Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информатика»

Г. П. Косинов

ПРОГРАММИРОВАНИЕ

ПРАКТИКУМ
по одноименной дисциплине для студентов специальности 1-40 04 01 «Информатика и технологии программирования» дневной формы обучения

Электронный аналог печатного издания

Гомель 2017

УДК 004.42(075.8)
ББК 32.973я73
К71

Рекомендовано к изданию научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 11 от 27.06.2016 г.*)

Рецензент: доц. каф. «Информационные технологии» ГГТУ им. П. О. Сухого канд. физ.-мат. наук, доц. *Е. Г. Стародубцев*

Косинов, Г. П.

К71 Программирование : практикум по одноим. дисциплине для студентов специальности 1-40 04 01 «Информатика и технологии программирования» днев. формы обучения / Г. П. Косинов. – Гомель : ГГТУ им. П. О. Сухого, 2017. – 65 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

ISBN 978-985-535-341-7.

Содержит 6 глав, в которых приведены краткие теоретические сведения, даны 30 вариантов заданий, представлен пример программного кода решения типовой задачи. Позволит студентам изучить синтаксис и семантику машинно-ориентированного языка низкого уровня Ассемблер.

Для студентов специальности 1-40 04 01 «Информатика и технологии программирования» дневной формы обучения.

УДК 004.42(075.8)
ББК 32.973

ISBN 978-985-535-341-7

Глава 1. Общая характеристика языка Ассемблер, его синтаксис. Структура .com и .exe программ

Чтобы программа выполнилась, она должна быть скомпилирована в исполняемый файл. Основные два формата исполняемых файлов – COM и EXE. Файлы типа COM содержат только скомпилированный код без какой-либо дополнительной информации о программе. Весь код, данные и стек такой программы располагаются в одном сегменте и не могут превышать 64 килобайта (кб). Файлы типа EXE содержат заголовок, в котором описывается размер файла, требуемый объем памяти, список команд в программе, использующих абсолютные адреса, которые зависят от расположения программы в памяти. EXE-файл может иметь любой размер. Формат EXE также используется для исполняемых файлов в различных версиях ОС DOS-расширителей и Windows. Технология получения исполняемого файла представлена на рис. 1.1.

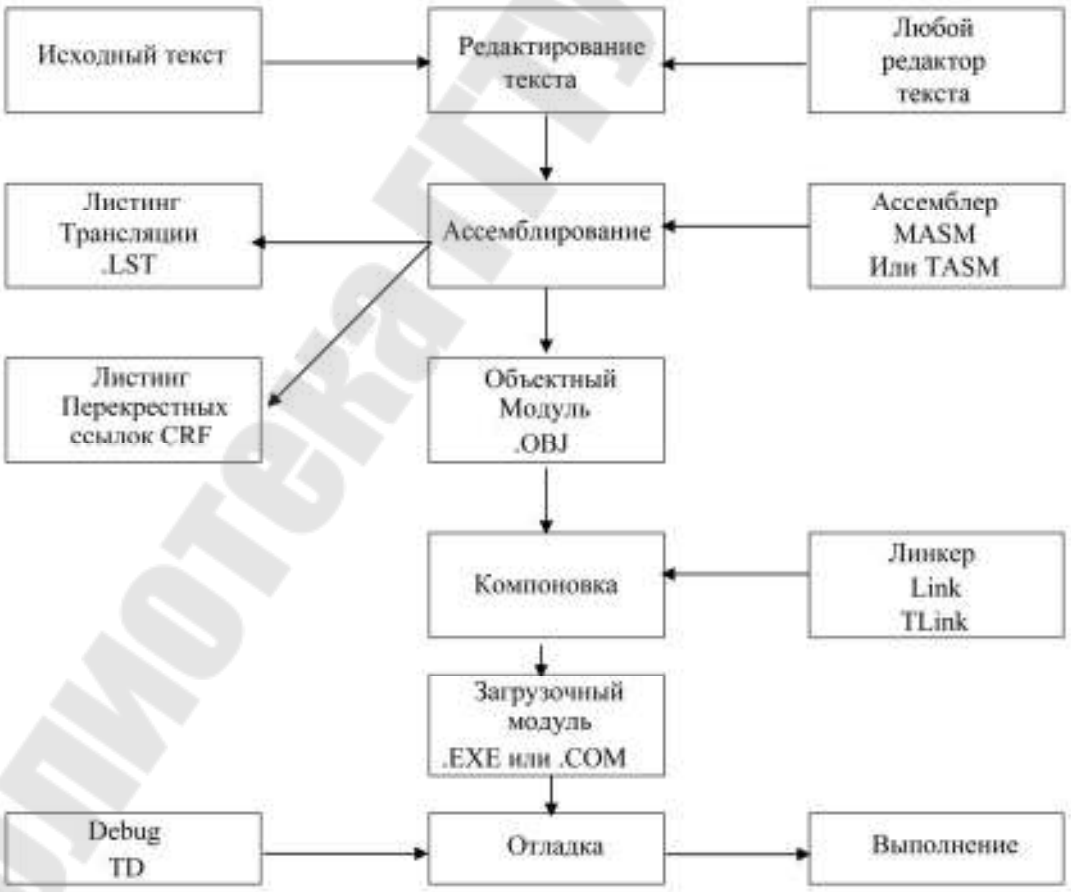

Рис. 1.1. Технология получения исполняемого файла

Модели памяти для создания исполняемого файла:

– TINY – код, данные и стек размещаются в одном и том же сегменте размером 64 кб;

– SMALL – код – в одном сегменте, а данные и стек – в другом;

– COMPACT – код – в одном сегменте, а данные могут располагаться в нескольких сегментах. Используются дальние указатели;

– MEDIUM – код – в нескольких сегментах, а все данные – в одном. Для доступа к данным используется только смещение, а для вызова подпрограмм – команды дальнего вызова процедур;

– LARGE, HUGE – и код, и данные могут занимать несколько сегментов;

– FLAT – то же, что и TINY, но используются 32-разрядные адреса. Максимальный размер сегмента – 4 гигабайта (Гб).

Если в явном виде в тексте программы не указана модель памяти, то по умолчанию создается EXE-файл с моделью памяти – SMALL.

1.1. Создание программы типа COM

Для разработки простой COM-программы, которая выводит на экран текст «Hello World!», необходимо в любом тестовом редакторе ввести текст:

```
; hello-l.asm
; Выводит на экран сообщение "Hello World!" и завершается
  .model  tiny              ; модель памяти, используемая для COM
  .code                     ; начало сегмента кода
  org  100h                 ; начальное значение счетчика – 100h
  main:  mov  ah,9          ; номер функции DOS – в AH
  mov  dx,offset message ; адрес строки – в DX
  int  21h                  ; вызов системной функции DOS
  ret                       ; завершение COM-программы
  message db  "Hello World!",0Dh,0Ah,'$' ; строка для вывода
  end  main                 ; конец программы.
```

После этого сохранить тестовый файл под именем hello-1.asm.

Для превращения программы в исполняемый файл нужно сначала вызвать транслятор для компиляции ее в объектный файл с именем hello-1.obj, набрав в командной строке следующую команду:

ml /c hello-l.asm,

где /c – параметр, который указывает на то, что нужно оттранслировать исходный файл (после этого будет создан объектный файл hello-l.obj).

После этого нужно вызвать компоновщик для создания hello-l.com, набрав в командной строке следующую команду:

link /t hello-l.obj , hello-l.com;

где /t – параметр, который указывает на то, что нужно создать исполняемый файл формата .com, затем указывается объектный файл и название исполняемого файла, который должен быть создан.

В данном примере трансляции и компоновки отсутствуют параметры, которые необходимы для создания листинга и включения информации для отладчика CodeView, так как программа простая и необходимости в этих действиях нет.

Рассмотрим исходный текст программы, чтобы понять, как она работает.

Первая строка определяет модель памяти **TINY**, в которой сегменты кода, данных и стека объединены. Эта модель предназначена для создания файлов типа COM.

Директива **.CODE** начинает сегмент кода, который в нашем случае также должен содержать и данные.

ORG 100h устанавливает значение программного счетчика в 100h, так как при загрузке COM-файла в память DOS занимает первые 256 байт (100h) блоком данных PSP и располагает код программы только после этого блока. Все программы, которые компилируются в файлы типа COM, должны начинаться с этой директивы.

Метка **MAIN** располагается перед первой командой в программе и будет использоваться в директиве END, чтобы указать, с какой команды начинается программа.

Команда **MOV AH,9** помещает число 9 в регистр AH. Это номер функции DOS «вывод строки».

Команда **MOV DX,OFFSET MESSAGE** помещает в регистр DX смешение метки **MESSAGE** относительно начала сегмента данных, который в нашем случае совпадает с сегментом кода.

Команда **INT 21h** вызывает системную функцию DOS. Эта команда – основное средство взаимодействия программ с операционной системой. В нашем примере вызывается функция DOS номер 9 – вывести строку на экран. Эта функция выводит строку от начала, адрес которого задается в регистрах DS:DX, до первого встреченного сим-

вола $. При загрузке COM-файла регистр DS автоматически загружается сегментным адресом программы, а регистр DX был загружен предыдущей командой.

Команда **RET** используется обычно для возвращения из процедуры. DOS вызывает COM-программы так, что команда RET корректно завершает программу.

Следующая строка программы HELLO-1.ASM определяет строку данных, содержащую текст «Hello World!», управляющий символ ASCII «возврат каретки» с кодом **0Dh**, управляющий символ ASCII «перевод строки» с кодом **0Ah** и символ **«$»**, завершающий строку. Эти два управляющих символа переводят курсор на первую позицию следующей строки точно так же, как в строках на языке С действует последовательность «\n».

И наконец, директива **END** завершает программу, одновременно указывая, с какой метки должно начинаться выполнение программы.

Напомним, что файлы типа COM содержат только скомпилированный код без какой-либо дополнительной информации о программе. Весь код, данные и стек такой программы располагаются в одном сегменте и не могут превышать 64 кб.

1.2. Создание программы типа EXE

EXE-программы немного сложнее в исполнении, но для них отсутствует ограничение размера в 64 кб, так что все достаточно большие программы используют именно этот формат.

```
; hello-2.asm
; Выводит на экран сообщение "Hello World!" и завершается
   .model   small      ; модель памяти, используемая для EXE
   .stack   100h       ; сегмент стека размером в 256 байт
   .data
      message db   "Hello World!",0Dh,0Ah,'$'
   .code
main:   mov   ax,DGROUP   ; сегментный адрес строки message
   mov   ds,ax       ; помещается в DS
   mov   dx,offset message
   mov   ah,9
   int      21h      ; функция DOS "вывод строки"
   mov   ax,4C00h
   int      21h      ; функция DOS "завершить программу"
   end   main
```

Для трансляции программы нужно в командной строке написать команду:

ml /c hello-2.asm

Для компоновки нужно записать:

link hello-2.obj;

В этом примере определяются три сегмента – сегмент стека директивой .STACK размером в 256 байт, сегмент кода, начинающийся с директивы .CODE, и сегмент данных, начинающийся с .DATA и содержащий строку. При запуске EXE-программы регистр DS уже не содержит адреса сегмента со строкой message (он указывает на сегмент, содержащий блок данных PSP), а для вызова используемой функции DOS этот регистр должен иметь сегментный адрес строки. Команда MOV AX,DGROUP загружает в AX сегментный адрес группы сегментов данных DGROUP, а MOV DS,AX копирует его в DS. Для ассемблеров MASM и TASM можно использовать вместо DGROUP предопределенную метку «@data», но единственная модель памяти, в которой группа сегментов данных называется иначе, – FLAT (ей мы пока пользоваться не будем). И наконец, программы типа EXE должны завершаться системным вызовом DOS 4Ch: в регистр AH помещается значение 4Ch, в регистр AL помещается код возврата (в данном примере код возврата 0 и регистры AH и AL загружаются одной командой MOV AX,4C00h), после чего вызывается прерывание 21h.

Файлы типа EXE содержат заголовок, где описывается размер файла, требуемый объем памяти, список команд в программе, использующих абсолютные адреса, которые зависят от расположения программы в памяти, и т. д. EXE-файл может иметь любой размер.

1.3. Язык Ассемблер, его синтаксис

Программа на языке ассемблера состоит из строк, имеющих следующий вид:

метка команда/директива операнды ; комментарий

Причем все эти поля необязательны. Метка может быть любой комбинацией букв английского алфавита, цифр и символов _, $, @, ?, но цифра не может быть первым символом метки, а символы $ и ? иногда имеют специальные значения и обычно не рекомендуются к использованию. Большие и маленькие буквы по умолчанию не различа-

ются, но различие можно включить, задав ту или иную опцию в командной строке ассемблера. Во втором поле, поле команды, может располагаться команда процессора, которая транслируется в исполняемый код, или директива, которая не приводит к появлению нового кода, а управляет работой самого ассемблера. В поле операндов располагаются требуемые командой или директивой операнды (т. е. нельзя указать операнды и не указать команду или директиву). И наконец, в поле комментариев, начало которого отмечается символом; (точка с запятой), можно написать все что угодно – текст от символа «;» до конца строки не анализируется ассемблером.

1.4. Регистры процессора

Регистр процессора – сверхбыстрая оперативная память внутри процессора, предназначенная, прежде всего, для хранения промежуточных результатов вычисления.

