МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ»

ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ

Учебно-методическое пособие

Составители:
Г.Э. Вощинская,
Е.Е. Михайлова,
Е.М. Лещенко

Воронеж
Издательский дом ВГУ
2015

Утверждено научно-методическим советом факультета прикладной математики, информатики и механики 5 ноября 2015 г., протокол № 3

Рецензент – заведующий кафедрой ПиИТ ф-та ФКН ВГУ Н.А. Тюкачев

Учебно-методическое пособие подготовлено на кафедре программного обеспечения и администрирования информационных систем факультета прикладной математики, информатики и механики Воронежского государственного университета.

Рекомендовано студентам 3-го курса факультета прикладной математики, информатики и механики Воронежского государственного университета для проведения лабораторных занятий по предмету «Программирование на ассемблере».

Для специальности: 02.03.03 – Математическое обеспечение и администрирование информационных систем

Содержание

Введение

Хотя язык ассемблера относительно редко используется на практике, его изучение является необходимой частью подготовки профессиональных программистов, поскольку позволяет лучше понять принципы работы компьютера, операционных систем и трансляторов с языков высокого уровня, а также, в случае необходимости, создавать высокоэффективные программы.

Языки высокого уровня были разработаны для того, чтобы скрыть особенности конкретных компьютеров. Язык ассемблера, в свою очередь, создан для использования конкретной специфики компьютера.

Язык ассемблера – это символьная форма записи машинного языка, его использование существенно упрощает написание машинных программ.

В настоящее время практически все программисты сначала изучают язык высокого уровня (Паскаль, Си++ и т.п.), в котором многие проблемы реализации алгоритмов скрыты. Изучение языка ассемблера позволяет понять, во что превращаются удобные, лаконичные и мощные конструкции языков высокого уровня. Это дисциплинирует стиль и приучает к аккуратности программирования. Например, становится понятно, что не стоит ради «красоты» писать процедуры с длинным списком параметров и несколькими операторами.

Данное методическое пособие содержит краткое описание команд ассемблера x86, примеры программирования и задания для самостоятельного выполнения по курсу «Программирование на ассемблере».

Начальные сведения о языке ассемблера

Ассемблер – это алгоритмический язык низкого уровня. Для каждой архитектуры свой ассемблер. Ассемблером называют и компилятор с языка ассемблера.

Программа на языке ассемблера – это последовательность предложений:

<предложение>
<предложение>
…

Предложения по смыслу делятся на

- комментарии,
- команды,
- директивы.

Каждое предложение начинается с отдельной строки и имеет следующий формат:

метка: команда операнды; комментарий
или
метка: 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 распознаются, 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 (то есть 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. Пустая строка – это тоже комментарий. Комментарий может быть многострочным:

COMMENT < *маркер*> текст <*маркер*>

Например,

```
COMMENT * все
это является
комментарием* и это тоже
```

Представление команд

Машинная команда занимает от 1 до 6 байт.

Формат команды

КОП [*операнды*]

КОП – код операции – один или два байта. В программе на языке ассемблера записывается мнемоническое обозначение КОП.

Команда может иметь от 0 до 2 операндов.

Операнд определяет, где находятся данные:

- непосредственно в команде,
- в регистре,
- в оперативной памяти.

Для команд с двумя операндами возможны следующие сочетания:

- регистр – регистр,
- регистр – память,
- регистр – непосредственный операнд,
- память – непосредственный операнд.

Тип операнда определяется форматом его записи. При этом содержимое регистра может использоваться или как сам операнд, или как составляющая его адреса.

В команде указывается имя регистра. В ассемблере закреплены следующие названия регистров.

Регистры общего назначения

32-битные регистры: EAX (аккумулятор), EBX (база), ECX (счетчик), EDX (регистр данных). Младшие 16 бит каждого из этих регистров применяются как самостоятельные регистры с именами AX, BX, CX, DX. 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. Еще четыре регистра общего назначения – ESI (индекс источника), EDI (индекс приемника), EBP (указатель базы), ESP (указатель стека).

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 начал всех сегментов равны). В процессорах Intel предусмотрено шесть 16-битных регистров – CS, DS, ES, FS, GS, SS, где хранятся селекторы. В отличие от DS, ES, GS, FS, которые называются регистрами сегментов данных, CS и SS отвечают за сегменты двух особенных типов – сегмента кода и сегмента стека.

Еще один регистр, используемый для организации ветвлений в программе, – регистр флагов FLAGS. Каждый его бит устанавливается в 1 при определенных условиях.

0	NT	IOPI		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CF – флаг переноса. Устанавливается в 1, если результат предыдущей операции не уместился в приемнике и произошел перенос из 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.

