ш | Щ | ъ | ы | ь | э | ю | я |

Символы с кодами 0 и FF не отображаются. Символ с кодом 20 – пробел. Коды символов даны в шестнадцатеричной системе счисления.

Символьные данные описываются в программе на ассемблере точно также, как и любые другие типа байт, с помощью директивы db. При этом можно описывать значения как код символа, или как сам символ, заключенный в апострофы.

Например,

A db ‘This is String’

B db ‘C’

C db 30h,33h,34h ; эквивалент C db ‘034’

Если длина строки в программе может быть переменной, обычно используется какой-либо код, сигнализирующий об окончании строки (терминальный символ). Обычно это символ с кодом «00». Такая строка называется строкой ASCIIZ (от Zero – ноль).

Например,

A db ‘This is string ASCIIZ’,0

### Преобразование чисел в символьную форму

Почти непременным атрибутом любой программы является вывод на экран не только текстовой, но и числовой информации. Если вывод текстов осуществляется весьма просто – достаточно описать выводимый текст с помощью оператора db и использовать затем подходящую функцию DOS или Windows (кодовая страница), то для вывода числа его необходимо сначала

преобразовать в символьную форму. Действительно, если послать на экран, например, код 5, мы увидим изображение трефового туза. Чтобы получить на экране цифру 5, надо послать код ASCII этого символа, т.е. число 35h. Таким образом, для вывода числа необходимо каждую цифру этого числа заменить кодом ASCII его изображения. Например, для вывода на экран десятичного изображения 13508 следует послать на экран последовательность кодов 31h, 33h, 35h, 30h, 38h. При выводе на экран 16-ричных чисел в последовательность кодов цифр потребуется включать и коды латинских букв.

## Команды работы с цепочками данных

**Цепочкой данных** называют последовательность байт или слов, размещенных в смежных ячейках памяти. Существует 5 команд обработки двух цепочек, обе из которых находятся в памяти. Обычно обработка цепочек данных производится поэлементно в программных циклах.

Все команды для работы со строками считают, что адрес источника всегда равен DS:SI, а адрес получателя равен ES:SI. Это означает, что адрес ячейки-источника или ячейки-получателя следует загрузить в сегментные регистры до выполнения команды. При использовании модели tiny такие действия не требуются. После каждой команды содержимое SI и DI изменяется на один или два байта. Если флажок DF = 1, то уменьшается, если DF = 0, то увеличивается. Состояние флага направления можно изменить безоперандными командами CLD и STD.

CLD ; DF:=0

STD ; DF:=1

### Передать байт/слово

Команды передачи байта/слова из области памяти, адресуемой регистром DI, в область памяти, адресуемую регистром SI, модифицируют указатели SI и DI.

| MOVSB | MOVSB |
|---|---|
| Действие: DS:[DI] := | Действие: ES:[DI] := |
| DS:[SI] | DS:[SI] |
| SI := SI +/– 1 | SI := SI +/– 2 |
| DI := DI +/– 1 | DI := DI +/– 2 |

### Сравнение байта/слова

CMPSB/CMPSW

Действие: ES:[DI] -DS:[SI]

SI := SI +/– 2

DI := DI +/– 2

Изменяется регистр флагов.

### Сканирование байта/слова

SCASB/SCASW

Вычитание байта/слова, адресуемого регистром DI из регистра AL/AX

AL := AL – ES:[DI] (AX := AX – ES:[DI])

DI := DI +/– 1 (DI := DI +/– 2)

### Загрузить байт/слово в al/ax

LODSB/LODSW

AL := ES:[DI]

AX := ES:[DI] (AL:= ES:[DI], AH := ES:[DI+1])

### Запомнить байт/слово

STOSB/STOSW

ES:[DI] := AL

**Пример:** Сравнение строк

```
…
MOV SI,OFFSET STRNG1
MOV DI, OFFSET STRNG2
MOV CX, STRNG1_LEN
CLD
NEXT: (REPZ) CMPSB
JNE NOT_SAME
LOOP NEXT
JMP SAME
NOT_SAME:
…
SAME:
…
```

STRNG1 DB ‘DEFAULTSTRING‘,0

STRNG2 DB ‘….’,0

STRNG1_len DB 14

### Префикс повторения

Без использования префикса команды обрабатывают только один байт или слово. Префикс же позволяет выполнить команды несколько раз, организуя цикл с одной командой. Префикс следует использовать перед цепочечной командой. Он заставляет процессор повторять команду до тех пор, пока не будет выполнено некоторое условие. После каждого выполнения команды регистр CX автоматически уменьшается на единицу.

### Префикс REPE/REPZ

Команда повторяется, пока регистр CX не станет равен нулю или ZF не станет равен нулю.

Пример:

REPE MOVSB

Если CX=0A, то команда, использованная в примере, перешлет десять байт из одной области памяти в другую.

### Префикс REPNE/REPNZ

Команда повторяется, пока регистр CX не станет равен нулю или ZF не станет равен единице.

Пример:

```
MOV DI, OFFSET STRNG2
MOV AL,0
MOV CX, 0200
CLD
REPNZ SCASB
SUB DI, OFFSET STRING1
```

В программе для загрузки адреса строки можно использовать атрибутивные операторы. **Атрибутивные операторы** не являются командами процессора, они лишь производят расчет необходимых цифровых значений, избавляя от необходимости просчитывать их вручную. Атрибутивные операторы приведены ниже.

ОПЕРАТОР LENGTH

Возвращает число единиц, назначенных переменной.

FEES DW 100 DUP (0)

MOV CX, LENGTH FEES эквивалентна MOV CX, 100.

ОПЕРАТОР SIZE

Аналогичен LENGTH, но вместо числа единиц он возвращает число байт.

ДИРЕКТИВА SEG

Возвращает значение сегментного адреса переменной или метки.

ДИРЕКТИВА OFFSET

Возвращает значение смещения переменной или метки. Команда Mov bx, offset OPER_1 заставляет смещение операнда ассемблироваться как непосредственный операнд, а во время

выполнения смещение загружается в регистр BX (команда bx, oper1 загружает в BX содержимое переменной BX).

Оператор OFFSET служит для загрузки эффективного адреса. В отладчике команда MOV SI,offset A будет выглядеть как MOV SI,XXXX (непосредственный операнд). Команда MOV SI,A приведет к загрузке в SI слова из ячейки памяти, а не ее адреса. Поэтому необходимо использовать оператор offset. Вместо этого можно использовать особую команду LEA (загрузка эффективного адреса) LEA SI,A = MOV SI, offset A. Команда LEA SI,A обычно транслируется (соответственно, выглядит при загрузке программы в отладчик) как MOV. Хотя это и отдельная команда процессора.

В отладчике Insight редактировать данные (ячейку памяти) можно в нижнем окне, используя комбинацию F10+E+D, используя адрес из команды загрузки. В табл. 8.1 приведено соответствие команд загрузки адресов в отладчике и текстовом файле.

Таблица 8.1

| Команда (в ассемблере) | Команда (в отладчике) | Действие |
|---|---|---|
| MOV SI,A | MOV SI,[XXXX] | В регистр SI загружается слово из памяти по адресу DS:XXXX (ячейка, описанная в программе как A) |
| MOV SI,offset A | MOV SI,XXXX | В регистр SI загружается адрес ячейки памяти (XXXX), описанной как A |
| LEA SI,A | MOV SI,XXXX<br>LEA SI,[XXXX] | В регистр SI загружается адрес ячейки памяти (XXXX), описанной как A |

Выполнение задания:

1) Разработать алгоритм работы программы в соответствии с индивидуальным заданием. При разработке алгоритма следует учитывать возможности команд микропроцессора. В данном случае может быть целесообразно описывать алгоритм не с помощью блок-схемы, а словами или специальными схемами, иллюстрирующими работу алгоритма на модели строк.

В общем случае алгоритм обработки строки символов также представляет собой алгоритм обработки массива, только элементы его являются кодами символов. А также, учитывая специальные команды для обработки цепочек (для числовых массивов их, кстати, тоже можно применять), можно сделать алгоритм более понятным и лаконичным.

**Пример.** Преобразовать строку ASCII в верхний регистр (т.е. заменить строчные буквы на прописные).

Алгоритм можно описать так:

1. Повторять пока не закончится строка

a. Загрузить символ из строки

b. Преобразовать строчные буквы в прописные (так как коды строчных букв отличаются на 20h, то надо вычесть 20h из кода символа)

c. Записать символ в строку.

| s | t | r | g | 0 |
|---|---|---|---|---|

lodsb
AL:=AL+20h

stosb

| S | T | R | G | 0 |
|---|---|---|---|---|

2) Написать программу на языке ассемблера по разработанному алгоритму, откомпилировать, загрузить в отладчик и проверить результаты работы.

Пишем программу:

1) стандартная часть

```
.model tiny
.code
  org 100h
start:
…
ret
…
end start.
```

3) Вписываем тестовые данные – исходная строка и результат. Если при написание алгоритма не позаботиться о проверке исходных данных (являются ли они строчными латинскими буквами?), то и тестовые данные должны быть такими, чтобы программа сработала корректно.

В ряде случаев, когда использование программы ограничено и, пользователь точно знает о механизме её работы, использовать многочисленные проверки исходных данных, тратя время на разработку этих проверок (как это ещё называется в программировании, сделать «защиту от дурака») не имеет смысла. В этом случае говорят об ограничениях, наложенных на работу программы. Если в коммерческом программном продукте такое допущение весьма ограничено, то в программах «для себя» – наилучший метод оптимально быстро разработать программу.

Программа будет выглядеть следующим образом.

```
.model tiny
.code
  org 100h
start:
…
ret
S1 db ‘string’,0
S2 db 6 dup(?)
end start
```

4) Сначала загружаем исходные данные, затем пишем основной цикл. Программа будет выглядеть следующим образом.

```
.model tiny
.code
  org 100h
start:
 mov si,offset s1 ; загрузить адрес строки-источника
 mov di,offset s2 ; загрузить адрес строки-получателя
 ;цикл обработки строки:
m:  lodsb
   cmp al,0 ;конец строки?
   Je ex
      Sub al,20h ;преобразовать
      Stosb       ;записать
   Jmp m
Ex:
ret
```

```
S1 db 'string',0
S2 db 6 dup(?)
end start
```

5) Компилируем и загружаем в отладчик. Выполняя программу в отладчике, наблюдаем правильность её работы, анализируя информацию в окне данных. Адреса, по которым располагаются строки при загрузке программы в память можно узнать из команд mov si,offset s1 и mov di, offset s2, которые будут отображаться в отладчике с конкретными значениями адресов, например, как mov si,0130 и mov di,0135.