Регистр представляет собой цифровую электронную схему, служащую для временного хранения двоичных чисел. В процессоре имеется значительное количество регистров, большая часть которых используется самим процессором и недоступна программисту (например, при выборке из памяти очередной команды она помещается в регистр команд, и программист обратиться к этому регистру не может). Имеются также регистры, которые в принципе программно доступны, но обращение к ним осуществляется из программ операционной системы (например, управляющие регистры и теневые регистры дескрипторов сегментов). Этими регистрами пользуются в основном разработчики операционных систем.

По назначению регистры различаются на:

– флаговые – хранят признаки результатов арифметических и логических операций;

– общего назначения – хранят операнды арифметических и логических выражений, индексы и адреса;

– индексные – хранят индексы исходных и целевых элементов массива;

– указательные – хранят указатели на специальные области памяти (указатель текущей операции, указатель базы, указатель стека);

– сегментные – хранят адреса и селекторы сегментов памяти;

– управляющие – хранят информацию, управляющую состоянием процессора, а также адреса системных таблиц.

Процессор Intel x86 после включения питания оказывается в так называемом режиме реальной адресации памяти. Большинство операционных систем переводят его в защищенный режим, позволяющий им обеспечивать многозадачность, распределение памяти и другие функции. Пользовательские программы в таких ОС работают в режиме V86, из которого им доступно все то же, что и из реального режима, кроме команд, относящихся к управлению защищенным режимом. Так как ОС мы проектировать не собираемся, то дальше мы будем рассматривать все то, что доступно программисту в режиме V86, т. е. в подавляющем большинстве случаев.

Начиная с 80386, процессоры Intel предоставляют 16 основных регистров для пользовательских программ плюс еще 11 регистров для работы с мультимедийными приложениями (MMX(Multimedia Extension)) и числами с плавающей запятой (FPU/NPX (Float Point Unit / Numerical Processor Extension)). Все команды так или иначе изменяют значения регистров, и всегда быстрее и удобнее обращаться к регистру, чем к памяти.

Из реального (но не из виртуального) режима, помимо основных регистров, доступны также регистры управления памятью (GDTR, IDTR, TR, LDTR); регистры управления (CR0, CR1–CR4); отладочные регистры (DR0–DR7) и машинно-специфичные регистры, но они не применяются для решения повседневных задач.

Регистры общего назначения. 32-битные регистры EAX (аккумулятор), EBX (база), ECX (счетчик), EDX (регистр данных) могут использоваться без ограничений для любых целей – временного хранения данных, аргументов или результатов различных операций. Название регистров происходит от того, что некоторые команды применяют их специальным образом: так, аккумулятор часто необходим 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 работает как счетчик в циклах и строковых операциях, а регистр-база – при так называемой адресации по базе. Младшие 16 бит каждого их этих регистров применяются как самостоятельные регистры с именами AX, BX, CX, DX. На самом деле в процессорах 8086–80286 все регистры были 16-битными и назывались именно так, а в 32-битные EAX–EDX появились с введением 32-битной архитектуры в 80386. Кроме этого, отдельные байты в 16-битных регистрах AX–DX тоже могут использоваться как 8-битные регистры и иметь свои имена. Старшие байты

этих регистров называются AH, BH, CH, DH, а младшие – AL, BL, CL, DL (см. рис. 1.2).

Остальные четыре регистра – ESI (индекс источника), EDI (индекс приемника), EBP (указатель базы), ESP (указатель стека) – имеют более конкретное назначение и применяются для хранения всевозможных временных переменных. Регистры ESI и EDI необходимы в строковых операциях, EBP и ESP – при работе со стеком. Так же как в случае с регистрами EAX–EDX, младшие половины этих четырех регистров называются SI, DI, BP и SP, соответственно, и в процессорах до 80386 только они и присутствовали.

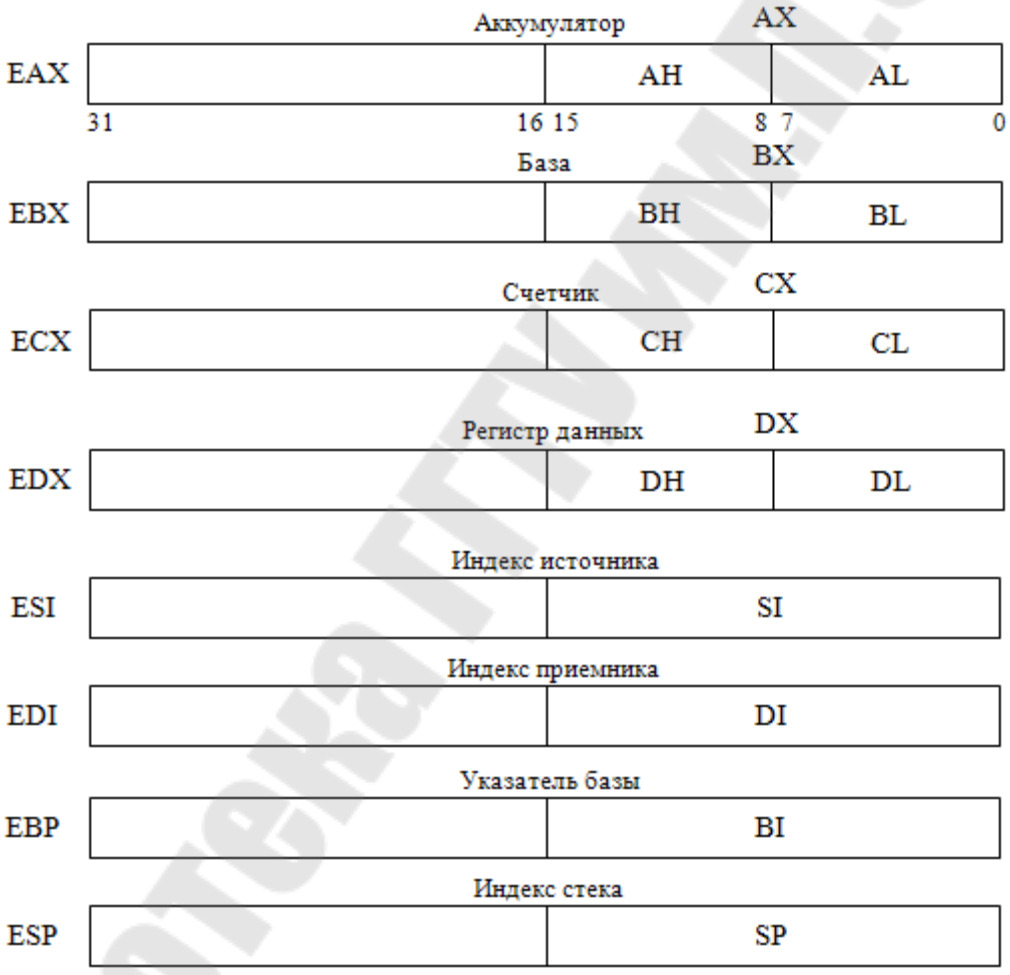

Рис. 1.2. Регистры общего назначения

Сегментные регистры. При использовании сегментированных моделей памяти для формирования любого адреса нужны два числа – адрес начала сегмента и смещение искомого байта относительно этого начала (в бессегментной модели памяти flat адреса начала всех сегментов равны). Операционные системы (кроме DOS) могут разме-

щать сегменты, с которыми работает программа пользователя, в разных местах памяти и даже временно записывать их на диск, если памяти не хватает. Так как сегменты способны оказаться где угодно, программа обращается к ним, применяя вместо настоящего адреса начала сегмента 16-битное число, называемое селектором. В процессорах Intel предусмотрены шесть 16-битных регистров – CS, DS, ES, FS, GS, SS, где хранятся селекторы. Это означает, что в любой момент можно изменить параметры, записанные в этих регистрах.

В отличие от DS, ES, GS, FS, которые называются регистрами сегментов данных, CS и SS отвечают за сегменты двух особенных типов – сегмент кода и сегмент стека. Первый содержит программу, исполняющуюся в данный момент, следовательно, запись нового селектора в этот регистр приводит к тому, что далее будет исполнена не следующая по тексту программы команда, а команда из кода, находящегося в другом сегменте, с тем же смещением. Смещение очередной выполняемой команды всегда хранится в специальном регистре EIP (указатель инструкции, 16-битная форма IP), запись в который также приведет к тому, что далее будет исполнена какая-нибудь другая команда. На самом деле все команды передачи управления – перехода, условного перехода, цикла, вызова подпрограммы и т. п. – и осуществляют эту самую запись в CS и EIP.

Стек. Стек – организованный специальным образом участок памяти, который используется для временного хранения переменных, передачи параметров вызываемым подпрограммам и сохранения адреса возврата при вызове процедур и прерываний. Легче всего представить стек в виде стопки листов бумаги (это одно из значений слова «stack» в английском языке) – Вы можете класть и забирать листы только с вершины стопки. Поэтому, если записать в стек числа 1, 2, 3, то при чтении они окажутся в обратном порядке – 3, 2, 1. Стек располагается в сегменте памяти, описываемом регистром SS, и текущее смещение вершины стека отражено в регистре ESP, причем во время записи значение этого смещения уменьшается, т. е. он «растет вниз» от максимально возможного адреса (см. рис. 1.3). Такое расположение стека «вверх ногами» может быть необходимым, к примеру, в бессегментной модели памяти, когда все сегменты, включая сегменты стека и кода, занимают одну и ту же область – память целиком. Тогда программа исполняется в нижней области памяти, в области малых адресов, и EIP растет, а стек располагается в верхней области памяти, и ESP уменьшается. При вызове подпрограммы параметры в большинстве случаев помешают в стек, а в EBP

записывают текущее значение ESP. Если подпрограмма использует стек для хранения локальных переменных, ESP изменится, но EBP можно будет использовать для того, чтобы считывать значения параметров напрямую из стека (их смещения запишутся как EBP + номер параметра).

Рис. 1.3. Стек

Регистр флагов. Еще один важный регистр, использующийся при выполнении большинства команд, – регистр флагов. Его младшие 16 бит, представлявшие собой весь этот регистр до процессора 80386, называются FLAGS. В E FLAGS каждый бит является флагом, т. е. устанавливается в 1 при определенных условиях или установка его в 1 изменяет поведение процессора. Все флаги, расположенные в старшем слове регистра, имеют отношение к управлению защищенным режимом, поэтому будем рассматривать только регистр FLAGS (см. рис. 1.4.):

– CF – флаг переноса. Устанавливается в 1, если результат предыдущей операции не уместился в приемнике и произошел перенос старшего бита, или, если требуется заем (при вычитании), в противном случае – в 0. Например, после сложения слова 0FFFFh и 1, если регистр, в который надо поместить результат, – слово, в него будет записано 0000h и флаг CF=1;

– PF – флаг четности. Устанавливается в 1, если младший байт результата предыдущей команды содержит четное число битов, равных 1, и в 0, если нечетное. Это не тоже самое, что делимость на два.

Число делится на 2 без остатка, если его самый младший бит равен нулю, и не делится, когда он равен 1;

– AF – флаг полупереноса или вспомогательного переноса. Устанавливается в 1, если в результате предыдущей операции произошел перенос (или заем) из третьего бита в четвертый. Этот флаг используется автоматически командами двоично-десятичной коррекции;

– ZF – флаг нуля. Устанавливается в 1, если результат предыдущей команды – ноль;

– TF – флаг ловушки. Он был предусмотрен для работы отладчиков, не использующих защищенный режим. Установка его в 1 приводит к тому, что после выполнения каждой программной команды управление временно передается отладчику (вызывается прерывание 1 – описание команды INT);

– IF – флаг прерываний. Сброс этого флага приводит к тому, что процессор перестает обрабатывать прерывания от внешних устройств (описание команды INT). Обычно его сбрасывают на короткое время для выполнения критических участков кода;

– DF – флаг направления. Он контролирует поведения команд обработки строк: когда он установлен в 1, строки обрабатываются в сторону уменьшения адресов, когда DF=0 – наоборот;

– OF – флаг переполнения. Он устанавливается в 1, если результат предыдущей арифметической операции над числами со знаком выходит за допустимые для них пределы. Например, если при сложении двух положительных чисел получается число со старшим битом, равным единице, т. е. отрицательное, и наоборот.

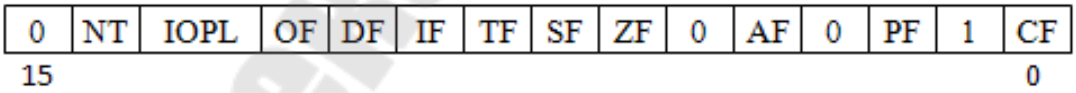

Рис. 1.4. Регистр флагов

Флаги IOPL (уровень привилегий ввода-вывода) и NT (вложенная задача) применяются в защищенном режиме.

1.5. Работа с командой прерываний INT

Особым видом являются подпрограммы, вызываемые по прерыванию командой INT. Прерывания бывают двух видов:

– аппаратные – это сигнал от любого устройства системы для процессора, который по этому сигналу должен обслужить данное устройство;

– программные – создается программами BIOS или DOS для вызова сервисных подпрограмм.

Команда прерывания

INT *number*

вызывает подпрограммы операционной системы. Эти прерывания имеют номера от 0 до FFh. Перед вызовом команды INT в регистр AX помещают номер функции, который определяет необходимую подпрограмму. Другие регистры тоже могут использоваться в прерывании.