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.

SF – флаг знака. Он всегда равен старшему биту результата.

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).

IF – флаг прерываний. Сброс этого флага в 0 приводит к тому, что процессор перестает обрабатывать прерывания от внешних устройств. Обычно его сбрасывают на короткое время для выполнения критических участков кода.

DF – флаг направления. Он контролирует поведение 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, то есть отрицательное, и наоборот.

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

Конструкции ветвления реализуются в языке ассемблера проверкой нужного флага.

Регистровая адресация

Операнды могут располагаться в любых регистрах общего назначения и сегментных регистрах. Значение операнда – содержимое регистра.

Некоторые команды используют определенные регистры специальным образом, при этом в команде регистр не указывается.

Пример команды с регистровой адресацией.

MOV AX,CX – пересылка содержимого регистра CX в регистр AX.

Непосредственный операнд

Некоторые команды позволяют использовать константы в качестве операнда-источника. В качестве непосредственного операнда могут быть числовые или символьные данные. Размер непосредственной константы должен быть согласован с размером второго операнда. Использование непосредственного операнда эффективнее, чем определение числовой константы в памяти.

Например, MOV AX, 500 – помещает в регистр AX число 500.

Прямая адресация

Исполнительный адрес является составной частью команды. Этот исполнительный адрес добавляется к содержимому сегментного регистра. Обычно прямая адресация применяется, если операндом служит метка.

Например, MOV AX, TABLE – пересылка значения, находящегося в оперативной памяти по адресу TABLE, в регистр AX.

Если содержимое сегмента данных:

0000		
0001	BB	TABLE
0002	AA	
0003		

Здесь значение метки TABLE равно 0001 относительно сегмента данных. Сегментный регистр по умолчанию – DS.

В результате содержимое регистра:

AX	AABB

Косвенная адресация

Исполнительный адрес операнда содержится в BX, BP, SI или в DI. Начиная с процессора 80386 и старше можно использовать EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.

Например, MOV AX,[BX] – пересылка значения, находящегося по адресу, взятому из регистра BX, в регистр AX. По умолчанию здесь в качестве сегментного регистра используется DS.

Адресация по базе

Исполнительный адрес вычисляется сложением сдвига и содержимого регистра, указанного в команде, – одного из регистров BX, BP, SI, DI, EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.

Например, MOV AX, [BP]+4

BP	001A

Содержимое сегмента данных:

001A	
001B	
001C	
001D	
001E	BB
001F	AA

Результат команды:

AX	AABB

Адресация с индексированием

Исполнительный адрес вычисляется как сумма значений сдвига и индексного регистра. В качестве 16-разрядных регистров можно использовать DI или SI, из 32-разрядных EAX, EBX, ECX, EDX, ESI, EDI.

Например, MOV AX,TABLE[DI] – пересылка значения, адрес которого вычисляется как сумма адреса TABLE и содержимого регистра DI.

DI	0004

Содержимое сегмента данных:

0001		TABLE
0002		
0003		
0004		
0005	BB	
0006	AA	

Результат команды:

AX	AABB

Адресация по базе с индексированием

Исполнительный адрес вычисляется как сумма значений базового регистра, индексного регистра и, возможно, сдвига. Из 16-битных регистров можно складывать только BX+SI, BX+DI, BP+SI, BP+DI, а из 32-битных – все восемь регистров общего назначения. Например,

MOV AX, [BX][SI]+2 или MOV AX, [BX+SI]+2

Основные непривилегированные команды

MOV *приемник, источник*

Пересылка данных. Копирует содержимое источника в приемник, источник не изменяется. В качестве источника для MOV могут использоваться: число (непосредственный операнд), регистр общего назначения, сегментный регистр или переменная (то есть операнд, находящийся в памяти); в качестве приемника: регистр общего назначения, сегментный регистр (кроме CS) или переменная. Оба операнда должны быть одного и того же размера – байт, слово или двойное слово. Нельзя выполнять пересылку данных с помощью MOV из одной переменной в другую, из одного сегментного регистра в другой, и нельзя помещать в сегментный регистр непосредственный операнд – эти операции выполняют двумя командами MOV (из сегментного регистра в обычный и уже из него – в другой сегментный), или парой команд PUSH/POP.

XCHC *операнд1,операнд2*

Обмен операндов между собой. Содержимое операнда2 копируется в операнд1, а старое содержимое операнда1 – в операнд2. XCHG можно выполнять над двумя регистрами или над регистром и переменной.