В ряде заданий требуется применить определенную смекалку, позволяющую эффективно решить задачу. Например, задача составления частотного словаря символов, дающая при программировании методом «brute force», то есть грубой силы (без оптимизации, первым попавшимся методом), весьма громоздкую программу, на самом деле может быть очень изящно решена используя тот факт, что каждый символ имеет свой код, который можно рассмотреть как адрес (индекс) в массиве, представляющим собой частотный словарь.

Программа будет выглядеть следующим образом.

```
….
Mov cx,length s
M:  Lodsb ; загрузим в al символ
Mov bl,al
Mov dh,0 ;сформируем из кода символа адрес (в массиве)
Inc byte ptr Chast[bx] ;увеличим счетчик символов с данным кодом
```

Loop M

…

s db ‘This is string ….’

Chast db 256 dup(0)

При программировании довольно часто приходится преобразовывать символьные данные в числовые и обратно. Это не одно и тоже – число «235» будет записано в байте памяти или регистре как “EBh”, в то время как его символьное представление (необходимое в дальнейшем для вывода на экран или ввода с клавиатуры) будет занимать 3 байта ‘235’ – 32h, 33h, 35h.

Для того чтобы преобразовывать символьное представление в число, необходимо вспомнить, что 235 = 5+3·10+2·100 и т.д., то есть преобразование можно осуществить последовательным умножением цифры на множитель в цикле, увеличивая сам множитель. Программа будет выглядеть следующим образом.

…

Mov cx,3 ; количество цифр в числе

std ; читаем строку справа-налево

m:  lodsb

sub al,30h   ; получаем саму цифру из её кода символа, 35h-30h=5 и т.д.

mul mn       ; домножаем на множитель – вес текущего разряда

add bx,ax    ; накапливаем сумму

mov al,mn   ; увеличиваем множитель в 10 раз (переходим к следующему разряду)

mul des

```
mov mn,al
loop m
…
mn db 1
des db 10
```

Приведенный кусок программы соответствует числам типа байт. Для обратного преобразования надо соответственно осуществлять последовательное деление на основании системы счисления:

235/10=23 ост **5**

23/10=2 ост **3**

2/10=0 ост **2**

Как и в случае перевода чисел, результат получается «в обратном порядке». Для этих вычислений можно использовать операции со стеком.

**Стеком** называется область памяти, построенная по принципу LIFO – LAST IN FIRST OUT «Последним зашел – первым вышел». Иллюстрация данного принципа – магазин автомата, детская пирамидка. Стек – самая обычная область памяти, все специфические свойства которой определяются доступными операциями – затолкнуть в стек (положить новое кольцо), вытолкнуть из стека (снять кольцо).

Стек «растет» сверху-вниз (заталкивание соответствует уменьшению адреса указателя, выталкивание – увеличению). Размер ячейки памяти – слово (2 байта).

Стек используется для:

1. сохранения в памяти значений регистров или переменных;
2. сохранения адресов возврата из подпрограмм;
3. передачи параметров подпрограммам.

Содержимое стека демонстрируется в отдельном окне отладчика (справа посередине). При использовании программ с моделью памяти tiny стек начинается с конца сегмента (SP=FFFE).

Команда

```
PUSH r/m16      (SS:SP<-r/m16, SP<-SP-2)
POP r/m16       (r/m16<-SS:SP, SP<-SP+2)
```

Обозначение r/m16 (регистр/память/шестнадцатиразрядный) используется в качестве общего для описания возможного диапазона операндов у команды. В данном случае это могут быть регистры (в том числе сегментные) или ячейки памяти размером в 2 байта. Пример: сохранить значения регистров перед умножением.

```
PUSH AX
PUSH DX
MUL BX
MOV LOW_RESULT,AX
MOV HI_RESULT,DX
POP DX
POP AX
```

Следует обратить внимание, что регистры достаются из стека в обратной последовательности. В принципе, можно обратится к любому записанному в стек значению (с помощью команд MOV (любые типы адресации) ), что в отдельных случаях

(при анализе параметров, переданных подпрограмме) необходимо, так как стек расположен в той же оперативной памяти, что и программа и данные.

**Пример:** участок программы преобразования числа типа байт в строку

```
…
Mov cx,0 ; обнулили счетчик цифр
M: mov ah,0
     div des
xchg ah,al
add al,30h ; преобразовали цифру в символ
push ax    ; запомнили в стеке
inc  cx
xchg ah,al
cmp al,0  ; выполнили деление до конца?
jne m
     pop ax
     stosb
     loop ex
…
```

Следует оформить эти программы в виде подпрограмм для того, чтобы была возможность использовать их в последующих работах для осуществления ввода/вывода числовых данных в программе.

Для студентов следует выполнить задачи в соответствии с вариантами индивидуальных заданий, приведённых в пункте 9 данного пособия.

# 8. ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ

## Практическая работа №1. Часть 1

Тема: Системы счисления.

Задание:

1. Перевести числовые данные своей даты рождения (ДД.ММ.ГГГГ) в шестнадцатеричную и двоичную системы счисления.
2. В шестнадцатеричной системе счисления выполнить расчет по формуле ДД·ДД-ГГГГ+ММ. Отрицательный результат представить в дополнительном коде.
3. Перевести результат в десятичную систему счисления и проверить правильность расчетов, произведя вычисление формулы в десятичной системе счисления.

## Практическая работа №1. Часть 2

Тема: Изучение работы с отладчиком Insight.

Задание:

1. Изучить режимы и основные функции работы программы отладчика Insight.
2. Ввести и выполнить программу, осуществляющую вычисления в соответствии с частью 1 практической работы.
3. Сравнить результаты ручных вычислений и программы.

## Практическая работа №2

Тема: программирование ветвящихся алгоритмов.

Задание:

1. Составить блок-схему алгоритма вычисления условной функции
2. Составить программу на языке ассемблера микропроцессора Intel 8086 по разработанному алгоритму.
3. Ввести программу в отладчик Insight.
4. Подобрать наборы тестовых данных для проверки.
5. Выполнить тестирование и отладку программы.

Варианты заданий.

1.

$$F = \begin{cases} A - B, \text{ если } A < B < C \\ (A + B) / C, \text{ если } A < B \geq C \\ A * C - B, \text{ иначе} \end{cases}$$

2.

$$F = \begin{cases} A + B, \text{ если } A \geq B < C \\ (A - B) * C, \text{ если } A \geq B \geq C \\ A / C - B, \text{ иначе} \end{cases}$$

3.

$$F = \begin{cases} A * B, \text{ если } A > B < C \\ (A * B) / C, \text{ если } A > B \geq C \\ A + C / B, \text{ иначе} \end{cases}$$

4.

$$F = \begin{cases} A/B, \text{ если } A \geq B < C \\ (A-B)*C, \text{ если } A < B < C \\ A*C+B, \text{ иначе} \end{cases}$$

5.

$$F = \begin{cases} A-B, \text{ если } A < B \geq C \\ (A-B)/C, \text{ если } A > B \geq C \\ A*C+B, \text{ иначе} \end{cases}$$

6.

$$F = \begin{cases} A*B, \text{ если } A > B < C \\ (C-B)*A, \text{ если } A > B \geq C \\ A/C+B, \text{ иначе} \end{cases}$$

7.

$$F = \begin{cases} C-B, \text{ если } A < B < C \\ A*B/C, \text{ если } A < B \geq C \\ B*C-B, \text{ иначе} \end{cases}$$

8.

$$F = \begin{cases} C+B, \text{ если } A \leq B < C \\ (C-B)*A, \text{ если } A \leq B \geq C \\ A*C+B, \text{ иначе} \end{cases}$$

9.

$$F = \begin{cases} A/C, \text{ если } A < B < C \\ (A-B)*C, \text{ если } A < B \geq C \\ A/(C*B), \text{ иначе} \end{cases}$$

10.

$$F = \begin{cases} A-B, \text{ если } A < B < C \\ (A+B)/C, \text{ если } A < B \geq C \\ A*C-B, \text{ иначе} \end{cases}$$

11.

$$F = \begin{cases} B - C, \text{ если } A > B > C \\ (A + C) / B, \text{ если } A > B \leq C \\ C * A + B, \text{ иначе} \end{cases}$$

12.

$$F = \begin{cases} B - C, \text{ если } A > B < C \\ A + B * C, \text{ если } A \leq B < C \\ C / (A + B), \text{ иначе} \end{cases}$$

13.

$$F = \begin{cases} C * A, \text{ если } A < C < B \\ A + B + C, \text{ если } A < C \geq B \\ (C - A) / B, \text{ иначе} \end{cases}$$

14.

$$F = \begin{cases} A - B, \text{ если } C < A < B \\ C + A / B, \text{ если } C < A \geq B \\ (C - B) * A, \text{ иначе} \end{cases}$$

15.

$$F = \begin{cases} A - B, \text{ если } C < B < A \\ (C + B) / A, \text{ если } C \geq B < A \\ A + B * C, \text{ иначе} \end{cases}$$

16.

$$F = \begin{cases} A - C, \text{ если } A < B < C \\ (C - B) / A, \text{ если } A < B \geq C \\ C * B - A, \text{ иначе} \end{cases}$$

17.

$$F = \begin{cases} A + B, \text{ если } A \geq B < C \\ (B - C) / A, \text{ если } A \geq B \geq C \\ A * B - C, \text{ иначе} \end{cases}$$

18.

$$F = \begin{cases} A * B, \text{ если } C < B < A \\ B / C, \text{ если } C < B \geq A \\ A + C - B, \text{ иначе} \end{cases}$$

19.

$$F = \begin{cases} A / C, \text{ если } A < B < C \\ (C * B) / A, \text{ если } A < B \geq C \\ B * A + C, \text{ иначе} \end{cases}$$