Процессор выполняет команду прерывания, используя таблицу векторов прерываний. Таблица векторов прерываний занимает самые нижние 1024 байта памяти. Каждый элемент таблицы – это 32-битный адрес «сегмент: смещение», указывающий на подпрограммы операционной системы. Когда происходит вызов команды INT процессор производит следующие действия:

– параметр number сообщает процессору местонахождение вектора в таблице векторов прерываний;

– процессор передает управление по указанному в векторе прерываний адресу процедуры обработки прерывания (ПОП);

– подпрограмма DOS или BIOS, расположенная по указанному адресу, начинает выполняться и передает управление назад, когда будет достигнута команда IRET.

Команда возврата из прерывания IRET отдает управление вызывающей программе, которая продолжает работу со следующей за прерыванием командой.

1.6. Практическая часть

Задание. Написать 2 программы, которые создают файлы типа COM и EXE, в которых необходимо вывести на экран свои Фамилию, Имя и Отчество, факультет и номер группы.

Глава 2. Программирование линейных алгоритмов

2.1. Директивы определения данных

В общем случае все директивы объявления данных имеют такой синтаксис:

[имя] директива dup_выражение [,dup_выражение]

Синтаксис параметра dup_выражение может быть следующим:

? (неинициализированные данные)
значение (значение элемента данных)
количество_повторов DUP(dup_выражение [,dup_выражение]) (объявление и инициализация массивов)

К директивам объявления и инициализации простых данных относится следующее.

DB (Define Byte) – определить байт.

Директивой db можно задавать следующие значения:

– выражение или константу, принимающую значение из диапазона: для чисел со знаком –128...+127; для чисел без знака 0...255;

– 8-битовое относительное выражение, использующее операции HIGH и LOW;

– символьную строку из одного или более символов. Строка заключается в кавычки. В этом случае определяется столько байт, сколько символов в строке.

DW (Define Word) – определить слово.

Директивой dw можно задавать следующие значения:

– выражение или константу, принимающую значение из диапазона: для чисел со знаком –32768...+32767; для чисел без знака 0...65535;

– 1- или 2-байтовая строка, заключенная в кавычки.

DD (Define Double word) – определить двойное слово.

Директивой dd можно задавать следующие значения:

– выражение или константу, принимающую значение из диапазона: для чисел со знаком –2147483648...+2147483647; для чисел без знака 0...4 294 967 295;

– относительное или адресное выражение, состоящее из 16-битового адреса сегмента и 16-битового смещения;

– строку длиной до 4 символов, заключенную в кавычки.

DQ (Define Quarter word) – определить учетверенное слово.

DF (Define Far word) – определить указатель дальнего слова.

DP (Define Pointer) /I определить указатель 48 бит.

DT (Define Ten Bytes) – определить 10 байт.

Для резервирования памяти под массивы используется директива DUP:

```
area   dw 10 dup(?)      ;резервируется память объемом  10 слов
string db   20 dup('*')  ;строка заполняется кодом символа '*'
array  dw   3 dup(8)     ;массив из 3 слов инициализ. числом 8
t      db   4 dup(5 dup(8))   ;20 восьмерок
```

В Turbo Assembler есть две директивы, которые можно использовать для присвоения значения идентификатору: EQU и "=". При использовании директивы EQU можно присваивать идентификаторам как строковые, так числовые значения:

имя EQU выражение

Здесь «выражение» может быть псевдонимом, числом или строкой:

```
num        equ   2*5          ;число
str        equ   'hello'      ;строка
```

Переопределение идентификатора с помощью другой директивы EQU допускается только в том случае, если раньше ему было присвоено строковое значение.

Пример

Листинг 1. Пример применения директив определения данных:

```
;определение байта
v1      db   ?              ;не инициализировано
v2      db   'ИП21'         ;символьная строка
v3      db   6              ;десятичная константа
v4      db   0afh           ;шестнадцатиричная константа
v5      db   0110100b         ;двоичная константа
;определение слова
w1      dw   bff3h          ;шестнадцатеричная константа
w2      dw   11101111b          ;двоичная константа
w3      dw   2,24,5,7           ;4 константы
w4      dw   23 dup(*)          ;23 звездочки
;определение двойного слова
d1      dd     ?                ;не определено
d2      dd     'uAudf'          ;символьная строка
```

```
d3      dd      08734           ;десятичная константа
d4      dd      087h,85fh       ;две константы
;определение учетверенного слова
v1dq    dq      ?               ;не определено
v2dq    dq      a83dh           ;константа
```

2.2. Команды передачи данных

Команда MOV копирует данные из операнда-источника в операнд-получатель. Она от носится к группе команд пересылки данных (data transfer) и используется в любой программе. Команда MOV является двуместной (т. е. имеет два операнда): первый операнд оп ределяетполучателя данных (destination), а второй – источник данных (source):

MOV получатель, источник

При выполнении этой команды изменяется содержимое операнда-получателя, а содержимое операнда-источника не меняется. Принцип пересылки данных справа налево соответствует принятому в операторах присваивания языков высокого уровня, таких как C++ или Java:

получатель = источник;

Практически во всех командах ассемблера операнд-получатель находится слева, а операнд-источник – справа.

В команде MOV могут использоваться самые разные операнды. Кроме того, необходимо учитывать следующие правила и ограничения.

Оба операнда должны иметь одинаковую длину.

В качестве одного из операндов обязательно должен использоваться регистр (т. е. пересылки типа «память-память» в команде MOV не поддерживаются).

В качестве получателя нельзя указывать регистры CS, EIP и IP.

Нельзя переслать непосредственно заданное значение в сегментный регистр.

Команда MOVZX (Move With Zero-Extend, или переместить и дополнить нулями) копирует содержимое исходного операнда в больший по размеру регистр получателя данных. При этом оставшиеся неопределенными биты регистра-получателя (как правило, старшие 16 или 24 бита) сбрасываются в ноль. Эта команда используется только при работе с беззнаковыми целыми числами.

В качестве операнда-получателя может быть задан только 16- или 32-разрядный регистр. На рис. 2.1 показано, как 8-разрядный исходный операнд загружается с помощью команды MOVZX в 16-разрядный регистр.

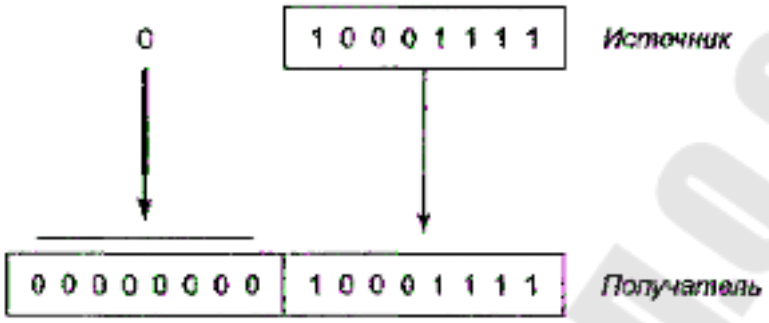

Рис. 2.1. Иллюстрация работы команды MOVZX

В приведенном ниже примере используются все три варианта команды MOVZX с разными размерами операндов:

```
MOV     bx,0a69bh
MOVZX   eax, bx          ; eax=0000a69bh
MOVZX   edx, bl          ;edx=0000009bh
MOVZX   cx, bl           ;cx=009bh
```

Команда MOVSX (Move With Sign-Extend, или переместить и дополнить знаком) копирует содержимое исходного операнда в больший по размеру регистр получателя данных, также как и команда MOVZX. При этом оставшиеся неопределенными биты регистра-получателя (как правило, старшие 16 или 24 бита) заполняются значением знакового бита исходного операнда. Эта команда используется только при работе со знаковыми целыми числами.

При загрузке меньшего по размеру операнда в больший по размеру регистр с помощью команды MOVSX, знаковый разряд исходного операнда дублируется (т. е. переносится или расширяется) во все старшие биты регистра-получателя. Например, при загрузке 8-разрядного значения 10001111b в 16-разрядный регистр, оно будет помещено в младшие 8 битов этого регистра. Затем, как показано на рис. 2.2, старший бит исходного операнда переносится во все старшие разряды регистра-получателя.

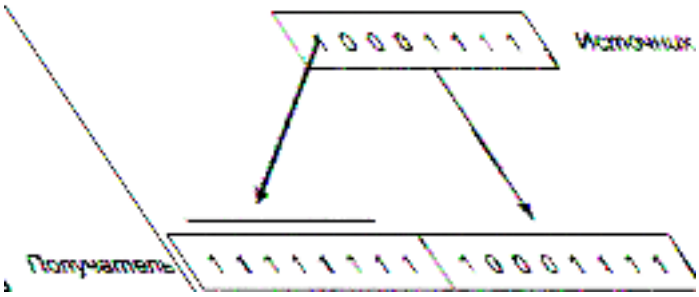

Рис. 2.2. Иллюстрация работы команды MOVSX

В приведенном ниже примере используются все три варианта команды MOVSX с разными размерами операндов:

```
MOV     bx,0a69bh
MOVSX   eax, bx          ; eax=ffffa69bh
MOVSX   edx, bl          ;edx=ffffff9bh
MOVSX   cx, bl           ;cx=ff9bh
```

2.3. Арифметические команды

Микропроцессоры семьи INTEL выполняют четыре основных математических действия с модификациями над 8-, 16- и 32-разрядными операндами по правилам арифметики с учетом знака и без его учета. Данные со знаком представляются в дополнительном коде. Операции выполняются над двоичными, а также упакованными и неупакованными двоично-десятичными числами. При реализации арифметических операций устанавливаются шесть флажков (шесть признаков результата): CF, AF, ZF, SF, PF, OF.

Команды сложения языка Ассемблер даны в табл. 1.

Таблица 1

Команды сложения языка Ассемблер

Действие	Мнемоника и формат	Описание
Сложить	ADD R1, R2	(R1)=(R1)+(R2)
Сложить с учетом переноса	ADC R1, R2	(R1)=(R1)+(R2)+(CF)
Инкрементирование	INC OPR	OPR=(OPR)+1

В таблице приняты такие сокращения:
R1 – операнд-приемник;
R2 – операнд-источник;
OPR – операнд.

Для сложения двух чисел, представленных различными типами данных, требуется выполнить приведение числа с наименьшим количеством бит к числу с наибольшим количеством бит. Для реализации алгоритма приведения можно использовать несколько вариантов составления программы. Один из них основывается на использовании оператора преобразования типа ptr (PTR – операция назначения типа mov bx, WORD PTR [100H]), а второй – на свойстве регистров микропроцессора. Регистры, как известно, могут быть представлены либо в двухбайтном виде (ax), либо в виде старшего байта (ah) и младшего байта (al). Следовательно, можно для преобразования числа из однобайтного в двухбайтное записать в регистр (al) значение однобайтного числа, а в регистр (ah) – нулевое значение. В итоге регистр (ax) будет содержать результат преобразования.

Команды вычитания языка Ассемблер даны в табл. 2.

Таблица 2

Команды вычитания языка Ассемблер

Действие	Мнемоника и формат	Описание
Вычесть	SUB R1, R2	(R1)=(R1)-(R2)
Вычесть с заемом	SBB R1, R2	(R1)=(R1)-(R2)-(CF)
Декрементирование	DEC OPR	OPR=(OPR)-1
Заменить знак	NEG OPR	OPR=инверсия(OPR) +1
Сравнить	CMP OPR 1, OPR 2	(OPR1)-(OPR2)

Команда NEG выполняет дополнение операнда до 2.

Команда CMP выполняет вычитание, устанавливает признаки результата, но не запоминает результат.

Команды умножения языка Ассемблер даны в табл. 3.

Таблица 3

Команды умножения языка Ассемблер

Действие	Мнемоника и формат	Описание
Умножить с учетом знака	IMUL R2	Операнды – байты: (AX)=(AL)*(R2) Операнды – слова: (DX:AX)=(AX)*(R2)
Умножить без учета знака	MUL R2	Аналогично команде IMUL, но операнды и произведение беззнаковые

Команда MUL умножает без знака содержимое аккумулятора (AL или AX) и операнда и размещает результат двойной длины в аккумуляторе с его распространением (AL и AH для 8-разрядных операндов, AX и – DX для 16-разрядных). Флажки CF и OF устанавливаются в 1, если старшая половина результата не равняется 0.

Команда IMUL подобна MUL, но выполняет умножение чисел со знаком. Флажки CF и OF устанавливаются, если старшая половина результата не является расширением знакового бита младшей половины.

Команды деления языка Ассемблер даны в табл. 4.

Таблица 4

Команды деления языка Ассемблер

Действие	Мнемоника и формат	Описание
Разделить со знаком	IDIV R2	Делитель – байт: (AL)=частное (AX)/(R2) (AH)=остаток (AX)/(R2) Делитель – слово: (AX)=частное (DX:AX)/(R2), (DX)=остаток (DX:AX)/(R2), Частное и остаток имеют знаки, знак остатка равняется знаку делимого
Разделить без знака	DIV R2	Команда аналогична команде IDIV, но операнды, частное и остаток беззнаковые
Превратить байт в слово	CBW	Расширяет знак AL в AH
Превратить слово в двойное слово	CWD	Расширяет знак AX в DX

Команда DIV реализует операцию деления без знака содержимого аккумулятора и его расширения (AL и AH для 8-разрядных операндов, AX и DX – для 16-разрядных) на операнд и размещает частное в аккумуляторе (AL или AX) и остаток – в расширении аккумулятора (AH или DX). Флажки не формируются, а деление на 0 дает прерывание с вектором 0. Команда IDIV работает аналогично, но учитывает знак.