PUSH *источник*

Поместить данные в стек. Помещает содержимое источника в стек. Источником может быть регистр, сегментный регистр, непосредственный операнд или переменная. Фактически эта команда уменьшает ESP на размер источника в байтах (2 или 4) и копирует содержимое источника в память по адресу SS:[ESP]. Команда PUSH почти всегда используется в паре с POP (считать данные из стека). Поэтому, чтобы скопировать содержимое одного сегментного регистра в другой (что нельзя выполнить одной командой MOV), можно использовать такую последовательность команд:

```
PUSH CS
POP DS ; Теперь DS указывает на тот же сегмент, что и CS.
```

POP *приемник*

Считать данные из стека. Помещает в приемник слово или двойное слово, находящееся в вершине стека, увеличивая ESP на 2 или 4 соответственно. POP выполняет действие, полностью обратное PUSH. Приемником может быть регистр общего назначения, сегментный регистр, кроме CS (чтобы загрузить CS из стека, надо воспользоваться командой RET), или переменная. Если в роли приемника выступает операнд, использующий ESP для косвенной адресации, команда POP вычисляет адрес операнда уже после того, как она увеличивает ESP.

LEA *приемник, источник*

Вычисление эффективного адреса. Вычисляет эффективный адрес источника (переменная) и помещает его в приемник (регистр). Если адрес – 32-битный, а регистр-приемник – 16-битный, старшая половина вычисленного адреса теряется; если наоборот, то есть приемник 32-битный, а адресация 16-битная, то вычисленное смещение дополняется нулями.

Двоичная арифметика

Все команды этой группы, кроме команд деления и умножения, изменяют флаги OF, SF, ZF, AF, CF, PF в соответствии с назначением каждого из них.

ADD *приемник, источник*

Команда выполняет арифметическое сложение приемника и источника, помещает сумму в приемник, не изменяя содержимое источника. Приемник может быть регистром или переменной, источник – числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Команда ADD никак не различает числа со знаком и без знака, но, употребляя значения флагов CF (перенос при сложении чисел без знака), OF (перенос при сложении чисел со знаком) и SF (знак результата), разрешается применять ее и для тех, и для других.

ADC *приемник, источник*

Сложение с переносом. Эта команда аналогична ADD, но при этом выполняет арифметическое сложение приемника, источника и флага CF. Пара команд ADD/ADC используется для сложения чисел повышенной точности.

SUB *приемник, источник*

Вычитает источник из приемника и помещает разность в приемник. Приемник может быть регистром или переменной, источник – числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Точно так же, как и команда ADD, SUB не делает различий между числами со знаком и без знака, но флаги позволяет использовать ее и для тех, и для других.

SBB *приемник, источник*

Вычитание с займом. Эта команда аналогична SUB, но она вычитает из приемника значение источника и дополнительно вычитает значение флага CF.

IMUL *источник*
IMUL *приемник, источник*
IMUL *приемник, источник1, источник2*

Умножение чисел со знаком. Эта команда имеет три формы, различающиеся числом операндов:

1. **IMUL** *источник*
   источник (регистр или переменная) умножается на AL, AX или EAX (в зависимости от размера операнда), и результат располагается в AX, DX:AX или EDX:EAX соответственно.
2. **IMUL** *приемник, источник*
   источник (число, регистр или переменная) умножается на приемник (регистр), и результат заносится в приемник.
3. **IMUL** *приемник, источник1, источник2*
   источник1 (регистр или переменная) умножается на источник2 (число), и результат заносится в приемник (регистр). Во всех трех вариантах считается, что результат может занимать в два раза больше места, чем размер источника. В первом случае приемник автоматически оказывается очень большим, но во втором и третьем случаях существует вероятность переполнения и потери старших битов результата. Флаги OF и CF будут равны единице, если это произошло, и нулю, если результат умножения поместился целиком в приемник (во втором и третьем случаях) или в младшую половину приемника (в первом случае). Значения флагов SF, ZF, AF и PF после команды IMUL не определены.

MUL *источник*

Умножение чисел без знака. Выполняет умножение содержимого источника (регистр или переменная) и регистра AL, AX, EAX (в зависимости от размера источника) и помещает результат в AX, DX:AX, EDX, EAX соответственно. Если старшая половина результата (AH, DX, EDX) содержит только нули (результат целиком поместился в младшую половину), флаги CF и OF устанавливаются в 0, иначе – в 1. Значение остальных флагов (SF, ZF, AF и PF) не определено.

IDIV *источник*

Выполняет целочисленное деление со знаком AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL, AX или EAX, а остаток – в AH, DX или EDX соответственно. Результат всегда округляется в сторону нуля, знак остатка совпадает со знаком делимого, абсолютное значение остатка меньше абсолютного значения делителя. Флаги CF, OF, SF, ZF, AF и PF после этой команды не определены, а переполнение или деление на ноль вызывает исключение #DE (ошибка при делении) в защищенном режиме и прерывание 0 – в реальном.