20.

$$F = \begin{cases} A - C, \text{ если } B < A < C \\ C / (A + B), \text{ если } B < A \geq C \\ A * B * C, \text{ иначе} \end{cases}$$

21.

$$F = \begin{cases} C - B, \text{ если } A \geq B < C \\ A / (C - B), \text{ если } A < B < C \\ A * B - C, \text{ иначе} \end{cases}$$

22.

$$F = \begin{cases} B / A + C, \text{ если } A \geq B < C \\ C * C - A, \text{ если } A < B < C \\ (A - C) / B, \text{ иначе} \end{cases}$$

23.

$$F = \begin{cases} B + A, \text{ если } C \geq B < A \\ A * C - B, \text{ если } C < B < A \\ A / C, \text{ иначе} \end{cases}$$

24.

$$F = \begin{cases} C - B, \text{ если } C < B \geq A \\ A / (B - C), \text{ если } C < B < A \\ A * C + B, \text{ иначе} \end{cases}$$

25.

$$F = \begin{cases} B - C, \text{ если } C \geq A < B \\ A * A * B, \text{ если } C < A < B \\ C / (B - C + A), \text{ иначе} \end{cases}$$

26.

$$F = \begin{cases} (C - A) * B, \text{ если } A \geq B < C \\ C - B, \text{ если } A < B < C \\ A * B / C, \text{ иначе} \end{cases}$$

27.

$$F = \begin{cases} C * B, \text{ если } B \geq A < C \\ A + B, \text{ если } B < A < C \\ (A * B) / - C, \text{ иначе} \end{cases}$$

28.

$$F = \begin{cases} C - A, \text{ если } B < C \geq A \\ C * (A - B), \text{ если } B < C < A \\ C * B - A, \text{ иначе} \end{cases}$$

29.

$$F = \begin{cases} B * B - A * A, \text{ если } A < B < C \\ A / C, \text{ если } A \geq B < C \\ A - C, \text{ иначе} \end{cases}$$

30.

$$F = \begin{cases} C * B, \text{ если } A < B \geq C \\ C / (B - A), \text{ если } A < B < C \\ B * B - C, \text{ иначе} \end{cases}$$

31.

$$F = \begin{cases} A - B * C, \text{ если } A < B \geq C \\ A + B, \text{ если } A < B < C \\ A / C, \text{ иначе} \end{cases}$$

32.

$$F = \begin{cases} C + A, \text{ если } C \geq A < B \\ B/(A + C), \text{ если } C < A < B \\ C * B + A, \text{ иначе} \end{cases}$$

33.

$$F = \begin{cases} C - B, \text{ если } B < A \geq C \\ A/(C - B), \text{ если } B < A < C \\ A * B - C, \text{ иначе} \end{cases}$$

## Практическая работа №3.

Тема: программирование циклических алгоритмов, операции с памятью.

Задание.

1. Составить блок-схему алгоритма обработки массивов данных
2. Составить программу на языке ассемблера микропроцессора Intel 8086 по разработанному алгоритму.
3. Откомпилировать программу с использованием компилятора TASM.
4. Отладить программу, загрузив выполняемый модуль в отладчик.

Варианты заданий.

Вариант 1. Задан массив A[N] из элементов типа word (целое 16-ти разрядное без знака). Составить программу суммирования элементов массива. Если сумма не умещается в двухбайтном

числе, предусмотреть формирования многобайтного результата с указанием его длины в особой ячейке памяти.

Вариант 2. Задан массив A[N] из элементов типа integer (целое 16-ти разрядное со знаком). Составить программу суммирования элементов массива и абсолютных значений элементов массива.

Вариант 3. Задан массив A[N] из элементов типа byte (целое 8-ми разрядное без знака). Составить программу суммирования элементов массива. Если сумма не умещается в однобайтном числе, предусмотреть формирование многобайтного результата с указанием его длины в особой ячейке памяти.

Вариант 4. Задан массив A[N] из элементов типа short integer (целое 8-ми разрядное со знаком). Составить программу суммирования элементов массива. Если результат не умещается в однобайтном числе, предусмотреть формирование многобайтного результата с указанием его длины в особой ячейке памяти. Отрицательный многобайтный результат должен быть в дополнительном коде.

Вариант 5. Задан массив A[N] из элементов типа word (целое 16-ти разрядное без знака). Составить программу нахождения максимального и минимального элемента. Разместить индексы максимального и минимального элемента в отдельных ячейках памяти.

Вариант 6. Задан массив A[N] из элементов типа integer (целое 16-ти разрядное со знаком). Составить программу нахождения максимального и минимального элемента. Разместить

индексы максимального и минимального элемента в отдельных ячейках памяти.

Вариант 7. Задан массив A[N] из элементов типа byte (целое 8-ми разрядное без знака). Составить программу нахождения максимального и минимального элемента. Разместить индексы максимального и минимального элемента в отдельных ячейках памяти. Подсчитать среднее арифметическое максимума и минимума.

Вариант 8. Задан массив A[N] из элементов типа short integer (целое 8-ми разрядное со знаком). Составить программу нахождения максимального и минимального элемента.

Подсчитать среднее арифметическое элементов массива.

Вариант 9. Задан массив A[N] из элементов типа word (целое 16-ти разрядное без знака). Составить программу сортировки массива по возрастанию.

Вариант 10. Задан массив A[N] из элементов типа integer (целое 16-ти разрядное со знаком). Составить программу сортировки массива по возрастанию.

Вариант 11. Задан массив A[N] из элементов типа byte (целое 8-ми разрядное без знака). Составить программу сортировки массива по возрастанию.

Вариант 12. Задан массив A[N] из элементов типа short integer (целое 8-ми разрядное со знаком). Составить программу сортировки массива по возрастанию.

Вариант 13. Задан массив A[N] из элементов типа word (целое 16-ти разрядное без знака). Составить программу сортировки массива по убыванию.

Вариант 14. Задан массив A[N] из элементов типа integer (целое 16-ти разрядное со знаком). Составить программу сортировки массива по убыванию.

Вариант 15. Задан массив A[N] из элементов типа byte (целое 8-ми разрядное без знака). Составить программу сортировки массива по убыванию.

Вариант 16. Задан массив A[N] из элементов типа short integer (целое 8-ми разрядное со знаком). Составить программу сортировки массива по убыванию.

Вариант 17. Задан массив A[N] из элементов типа word (целое 16-ти разрядное без знака). Составить программу поиска элемента X. Если элемент найден, программа должна определить его индекс в массиве и занести его индекс в особую ячейку памяти.

Вариант 18. Задан массив A[N] из элементов типа integer (целое 16-ти разрядное со знаком). Составить программу поиска элемента X. Если элемент найден, программа должна определить его индекс в массиве и занести его индекс в особую ячейку памяти.

Вариант 19. Задан массив A[N] из элементов типа byte (целое 8-ми разрядное без знака). Составить программу поиска элемента X. Если элемент найден, программа должна определить его индекс в массиве и занести его индекс в особую ячейку памяти.

Вариант 20. Задан массив A[N] из элементов типа short integer (целое 8-ми разрядное со знаком). Составить программу поиска элемента X. Если элемент найден, программа должна

определить его индекс в массиве и занести его индекс в особую ячейку памяти.

Вариант 21. Заданы массивы A[N], B[N] из элементов типа word (целое 16-ти разрядное без знака). Составить программу, формирующую массив C[N] из суммы элементов массивов A и B. (C[N]=A[i]+B[i]). Размерность элементов массива C[N] должна обеспечивать корректное суммирование (если результат не умещается в 16-ти разрядах).

Вариант 22. Заданы массивы A[N], B[N] из элементов типа integer (целое 16-ти разрядное со знаком). Составить программу, формирующую массив C[N] из разности элементов массивов A и B. (C[i]=A[i]-B[i]). Размерность элементов массива C[N] должна обеспечивать корректное вычитание (если результат не умещается в 16-ти разрядах).

Вариант 23. Заданы массивы A[N], B[N] из элементов типа byte (целое 8-ми разрядное без знака). Составить программу, формирующую массив C[N] из суммы элементов массивов A и B. (C[i]=A[i]+B[i]). Размерность элементов массива C[N] должна обеспечивать корректное суммирование (если результат не умещается в 8-ми разрядах).

Вариант 24. Заданы массивы A[N], B[N] из элементов типа short integer (целое 8-ми разрядное со знаком). Составить программу, формирующую массив C[N] из произведения элементов массивов A и B. (C[i]=A[i]·B[i]). Размерность элементов массива C[N] должна обеспечивать корректное умножение (если результат не умещается в 8-ми разрядах).

Вариант 25. Заданы массивы A[N], B[N] из элементов типа integer (целое 16-ти разрядное со знаком). Составить программу, формирующую массив C[N] из суммы элементов массивов A и B. (C[i]=A[i]+B[i]). Размерность элементов массива C[N] должна обеспечивать корректное суммирование (если результат не умещается в 16-ми разрядах).

Вариант 26. Заданы массивы A[N], B[N] из элементов типа word (целое 16-ти разрядное со знаком). Составить программу, формирующую массив C[N] из разности элементов массивов A и B. (C[i]=A[i]-B[i]). Программа должна обеспечивать корректное вычитание (если результат меньше 0).

Вариант 27. Заданы массивы A[N], B[N] из элементов типа byte (целое 8-ми разрядное без знака). Составить программу, формирующую массив C[N] из произведения элементов массивов A и B. (C[i]=A[i]·B[i]). Размерность элементов массива C[N] должна обеспечивать корректное умножения (если результат не умещается в 8-ми разрядах).

Вариант 28. Заданы массивы A[N], B[N] из элементов типа short integer (целое 8-ми разрядное со знаком). Составить программу, формирующую массив C[N] из разности элементов массивов A и B. (C[i]=A[i]-B[i])

Вариант 29. Заданы массивы A[N],B[N],C[N] из элементов типа word (целое 16-ти разрядное без знака). Составить программу, формирующую массив F[N] в соответствии с Вашим вариантом практической работы №2.