2.4. Практическая часть

Задание. Написать программу для вычисления арифметического выражения.

Вычисление арифметического выражения дано в в табл. 5.

Таблица 5

Вычисление арифметического выражения

№	Функция	№	Функция
1	y =5*K+L/J+J^2-100	16	D=(F-3*X)/(F-X^2)+F*X
2	Z=(K^4-K^3+3*L)/M	17	P=(M*V-H)/(H^3-J)+4*M
3	C=F+K^3-(5*B)/(F^2)	18	K=(5*Y-4*Y^2)/(J+2*B)+Y
4	G=8*H+(U/I-4*Y)/K	19	T=(F-4*X)/(X^3+F)-(F-X)^2
5	L=T/(P+6*Y)+P^3-5*Y	20	G=(J-4*I)/(U-8)*U^3-5*I
6	J=(5*A*S-7)/(N+6*M)	21	A=C^3+(B-C)/(D+B)-5*D
7	H=8*G/(J-10)+(H+4*J)	22	H=G^4+T/(G-T)+(G+T)^2
8	F=(B^4-H^3)/(X^2)-(B-H)/X	23	y =-K+L/(J+3*J^2)-L^3
9	D=(F+X)/(3*F-X)+F^3	24	Z=(L^2-L*K^3)/M-L*K
10	P=4*V-H/(H-6*J)+2*M	25	C=-F^2+(5*K-B)/(K+F^2)
11	K=2*Y^2-3*Y^3/(J-4*B)	26	G=(H+U/I)-4*Y/(K+2*H)
12	T=(F+3*X)/(X-F)-(F+X)^2	27	L=T*(P-Y)+P/(T^3-2*Y)
13	G=(J+6*I)/U-5*U^3-I	28	J=A/S-(N-6*A)/(2*S^2-A)
14	A=3*C-(B+4*C)/(D-B)+D^3	29	H=G*(J-3)+H/(4*J-G^2)
15	H=5*G-4*T/(G+3*T)+G^2	30	F=B*H*(X^2)-B*H/X+B^3

Пример программного кода для вычисления выражений:

X=(A^2-B^3+3*C)/(2*A^2 – C) Y=(B^3 – X/4)/A

Листинг программы:

```
; arifm-2.asm
; Основные арифметические операции
   .model   small      ; модель памяти, используемая для EXE
   .stack   100h       ; сегмент стека размером в 256 байт
   .data
        a    db    -3
 b    db     5
        c    db     2
        x    db     ?
        y    db     ?
   .code
    main:   mov    ax,data    ; сегментный адрес данных
```

```
    mov    ds,ax        ; помещается в DS
      mov     al,a
      imul    al      ;al=a^2
      mov     bh,al   ;a^2->bh
      mov     al,b
      imul    al      ;al=b^2
      imul    b       ;al=b^3
      mov     bl,al   ;b^3->bl
      mov     al,3h
      imul    c       ;al=3*c
      add     al,bh   ;a^2+3c
      sub     al,bl   ;a^2-b^3+3c
      mov     ch,al
      mov     al,2
      imul    bh      ;2*a^2
      sub     al,c
      mov     cl,al   ;2*a^2-c
      mov     al,ch   ;\
      cbw             ; в ax – числитель
      idiv    cl
      mov     x,al
;закончено вычисление 1-й формулы
      mov     cl,4h
      cbw
      idiv    cl     ;x/4
      neg     al     ;-x/4
      add     al,bl  ;b^3-x/4
      cbw
      idiv    a
      mov     y,al
    mov    ax,4C00h
    int      21h        ; функция DOS "завершить программу"
    end    main
```

Глава 3. Программирование разветвляющихся алгоритмов

3.1. Команды сравнения данных

Команда CMP вычитает исходный операнд из операнда получателя данных и в зависимости от полученного результата устанавливает флаги состояния процессора. При этом в отличие от команды SUB значение операнда получателя данных не изменяется.

CMP получатель, источник

В команде CMP используются аналогичные команде AND типы операндов.

Флаги. Команда CMP изменяет состояние следующих флагов (табл. 6): CF (флаг переноса); ZF (флаг нуля); SF (флаг знака): OF (флаг переполнения); AF (флаг служебного переноса); PF (флаг четности). Они устанавливаются в зависимости от значения, которое было бы получено в результате применения команды SUB. Например, как показано в табл. 6, после выполнения команды CMP по состоянию флагов нуля (ZF) и переноса (CF) можно судить о величинах сравниваемых между собой беззнаковых операндов.

Таблица 6

Состояние флагов после сравнения беззнаковых операндов с помощью команды CMP

Значение операндов	ZF	CF
получатель < источник	0	1
получатель > источник	0	0
получатель = источник	1	0

Если сравниваются два операнда со знаком, то, кроме флагов ZF и CF, нужно учитывать еще и флаг знака (SF), как показано в табл. 7.

Таблица 7

Состояние флагов после сравнения операндов со знаком с помощью команды CMP

Значение операндов	Состояние флагов
получатель < источник	SF ≠ OF и ZF = 0
получатель > источник	SF = OF и ZF = 0
получатель = источник	ZF = 1

Команда CMP очень важна, поскольку она используется практически во всех основных условных логических конструкциях. Если после команды CMP поместить команду условного перехода, то полученная конструкция на языке ассемблера будет аналогична оператору IF языка высокого уровня.

Пример. При сравнении числа 5, находящегося в регистре EAX, с числом 10, устанавливается флаг переноса CF, поскольку при вычитании числа 10 из 5 происходит заем единицы:

```
MOV   eax, 5
CMP   eax, 10                ; CF=1
```

При сравнении содержимого регистров eax и ecx, в которых содержатся одинаковые числа 15, устанавливается флаг нуля (ZF), так как в результате вычитания этих чисел получается нулевое значение:

```
MOV   eax, 15
MOV   ecx, 15
CMP   eax, ecx               ; ZF=1
```

3.2. Булевы операции

Операции И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и НЕ положены в основу работы логических схем компьютера, а также его программного обеспечения.

В системе команд процессоров семейства IA-32 предусмотрены команды AND, OR, XOR, NOT, TEST и BT, выполняющие перечисленные выше булевы операции между байтами, словами и двойными словами (табл. 8).

Таблица 8

Логические команды процессора

Команда	Описание
AND	Выполняет операцию логического И между двумя операндами
OR	Выполняет операцию логического ИЛИ между двумя операндами
XOR	Выполняет операцию исключающего ИЛИ между двумя операндами
NOT	Выполняет операцию логического отрицания (НЕ) единственного операнда

Окончание табл. 8

Команда	Описание
TEST	Выполняет операцию логического И между двумя операндами, устанавливает соответствующие флаги состояния процессора, но результат операции не записывается вместо операнда получателя данных
BT, BTC, BTR, BTS	Копирует бит операнда получателя, номер *n* которого задан в исходном операнде, во флаг переноса (CF), а затем, в зависимости от команды, тестирует, инвертирует, сбрасывает или устанавливает этот же бит операнда получателя

Команда AND. Команда AND выполняет операцию логического И между соответствующими парами битов операндов команды и помещает результат на место операнда получателя данных:

AND получатель, источник

Команда AND может работать с 8-, 16- или 32-разрядными операндами, причем длина у обоих операндов должна быть одинаковой. При выполнении операции поразрядного логического И значение результата будет равно 1 только в том случае, если оба бита пары равны 1. В табл. 9 приведена таблица истинности для операции логического И.

Таблица 9

Таблица истинности для операции логического И.

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

Команда AND обычно используется для сброса отдельных битов двоичного числа (например, флагов состояния процессора) по заданной маске. Если бит маски равен 1, значение соответствующего разряда числа не изменяется (в этом случае говорят, что разряд замаскирован), а если равен 0 – то сбрасывается. В качестве примера на рис. 3.1 показано, как можно сбросить четыре старших бита 8-разрядного двоичного числа. Для выполнения этой операции можно воспользоваться двумя командами:

```
MOV al, 00111011b
AND  al,00001111b
```

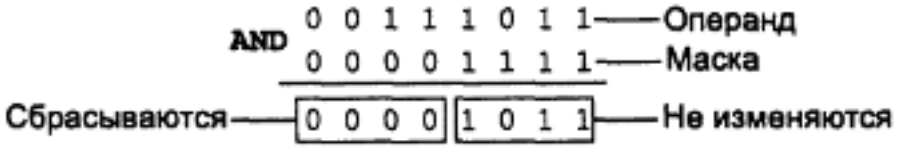

Рис. 3.1. Сброс битов по маске с помощью команды AND

Команда AND всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Команда OR. Команда OR выполняет операцию логического ИЛИ между соответствующими парами битов операндов команды и помещает результат на место операнда получателя данных:

OR получатель, источник

Команда OR может работать с 8-, 16- или 32-разрядными операндами, причем длина у обоих операндов должна быть одинаковой. При выполнении операции поразрядного логического ИЛИ значение результата будет равно 1, если хотя бы один из битов пары операндов равен 1. В табл. 10 приведена таблица истинности для операции логического ИЛИ.

Таблица 10

Таблица истинности для операции логического ИЛИ

X	***Y***	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

Команда OR обычно используется для установки в единицу отдельных битов двоичного числа (например, флагов состояния процессора) по заданной маске. Если бит маски равен 0, значение соответствующего разряда числа не изменяется, а если равен 1 – то устанавливается в 1. В качестве примера на рис. 3.2 показано, как можно установить четыре младших бита 8-разрядного двоичного числа, выбрав в качестве маски число 0Fh. Значение старших битов числа при этом не меняется.

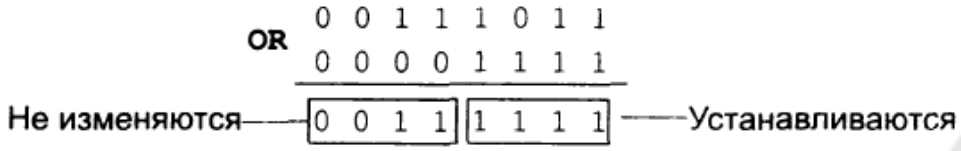

Рис. 3.2. Установка битов по маске с помощью команды OR

С помощью команды OR можно преобразовать двоичное число, значение которого находится в диапазоне от 0 до 9 в ASCII-строке. Для этого нужно установить в единицу биты 4 и 5. Например, если в регистре AL находится число 05h, то чтобы преобразовать его в соответствующий ASCII-код, нужно выполнить операцию OR регистра AL с числом 30h. В результате получится число 35h, которое соответствует ASCII-коду цифры 5 (рис. 3.3).

```
    0 0 0 0 0 1 0 1   05h
OR  0 0 1 1 0 0 0 0   30h
    -----------------
    0 0 1 1 0 1 0 1   35h, '5'
```

Рис. 3.3. Преобразование двоичного числа в ASCII-код
с помощью команды OR

На языке ассемблера подобное преобразование можно записать так:

```
MOV   dl,  6
OR    dl, 30h                ;ASCII – код для числа 6
```

Команда OR всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата. Например, с помощью команды OR можно определить, какое значение находится в регистре (отрицательное, положительное или нуль). Для этого вначале нужно выполнить команду OR, указав в качестве операндов один и тот же регистр, например:

```
OR    dl, dl
```

а затем – проанализировать значения флагов, как показано в табл. 11.

Таблица 11

Определение значения числа по флагам состояния процессора

Флаг нуля (ZF)	Флаг знака (SF)	Значение числа
0	0	Больше нуля
1	0	Равно нулю
0	1	Меньше нуля

Команда XOR. Команда XOR выполняет операцию ИСКЛЮЧАЮЩЕГО ИЛИ между соответствующими парами битов операндов команды и помещает результат на место операнда получателя данных:

XOR получатель, источник

Команда XOR может работать с 8-, 16- или 32-разрядными операндами, причем длина у обоих операндов должна быть одинаковой. При выполнении операции поразрядного ИСКЛЮЧАЮЩЕГО ИЛИ значение результата будет равно 1, если значения битов пары операндов различны, и 0 – если значения битов равны. В табл. 12 приведена таблица истинности для операции логического ИСКЛЮЧАЮЩЕГО ИЛИ.

Таблица 12

Таблица истинности для операции ИСКЛЮЧАЮЩЕГО ИЛИ

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	1

Как следует из таблицы, при выполнении операции ИСКЛЮЧАЮЩЕГО ИЛИ с нулевым битом получается исходное значение бита, а с единичным битом – значение исходного бита инвертируется. Операция ИСКЛЮЧАЮЩЕГО ИЛИ обладает свойством реверсивности – если ее выполнить дважды с одним и тем же операндом, то значение результата инвертируется. Как показано в табл. 13, если два раза подряд выполнить операцию ИСКЛЮЧАЮЩЕГО ИЛИ между битами X и Y, то в результате получится исходное значение бита X.