DIV *источник*

Выполняет целочисленное деление без знака AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная) и по-

мещает результат в AL, AX или EAX, а остаток – в AH, DX или EDX соответственно. Результат всегда округляется в сторону нуля, абсолютное значение остатка меньше абсолютного значения делителя. Флаги CF, OF, SF, ZF, AF и PF после этой команды не определены, а переполнение или деление на ноль вызывает исключение #DE (ошибка при делении) в защищенном режиме и прерывание 0 – в реальном.

INC *приемник*

Увеличивает приемник (регистр или переменная) на 1. Единственное отличие этой команды от ADD *приемник*, 1 состоит в том, что флаг CF не затрагивается. Остальные арифметические флаги (OF, SF, ZF, AF, PF) устанавливаются в соответствии с результатом сложения.

DEC *приемник*

Уменьшает приемник (регистр или переменная) на 1. Единственное отличие этой команды от SUB *приемник,1* заключается в том, что флаг CF не затрагивается. Остальные арифметические флаги (OF, SF, ZF, AF, PF) устанавливаются в соответствии с результатом вычитания.

NEG *приемник*

Изменение знака. Выполняет над числом, содержащимся в приемнике (регистр или переменная), операцию дополнения до двух. Эта операция эквивалентна обращению знака операнда, если рассматривать его как число со знаком. Если приемник равен нулю, флаг CF устанавливается в 0, иначе – в 1. Остальные флаги (OF, SF, ZF, AF, PF) назначаются в соответствии с результатом операции.

CMP *приемник, источник*

Сравнивает приемник и источник и устанавливает флаги. Действие осуществляется путем вычитания источника (число, регистр или переменная) из приемника (регистр или переменная; приемник и источник не могут быть переменными одновременно), причем результат вычитания никуда не записывается. Единственным следствием работы этой команды оказывается изменение флагов CF, OF, SF, ZF, AF и PF. Обычно команду CMP используют вместе с командами условного перехода (Jcc).

Логические операции

AND *приемник, источник*

Команда выполняет побитовое логическое И над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 1, только если соответствующие биты обоих операндов были равны 1, и равен 0 в остальных случаях.

Наиболее часто AND применяют для выборочного обнуления отдельных битов. Флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен.

OR *приемник, источник*

Выполняет побитовое логическое ИЛИ над приемником (регистр или переменная) и источником (число, регистр или переменная) и помещает результат в приемник; источник и приемник не могут быть переменными одновременно. Любой бит результата равен 0, только если соответствующие биты обоих операндов были равны 0, и равен 1 в остальных случаях. Команду OR чаще всего используют для выборочной установки отдельных битов. При выполнении команды OR флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен.

XOR *приемник, источник*

Выполняет побитовое логическое исключающее ИЛИ над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 1, если соответствующие биты операндов различны, и нулю – в противном случае. При выполнении команды OR флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен.

NOT *приемник*

Каждый бит приемника (регистр или переменная), равный нулю, устанавливается в 1, и каждый бит, равный 1, сбрасывается в 0. Флаги не затрагиваются.

TEST *приемник, источник*

Вычисляет результат действия побитового логического И над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и устанавливает флаги SF, ZF и PF в соответствии с полученным показателем, не сохраняя результата (флаги OF и CF обнуляются, значение AF не определено). TEST, так же как и CMP, используется в основном в сочетании с командами условного перехода (Jcc).

Операции сдвигов

SAR *приемник, счетчик*	Арифметический сдвиг вправо
SAL *приемник, счетчик*	Арифметический сдвиг влево
SHR *приемник, счетчик*	Логический сдвиг вправо
SHL *приемник, счетчик*	Логический сдвиг влево

Эти четыре команды выполняют двоичный сдвиг приемника (регистр или переменная) вправо (в сторону младшего бита) или влево (в сторону