Вариант 30. Заданы массивы A[N],B[N],C[N] из элементов типа integer (целое 16-ти разрядное со знаком). Составить программу, формирующую массив F[N] в соответствии с Вашим вариантом практической работы №2.

Вариант 31. Задан массив A[2,2]. Найти определитель.

Вариант 32. Задан массив A[3,3] Найти сумму элементов массива.

Вариант 33. Задан массив A[N,N] Найти минимум и максимум, указать их индексы.

## Практическая работа №4

Тема: обработка символьной информации.

Задание.

1. Составить алгоритм решения задачи.
2. Составить программу на языке ассемблера микропроцессора Intel 8086 по разработанному алгоритму.
3. Откомпилировать программу с использованием компилятора TASM
4. Отладить программу, загрузив выполняемый модуль в отладчик.

Варианты заданий.

Для всех вариантов считается, что строка ASCII – строка, состоящая из букв, цифр и пробелов (символов) и заканчивающаяся символом с кодом 00. Слова разделены пробелом.

Вариант 1. Преобразовать строку ASCII в верхний регистр (т.е. заменить строчные буквы на прописные).

Вариант 2. Задана строка ASCII. Преобразовать первые символы слов в верхний регистр. (т.е. заменить строчные буквы на прописные).

Вариант 3. Преобразовать строку ASCII в нижний регистр (т.е. заменить прописные буквы строчными).

Вариант 4. Задана строка ASCII. Преобразовать все символы, кроме первых символов слов, в нижний регистр (т.е. заменить прописные буквы строчными).

Вариант 5. Посчитать количество слов в строке ASCII.

Вариант 6. Посчитать количество символов в строке ASCII.

Вариант 7. Убрать «лишние» пробелы в строке (т.е. если слова разделены несколькими пробелами, то разделить их только одним).

Вариант 8. Задана строка ASCII и символ. Подсчитать, сколько раз данный символ повторяется в строке.

Вариант 9. Задана строка ASCII. Определить наиболее часто повторяющийся символ.

Вариант 10. Целое число со знаком записано в виде строки ASCII. Преобразовать в число типа integer.

Вариант 11. Целое число без знака записано в виде строки ASCII. Преобразовать в число типа word.

Вариант 12. Целое число со знаком записано в виде строки ASCII. Преобразовать в число типа short integer.

Вариант 13. Целое число без знака записано в виде строки ASCII. Преобразовать в  число типа byte.

Вариант 14.  Задана строка  ASCII. Осуществить поиск подстроки (определить номер символа, с которого начинается подстрока в строке).

Вариант 15. Заданы две строки ASCII. Сформировать третью по принципу: первое слово первой строки – первое слово второй строки – второе слово первой строки – второе слово второй строки – третье слово первой строки и т.д.

Вариант 16. Заданы две строки ASCII. Сформировать третью, как результат склеивания первой и второй, причем в первой строке изменить последовательность символов на обратную (справа-налево).

Вариант 17. Заданы две строки ASCII. Сформировать третью по принципу: первое слово первой строки – последнее слово второй строки – второе слово первой строки – предпоследнее слово второй строки – третье слово первой строки и т.д.

Вариант 18. Заданы две строки ASCII. Сформировать третью, как результат склеивания первой и второй, причем во второй строке изменить последовательность символов на обратную (справа-налево).

Вариант 19. Задана строка ASCII и два символа. Преобразовать строку так, чтобы  первый символ был заменен на второй.

Вариант 20. Задана строка ASCII. Преобразовать ее, заменив последовательность символов в словах на обратную (справа-налево).

Вариант 21. Шестнадцатеричное число представлено в виде строки ASCII. Преобразовать данную строку во внутреннее представление.

Вариант 22. Двоичное число представлено в виде строки ASCII. Преобразовать данную строку во внутреннее представление.

Вариант 23. Задана строка ASCII. Преобразовать ее так, чтобы символы в каждом слове располагались по алфавиту.

Вариант 24. Задана строка ASCII. Преобразовать ее, заменив последовательность символов на обратную.

Вариант 25. Составить статистику строки ASCII (предложения) – количество букв, цифр, пробелов.

Вариант 26. Составить статистику строки ASCII (предложения) – количество слов, символов, пробелов.

Вариант 27. Составить статистику строки ASCII (предложения) – средняя длина слова, наиболее часто повторяющийся символ.

Вариант 28. Задана строка ASCII. Составить частотный словарь символов (сколько раз встречается каждый символ в строке).

Вариант 29. Задана строка ASCII. Составить частотный словарь слов (сколько раз встречается каждое слово в строке).

Вариант 30. Задана строка ASCII. Преобразовать ее так, чтобы русские буквы заменить на латинские (составить транслитерацию).

Вариант 31. Задана строка ASCII. Преобразовать ее, заменив русские буквы латинскими в соответствии с расположением на клавиатуре (исправление ошибки неверного ввода).

Вариант 32. Задана строка ASCII. Преобразовать ее, заменив латинские буквы русскими в соответствии с расположением на клавиатуре (исправление ошибки неверного ввода).

Вариант 33. Задана строка ASCII. Преобразовать ее, заменив цифры символами в соответствии с расположением на клавиатуре (исправление ошибки неверного ввода).

## 9. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Позиционные системы счисления.

2. Перевод из одной позиционной системы счисления в другую.

3. Арифметические операции в системах счисления.

4. Представление информации в процессоре.

5. Определение и смысл дополнительного кода.

7. Получение дополнительного кода.

6. Ресурсы программиста микропроцессора 8086. Регистры – определение, размерность.

7. Названия и назначение регистров общего назначения.

8. Организация памяти в МП 8086. Физический и логический адреса. Регистры.

9. Общий вид команды процессора, примеры.

10. Команды сложения и вычитания.

11. Команды умножения и деления.

12. Команды пересылки данных.

13. Регистр флагов. Назначение некоторых из них.

14. Стадии создания программного обеспечения.

15. Команда сравнения и команды условных переходов.

16. Стадии разработки программы на языке ассемблера.

17. Описание данных в языке ассемблера.

18. Виды адресации.

19. Описание и обработка массивов данных.

20. Команды обработки цепочек данных (строковые). Общие принципы.

21. Команды копирования, сравнения цепочек данных. Примеры применения.

22. Команды загрузки/сохранения элемента цепочки данных, команда сканирования. Примеры применения.

23. Команды организации цикла.

24. Обработка символьных данных. Общие принципы.

25. Описание символьных данных в ассемблере. ASCII – код (общий принцип).

26. Преобразование числа из внутреннего представления в символьное и обратно (общий принцип или конкретный пример).

**Практические задачи**

1. Найти дополнительный код числа.
2. Перевести число из одной системы счисления в другую.
3. Написать программу вычисления простейшей арифметической функции (A+B-C и т.п.).
4. Написать блок-схему вычисления условной функции (с одним условием).
5. Написать программу суммирования элементов массива.
6. Написать программу поиска элемента в строке.
7. Написать программу вывода символа на экран.
8. Написать программу вывода строки на экран.
9. Написать программу вывода числа на экран.
10. Написать программу вычисления простейшей логической функции. (A·B/C и т.п.).

# 10. РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Абель П. Язык ассемблера для 1ВМ РС и программирования. — М.: Высшая школа, 1992.

2. Брэдли Д. Профаммирование на языке ассемблера для персональной ЭВМ фирмы 1ВМ. — М.: Радио и связь, 1988.

3. Гольштейн Е. Г., Юдин Д. Б. Задачи линейного программирования транспортного типа. — М.: Наука, 1969.

4. Зубков С.В. Assembler для DOS, Windows и Unix. — 2-е изд., испр. и доп. — М.: ДМК, 2000.

5. Искусство программирования на Ассемблере. Лекции и упражнения.: Голубь Н.Г. – 2-е изд., испр. и доп. СПб.: ООО “ДИАСОФТЮП”, 2002. – 656 с.

6. Калихман И. Л., Войтенко М. А. Динамическое программирование в примерах и задачах. — М.: Высшая школа, 1979.

7. Корбут А. А., Финкелъштейн Ю. Ю. Дискретное программирование. — М.: Наука, 1969.

8. Новиков Ф. А. Дискретная математика для программистов. — СПб.: Питер, 2000.

9. Нортон П., Соухэ Д. Язык ассемблера для 1ВМ РС. — М.: Компьютер, 1993.

10. Пильщиков В.Н. Программирование на языке ассемблера 1ВМ РС.-М.: ДИАЛОГ-МИФИ, 1999.

11. Ровдо А.А. Микропроцессоры от 8086 до Pentium III Xeon и AMD-К6-3. — М.: ДМК, 2000.

12. Рудаков П.И., Финогенов К.Г. Язык ассемблера: уроки программирования. — М.: ДИАЛОГ-МИФИ, 2001.

13. Сван Т. Освоение ТигЬо Assembler. — К.: Диалектика, 1996.

14. Сигал И.Х., Иванова А.П. Введение в прикладное дискретное программирование: модели и вычислительные алгоритмы: Учеб. пособие. – Изд. 2-е, испр. – М.: ФИЗМАТЛИТ, 2003. – 240 с. – ISBN 5-9221-0377-6.

15. Скэнлон Л. Персональные ЭВМ 1ВМ РС и XT. Программирование на языке ассемблера. — М.: Радио и связь, 1989.

16. Схрейвер А. Теория линейного и целочисленного программирования. Т. 2. — М.: Мир, 1991.

17. Юров В.И. Assembler. Практикум. 2-е изд. – СПб.: Питер, 2006. – 399 с.

18. Юров В.И. Assembler. Учебник. – СПб.: Питер, 2001.

19. Юров В.И. Assembler. Специальный справочник. – СПб.: Питер, 2000.

# ПРИЛОЖЕНИЕ 1. РАБОТА С ОТЛАДЧИКОМ INSIGHT, КОМПИЛЯТОРОМ TASM И КОМПОНОВЩИКОМ TLINK

Отладчик Insight запускается путем запуска файла insight.com из каталога Insight. Отладчик функционирует в двух основных режимах – режим выполнения и режим редактирования. В режиме выполнения отладчик готов к выполнению команды (команд) из набранной в режиме редактирования или загруженной программы. Загрузить программу в отладчик можно, указав её в качества параметра командной строки или воспользовавшись пунктом меню Load.