Таблица 13

Демонстрация свойства реверсивности операции ИСКЛЮЧАЮЩЕГО ИЛИ

X	Y	$X \oplus Y$	$(X \oplus Y) \oplus Y$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Команда XOR всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Команда NOT. Команда NOT позволяет выполнить инверсию всех битов операнда, в результате чего получается обратный код числа.

Например, обратный код числа F0h равен 0Fh:

```
MOV   dl,  11110000b
NOT   dl                          ;dl = 00001111b
```

Команда NOT не изменяет флаги процессора.

Команда TEST. Команда TEST выполняет операцию поразрядного логического И между соответствующими парами битов операндов и в зависимости от полученного результата устанавливает флаги состояния процессора. При этом, в отличие от команды AND, значение операнда получателя данных не изменяется. В команде TEST используются аналогичные команде AND типы операндов. Обычно команда TEST применяется для анализа значения отдельных битов числа по маске.

Пример. Тестирование нескольких битов. С помощью команды TEST можно определить состояние сразу нескольких битов числа. Предположим, мы хотим узнать, установлен ли нулевой и третий биты регистра AL. Для этого можно воспользоваться такой командой:

```
TESTt   al,00001001b    ; Тестируем биты 0 и 3
```

Команда TEST всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата выполнения операции логического И (как и команда AND).

Установка и сброс отдельных флагов состояния процессора. Для установки или сброса флагов нуля (ZF), знака (SF), переноса (CF)

и переполнения (OF)существуют несколько способов, и большинство из них изменяет значение операнда получателя данных. Чтобы установить флаг нуля (ZF), необходимо выполнить команду AND с нулевым операндом, а для сброса этого флага – команду OR с единичным операндом, как показано ниже:

```
AND   al, 0          ;ZF устанавливается
OR    al, 1          ;ZF сбрасывается
```

Для установки флага знака (SF) выполните команду OR, у которой старший бит операнда установлен в 1, а для сброса этого флага – команду AND, у которой старший бит операнда сброшен в 0, как показано ниже:

```
AND   al, 7fh        ;SF сбрасывается
OR    al, 80h        ;SF устанавливается
```

Для установки и сброса флага переноса (CF) предусмотрены специальные команды: STC (SeT Carry flag) и CLC (CLear Carry flag):

```
clc                  ;CF сбрасывается
stc                  ;CF устанавливается
```

Чтобы установить флаг переполнения (OF), нужно сложить два положительных числа так, чтобы в результате получилась отрицательная сумма. Для сброса этого флага достаточно выполнить команду OR с нулевым операндом, как показано ниже:

```
MOV   al, 7fh        ;al = +127
INC   al             ; al = 80h (-128),  OF = 1
OR    al, 0          ;OF сбрасывается
```

3.3. Команды условного перехода

В системе команд процессоров IA-32 не предусмотрена поддержка условных логических структур, характерных для языков высокого уровня. Однако на языке ассемблера с помощью набора команд сравнения и условного перехода можно реализовать логическую структуру любой сложности. В языке высокого уровня любой условный оператор выполняется в два этапа. Сначала вычисляется значение условного выражения, а затем, в зависимости от его результата, выполняются те или иные действия. Проводя аналогию с языком ассемблера, можно сказать, что сначала выполняются такие команды, как CMP, AND или SUB, влияющие на флаги состояния процессора.

Затем выполняется команда условного перехода, которая анализирует значение нужных флагов и, в случае если они установлены, выполняет переход по указанному адресу.

Пример. С помощью команды CMP значение регистра AL сравнивается с нулем. Команда JZ (Jump if Zero, или переход, если ноль) передает управление по метке L1, если в результате выполнения команды CMP был установлен флаг ZF:

```
CMP   al, 0
JZ    Metka                ;переход если ZF=1
……..
……..
Metka:
```

Команда Jcond. Команда условного перехода передает управление по указанной метке в случае, если установлен соответствующий флаг состояния процессора. Если флаг сброшен, то выполняется следующая за ней команда. Синтаксис команд условного перехода следующий:

Jcond метка_перехода

Здесь вместо параметра cond нужно подставить аббревиатуру нужного условия, которая будет определять состояние одного или нескольких флагов состояния процессора.

При выполнении любой команды условного перехода процессор вначале проверяет состояние соответствующих флагов регистра EFLAGS. Если он обнаруживает, что флаги, соответствующие данному условию, установлены, выполняется переход по указанной метке. В противном случае управление передается команде, следующей за командой условного перехода.

Типы команд условного перехода. В системе команд предусмотрено много типов команд условного перехода. Удобно разбить весь этот набор команд на четыре группы:

– выполняющие переход в зависимости от значения флагов состояния процессора;

– выполняющие переход в зависимости от равенства операндов или равенства нулю регистра ECX (CX);

– использующиеся после команд сравнения беззнаковых операндов;

– использующиеся после команд сравнения операндов со знаком.

В табл. 14 перечислены команды первой группы, выполняющие переход в зависимости от значения флагов состояния процессора: ZF, CF, OF, PF и SF.

Таблица 14

Команды, выполняющие переход в зависимости от значения флагов состояния процессора

Мнемоника	Описание	Состояние флагов
JZ	Переход, если нуль	ZF = 1
JNZ	Переход, если не нуль	ZF = 0
JC	Переход, если перенос	CF = 1
JNC	Переход, если нет переноса	CF = 0
JO	Переход, если переполнение	OF = 1
JNO	Переход, если нет переполнения	OF = 0
JS	Переход, если флаг знака установлен	SF = 1
JNS	Переход, если флаг знака сброшен	SF = 0
JP	Переход, если флаг честности установлен	PF = 1
JNP	Переход, если флаг честности сброшен	PF = 0

Сравнение на равенство. В табл. 15 перечислены команды, выполняющие переход в зависимости от равенства операндов или равенства нулю регистра ECX (CX). В таблице через *левоп* и *правоп* обозначим, соответственно, левый (получатель данных) и правый (источник данных) операнды команды CMP:

CMP левоп,правоп

Подобное название операндов отражает порядок их следования в операторах отношения, использующихся в алгебраических выражениях. Например, в выражении X < Y, X называется левым операндом (левоп), а Y – правым (правоп).

Таблица 15

Команды, выполняющие переход в зависимости от равенства операндов

Мнемоника	Описание	Состояние флагов
JE	Переход, если равны (*ЛЕВОП = ПРАВОП*)	ZF = 1
JNE	Переход, если не равны (*ЛЕВОП ≠ ПРАВОП*)	ZF = 0
JCXZ	Переход, если регистр CX = 0	–
JECXZ	Переход, если регистр ECX = 0	–

Сравнение беззнаковых чисел. Команды условного перехода, использующиеся после команд сравнения беззнаковых операндов, перечислены в табл. 16. Этот тип команд условного перехода используется в случае, если нужно сравнить два беззнаковых числа, таких как 7Fh и 80h, и при этом предполагается, что первое число (7Fh) должно быть меньше второго (80h), а не наоборот, как в случае операндов со знаком.

Таблица 16

Команды условного перехода, использующиеся после команд сравнения беззнаковых операндов

Мнемоника	Описание	Состояние флагов
JA	Переход, если выше (*ЛЕВОП > ПРАВОП*)	CF = 0 и ZF = 0
JNBE	Переход, если не ниже или равно (синоним команды JA)	CF = 0 и ZF = 0
JAE	Переход, если выше или равно (*ЛЕВОП >= ПРАВОП*)	CF = 0 или ZF = 1
JNB	Переход, если не ниже (синоним команды JAE)	CF = 0 или ZF = 1
JB	Переход, если ниже (*ЛЕВОП < ПРАВОП*)	CF = 1 и ZF = 1
JNAE	Переход, если не выше или равно (синоним команды JB)	CF = 1 и ZF = 1
JBE	Переход, если ниже или равно (*ЛЕВОП <= ПРАВОП*)	CF = 1 или ZF = 1
JNA	Переход, если не выше (синоним команды JBE)	CF = 1 или ZF = 1

Сравнение чисел со знаком. Команды условного перехода, использующиеся после команд сравнения операндов со знаком, перечислены в табл. 17. Этот тип команд условного перехода используется в случае, если сравниваемые операнды должны интерпретироваться как числа со знаком. Например, при сравнении чисел 7Fh и 80h, результат будет зависеть от того, являются ли эти числа беззнаковыми или содержат знак. Поэтому команды JA и JG будут работать поразному.

```
MOV   al, 7fh          ; 7fh = +127
CVP   al, 80h          ;  80h может быть +128 или -128
JA    IsAbove          ; нет перехода ввиду +127  < +128
JG    IsGreater          ; переход ввиду +127 > -128
```

В этом примере команда JA не выполняет переход, так как беззнаковое число 7Fh меньше, чем беззнаковое число 8Oh. В то же время, команда JG выполняет переход, поскольку число +127 больше, чем –128.

Таблица 17

Команды условного перехода, использующиеся после команд сравнения операндов со знаком

Мнемоника	Описание	Состояние флагов
JG	Переход, если больше (*ЛЕВОП* > *ПРАВОП*)	SF = OF и ZF = 0
JNLE	Переход, если меньше или равно (синоним команды JG)	SF = OF и ZF = 0
JGE	Переход, если больше или равно (*ЛЕВОП* >= *ПРАВОП*)	SF = 0 или ZF = 1
JNL	Переход, если не меньше (синоним команды JGE)	SF = 0 или ZF = 1
JL	Переход, если меньше (*ЛЕВОП* < *ПРАВОП*)	SF ≠ OF и ZF = 0
JNGE	Переход, если не больше или равно (синоним команды JL)	SF ≠ OF и ZF = 0
JLE	Переход, если меньше или равно (*ЛЕВОП* <= *ПРАВОП*)	SF ≠ OF или ZF = 1
JNG	Переход, если не больше (синоним команды JBE)	SF ≠ OF или ZF = 1

3.4. Практическая часть

Задание. Программирование разветвляющихся алгоритмов.

Разработать программу для формирования функции $z(x,y)$. Подобрать набор тестов для проверки правильности работы программного фрагмента. Вычислить значения функции z для тестовых значений x и y. Индивидуальные задания приведены в табл. 18.

Таблица 18

Индивидуальные задания

№	Вид функции	№	Вид функции
1	$y=\begin{cases}6x, & \text{если } x>5\\ x, & \text{если } 1\le x\le 5\\ 4x^2 & \text{– в остальных случаях}\end{cases}$	2	$y=\begin{cases}3x^2, & \text{если } x>10\\ 3x, & \text{если } x\le 2\\ 4x & \text{– в остальных случаях}\end{cases}$

Продолжение табл. 18

№	Вид функции	№	Вид функции
3	$y=\begin{cases} x, & \text{если } x>20 \\ \dfrac{100}{x}, & \text{если } 1\le x\le 20 \\ x+2 & \text{– в остальных случаях} \end{cases}$	4	$y=\begin{cases} 2x, & \text{если } x>10 \\ 4x, & \text{если } 1\le x\le 10 \\ 6x & \text{– в остальных случаях} \end{cases}$
5	$y=\begin{cases} \dfrac{75}{x}, & \text{если } x>6 \\ 7x, & \text{если } 1\le x\le 6 \\ \dfrac{x}{5} & \text{– в остальных случаях} \end{cases}$	6	$y=\begin{cases} 5x, & \text{если } x>2 \\ \lvert x\rvert, & \text{если } -10\le x\le -3 \\ 2x^2 & \text{– в остальных случаях} \end{cases}$
7	$y=\begin{cases} 4x, & \text{если } x>12 \\ x+60, & \text{если } 5\le x\le 12 \\ 2x^4 & \text{– в остальных случаях} \end{cases}$	8	$y=\begin{cases} 7x, & \text{если } x>3 \\ \lvert x-9,\rvert & \text{если } x\le 0 \\ x^2+2 & \text{– в остальных случаях} \end{cases}$
9	$y=\begin{cases} x-90, & \text{если } x>10 \\ 6x, & \text{если } 1\le x\le 9 \\ x^2 & \text{– в остальных случаях} \end{cases}$	10	$y=\begin{cases} x^3, & \text{если } x>8 \\ 2x^2, & \text{если } x\le 0 \\ x+3 & \text{– в остальных случаях} \end{cases}$
11	$y=\begin{cases} 8x, & \text{если } x>5 \\ x^2, & \text{если } -5\le x\le 5 \\ \lvert x+7 & \text{– в остальных случаях} \end{cases}$	12	$y=\begin{cases} x+99, & \text{если } x>8 \\ \dfrac{50}{x}, & \text{если } 1\le x\le 8 \\ x^2 & \text{– в остальных случаях} \end{cases}$
13	$y=\begin{cases} 1-3x, & \text{если } x>0 \\ x-10, & \text{если } -5\le x\le 0 \\ x^2 & \text{– в остальных случаях} \end{cases}$	14	$y=\begin{cases} x^4, & \text{если } x<10 \\ x-20, & \text{если } x>15 \\ 5x & \text{– в остальных случаях} \end{cases}$
15	$y=\begin{cases} x-9, & \text{если } x>5 \\ x^2, & \text{если } 1\le x\le 5 \\ x^3 & \text{– в остальных случаях} \end{cases}$	16	$y=\begin{cases} 2x^2, & \text{если } x>8 \\ \lvert x-10\rvert, & \text{если } x\le 5 \\ x+7 & \text{– в остальных случаях} \end{cases}$
17	$y=\begin{cases} x-15, & \text{если } x>10 \\ \dfrac{1}{x}, & \text{если } 4\le x\le 10 \\ x-10 & \text{– в остальных случаях} \end{cases}$	18	$y=\begin{cases} 6x, & \text{если } x<10 \\ x-5, & \text{если } 10\le x\le 20 \\ \lvert x+5\rvert & \text{– в остальных случаях} \end{cases}$
19	$y=\begin{cases} \dfrac{88}{x}, & \text{если } x>6 \\ \lvert x-100\rvert, & \text{если } 1\le x\le 6 \\ \dfrac{x}{5} & \text{– в остальных случаях} \end{cases}$	20	$y=\begin{cases} x^4, & \text{если } x>4 \\ 2x^3, & \text{если } x\le 0 \\ \lvert x-50\rvert & \text{– в остальных случаях} \end{cases}$