старшего бита) на значение счетчика (число или регистр CL, из которого учитываются только младшие 5 бит, принимающие значения от 0 до 31). Операция сдвига на 1 эквивалентна умножению (сдвиг влево) или делению (сдвиг вправо) на 2. Команды SAL и SHL выполняют одну и ту же операцию (на самом деле это одна и та же команда) – на каждый шаг сдвига старший бит заносится в CF, все биты сдвигаются влево на одну позицию, и младший бит обнуляется. Команда SHR осуществляет прямо противоположную операцию: младший бит заносится в CF, все биты сдвигаются на 1 вправо, старший бит обнуляется. Эта команда эквивалентна беззнаковому целочисленному делению на 2. Команда SAR действует по аналогии с SHR, только старший бит не обнуляется, а сохраняет предыдущее значение. SAR, таким образом, эквивалентна знаковому делению на 2, но, в отличие от IDIV, округление происходит не в сторону нуля, а в сторону отрицательной бесконечности. Сдвиги, большие 1, эквивалентны соответствующим сдвигам на 1, выполненным последовательно. Сдвиги на 1 изменяют значение флага OF: SAL/SHL устанавливают его в 1, если после сдвига старший бит изменился (то есть старшие два бита исходного числа не были одинаковыми), и в 0, если старший бит остался тем же. SAR устанавливает OF в 0, а SHR – в значение старшего бита исходного числа. Для сдвигов на несколько битов значение OF не определено. Флаги SF, ZF, PF назначаются всеми сдвигами в соответствии с результатом, параметр AF не определен (кроме случая, когда счетчик сдвига равен нулю: ничего не происходит, и флаги не изменяются).

Команды передачи управления

JMP *операнд*

Безусловный переход. Передает управление в другую точку программы, не сохраняя какой-либо информации для возврата. Операндом может быть непосредственный адрес для перехода (в программах используют имя метки, установленной перед командой, на которую выполняется переход), а также регистр или переменная, содержащая адрес.

В зависимости от типа перехода различают:

- переход типа ***short*** (короткий переход) – если адрес перехода находится в пределах -128...+127 байт от команды JMP;
- переход типа ***near*** (ближний переход) – если адрес перехода находится в том же сегменте памяти, что и команда JMP;
- переход типа ***far*** (дальний переход) – если адрес перехода находится в другом сегменте. Дальний переход может выполняться и в тот же самый сегмент при условии, что в сегментной части операнда указано число, совпадающее с текущим значением CS.

Jcc *метка*

Условный переход. Это набор команд, выполняющих переход (типа *short* или *near*), если удовлетворяется соответствующее условие, которым в каждом случае реально является состояние тех или иных флагов. Но, когда команда из набора Jcc используется сразу после CMP, условия приобретают формулировки, соответствующие отношениям между операндами CMP. Например, если операнды CMP были равны, то команда JE, выполненная сразу после CMP, осуществит переход. Операнд для всех команд из набора **Jcc** – 8-битное или 32-битное смещение относительно текущей команды. Слова выше и ниже при описании команд относятся к сравнению чисел без знака; слова больше и меньше учитывают знак.

Команды **Jcc** не поддерживают дальних переходов, поэтому, если требуется выполнить условный переход на дальнюю метку, необходимо использовать команду из набора Jcc с обратным условием и дальний JMP.

JA JBE	CF = 0 и ZF = 0	Если выше Если не ниже и не равно
JAE JNB JNC	CF = 0	Если выше или равно Если не ниже Если нет переноса
JB JNAE JC	CF=1	Если ниже Если не выше и но равно Если перенос
JBE JNA	CF = 1 или ZF = 1	Если ниже или равно Если не выше
JE JZ	ZF= 1	Если равно Если ноль
JG JNLE	ZF = 0 и SF = OF	Если больше Если не меньше и не равно
JGE JNL	SF = OF	Если больше или равно Если не меньше
JL JNGE	SF <> OF	Если меньше Если не больше и не равно
JLE JNG	ZF = 1 или SF <> OF	Если меньше или равно Если не больше
JNE JNZ	ZF = 0	Если не равно Если не ноль
JNO	OF = 0	Если нет переполнения
JO	OF=1	Если есть переполнение
JNP JPO	PF = 0	Если нет четности Если нечетное
JP JPE	PF=1	Если есть четность Если четное
JNS	SF = 0	Если нет знака
JS	SF=1	Если есть знак

JCXZ *метка*	переход, если CX = 0
JECXZ *метка*	переход, если ECX = 0

Выполняет ближний переход на указанную метку, если регистр CX или ECX (для JCXZ и JECXZ соответственно) равен нулю. Так же как и команды из серии Jcc, JCXZ и JECXZ не могут выполнять дальних переходов.

LOOP *метка*

Цикл. Уменьшает регистр ECX на 1 и выполняет переход типа *short* на метку (которая не может быть дальше расстояния -128...+ 127 байт от команды LOOP), если ECX не равен нулю. Эта команда используется для организации циклов, в которых регистр ECX (или CX при 16-битной адресации) играет роль счетчика. LOOP не изменяет значения флагов.