В режиме редактирования можно редактировать команды, содержимое регистров, данные (содержимое памяти) и флаги.

Для перехода в режим редактирования войдите в меню (F10), выберете подпункт Edit (E) (редактирование), выберете объект редактирования (A – assembler – редактирование команд (кода), D – dump – редактирование памяти (данных), R – register – редактирование регистров, F – flags – редактирование флагов).

Для возврата в режим выполнения нажмите ESC.

При редактировании команд, необходимо подтверждать изменения клавишей Enter, иначе изменения не будут запомнены.

Рис. П.1.1

Таблица П.1.1. Основные комбинации клавиш

| Клавиши | Действие отладчика |
|---|---|
| F2 | Установить точку останова на выбранной команде Выбор команды осуществляется путем перемещения **селектора команд** (серой полосы в окне кода) к нужной команде. Выбранная команда и та, которая будет выполняться – могут быть разные. |
| Ctrl-F2 | Загрузить программу заново (работает только для загруженных, а не набранных непосредственно в отладчике, программ) |
| F3 | Загрузить файл (загружаются файлы типа com и exe) |
| F4 | «Выполнить до». Выполняются все команды до выбранной. Если по алгоритму программы выполнения этой команды не произойдет, программа выполнится целиком. |
| Alt-F5 | Посмотреть пользовательский экран. Если в программе предусмотрены операции вывода на экран, то с помощью этой комбинации можно посмотреть, что было выведено на экран. Показать окна отладчика можно, ещё раз повторив нажатие Alt-F5 |
| F5 | Обновить экран. Может быть полезно для контроля за данными, изменяющимися в обработчиках прерываний, которые мы не можем отладить по одной команде. |
| F7 | Выполнить текущую команду. (Выполняется одна команда, на которую указывает **указатель выполняемой команды** независимо от того, есть ли он в окне кода). Адрес команды, на которую указывает указатель, совпадает со значением регистра IP в окне регистров. Для того, чтобы установить указатель на определенную команду, измените значение IP, вписав туда адрес нужной команды. |
| F8 | Выполнить текущую команду. Отличается от F7 тем, что команды вызова подпрограмм (CALL), прерываний (INT), цикла (LOOP), строковую операцию с префиксом повторения (REP) выполняет как одну команду. Обязательно необходимо нажимать F8 при вызове прерывания при вызовах функций операционных систем, иначе вы начнете отлаживать операционную систему. |
| Ctrl-F9 | Запустить. Просто запускает программу с текущей команды до конца программы. Имеет смысл только в интерактивных программах. |
| F10, Alt | Вход в меню |

| Alt-X | Выход из программы |
|---|---|

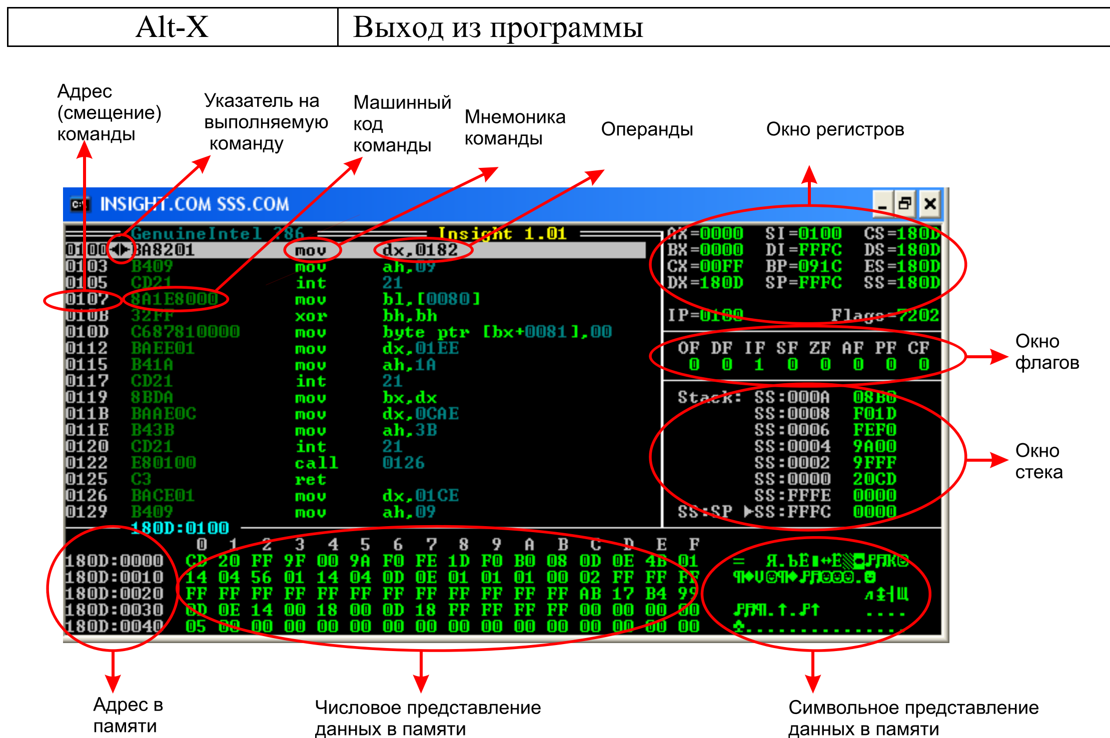



Рис. П.1.2

Компилятором называется программа, осуществляющая перевод текстового представления программы в её машинный код (конечный результат представления программы на любом языке программирования). Компилятор ассемблера является так называемым «компилятором 1 в 1», то есть каждой команде в программе строго соответствует машинная команда процессора, в отличие от языков высокого уровня.

Для обработки компилятором текст программы создается в текстовом файле в формате ASCII (DOS текст). В ОС Windows такие файлы создаются редактором Notepad и имеют расширение Txt. Более удобно для создания программы в текстовом файле пользоваться встроенным редактором файлового менеджера (FAR и другие). Для того, чтобы создать текстовый файл, перейдите в каталог, где вы будете работать (**C:\student** или другой, разрешенный для работы) и нажмите **Shift-F4.** В открывшемся окне наберите имя файла с расширением ASM. Например, lab3.asm. В открывшемся окне создайте текст программы, сохраните его при необходимости на диск (**F2**). Можно также не создавать заново текст, чтобы не повторять стандартные участки программы, а редактировать уже имеющейся файл, удаляя из него ненужные и добавляя новые команды. Для этого выберете нужный файл и нажмите **F4** для редактирования его. Учтите, что старое содержимое файла при этом будет потеряно, если Вы не сохраните файл под другим именем (**shift-F2**).

После создания файла необходимо откомпилировать его. Для этого скопируйте в каталог, в котором вы работаете, файлы **tasm.exe** (компилятор) и **tlink.exe** (компановщик) из каталога **C:\inst\insight\tasm** (копирование в FAR производится с помощью клавиши **F5**) . Для того, чтобы

откомпилировать свою программу, наберите в командной строке (под панелями с именами файлов, начинающейся с приглашения **C:\)** **tasm.exe lab3.asm** , где lab3.asm – имя вашего файла. Чтобы увидеть результаты программы, уберите с экрана панели с именами файлов с помощью **CTRL-O,** таким же образом их можно потом вернуть на место. Чтобы не набирать названия файлов в командной строке, вы можете перенести в командную строку имена файлов, выбрав их поочередно в панели и нажав **CTRL-ENTER**.

Компилятор выдает информацию об успешности процесса выполнения в виде:

Turbo Assembler  Version 2.0  Copyright (c) 1988, 1990 Borland International

Assembling file:   lab3.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  441k

Если Error messages: none, то есть ошибок при компиляции не возникло, на диске появится файл с тем же именем, что и исходный, но с раширением **.obj** (объектный модуль), например **lab3.obj**.  Если имеется список ошибок:

Turbo Assembler  Version 2.0  Copyright (c) 1988, 1990 Borland International

Assembling file:   lab3.ASM
**Error** lab3.ASM(11) Undefined symbol: AR
Error messages:    1
Warning messages:  None
Passes:            1
Remaining memory:  441k

То необходимо устранить их, отредактировав файл с программой, после чего повторить процедуру компиляции. (Чтобы не повторять набор команд, можно воспользоваться историей команд, нажав **ALT-F8**, и выбрать в списке последних набранных команд нужную) После имени файла в скобках указывается строка в программе, где обнаружена ошибка. Внимательно посмотрите на команду в этой строке. Если ошибок в ней нет, возможно, ошибка допущена в другом месте программы, из-за чего данная команда не может быть корректно распознана. Например, в данном случае в команда ссылается на неопределенный символ Ar – вероятно, вы забыли описать переменную с таким названием.

После успешной процедуры компиляции, необходимо перевести объектный модуль в выполняемый файл (.com). для этого выполните команду

(сформировав её в командной строке так же, как и при компиляции): **tlink.exe lab3.obj /t** Если компоновка прошла успешно, то в каталоге появится файл **lab3.com.**

Это выполняемый файл программы. Его можно просто запустить, как и любую другую программу. Однако, поскольку в нем отсутствуют команды вывода на экран (их вы научитесь применять во втором семестре), результата мы таким образом не увидим. Для того, чтобы протестировать и отладить программу, загрузите её в отладчик Insight. Для этого скопируйте его в ваш рабочий каталог, и запустите из командной строки **Insight.com lab3.com.** Теперь вы можете работать в отладчике с вашим программным кодом.

Основные отличия программы при написание в текстовом файле и представления в отладчике.

А) В текстовом файле необходимо указывать некоторые специальные операторы, указывающие компилятору на тип выполняемой программы. Типовая структура текста ассемблерной программы (для типа выполняемых файлов .com , все практические задания в рамка курса мы будем выполнять именно в виде таких файлов):

```
.model tiny        ; спецификатор модели памяти (односегментная)
.code              ; указатель начала сегмента кода
  org 100h         ; переопределения счетчика адресов
  start:           ; точка входа (может называться как угодно, но
должна быть)
……                 ; команды вашей программы
                        ret     ; команда возврата (в конце
                   основной com-программы – выход ; в
                   операционную систему
a db ….            ; описание данных в вашей программе

                        end start       ; указание конца файла и
                   точки входа. После этой строки не
                        ; анализируются компилятором
```

Можно использовать как прописные, так и строчные буквы.