Окончание табл. 18

№	Вид функции	№	Вид функции
21	$y=\begin{cases}8x, & \text{если } x>2\\ \lvert x+1\rvert, & \text{если } -10\le x\le -3\\ x^3 & \text{– в остальных случаях}\end{cases}$	22	$y=\begin{cases}x-77, & \text{если } x>3\\ \lvert x-4,\rvert & \text{если } x\le 0\\ x^2+2 & \text{– в остальных случаях}\end{cases}$
23	$y=\begin{cases}7x, & \text{если } x>10\\ \lvert x-68\rvert, & \text{если } 5\le x\le 10\\ 8-x^2 & \text{– в остальных случаях}\end{cases}$	24	$y=\begin{cases}x^3, & \text{если } x>3\\ 9x^2, & \text{если } x\le 0\\ x-6 & \text{– в остальных случаях}\end{cases}$
25	$y=\begin{cases}8x, & \text{если } x>3\\ x^2, & \text{если } 0\le x\le 3\\ \lvert x-7\rvert & \text{– в остальных случаях}\end{cases}$	26	$y=\begin{cases}2x, & \text{если } x>30\\ \dfrac{150}{x}, & \text{если } 1\le x\le 15\\ x^2 & \text{– в остальных случаях}\end{cases}$
27	$y=\begin{cases}\lvert 1-3x\rvert, & \text{если } x>0\\ 3x-7, & \text{если } -50\le x\le 0\\ x^2 & \text{– в остальных случаях}\end{cases}$	28	$y=\begin{cases}4x^2, & \text{если } x<10\\ x-20, & \text{если } x>15\\ \lvert 5-x\rvert & \text{– в остальных случаях}\end{cases}$
29	$y=\begin{cases}3x, & \text{если } x>5\\ \dfrac{130}{x}, & \text{если } -10\le x\le -1\\ \lvert x+5\rvert & \text{– в остальных случаях}\end{cases}$	30	$y=\begin{cases}3x, & \text{если } x>10\\ \lvert x-55\rvert, & \text{если } 1\le x\le 10\\ 2x-9 & \text{– в остальных случаях}\end{cases}$

Пример программы вычисления значения z по формуле, приведенной ниже, показан на рис. 3.4.

Вид функции
$z=y+\begin{cases}x^2-5, & \text{если } x-y=5\\ x+y, & \text{если } x-y=10\\ x^3-y & \text{– в остальных случаях}\end{cases}$

Рис. 3.4. Пример программы вычисления значения z

Листинг программы:

```
.model small
    .386
    .stack 256h
    .data
      ;здесь располагаются переменные
```

```
 x db 2
 y db 1
 ans dw ?
 outp db 3 dup('$')
.code
start:
 mov ax, @data
 mov ds, ax
 xor ax,ax
                    ; начало основной части
 mov  al, x
 mov  bl, y
 sub al, bl                ; x-y
 cmp al,5                  ; if x-y=5
 je M1
 cmp al,10                 ; if x-y=10
 je M2

                    ;вычисление x^3-y
 mov bl, x
 mov al, x
 imul bl
 imul bl
 xor bx, bx
 mov bl, y
 sub ax, bx
 mov cx, ax
 jmp exit
M1:                        ; вычисление x^2-5
 mov bl, x
 mov al, x
 imul bl
 sub ax, 5
 mov cx, ax
 jmp exit
 M2:                       ; вычисление x+y
 mov al, x
 add al, y
 mov cl, al
```

```
exit:
  xor bx, bx
  mov bl, y                    ;добавим y
  mov ax, cx
  add ax, bx
                               ; подготавливаем вывод результата
aam                            ;преобразует двоичное содержимое AX
;в формат неупакованного BCD-числа.
add ah,30h                     ;преобразование в символьный вид для
;вывода
add al,30h
mov [outp], ah                 ;помещаем в переменную
mov [outp+1], al

lea dx,outp   ;выводим на экран
mov ah, 09h
int 21h

mov ah, 4ch
int 21h

end start
```

Глава 4. Циклические алгоритмы и обработка строк

4.1. Цикл LOOP. Обработка строк. Цепочечные команды

Для организации цикла предназначена команда LOOP. У этой команды один операнд – имя метки, на которую осуществляется переход. В качестве счетчика цикла используется регистр CX. Команда LOOP выполняет декремент CX, а затем проверяет его значение. Если содержимое CX не равно нулю, то осуществляется переход на метку, иначе управление переходит к следующей после LOOP команде.

Содержимое CX интерпретируется командой как число без знака. В CX нужно помещать число, равное требуемому количеству повторений цикла. Понятно, что максимально может быть 65535 повторений. Еще одно ограничение связано с дальность перехода. Метка должна находиться в диапазоне –127…+128 байт от команды LOOP.

Все команды для работы со строками считают, что строка-источник находится по адресу DS:SI (или DS:ESI), т. е. в сегменте памяти, указанном в DS со смещением в SI, а строка-приемник – соответственно, в ES:DI (или ES:EDI). Кроме того, все строковые команды работают только с одним элементом строки (байтом, словом или двойным словом) за один раз. Направление просмотра строки зависит от значения флага направления DF. Изменить флаг DF можно командами:

– очистка флага DF (clear DF) – CLD (DF:=0) – просмотр слева направо;

– установка флага DF (set DF) – STD (DF:=1) – просмотр справа налево. Для того чтобы команда выполнялась над всей строкой, необходим один из префиксов повторения операций.

Префиксы повторения команд обработки строковых примитивов даны в табл. 19.

Таблица 19

Префиксы повторения команд обработки строковых примитивов

Префикс	Описание
REP	Повторять команду, пока ECX > 0
REPZ, REPE	Повторять команду, пока ECX > 0 и флаг нуля установлен (ZF = 1)
REPNZ, REPNE	Повторять команду, пока ECX > 0 и флаг нуля сброшен (ZF = 0)

Любой из префиксов выполняет следующую за ним команду строковой обработки столько раз, сколько указано в регистре ECX (или CX, в зависимости от разрядности адреса), уменьшая его при каждом выполнении команды на 1. Кроме того, префиксы REPZ и REPE прекращают повторения команды, если флаг ZF сброшен в 0, и префиксы REPNZ и REPNE прекращают повторения, если флаг ZF установлен в 1. Префикс REP обычно используется с командами INS, OUTS, MOVS, LODS, STOS, а префиксы REPE, REPNE, REPZ и REPNZ – с командами CMPS и SCAS. Поведение префиксов не с командами строковой обработки не определено.

Команды MOVSB, *MOVSW и MOVSD*. Копирует один байт (MOVSB), слово (MOVSW) или двойное слово (MOVSD) из памяти по адресу DS:ESI (или DS:SI, в зависимости от разрядности адреса) в память по адресу ES:EDI (или ES:DI).

После выполнения команды регистры ESI (SI) и EDI (DI) увеличиваются на 1, 2 или 4 (если копируются байты, слова или двойные слова), если флаг DF = 0, и уменьшаются, если DF = 1. При использовании с префиксом REP команда MOVS выполняет копирование строки длиной в ECX (или CX) байт, слов или двойных слов. Типы команд MOVS даны в табл. 20.

Таблица 20

Типы команд MOVS

Команда	Описание
MOVSB	Копирует последовательность байтов
MOVSW	Копирует последовательность слоев
MOVSD	Копирует последовательность двойных слоев

Команды CMPSP, *CMPSW и CMPSD*. Сравнивает один байт (CMPSB), слово (CMPSW) или двойное слово (CMPSD) из памяти по адресу DS:ESI (или DS:SI, в зависимости от разрядности адреса) с байтом, словом или двойным словом по адресу ES:EDI (или ES:DI) и устанавливает флаги аналогично команде CMP.

После выполнения команды регистры ESI (SI) и EDI (DI) увеличиваются на 1, 2 или 4 (если сравниваются байты, слова или двойные слова), если флаг DF = 0, и уменьшаются, если DF = 1. При использовании с префиксом REP команда CMPS выполняет сравнение строки длиной в ECX (или CX) байт, слов или двойных слов, но чаще ее ис-

пользуют с префиксами REPNE/REPNZ или REPE/REPZ. В первом случае сравнение продолжается до первого несовпадения в сравниваемых строках, а во втором – до первого совпадения. Типы команд CMPS даны в табл. 21.

Таблица 21

Типы команд CMPS

Команда	Описание
CMPSB	Сравнивает последовательность байтов
CMPSW	Сравнивает последовательность слоев
CMPSD	Сравнивает последовательность двойных слоев

Команды SCASB, SCASW и SCASD. Сравнивает содержимое регистра AL (SCASB), AX (SCASW) или EAX (SCASD) с байтом, словом или двойным словом из памяти по адресу ES:EDI (или ES:DI, в зависимости от разрядности адреса) и устанавливает флаги аналогично команде CMP.

После выполнения команды регистр EDI (DI) увеличивается на 1, 2 или 4 (если сканируются байты, слова или двойные слова), если флаг DF = 0, и уменьшается, если DF = 1. При использовании с префиксом REP команда SCAS выполняет сканирование строки длиной в ECX (или CX) байт, слов или двойных слов, но чаще ее используют с префиксами REPNE/REPNZ или REPE/REPZ. В первом случае сканирование продолжается до первого элемента строки, отличного от содержимого аккумулятора, а во втором – до первого совпадающего.

Команды STOSB, *STOSW и STOSD*. Копирует регистр AL (STOSB), AX (STOSW) или EAX (STOSD) в память по адресу ES:EDI (или ES:DI, в зависимости от разрядности адреса).

После выполнения команды регистр EDI (DI) увеличивается на 1, 2 или 4 (если копируется байт, слово или двойное слово), если флаг DF = 0, и уменьшается, если DF = 1. При использовании с префиксом REP команда STOS заполнит строку длиной в ECX (или CX) числом, находящимся в аккумуляторе.

Команды LODSB, *LODSW и LODSD*. Копирует один байт (LODSB), слово (LODSW) или двойное слово (LODSD) из памяти по адресу DS:ESI (или DS:SI, в зависимости от разрядности адреса) в регистр AL, AX или EAX, соответственно.

При использовании с префиксом REP команда LODS выполнит копирование строки длиной в ECX (или CX), что приведет к тому, что в аккумуляторе окажется последний элемент строки. На самом деле эту команду используют без префиксов, часто внутри цикла в паре с командой STOS, так что LODS считывает число, другие команды выполняют над ним какие-нибудь действия, а затем STOS записывает измененное число в то же место в памяти.

4.2. Команды ввода-вывода символов и строк

Для вывода информации на экран используется фнкция 09h. Она выводит строку целиком, пока не встретиться символ конца строки – $.

```
LEA dx, ska        ;в dx переменную для вывода
MOV ah, 09h    ;в ah номер функции
INT 21h            ; выполнение функции
```

Для ввода информации используется фнкция 0Ah. Ввод строки символов заканчивается нажатием клавиши Enter.

```
LEA dx, n   ; в dx размер вводимой строки
MOV ah, 0Ah   ; в ah функцию
INTt 21h
```

Еще одна функция вывода информации на экран. Она выводит не всю строку целиком, а заданное число символов.

```
LEA dx, ska        ;строка для вывода
MOV ah, 40h
MOV cx, 1  ; количество символов для вывода начиная с начала
INT   21h
```

Функция завершения программы – 4ch. В конце каждой программы.

```
MOV ah, 4ch
INT 21h
```

Функция 02 выводит цифру или число, после преобразования его в символ (добавление 30h).

```
ADD dx,30h
MOV ah,02
INT 21h
```

4.3. Практическая часть

Задание. Программирование алгоритмов обработки строк.

Разработать программу для обработки строковой информации с использованием цепочечных команд. Индивидуальные задания приведены в табл. 22.