LOOPE *метка*	Цикл, пока равно
LOOPZ *метка*	Цикл, пока ноль
LOOPNE *метка*	Цикл, пока не равно
LOOPNZ *метка*	Цикл, пока не ноль

Все перечисленные команды уменьшают регистр ECX на один, после чего выполняют переход типа short, если ECX не равен нулю и если выполняется условие. Для команд LOOPE и LOOPZ условием является равенство единице флага ZF, для команд LOOPNE и LOOPNZ – равенство флага ZF нулю. Сами команды LOOPcc не изменяют значений флагов, так что ZF должен быть установлен (или сброшен) предшествующей командой.

Директивы определения данных

[<*имя*>] **DB** <*операнд*> {,<*операнд*>}

Ассемблер вычисляет операнды и заносит их значения в последовательные байты памяти. Первому из них присваивается указанное имя.

Псевдокоманды определения переменных

Директива определения данных – это директива ассемблера, которая приводит к включению данных или кода в программу, хотя сама никакой команде процессора не соответствует. Ассемблер устанавливает тип (байт, слово, вещественное число и т. д.) данных, задает начальное значение и ставит в соответствие переменной метку, которая будет использоваться для обращения к этим данным. Директивы определения данных записываются в общем виде следующим образом:

имя_переменной **D*** *значение*

где D* – одна из нижеприведенных псевдокоманд:

DB – определить байт;

DW – определить слово (2 байта);

DD – определить двойное слово (4 байта);

DF – определить 6 байт (адрес в формате 16-битный селектор: 32-битное смещение);

DQ – определить учетверенное слово (8 байт);
DT – определить 10 байт (80-битные типы данных, используемые FPU).

Поле значения может содержать одно или несколько чисел, строк символов (взятых в одиночные или двойные кавычки), операторов **?** и DUP, разделенных запятыми. Все установленные таким образом данные окажутся в выходном файле, а имя переменной будет соответствовать адресу первого из указанных значений. Например, набор директив

```
text_string   DB 'Hello world!'
number        DW 7
table         DB 1,2,3,4,5,6,7,8,9,OAh,OBh,OCh,ODh,OEh,OFh
float number  DD 3.5e7
```

заполняет данными 33 байта. Первые 12 байт содержат ASCII-коды символов строки *Hello world!***,** и переменная text_string указывает на первую букву в этой строке, так что команда **`mov al, text_string`**
считает в регистр AL число 48h (код латинской буквы **H).**
Если вместо точного значения указан знак **?**, переменная считается неинициализированной и ее значение на момент запуска программы может оказаться любым. Если нужно заполнить участок памяти повторяющимися данными, используется специальный оператор DUP, имеющий формат
счетчик **DUP** (значение).
Например, определение

```
table_512w DW 512 DUP(?)
```

создает массив из 512 неинициализированных слов, на первое из которых указывает переменная *table_512w*. В качестве аргумента в операторе **DUP** могут выступать несколько значений, разделенных запятыми, и даже дополнительные вложенные операторы **DUP**:

```
Y DB -7,3DUP(0,2 DUP(?) );эквивалентно
Y DB -7,0,?,?,0,?,?,0,?,?
```

Организация программы

Каждая программа состоит из одного или нескольких сегментов, например
сегмент кода;
сегмент данных;
сегмент стека.

Обычно область памяти, в которой находятся команды, называют *сегментом кода,* область памяти с данными – *сегментом данных* и область памяти, отведенную под стек, – *сегментом стека.* Разумеется, ассемблер позволяет изменять устройство программы как угодно – помещать данные в сегмент кода, разносить код на множество сегментов, помещать стек в один сегмент с данными или вообще использовать один сегмент для всего.

Сегмент программы описывается директивами SEGMENT и ENDS.

имя_сегмента **SEGMENT** <параметры>

…

имя_сегмента **ENDS**

Имя сегмента – метка, которая будет использоваться для получения сегментного адреса, а также для комбинирования сегментов в группы. Параметры директивы SEGMENT необязательны. Они позволяют задать тип сегмента (например STACK, PUBLIC, PRIVATE), разрядность, выравнивание, режим READONLY. В программе может быть несколько сегментов с одним и тем же именем. Все они будут объединены. Адрес сегмента определяется одним из сегментных регистров. Директива ASSUME указывает ассемблеру, с каким сегментом или группой сегментов связан тот или иной сегментный регистр.

ASSUME *регистр :связь*,..

В качестве операнда *связь* могут использоваться имена сегментов, имена групп, выражения с оператором SEG или слово NOTHING, означающее отмену действия предыдущей ASSUME для данного регистра. Эта директива не изменяет значений сегментных регистров, а только позволяет ассемблеру проверять допустимость ссылок и самостоятельно вставлять при необходимости префиксы переопределения сегментов. Загрузка сегментных регистров, кроме CS, должна быть сделана в программе до использования какого-либо имени из соответствующего сегмента.