Б) В текстовом файле обычно не используются непосредственные адреса команд и данных. Компилятор сам считает адреса для каждой команды или переменной – это позволяет свободно добавлять команды и данные, не изменяя текста программы при изменении, хотя адреса команд и данных изменятся при добавлении/удалении команд в середину программы. Вместо адресов используются символические имена (идентификаторы) для переменных и меток. В отладчике переменных как таковых нет, там можно только редактировать ячейки памяти по фиксированным адресам. При загрузке программы в отладчик, вы будете видеть абсолютные адреса (табл. П.1.2).

Таблица П.1.2.

| В текстовом файле | В отладчике |
|---|---|
| Jl m1<br>Mov ax,bx<br>Jmp m3<br>m1: mov bx,ax<br>m3: | 0110 Jl 0117<br>0112 Mov ax,bx<br>0114 Jmp short 0119 *<br>0117 mov bx,ax<br>0119 |
| Mov ax,a[SI] | 0110 mov ax,[SI+0130]<br>*(0130 – адрес переменной а в памяти)* |
| Mov al,a[si]<br>Mov bx,b[di]<br>Mov ax,X<br>Mov X,5<br><br><br><br><br><br>A db 2,3,4<br><br>B dw 5,6,-2<br><br>X dw 4 | Mov al,[si+**0140**]<br>Mov bx,[di+**0170**]<br>Mov ax,[**0180**]<br>Mov byte ptr [**0180**],5 ** <br><br>*В окне данных в отладчике (внизу) по*<br>*адресу, соответствующему переменной А*<br>*(можно узнать из команды обращения к*<br>*данной переменной)*<br>0CB0:**0140** **02 03 04 00** 00 00 00 00 00 00 00<br><br>0CB0:**0170** **05 00 06 00 FE FF** 00 00 00 00<br><br>0CB0:**0180** **04 00** 00 00 00 |
| Числа указываются в десятичной системе счисления по умолчанию.<br>Можно также указывать в шестнадцатеричной, двоичной, восьмеричной:<br>A db 32<br>A db 20h<br>A db 00100000b<br>A db 40q | Все числа отображаются как шестнадцатеричные<br><br><br><br><br>0CB0:**0140** **20** |
| | **(short – указание на короткий переход, в пределах 127 байт, по умолчанию формируется компилятором, если метка расположена по программному коду ближе 127 байт)*<br>*** (указатели размера операнда – byte ptr и word ptr используются тогда, когда нельзя определить, слово или байт (5) необходимо записать в память? Иногда их приходится использовать и в тексте программы)* |

# ПРИЛОЖЕНИЕ 2. СИСТЕМА КОМАНД МИКРОПРОЦЕССОРА INTEL 8086

Обозначения в табл. П. 2.1: $n_T$ – число тактов, требуемое для выполнения команды; *E* – число тактов, затрачиваемое на вычисление исполнительного адреса EA операнда, находящегося в памяти; r (или rl, r2) – регистр ЦП; *seg* – сегментный регистр ЦП; *a* (или *A)* – аккумулятор *AL* или *AX;* mem – ячейка памяти; data – данные, непосредственно представленные в команде (data *L,* data *H* – младший и старший байты данных); port – имя или номер порта ввода – вывода; disp – константа смещения (disp *L,* disp *H* – младший и старший байты константы смещения); label – метка (адрес метки или ее имя); name – имя подпрограммы (или ее начальный адрес); addr – адрес (addr *L,* addr *H* – младший и старший байты адреса); type – тип или уровень прерывания; mod – режим адресации; reg – код регистра; r/m – регистр/память (поле в постбайте команды); *w* = 0 – размерность регистра (памяти или данных) 8 разрядов, *w*=1 – размерность регистра (памяти или данных) 16 разрядов; d=0 – пересылка из регистра, d=l – пересылка в регистр; *sw* = 01–16-разрядная константа, *sw*=ll – 8-разрядная константа; *v*=0 – счетчик=1, *v*=1 – счетчик=*(CX).*

Таблица П. 2.1

| Группа команд | № п.п. | Мнемокод | $n_r$ | Байты формата команды | | Описание команды |
|---|---|---|---|---|---|---|
| | | | | Нечетный | Четный | |
| Пересылки данных | 1 | MOV r, r | 2 | 1 0 0 0 1 0 d w | mod reg r/m | Pr←Pr |
| | 2 | MOV r, mem | 8+E | | | Pr←П |
| | 3 | MOV mem, r | 9+E | | | П← Pr |
| | 4 | MOV mem, data | 10+E | 1 1 0 0 0 1 1 w | mod 0 0 0 r/m | П←Д |
| | | | | data L | data H (w=1) | |
| | 5 | MOV r, data | 4 | 1 0 1 1 w reg | data L | Pr← Д |
| | | | | data H (w=1) | | |
| | 6 | MOV a, mem | 10 | 1 0 1 0 0 0 d w | addr L | A←П |
| | | | | Addr H | | |
| | 7 | MOV mem, a | 10 | | | П← A |
| | 8 | MOV seg, r | 2 | 1 0 0 0 1 1 d 0 | mod 0 seg r/m | Сегментный Pr←Pr |
| | 9 | MOV seg, mem | 8+E | | | Сегментный Pr←П |
| | 10 | MOV r, seg | 2 | | | Pr← сегментный Pr |
| | 11 | MOV mem, seg | 9+E | | | П← сегментный Pr |
| | 12 | PUSH | 10 | 1 1 1 1 1 1 1 1 | mod 1 1 0 r/m | Стек←Pr |
| | 13 | PUSH mem | 16+E | | | Стек←П |
| | 14 | PUSH r | 10 | 0 1 0 1 0 reg | | Стек←Pr |
| | 15 | PUSH seg | 10 | 0 0 0 seg 1 1 0 | | Стек← сегментный Pr |
| | 16 | POP r | 8 | 1 0 0 0 1 1 1 1 | mod 0 0 0 r/m | Pr←стек |
| | 17 | POP mem | 17+E | | | П←стек |
| | 18 | POP r | 8 | 0 1 0 1 1 reg | | Pr←стек |
| | 19 | POP seg | 8 | 0 0 0 seg 1 1 1 | | Сегментный Pr←стек |
| | 20 | XCHG r, mem | 17+E | mod reg r/m | | Pr←П |
| | 21 | XCHG r, r | 4 | | | Pr←Pr |
| | 22 | XCHG AX, r | 3 | 1 0 0 1 0 reg | | A←Pr |
| | 23 | IN port | 10 | 1 1 1 0 0 1 0 w | port | A←порт |
| | 24 | IN | 8 | 1 1 1 0 1 1 0 w | | A←(*DX*) |
| | 25 | OUT port | 10 | 1 1 1 0 0 1 1 w | port | Порт← A |
| | 26 | OUT | 8 | 1 1 1 0 1 1 1 w | | (*DX*) ← A |
| | 27 | LEA r | 2+E | 1 0 0 0 1 1 0 1 | mod reg r/m | Pr←EA |
| | 28 | LDS r, mem | 16+E | 1 1 0 0 0 1 0 1 | mod reg r/m | Pr и *DS*←П |
| | 29 | LES r, mem | 16+E | 1 1 0 0 0 1 0 0 | mod reg r/m | Pr и *ES*←П |
| | 30 | LAHF | 4 | 1 0 0 1 1 1 1 1 | | *AH*←*FL* |
| | 31 | SAHF | 4 | 1 0 0 1 1 1 1 0 | | *FL*←*AH* |
| | 32 | PUSHF | 10 | 1 0 0 1 1 1 0 0 | | Стек←*F* |
| | 33 | POPF | 8 | 1 0 0 1 1 1 0 1 | | *F*←стек |
| Арифмети-ческие действия | 34 | ADD rl, r2 | 3 | 0 0 0 0 0 0 d w | mod reg r/m | Pr←Pr+Pr |
| | 35 | ADD r, mem | 9+E | | | Pr←Pr+ П |
| | 36 | ADD mem, r | 16+E | | | П←Pr+Pr |
| | 37 | ADD r, data | 4 | 1 0 0 0 0 0 s w | mod 0 0 0 r/m | Pr←Pr+ Д |
| | | | | data L | data H (sw=01) | |
| | 38 | ADD mem, data | 17+E | | | П← П+Д |
| | 39 | ADD a, data | 4 | 0 0 0 0 0 1 0 w | data L | A← A+Д |
| | | | | data H (w=1) | | |
| | 40 | ADC r1, r2 | 3 | 0 0 0 1 0 0 d w | mod reg r/m | Pr←Pr+Pr+*CF* |
| | 41 | ADC r, mem | 9+E | | | Pr←Pr+П+*CF* |
| | 42 | ADC mem, r | 16+E | | | П←П+Pr+*CF* |
| | 43 | ADC r, data | 4 | 1 0 0 0 0 0 s w | mod 0 1 0 r/m | Pr←Pr+Д+*CF* |
| | 44 | ADC mem, data | 17+E | data L | data H (sw=01) | П←П+ Д +*CF* |
| | 45 | ADC a, data | 4 | 0 0 0 1 0 1 0 w | data L | A← A+Д+*CF* |
| | | | | data H (w=1) | | |
| | 46 | INC r | 2 | 1 1 1 1 1 1 1 w | mod 0 0 0 r/m | Pr←Pr+1 |
| | 47 | INC mem | 15+E | | | П←П+1 |
| | 48 | INC r | 2 | 0 1 0 0 0 reg | | Pr←Pr (8)+1 |
| | 49 | SUB rl, r2 | 3 | 0 0 1 0 1 0 d w | | Pr←Pr—Pr |
| | 50 | SUB r, mem | 9+E | | | Pr←Pr—П |
| | 51 | SUB mem, r | 16+E | | | П←П—Pr |
| | 52 | SUB r, data | 4 | 1 0 0 0 0 0 s w | mod 1 0 1 r/m | Pr←Pr— Д |
| | | | | data L | data H (sw=01) | |
| | 53 | SUB mem, data | 17+E | | | П←П—Д |
| | 54 | SUB a, data | 4 | 0 0 1 0 1 1 0 w | data L | A← A—Д |
| | | | | data H (w=1) | | |