Таблица 22

Индивидуальные задания

№	Задание
1	Определить количество слов, которые содержат ровно 3 буквы «А»
2	В строке, состоящей из латинских букв, каждый четный символ записать с большой буквы, нечетный – с маленькой
3	Дана строка. Отобрать слова в новую строку, в которых два раза встречается символ «x»
4	В строке заменить все малые латинские буквы на большие и наоборот
5	Из строки удалить символ, стоящий до и после «a»
6	Поставить каждый последний символ слова в начало слова
7	Из двух исходных строк сделать одну по следующему правилу: первое слово первой строки, затем первое слово второй строки, второе слово первой и т. д.
8	Найти процент слов, которые начинаются с маленькой гласной буквы английского алфавита
9	Дано предложение, состоящее из нескольких слов, разделенных одним или несколькими пробелами. Написать программу, печатающую все слова, начинающиеся на заданную букву
10	Из исходной строки выделить две подстроки. В первую должны войти слова, заканчивающиеся на «к», во вторую – все остальные слова
11	Найти самое короткое слово в строке
12	В словах, следующих за каждым из слов «татьяна», «влад», «тихон», заменить первую букву на прописную
13	Из исходной строки во вторую строку поместить слова, в которых нет буквы «z»
14	В строке каждый первый символ в слове поставить в конец слова
15	Известно, что строка состоит только из строчных букв. В начало строки поместить русские буквы, в конец – латинские
16	Дана строка, состоящая из символов латинского алфавита, разделенных одним или несколькими пробелами. Преобразовать каждое слово в строке, удалив из него все вхождения последней буквы этого слова (количество пробелов между словами не изменять)
17	Удалить из строки слова, содержащие букву «a»

Окончание табл. 22

№	Задание
18	Дана строка, состоящая из символов латинского алфавита, разделенных одним или несколькими пробелами. Преобразовать каждое слово в строке, удалив из него все вхождения первой буквы этого слова (количество пробелов между словами не изменять)
19	Дана строка. В строке все слова переставить в обратном порядке и вывести на экран
20	Поменять каждую букву на следующую по алфавиту, а букву «z» заменить на букву «a»
21	Определить – входит ли в заданную строку подстрока
22	В исходной строке заменить все «two» на «2»
23	В языке используется латинский алфавит. Глагол прошедшего времени получается из глагола настоящего времени изменением порядка следования гласных (a, o, u, i, e) на обратный. Согласные остаются на своих местах. Например, глагол gabonery преобразуется в gybenora. Задается глагол настоящего времени. Преобразовать его в глагол прошедшего времени и напечатать
24	В строке вывести слово, в котором наибольшее число букв «c»
25	Удалить из строки все цифры
26	Из исходной строки в новую строку поместить все слова, которые начинаются и заканчиваются на одну и ту же букву
27	В строке найти минимальное слово (по длине) и вывести его
28	В строке, состоящей из латинских букв, удалить все гласные
29	Поменять местами первый и последний символы в каждом слове
30	Дана строка-предложение из символов латинского алфавита. Вывести самое длинное слово в предложении (если таких слов несколько, то вывести последнее из них)

Пример выполнения программы, которая вводит исходную строку и с использованием цепочечных команд выводит ее в обратном порядке:

```
.model small
.stack
.data
        crlf    db    0Dh, 0Ah, '$'
        a       db    9 dup(0), '$'       ;тут будет переписанная задом
наперед новая строка
        b       db    10,0,10 dup(0)  ;тут будет вводимая строка

.code
```

```
ASSUME DS:@data, es:@data

start:
        mov ax, @data
        mov ds, ax
        mov es, ax

        lea si, b+10        ;в si источник
        lea di, a           ;в di приемник

        mov ah, 0Ah         ;вводим строку
        lea dx, b
        int 21h

        lea dx, crlf;абзац
        mov ah, 9
        int 21h

        mov cx, 9  ; в cx записываем длину цепочки
let:
        std    ;установить флаг направления (Direction flag) = 1. Значит обработка цепочки будет идти с конца
        lods b ; извлекает элемент из цепочки b, начиная с конца и помещает его в ax
        cld ;установить флаг направления (Direction flag) = 0. Обработка цепочки будет идти с начала
        stos a ;извлекает элемент из ax, и помещает его в цепочку a.
    loop let     ;повторять пока не обнулится cx.

        lea dx, a           ;вывод получившейся цепочки на экран.
        mov ah, 9
        int 21h

        mov ah, 4ch
        int 21h

    end start
```

Глава 5. Числовые массивы. процедуры и макросы в ассемблере

5.1. Числовые массивы

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

Описание и инициализация массива в программе:

1. Можно перечислить элементы массива в поле операндов. При перечислении элементы разделяются запятыми:

```
;массив из 5 элементов. Размер каждого элемента 2 байта
mas1 dw    1, 2, 3, 4, 5
```

2. Можно использовать оператор повторения DUP:

```
;массивы из 5 элементов. Размер каждого элемента 2 байта
mas2        dw    5 dup (?)
mas3        dw    5 dup (0)
```

3. Инициализировать элементы массива можно и в цикле:

```
    mov cx, 5      ;значение счетчика циклов в cx
    mov si, 0      ;индекс начального элемента в si
    mov ax, 1      ;значение первого элемента массива
go:                ;метка начала цикла
    mov mas2[si], ax ;запись в массив

    inc si  ;продвижение к следующему элементу массива
    inc si  ;или add si,2 ,т.к. данные размером в слово
    inc ax ;инкремент ax
    loop go       ;повторить цикл
```

Все элементы массива располагаются в памяти компьютера последовательно. Для того чтобы локализовать определенный элемент массива, к его имени нужно добавить индекс. В языке Ассемблер индексы массивов – это обычные адреса. Нумерация элементов массива начинается с нуля.

Например, пусть дан массив:

```
mas    dw   5, 3, 7, 1, 2
```

Для того чтобы считать третий элемент – число 7 в регистр dx. Этот элемент отстоит от начала массива на 4 байта. Поэтому, чтобы

обратиться к этому элементу, к базовому адресу необходимо добавить смещение 4.

```
mov    si,4
mov     dx, mas[si]
```

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

5.2. Режимы адресации микропроцессора

Можно выделить семь различных режимов адресации: регистровая, непосредственная, прямая, косвенная регистровая, по базе, прямая с индексированием, по базе с индексированием.

При *регистровой адресации* МП извлекает операнд из регистра или загружает его в регистр. Например:

```
mov ax, cx        ; пересылка содержимого регистра cx в регистр ax
```

*Непосредственная адре*сация позволяет указать значение константы в качестве второго операнда команды. Например,

```
mov cx, 500       ; загрузить в регистр cx значение 500
```

Исполнительным адресом называется смещение операнда относительно начала сегмента, в котором он находится. Все имена (переменные, метки), используемые в программе, по-другому называются символическими адресами и являются смещениями относительно начала сегмента, в котором они находятся и представляют собой целое число.

При *прямой адресации* исполнительный адрес является составной частью команды. МП добавляет этот адрес к сдвинутому на 4 бита (т. е. умноженному на 16) содержимому регистра сегмента данных DS и получает 20-битовый физический адрес операнда. Обычно прямая адресация применяется, если операндом служит метка (символический адрес). Например:

```
mov ax, table   ;загрузка в регистр ax содержимого
                ;ячейки памяти (2 байта) с меткой table
```

При косвенной регистровой адресации исполнительный адрес операнда содержится в базовом регистре bx, регистре указателя базы bp или в индексном регистре si или di. Регистровые операнды косвенной адресации необходимо заключать в квадратные скобки. Например:

```
                ;загрузить в регистр ax содержимое ячейки,
mov ax, [bx]    ;исполнительный адрес, которой находится в
                ;регистре bx
```

Чтобы поместить смещение (*относительный адрес*) ячейки table в регистр bx, можно использовать директиву OFFSET. Например:

```
mov bx, offset table    ;поместить в регистр bx смещение
                        ;ячейки table регистре bx
```

Данный способ адресации эффективен в тех случаях, когда необходимо получить доступ к последовательности ячеек, начиная с данного адреса.

При адресации по базе исполнительный адрес вычисляется сложением содержимого регистра bx или bp и сдвига, измеряемого в байтах. Например, следующие команды выполняют одно и то же действие:

```
    mov ax, [bx]+4   ;загрузить в регистр ax содержимое по адресу,
mov ax, [bx+4]   ;отстоящему на 4 байта от ячейки, исполнитель-
ный
mov ax, 4[bx]    ;адрес которой находится в регистре bx
```

При *прямой адресации с индексированием* исполнительный адрес вычисляется как сумма значений сдвига и индексного регистра si или di. Этот тип адресации удобен для доступа к элементам таблицы, когда сдвиг указывает на начало таблицы, а индексный регистр – на ее элемент. Например:

```
mov di,2
                ;загрузить в al третий элемент
mov al, table[di]
                ; (со смещением 2) таблицы table
```

При *адресации по базе с индексированием* исполнительный адрес вычисляется как сумма значений базового и индексного регистров и, возможно, сдвига. Этот метод удобен при адресации двухмерных массивов, когда базовый регистр содержит начальный адрес массива, а значение сдвига и индексного регистра определяют смещение по строке и столбцу. Примеры допустимых форматов команды:

```
mov ax, [bx+2+di]
mov ax, [bx+2][di]
mov ax, [bx][di+2]
```

При всех способах адресации (кроме тех, где используется регистр bp) исполнительный адрес вычисляется относительно регистра сегмента ds. Для регистра bp регистром сегмента служит регистр сегмента стека ss. Для изменения сегментных регистров, принятых по умолчанию, можно использовать операцию изменения префикса сегмента. Например, команда

mov es:[bx], dx

пересылает слово из регистра dx в ячейку памяти, находящуюся в сегменте, адресуемом текущим содержимым сегментного регистра es (вместо ds) со смещением, находящемся в регистре bx.

5.3. Команды работы подпрограмм

В языке ассемблера подпрограмма оформляется в виде процедур. Для объявления процедур в одномодульных, простых программах можно использовать упрощенный вариант синтаксиса:

<имя> PROC <тип>
<тело процедуры>
RET
<имя> ENDP,

где PROC – директива, определяющая начало процедуры, <имя> – метка, служащая именем процедуры и задающая адрес точки входа в процедуру; <тип> – параметр, определяющий тип процедуры и может принимать два значения: near (ближний) и far (дальний). Если используется упрощенные формы определения сегментов, то тип процедуры задается выбранной моделью памяти и его можно не указывать. Для моделей tiny, small и compact определяется тип near, а для остальных – far; RET (Return) – последняя команда подпрограммы. Она осуществляет выход из подпрограммы и возврат в основную программу. Тип процедуры автоматически задает тип возврата из процедуры.

Вызов подпрограммы из основной программы осуществляется командой CALL (), которая имеет следующий синтаксис:

CALL <тип> <имя>,

где <тип> – параметр, определяющий тип перехода и может принимать значения near или far; <имя> – имя вызываемой процедуры.

Когда команда CALL вызывает процедуру и сохраняет в стек адрес следующей за CALL команды. При ближнем вызове в стек по-

мещается значение регистра IP, а при дальнем вызове – значение регистров CS и IP. После выполнения в процедуре команды RET – выполняется обратное действие.

Фрагмент программы вызова процедур

```
; фрагмент кода программы
main PROC
MOV AX, @data
MOV DS, AX
CALL      MySub
...
MOV AX, 4C00h
INT  21
main ENDP
MySub     PROC
...
RET
MySub     ENDP
```

Передача параметров в процедуру. Передавать фактические параметры процедуре можно по-разному. Используются несколько способов передачи параметров. Выбор конкретного способа для определенной задачи зависит от количества используемых параметров и сложности их обработки.

1. Передача через регистры: программа перед вызовом заносит входные параметры в некоторые регистры процессора, а после возврата выбирает из регистров значения результатов.

2. Передача параметров по значению: перед обращением к процедуре основная процедура вычисляет значения фактических параметров и записывает их в регистры.

3. Передача параметров по ссылке означает передачу адреса (имени) ячейки, соответствующей фактическому параметру, через регистр: основная программа записывает в какой-то регистр адрес фактической переменной, а процедура берет его оттуда. Используют регистры-модификаторы BX, BP, SI или DI, так как процедуре придется модифицировать по этому регистру.

5.4. Макросы и макроопределения

Одними из самых мощных языковых средств Ассемблера являются макроопределения. Макроопределением (или макросом) называется набор команд, которому присвоено имя и который вставляется

в программу всякий раз, когда Ассемблер встречает это имя в тексте программы. Макрос начинается директивой MACRO и заканчивается ENDM. Например, пусть описано макроопределение PUTSPACE, выводящее символ пробела на экран:

```
PUTSPACE macro
        mov   ah,2
        mov   dl,' '
        int     21h
  endm
```

Теперь в программе можно использовать слово PUTSPACE как имя команды, и Ассемблер заменит каждое такое слово на макрорасширение, на три команды, содержащиеся в макроопределении. Макроопределения могут вызываться с параметрами, следовательно, в зависимости от ситуации включаемый код будет генерироваться с различными параметрами, например, как макрос обмена значений в памяти:

```
CHANGE_VAR   macro
 var1, var2
        push    var1
        push    var2
        pop     var1
       pop     var2
endm
```

Теперь можно использовать макрос CHANGE_VAR для того, чтобы поменять значения двух областей памяти местами.

5.5. Практическая часть

Задание. Использование макросов и процедур.

Разработать программу для обработки числовых массивов с использованием макросов и процедур. Индивидуальные задания приведены в табл. 23.