Сегмент кода всегда адресуется регистром CS. Сегмент стека всегда адресуется регистром SS. Остальные сегментные регистры адресуют сегменты данных.

Действуют соглашения:

- адрес перехода всегда по CS,
- в строковых командах особые правила,
- в других командах
    - DS, если адрес не модифицируется или среди модификаторов нет BP,
    - SS, если среди модификаторов – BP.

Перечисленные директивы удобны для создания больших программ на ассемблере, состоящих из разнообразных модулей и содержащих множество сегментов. В повседневном программировании обычно используется ограниченный набор простых вариантов организации программы, известных как модели памяти.

Модели памяти и упрощенные директивы определения сегментов

Модели памяти задаются директивой .MODEL

.model *модель, язык, модификатор,*

где модель – одно из следующих слов:

TINY – код, данные и стек размещаются в одном и том же сегменте размером до 64 Кб. Эта модель памяти чаще всего используется при написании на ассемблере небольших программ;

SMALL – код размещается в одном сегменте, а данные и стек – в другом (для их описания могут применяться разные сегменты, но объединенные в одну группу). Эту модель памяти также удобно использовать для создания программ на ассемблере;

COMPACT – код размещается в одном сегменте, а для хранения данных могут использоваться несколько сегментов, так что для обращения к данным требуется указывать сегмент и смещение (данные дальнего типа);

MEDIUM – код размещается в нескольких сегментах, а все данные – в одном, поэтому для доступа к данным используется только смещение, а вызовы подпрограмм применяют команды дальнего вызова процедуры;

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

FLAT – то же, что и **TINY**, но используются 32-битные сегменты, так что максимальный размер сегмента, содержащего и данные, и код, и стек, – 4 Мб.

Язык – необязательный операнд, принимающий значения C, PASCAL, BASIC, FORTRAN, SYSCALL и STDCALL. Если он указан, подразумевается, что процедуры рассчитаны на вызов из программ на соответствующем языке высокого уровня, следовательно, если указан язык C, все имена ассемблерных процедур, объявленных как PUBLIC, будут изменены так, чтобы начинаться с символа подчеркивания, как это принято в C.

Модификатор – необязательный операнд, принимающий значения NEARSTACK (по умолчанию) или FARSTACK. Во втором случае сегмент стека не будет объединяться в одну группу с сегментами данных.

После того как модель памяти установлена, вступают в силу упрощенные директивы определения сегментов, объединяющие действия директив SEGMENT и ASSUME. Кроме того, сегменты, объявленные упрощенными директивами, не требуется закрывать директивой ENDS – они закрываются автоматически, как только ассемблер обнаруживает новую директиву определения сегмента или конец программы.

Директива .**CODE** описывает основной сегмент кода

.**CODE** *имя_сегмента*

эквивалентно

```
_TEXT SEGMENT word public 'CODE'
```

для моделей TINY, SMALL и COMPACT и

```
name_TEXT SEGMENT word public 'CODE'
```

для моделей MEDIUM, HUGE и LARGE name – имя модуля, в котором описан данный сегмент. В этих моделях директива .CODE также допускает необязательный операнд – имя определяемого сегмента, но все сегменты кода, описанные так в одном и том же модуле, объединяются в один сегмент с именем NAME_TEXT.

.**STACK** *размер*

Директива .STACK описывает сегмент стека и эквивалентна директиве

```
STACK SEGMENT para public 'stack'
```

Необязательный параметр указывает размер стека. По умолчанию он равен 1 Кб. Сегмент стека должен быть описан, т.к. его использует ОС при обработке прерываний. Стек располагается в сегменте памяти, адрес которого задан регистром SS. Очередная ячейка стека адресуется парой **SS:ESP**. Стек заполняется от конца к началу. Если в директиве SEGMENT задан параметр Stack, SS устанавливается на начало сегмента, а SP равно длине стекового сегмента, т.е. указывает на первую ячейку после стека.

Запись в стек выполняется командой

PUSH *op*

op (операнд) – регистр, сегментный регистр, переменная, непосредственный операнд. Размер операнда 2 или 4 байта.

Чтение из стека

POP *op*

op (операнд) такой же, как и у команды PUSH, кроме регистра CS. Команды **PUSH** и **POP** не проверяют на выход за границы стека. Проверку нужно делать в программе.

SP=0 ? – стек полон

SP=k ? – стек пуст, k – длина стека в байтах.