| Группа команд | № п.п. | Мнемокод | $n_r$ | Байты формата команды | | Описание команды |
|---|---|---|---|---|---|---|
| | | | | Нечетный | Четный | |
| Арифметические действия | 55 | SBB r1, r2 | 3 | 0 0 0 1 1 0 d w | mod reg r/m | Pr←Pr—Pr—*CF* |
| | 56 | SBB r, mem | 9+E | | | Pr←Pr—П—*CF* |
| | 57 | SBB mem, r | 16+E | | | П←П—Pr—*CF* |
| | 58 | SBB r, data | 4 | 1 0 0 0 0 0 s w<br>data L | mod 0 1 1 r/m<br>data H (sw=01) | Pr←Pr—Д—*CF* |
| | 59 | SBB mem, data | 17+E | | | П←П—Д—*CF* |
| | 60 | SBB a, data | 4 | 0 0 0 1 1 1 0 w<br>data H (w=1) | data L | A←A—Д |
| | 61 | DEC r | 2 | 1 1 1 1 1 1 1 w | mod 0 0 1 r/m | Pr←Pr—I |
| | 62 | DEC mem | 155+E | | | П←П—I |
| | 63 | DEC r | 2 | 0 1 0 0 1 reg | | Pr←Pr(8)—I |
| | 64 | NEG r | 3 | 1 1 1 1 0 1 1 w | mod 0 1 1 r/m | Pr←0—Pr |
| | 65 | NEG mem | 16+E | | | П←0—I |
| | 66 | CMP r1, r2 | 3 | 0 0 1 1 1 0 d w | mod reg r/m | Pr—Pr |
| | 67 | CMP r, mem | 9+E | | | Pr—П |
| | 68 | CMP mem, r | 16+E | | | П—Pr |
| | 69 | CMP r, data | 4 | 1 0 0 0 0 0 s w<br>data L | mod 1 1 1 r/m<br>data H (w=1) | Pr—Д |
| | 70 | CMP mem, data | 17+E | | | П—Д |
| | 71 | CMP a, data | 4 | 0 0 1 1 1 1 0 w<br>data H (w=1) | data L | A—Д |
| | 72 | MUL src | 71+E<br>124+E | 1 1 1 1 0 1 1 w | mod 1 0 0 r/m | *AX*←*AL* источник (при ϖ=0)<br>*DX, AX*←*AX* источник (при ϖ=1) |
| | 73 | IMUL src | 90+E<br>144+E | 1 1 1 1 0 1 1 w | mod 1 0 0 r/m | *AX*←*AL* источник (при ϖ=0)<br>*DX, AX*←*AX* источник (при ϖ=1)<br>со знаком |
| | 74 | DIV src | 90+E<br>112+E | 1 1 1 1 0 1 1 w | mod 1 1 0 r/m | AL←*AX/* источник, *AH* остаток (при ϖ=0)<br>AX←*D, AX/* источник, *DX* остаток (при ϖ=1) |
| | 75 | IDIV src | 112+E<br>177+E | 1 1 1 1 0 1 1 w | | AL←*AX/* источник, *AH* остаток (при ϖ=0)<br>AX←*D, AX/* источник, *DX* остаток (при ϖ=1) |
| | 76 | DAA | 4 | 0 0 1 0 0 1 1 1 | | *AL*←скорректированное *AL* (сложение, 2—10) |
| | 77 | DAS | 4 | 0 0 1 0 1 1 1 1 | | *AL*←скорректированное *AL* (вычитание, 2—10) |
| | 78 | AAA | 4 | 0 0 1 1 0 1 1 1 | | *AL*←скорректированное *AL* (сложение, ASCII) |
| | 79 | AAS | 4 | 0 0 1 1 1 1 1 1 | | *AL*←скорректированное *AL* (вычитание, ASCII) |
| | 80 | AAM | 83 | 1 1 0 1 0 1 0 0 | 0 0 0 0 1 0 1 0 | *AX*←скорректированное *AX* (умножение) |
| | 81 | AAD | 60 | 1 1 0 1 0 1 0 1 | 0 0 0 0 1 0 1 0 | *AX*←скорректированное *AX* (деление) |
| | 82 | CBW | 5 | 1 0 0 1 1 0 0 0 | | *AH*← знак *AL* |
| | 83 | CWD | 5 | 1 0 0 1 1 0 0 1 | | *DX*← знак *AX* |
| Сдвиги | 84 | SHL/SAL r | 2 | 1 1 0 1 0 0 v w | mod * 0 0 r/m | Логический влево (Pr) |
| | 85 | SHL/SAL mem | 15+E | | | Логический влево (П) |
| | 86 | SHR r | 2 | 1 1 0 1 0 0 v w | mod 1 0 1 r/m | Логический вправо (Pr) |
| | 87 | SHR mem | 15+E | | | Логический вправо (П) |
| | 88 | SAR r | 2 | 1 1 0 1 0 0 v w | mod 1 1 1 r/m | Арифметический вправо (Pr) |

| Группа команд | № п.п. | Мнемокод | $n_r$ | Байты формата команды | | Описание команды |
|---|---|---|---|---|---|---|
| | | | | Нечетный | Четный | |
| | 89 | SAR mem | 15+E | | | Арифметический вправо (п) |
| | 90 | ROL r | 8 | 1 1 0 1 0 0 v w | mod 0 0 0 r/m | Циклический влево (Pr) |
| | 91 | ROL mem | 20+E | | | Циклический влево(П) |
| | 92 | ROR к | 8 | 1 1 0 1 0 0 v w | mod 0 0 1 r/m | Циклический вправо (Pr) |
| | 93 | ROR mem | 20+E | | | Циклический вправо (П) |
| | 94 | RCL r | 8 | 1 1 0 1 0 0 v w | mod 0 1 0 r/m | Циклический через CF влево (Pr) |
| | 95 | RCL mem | 20+E | | | Циклический через CF влево (П) |
| | 96 | RCR к | 8 | 1 1 0 1 0 0 v w | mod 0 1 1 r/m | Циклический через CF вправо (Pr) |
| | 97 | RCR mem | 20+E | | | Циклический через CF вправо (П) |
| Логические действия | 98 | AND rl, r2 | 3 | 0 0 1 0 0 0 d w | mod reg r/m | Pr←PrΛPr |
| | 99 | AND r, mem | 9+E | | | Pr←PrΛП |
| | 100 | AND mem, r | 16+E | | | П←ПΛPr |
| | 101 | AND r, data | 4 | 1 0 0 0 0 0 0 w | mod 1 0 0 r/m | Pr←PrΛД |
| | | | | Data L | data H (w=1) | |
| | 102 | AND mem, data | 17+E | | | П←ПΛД |
| | 103 | AND a, data | 4 | 0 0 1 0 0 1 0 w | data L | A←AΛД |
| | | | | data H (w=1) | | |
| | 104 | OR rl, r2 | 3 | 0 0 0 0 1 0 d w | mod reg r/m | Pr←PrVPr |
| | 105 | OR r, mem | 9+E | | | Pr←PrVП |
| | 106 | OR mem, r | 16+E | | | |
| | 107 | OR r, data | 4 | 1 0 0 0 0 0 0 w | mod 0 0 1 r/m | Pr←PrVД |
| | | | | data L | data H (w=1) | |
| | 108 | OR mem, data | 17+E | | | П←ПVД |
| | 109 | OR a, data | 4 | 0 0 0 0 1 1 0 w | data L | A←AVД |
| | | | | Data H | | |
| | 110 | XOR rl, r2 | 3 | 0 0 1 1 0 0 d w | mod reg r/m | Pr←Pr+Pr |
| | 111 | XOR r, mem | 9+E | | | Pr←Pr+П |
| | 112 | XOR mem, r | 16+E | | | П←П+Пr |
| | 113 | XOR r, data | 4 | 1 0 0 0 0 0 0 w | mod 1 1 0 r/m | Pr←Pr+Д |
| | | | | | data H (w=1) | |
| Логические операции | 114 | XOR mem, data | 17+E | | | П←П⊕Д |
| | 115 | XOR a, data | 4 | 0 0 1 1 0 1 0 w | data L | A←A⊕Д |
| | | | | data H (w=1) | | |
| | 116 | TEST r1, r2 | 3 | 1 0 0 0 0 1 0 w | mod reg r/m | PrΛPr Результат |
| | 117 | TEST r, mem | 9+E | | | PrΛП определяется |
| | 118 | TEST r, data | 4 | 1 1 1 1 0 1 1 w | mod 0 0 0 r/m | PrΛД по флагам |
| | | | | data L | data H (w=1) | ПΛД |
| | 119 | TEST, mem data | 10+E | | | |
| Обработка строк | 120 | TEST a, data | 4 | 1 0 1 0 1 0 0 w | data L | AΛД |
| | | | | data H (w=1) | | |
| | 121 | NOT r | 3 | 1 1 1 1 0 1 1 w | mod 0 1 0 r/m | Pr←$\overline{Pr}$ |
| | 122 | NOT meme | 16+E | | | П←П |
| | 123 | MOVS | 17 | 1 0 1 0 0 1 0 1 | | П[*DI*] ←П[*SI*] |
| | 124 | CMPS | 22 | 1 0 1 0 0 1 1 w | | П[*SI*]– П[*DI*] |
| | 125 | SCAS | 15 | 1 0 1 0 1 1 1 w | | A– П[*DI*] |
| | 126 | LODS | 12 | 1 0 1 0 1 1 0 w | | A← П[*SI*] |
| | 127 | STOS | 10 | 1 0 1 0 1 0 1 w | | П[*DI*] ←A |
| | 128 | REPNE/PERNZ | 6 | 1 1 1 1 0 0 1 z | | *CX*←*CX* – 1, повторять пока *CX*≠0 и *ZF*=0 |
| | 129 | REP/REPE/PERZ | 6 | | | Повторять пока *CX*≠0 b *ZF*=1 |
| Безусловные переходы | 130 | JMP label | 7 | 1 1 1 0 1 0 0 1 | disp L | *IP*←*IP*+смещение |
| | | | | disp H | | |