Таблица 23

Индивидуальные задания

№	Задание
1	В одномерном массиве определить количество элементов, равных минимальному
2	В числовом массиве определить количество элементов, равных максимальному

Продолжение табл. 23

№	Задание
3	Найти сумму элементов, стоящих до минимального элемента
4	В одномерном массиве определить среднее арифметическое элементов, стоящих после максимального.
5	В одномерном массиве найти количество элементов больших, чем среднее значение элементов массива
6	Из исходного массива получить два. В первый записать элементы на местах кратных 3, во второй – элементы на местах кратных 4
7	Найти количество чисел между первым и последним нулем
8	Найти сумму элементов, стоящих после минимального
9	Из двух массивов, упорядоченных по убыванию, получить один новый, упорядоченный по возрастанию
10	В одномерном массиве переставить элементы наоборот
11	В одномерном массиве найти сумму элементов, стоящих до максимального
12	Дан массив чисел. Из него сделать два отдельных массива. В первый поместить только положительные, во второй – отрицательные числа исходного массива
13	Отсортировать массив по убыванию методом включения
14	Отсортировать массив по возрастанию методом пузырька
15	Отсортировать массив по убыванию методом выбора
16	Поменять местами максимальный среди отрицательных и максимальный среди положительных элементов в одномерном массиве
17	Найти сумму всех элементов, значения которых меньше чем модуль минимального элемента
18	Найти количество элементов, стоящих между первым отрицательным и первым положительным элементами
19	Определить количество элементов, стоящих после последнего нуля
20	Из двух исходных массивов сделать один, в который поместить все четные по значению элементы
21	В одномерном массиве найти произведение элементов стоящих между первым и последним отрицательным элементами массива
22	Из исходного массива получить два. В первый записать положительные элементы на нечетных местах, во второй – отрицательные элементы на четных местах
23	Найти сумму элементов стоящих между первым положительным и минимальным элементом
24	Из исходного массива получить два. В первый поместить элементы, стоящие до максимального, во второй – после максимального элемента
25	В одномерном массиве определить среднее арифметическое элементов, стоящих после первого нечетного элемента

Окончание табл. 23

№	Задание
26	В массиве определить среднее арифметическое элементов, значение которых делится на 2
27	Определить сумму элементов, стоящих между первым и вторым отрицательным элементами
28	Найти произведение тех элементов, которые стоят до первого положительного элемента
29	Найти произведение всех элементов кроме наименьшего
30	Поменять местами максимальный и минимальный элементы в одномерном массиве

Пример программы, выполняющей обмен местами максимального и минимального элемента (с использованием макросов и процедур):

```
.model small
.386

swaps     macro     var1, var2
          push  var1
          push  var2
          pop   var1
          pop   var2
endm

prolog    macro          ;связка ds с сегментом данных
     mov  ax, @data
          mov  ds, ax
endm
epilog    macro          ;стандартный выход
     mov  ah, 4ch
        int 21h
endm

puts  macro s            ;вывод строки на экран
          mov  ah, 9
          lea  dx, s
          int  21h
endm
putch macro c            ;вывод символа на экран
```

```
            mov   ah, 2
            mov   dl, c
            int     21h
endm
.stack 256h
.data
                        ;   исходный массив
 array       db          44,43,65,42,54,74,83,45,34,65,'$'
 len       dw      10
 min        db      ?
 max        db      ?
 minadr     dw      ?
 maxadr     dw      ?
 outp       db      '00,$'
.code
 ASSUME     ds:@data,es:@data
main:
   prolog

       ;puts array
       putch 0ah
       putch 0dh

       mov cx, len
       mov si, 0
       mov minadr, si
       mov maxadr, si
       mov dl, array[si]
       mov min, dl
       mov max, dl

compare:
      mov   dl, array[si]      ;помещаем элемент массива в dx
      mov al,dl
      mov bl,dl

      aam                      ; вывод элемента исходного массива
      add ah,30h
      add al,30h
```

```
        mov [outp], ah
        mov [outp+1], al
        puts outp
        mov dl,bl

        cmp   dl, min          ;сравнение dx с min
        jb      new_min        ;переход, если меньше
        jae max_term

new_min:
        mov min, dl
        mov minadr, si
        jmp   max_term

max_term:
        cmp   dl, max           ;сравнение dx с max
        ja      new_max         ;переход, если больше
        jbe next                ; иначе к следующему элементу

new_max:
        mov max, dl
        mov maxadr, si

next:
        inc    si               ;на следующий элемент
        dec    cx               ;условие для выхода из цикла
        jcxz   outputs          ;cx = 0? если да – на выход
        jmp    compare          ;нет – повторить цикл
outputs:

        putch 0ah               ; курсор на новую строку
        putch 0dh

        mov si, minadr          ; тут меняем элементы
        mov dl, max
        mov array[si], dl
        mov si, maxadr
        mov dl, min
        mov array[si], dl
```

```
                                ;вывод полученного массива
        mov cx, len
        mov si, 0
loops:
        mov   dl, array[si]          ;помещаем элемент массива в dx
        mov al,dl

        aam                          ;  число в BCD и затем в ASCII-код
        add ah,30h
        add al,30h
        mov [outp], ah
        mov [outp+1], al
        puts outp
        inc si
loop loops

exit:
    epilog
end main
```

Глава 6. Связь языка Ассемблер и C++

Наиболее общим использованием языка Ассемблер в настоящее время является применение его в качестве приложения к языку программирования высокого уровня. При разработке программы, как правило, обычно используют язык высокого уровня и лишь небольшую часть модулей пишут на языке Ассемблер. Язык Ассемблер используется тогда, когда критичны скорость работы программы или ее размер, или когда язык высокого уровня не обеспечивает доступ к полным возможностям или к аппаратным средствам. Существует два вида комбинирования Ассемблера и C++

Использование ассемблерных вставок. Ассемблерные коды в виде команд ассемблера вставляются в текст программы на языке C++. Компилятор языка распознает их как команды ассемблера и без изменений включает в формируемый им объектный код. Эта форма удобна, если надо вставить небольшой фрагмент.

Использование внешних процедур и функций. Это более универсальная форма комбинирования. У нее есть ряд преимуществ:

– написание и отладку программ можно производить независимо;

– написанные подпрограммы можно использовать в других проектах;

– облегчаются модификация и сопровождение подпрограмм.

При написании ассемблерных вставок используется следующий синтаксис:

```
_asm
{
текст программы на ассемблере
}
```

Если необходимо записать одну команду, то синтаксис ее будет:

```
_asm   Команда;
```

Для связи посредством внешних процедур создается многофайловая программа.

6.1. Практическая часть

Задание . Связь Ассемблера с языком C++.

Разработать программу для обработки числовых массивов с использованием ассемблерных вставок. Ввод и вывод числовых мАсси-

вов необходимо проводить в C++, а основную обработку – на Ассемблере. Индивидуальные задания приведены в табл. 24.

Таблица 24

Индивидуальные задания

№	Задание
1	Даны два отсортированных по возрастанию массива. Получить новый массив из элементов исходных, отсортированный по убыванию
2	Из исходного массива получить два: первый – с четными по значению элементами исходного, второй – с нечетными
3	Преобразовать исходный массив так, чтобы порядок его элементов имел следующий вид: сперва находились элементы, равные 0 , затем – положительные элементы, и в конце массива – отрицательные
4	Из двух исходных массивов создать новый из элементов, стоящих после минимального в каждом из исходных массивов
5	Из двух исходных массивов получить новый, в котором записать сперва второй массив в обратном порядке, затем первый – в прямом порядке
6	Из двух массивов получить новый, в который сперва записать элементы, принадлежащие промежутку [5, 10] из исходных массивов, затем записать все остальные элементы
7	Даны два массива. В первом массиве найти максимальный элемент и вставить после него все элементы второго массива
8	Объединить два массива в один следующим образом: сначала все отрицательные из исходных массивов, затем – все положительные
9	Из одного исходного массива получить три: в первый записать положительные элементы, во второй – отрицательные, в третий – нули
10	Из одного исходного массива сформировать два новых массива: в первый поместить четные элементы, во второй – нечетные элементы
11	Из одного исходного массива сформировать два новых: в первый записать числа, по модулю больше 3, во второй – остальные
12	Из одного исходного массива выделить три новых: в первый поместить элементы, по значению кратные 3, во второй – кратные 4, в третий – кратные 5
13	Из двух массивов получить один, в который записать сперва все элементы на четных местах из исходных массивов, потом – на нечетных
14	Из одного исходного массива получить два новых. В первый записать элементы до максимального среди отрицательных, во второй – элементы после максимального среди отрицательных
15	Из двух отсортированных по убыванию массивов получить один, отсортированный по возрастанию
16	Даны два массива чисел. Найти в каждом из них минимальный элемент среди положительных и поменять их местами

Окончание табл. 24

№	Задание
17	В массиве найти минимальный элемент. Сформировать новый массив из элементов исходного, которые кратны минимальному
18	Сформировать новый массив из элементов исходного, стоящих между первым и вторым отрицательными элементами
19	Из двух исходных массивов создать новый из элементов, стоящих между минимальным и максимальным в каждом из исходных массивов
20	Поменять местами в массиве максимальное среди элементов, стоящих на четных местах и максимальное среди элементов, стоящих на нечетных местах
21	Даны два массива. Первый из них отсортирован по убыванию массивов, второй – по возрастанию. Из этих массивов получить новый, отсортированный по возрастанию
22	Сформировать новый массив из элементов исходного, стоящих между между последним и предпоследним отрицательными элементами
23	Из двух исходных массивов создать новый из элементов, стоящих перед максимальным в каждом из исходных массивов
24	Из двух исходных массивов создать новый из элементов, стоящих после последнего числа 10 в каждом из исходных массивов.
25	Даны два массива. В первом массиве найти первое число 7 и вставить после него все элементы второго массива
26	Даны два массива чисел. Найти в каждом из них максимальный элемент среди отрицательных и поменять их местами
27	Сформировать новый массив из элементов исходного, стоящих между между первым и последним элементами, равными 10
28	Поменять местами в массиве минимальное среди элементов, стоящих на четных местах и минимальное среди элементов, стоящих на нечетных местах
29	Даны два массива. В первом массиве найти последнее число 7 и вставить после него все элементы второго массива, расположенные в обратном порядке
30	Найти произведение элементов, стоящих между последним и предпоследним числом 5

Пример программы с ассемблерными вставками. Найти произведение элементов, стоящих между последним и предпоследним числом 5

```
#include <iostream>
#pragma inline
```

```
using namespace std;

void output(int arr[], int size);

void main()
{
	setlocale(LC_ALL, "Russian");
	int i = 0;
	cout << "Введите кол-во элементов массива: ";
	cin >> i;
	int* arr = new int[i];

	for (int j = 0; j < i; j++)
	{
		cout << "arr[" << j << "]: ";
		cin >> arr[j];
	}

	system("cls");

	output(arr, i);

	int number = 0, to_find = 5, result = 1, last_index = (i-1) * 4,
index_l = 0, index_s = 0;

	/////
	_asm
	{
	start:

		mov esi, arr;
		mov edi,0
		mov edx, last_index;
		mov ebx, dword ptr[esi + edx];
		mov eax, dword ptr[esi];

	search:

		cmp edi, 2;
```

```
        jz prepare_to_multiplication;

        mov ebx, dword ptr[esi + edx];

        cmp ebx, to_find;
        jz finded;

        cmp edx, 0;
        jz finish;

        sub edx, 4; // ищем с конца (вычитаем)
        jmp search;

finded:

        inc number;
        mov edi, number;

        cmp edi, 1;
        jz last_i;

        cmp edi, 2;
        jz start_i;

last_i:
        mov index_l, edx;
        sub edx, 4;

        jmp search;

start_i:
        mov index_s, edx;
        sub edx, 4;

        jmp search;

prepare_to_multiplication:

        mov edi, index_l;
```

```
            mov esi, index_s;
            sub edi, esi;
            cmp edi, 4;
            jz finish;

            mov esi, arr;

            mov edi, index_l;
            mov ebx, dword ptr[esi + edi];

            mov edi, index_s;
            mov eax, dword ptr[esi + edi];
            add edi, 4;

        multiplication:

            mov eax, dword ptr[esi + edi];
            imul eax, result;
            mov result, eax;

            add edi, 4;
            cmp edi, index_l;
            jz finish;

            jmp multiplication;

        finish:
        }
        /////

        if (index_l – index_s == 4)
            cout << "Найденные элементы являются соседями..." << endl << "То есть произведение равняется 0!" << endl;
        else if (number == 2)
            cout << "Результат (произведение элементов, лежащих между последним и предпоследним числом 5): " << result << endl;

        else
```

```
                cout << "В массиве отсутствует 2 элемента со значением '5'" << endl;

        delete[] arr;

        system("pause");
}

void output(int arr[], int size)
{
        for (int i = 0; i < size; i++)
                cout << arr[i] << " ";
        cout << endl;
}
```

Содержание

Учебное электронное издание комбинированного распространения

Учебное издание

Косинов Геннадий Петрович

ПРОГРАММИРОВАНИЕ

Практикум по одноименной дисциплине для студентов специальности 1-40 04 01 «Информатика и технологии программирования» дневной формы обучения

Электронный аналог печатного издания

Редактор *Т. Н. Мисюрова*
Компьютерная верстка *Н. Б. Козловская*

Подписано в печать 12.05.17.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Ризография. Усл. печ. л. 3,95. Уч.-изд. л. 4,28.
Изд. № 95.
http://www.gstu.by

Издатель и полиграфическое исполнение
Гомельский государственный
технический университет имени П. О. Сухого.
Свидетельство о гос. регистрации в качестве издателя
печатных изданий за № 1/273 от 04.04.2014 г.
пр. Октября, 48, 246746, г. Гомель