| Группа команд | № п.п. | Мнемокод | $n_r$ | Байты формата команды | | Описание команды |
|---|---|---|---|---|---|---|
| | | | | Нечетный | Четный | |
| | 131 | JMP label | 2 | 1 1 1 0 1 0 1 1 | disp L | *IP*←*IP*+смещение, расширенное со знаком до 16 бит |
| | 132 | JMP label | 7+E | 1 1 1 1 1 1 1 1 | mod 1 0 0 r/m | *IP*←(*EA*) |
| | 133 | JMP label | 7 | 1 1 1 0 1 0 1 0<br>addr H<br>seg H | addr L<br>seg L | *IP*←адрес перехода<br>*CS*←адрес сегмента |
| | 134 | JMP label | 16+E | 1 1 1 1 1 1 1 1 | mod 1 0 1 r/m | *IP*←(*EA*); *CS* (*EA*+2) |
| | 135 | CALL name | 11 | 1 1 1 0 1 0 0 0<br>disp H | disp L | *SP*←*SP*—2; *SP*—1; *SP*←*IP* |
| | 136 | CALL name | 13+E | 1 1 1 1 1 1 1 1 | mod 0 1 0 r/m | *SP*←*SP*—2; *SP*—1; *SP*←*IP*<br>*IP*←(*EA*) |
| | 137 | CALL name | 20 | 1 0 0 1 1 0 1 0<br>addr H<br>seg H | add L<br>seg L | *SP*←*SP*—2; *SP*—1; *SP*←*CS*<br>*SP*←*SP*—2; *SP*—1;<br>*SP*←*IP*<br>*IP*←адрес перехода; CS←адрес сегмента |
| | 138 | RET | 8 | 1 1 0 0 0 0 1 1 | disp L | *IP*←(*SP*+1, *SP*): *SP*←*SP*+2 |
| | 139 | RET | 18 | 1 1 0 0 0 0 1 0<br>data H | data L | *IP*←(*SP*+1, *SP*): *SP*←*SP*+*D* |
| | 140 | RET | 18 | 1 1 0 0 1 0 1 1 | | *IP*←(*SP*+1, *SP*): *SP*←*SP*+2<br>*CS*←(*SP*+1, *SP*): *SP*←*SP*+2 |
| | 141 | RET | 18 | 1 1 1 0 0 1 0 1 0<br>Data H | data L | *IP*←(*SP*+1, *SP*): *SP*←*SP*+2<br>*CS*←(*SP*+1, *SP*): *SP*←*SP*+*D* |
| Условные переходы | 142 | JZ/JE label | 8/4 | 1 1 0 0 0 0 1 1 | disp | По равенству |
| | 143 | JNZ/JNE label | 8/4 | 0 1 1 1 0 1 0 1 | disp | По неравенству |
| | 144 | JL/JNGE label | 8/4 | 0 1 1 1 1 1 0 0 | disp | По меньше (со знаком) |
| | 145 | JNL/JGE label | 8/4 | 0 1 1 1 1 1 0 1 | disp | По не меньше (со знаком) |
| | 146 | JLE/JNG label | 8/4 | 0 1 1 1 1 1 1 0 | disp | По не больше (со знаком) |
| | 147 | JNLE/JG label | 8/4 | 0 1 1 1 1 1 1 1 | disp | По больше (со знаком) |
| | 148 | JB/JNAE label | 8/4 | 0 1 1 1 0 0 1 0 | disp | По меньше (без знака) |
| | 149 | JNB/JNA label | 8/4 | 0 1 1 1 0 0 1 1 | disp | По не меньше (без знака) |
| | 150 | JBE/JNA label | 8/4 | 0 1 1 1 0 1 1 0 | disp | По не больше (без знака) |
| | 151 | JNBE/JA label | 8/4 | 0 1 1 1 0 1 1 1 | disp | По больше (без знака) |
| | 152 | JP/JPE label | 8/4 | 0 1 1 1 1 0 1 0 | disp | По четному паритету |
| | 153 | JNP/JPO label | 8/4 | 0 1 1 1 1 0 1 1 | disp | По нечетному паритету |
| | 154 | JO label | 8/4 | 0 1 1 1 0 0 0 0 | disp | По переполнению |
| | 155 | JNO label | 8/4 | 0 1 1 1 0 0 0 1 | disp | По непереполнению |
| | 156 | JS label | 8/4 | 0 1 1 1 1 0 0 0 | disp | По минусу |
| | 157 | JNS label | 8/4 | 0 1 1 1 1 0 0 1 | disp | По плюсу |
| | 158 | LOOP label | 9 | 1 1 1 0 0 0 1 0 | disp | *IP*←*IP*+смещение, если *CX*≠0 |
| | 159 | LOOPNZ/LOOPNE label | 11 | 1 1 1 0 0 0 0 z | disp | *IP*←*IP*+смещение, если *Z*=0 |
| | 160 | | 11 | 1 1 1 0 0 0 1 1 | disp | *IP*←*IP*+смещение, если *Z*=1 |
| | 161 | | 9 | | disp | *IP*←*IP*+смещение, если *CX*=0 |

| Группа команд | № п.п. | Мнемокод | $n_r$ | Байты формата команды | | Описание команды |
|---|---|---|---|---|---|---|
| | | | | Нечетный | Четный | |
| Преры-вания | 162 | INT | 52 | 1 1 0 0 1 1 0 1 | type | Прерывание заданного типа |
| | 163 | INT | 52 | 1 1 0 0 1 1 0 0 | | Прерывание типа 3 |
| | 164 | INTO | 52 | 1 1 0 0 1 1 1 0 | | Прерывание при переполнении |
| | 165 | IRET | 24 | 1 1 0 0 1 1 1 1 | | Возврат без прерывания |
| Управ-ление процес-сором | 166 | STC | 2 | 1 1 1 1 1 0 0 1 | | $CF \leftarrow 1$ |
| | 167 | CMC | 2 | 1 1 1 1 0 1 0 1 | | $CF \leftarrow \overline{CF}$ |
| | 168 | CLC | 2 | 1 1 1 1 1 0 0 0 | | $CF \leftarrow 0$ |
| | 169 | STD | 2 | 1 1 1 1 1 1 0 1 | | $DF \leftarrow 1$ |
| | 170 | CLD | 2 | 1 1 1 1 1 1 0 0 | | $DF \leftarrow 0$ |
| | 171 | STI | 2 | 1 1 1 1 1 0 1 1 | | $IF \leftarrow 1$ |
| | 172 | CLI | 2 | 1 1 1 1 1 0 1 0 | | $IF \leftarrow 0$ |
| | 173 | HLT | 2 | 1 1 1 1 0 1 0 0 | | Останов, $IP \leftarrow IP+1$ |
| | 174 | WAIT | 3 | 1 0 0 1 1 0 1 1 | | Ожидание |
| | 175 | ESC | 7+E | 1 1 0 1 1 x | mod y r/m | Переход к работе сопроцессора |
| | 176 | LOCK | 2 | 1 1 1 1 0 0 0 0 | | Блокировка шин |
| | 177 | NOP | 2 | 1 0 0 1 0 0 0 0 | | Пустая операция |

Таблица П. 2.2. Влияние команд микропроцессора ВМ86 на значения флагов состояния

| Группы команд | Команды | Флаги состояния | | | | | |
|---|---|---|---|---|---|---|---|
| | | OF | CF | AF | SF | ZF | PF |
| Сложение и вычитание | ADD ADC SUB SBC<br>CMP NEG CMPS SCAS<br>INC DEC | +<br>+<br>+ | +<br>+<br>- | +<br>+<br>+ | +<br>+<br>+ | +<br>+<br>+ | +<br>+<br>+ |
| Умножение и деление<br>Десятичная коррекция | MUL IMUL<br>DIV IDIV<br>DAA DAS<br>AAA AAS<br>AAM AAD | +<br>?<br>?<br>?<br>? | +<br>?<br>+<br>+<br>? | ?<br>?<br>+<br>+<br>? | ?<br>?<br>+<br>?<br>+ | ?<br>?<br>+<br>?<br>+ | ?<br>?<br>+<br>?<br>+ |
| Логические операции | AND OR XOR TEST | 0 | 0 | ? | + | + | + |
| Сдвиги:<br>одиночный<br>многоразрядный<br><br>одиночный<br>многоразрядный<br>Восстановление флагов<br>Управление флагом пареноса | SHL SHR<br>SHL SHR<br>SAR<br>ROL ROR RCL RCR<br>ROL ROR RCL RCR<br>POPF IRET<br>SAHF<br>STC<br>CLC<br>CMC | +<br>?<br>0<br>+<br>?<br>+<br>-<br>-<br>-<br>- | +<br>+<br>+<br>+<br>+<br>+<br>+<br>1<br>0<br>* | ?<br>?<br>?<br>-<br>-<br>+<br>+<br>-<br>-<br>- | +<br>+<br>+<br>-<br>-<br>+<br>+<br>-<br>-<br>- | +<br>+<br>+<br>-<br>-<br>+<br>+<br>-<br>-<br>- | +<br>+<br>+<br>-<br>-<br>+<br>+<br>-<br>-<br>- |

| Группы Команд | Команды | Флаги состояния | | |
|---|---|---|---|---|
| | | DF | IF | TF |
| Восстановление флагов | POPF IRET | * | + | + |
| Прерывание<br>Управление флагами | INT INTO<br>STD<br>CLD | -<br>1<br>0 | 0<br>-<br>- | 0<br>-<br>- |
| Управление флагами | STI<br>CLI | -<br>- | 1<br>0 | -<br>- |

Примечание. + - влияет на флаг,0 – сбрасывает в «0»,1 – устанавливает в «1», ? – не определено, * – инвертирует, - не влияет

Мащенко Павел Евгеньевич. Романчиков Андрей Михайлович.
**АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ**
**МИКРОПРОЦЕССОРА INTEL 8086**

Учебное пособие

---

Подписано к печати_____________ Формат 60х84/16
Усл. печ. л ____________________ Тираж______экз.
Заказ №_________

---

127994, Москва, ул. Образцова, 15
Типография МИИТ