Для доступа к элементам стека можно использовать выражения:

SP	W1
+2	W2
+4	W3

Чтобы выбрать W3, можно установить BP на SP, т.е.

```
MOV BP, SP
MOV AX, [BP+4],
```

выражение [SP+4] использовать нельзя, т.к. SP не является модификатором, но ESP использовать можно.

.DATA
Описывает обычный сегмент данных и соответствует директиве
```
_DATA SEGMENT word public 'DATA'
```
.**DATA**?
Описывает сегмент неинициализированных данных:
```
_BSS SEGMENT word public 'BSS'
```
Этот сегмент обычно не включается в программу, а располагается за концом памяти, так что все описанные в нем переменные на момент загрузки программы имеют неопределенные значения.

.**CONST**
Описывает сегмент неизменяемых данных:
```
CONST SEGMENT word public 'CONST'
```
В некоторых операционных системах этот сегмент будет загружен так, что попытка записи в него может привести к ошибке.

.**FARDATA** – *имя сегмента*
Сегмент дальних данных:
```
имя_сегмента segment para private 'FAR_DATA'
```
Доступ к данным, описанным в этом сегменте, потребует загрузки сегментного регистра. Если не указан операнд, в качестве имени сегмента используется FAR_DATA

.**FARDATA?** *имя_сегмента*
Сегмент дальних неинициализированных данных:
```
имя_сегмента segment para private 'FAR_BSS'
```
Как и в случае с FARDATA, доступ к данным из этого сегмента потребует загрузки сегментного регистра. Если имя сегмента не указано, используется FAR_BSS. Во всех моделях памяти сегменты, представленные директивами .**DATA** :**DATA?**, .**CONST**, **.FARDATA** и .**FARDATA?**, а также сегмент, описанный директивой .**STACK**, если не был указан модификатор **FARSTACK**, и сегмент .**CODE** в модели **TINY** автоматически объединяются в группу с именем FLAT – для модели памяти **FLAT** или DGROUP – для всех остальных моделей. При этом сегментный регистр DS (и SS, если не было FARSTACK, и CS в модели TINY) настраивается на всю эту группу, как если бы была выполнена директива ASSUME.

Вывод на экран в текстовом режиме
Программа, написанная на ассемблере, так же как и программа, написанная на любом другом языке программирования, выполняется не сама по себе, а при помощи операционной системы. Операционная система выделяет области памяти для программы, загружает ее, передает ей управление и

обеспечивает взаимодействие программы с устройствами ввода-вывода, файловыми системами и другими программами (разумеется, кроме тех случаев, когда эта программа сама является операционной системой или ее частью). Способы взаимодействия программы с внешним миром различны для разных операционных систем. Для этого вызываются соответствующие функции. Обращение программы к операционной системе осуществляется посредством программных прерываний.

Команда **INT** 21h вызывает системную функцию DOS. Эта команда – основное средство взаимодействия программ с операционной системой.
Функция DOS номер 9 – выводит строку на экран. Адрес начала строки задается в регистрах DS:DX, признак конца – символ $(24h). Номер функции записывается в регистр AH=09h.

Системный вызов DOS 4Ch возвращает управление операционной системе. В регистр AH помещается значение 4Ch, в регистр AL – код возврата 0: MOV AX,4C00h.

Ввод строки с клавиатуры

Ввод строки с клавиатуры выполняется функцией 10 прерывания 21.

```
DS:DX:=АДРЕС БУФЕРА ВВОДА
MOV AH,0AH
INT 21H
```

Ввод пока не <*Enter*>

Буфер имеет следующую структуру:

max	n	S1	…	cr	…	
0	1	2		n+2		max+1

max – максимальное количество символов
n – фактическое количество символов
например,

```
BUF   DB  10,?,10 DUP(‘ ‘)
……
      LEA  DX,BUF
      MOV  AH,0AH
      INT  21H
```

Пример программы, выводящей на экран строку "Hello, World!", приведен ниже.

```
; HELLO.ASM
; ВЫВОДИТ НА ЭКРАН СООБЩЕНИЕ "HELLO, WORLD!" И ЗАВЕРШАЕТСЯ
;
    .MODEL   SMALL    ; МОДЕЛЬ ПАМЯТИ, ИСПОЛЬЗУЕМАЯ ДЛЯ
EXE
```

```
     .STACK   100H      ; СЕГМЕНТ СТЕКА = 256
     .CODE              ; НАЧАЛО СЕГМЕНТА КОДА
START:
     MOV  AX, DGROUP    ; СЕГМЕНТНЫЙ АДРЕС СТРОКИ MESSAGE
     MOV  DS, AX    ; ПОМЕСТИТЬ В DS
     MOV  DX,OFFSET MESSAGE  ; СМЕЩЕНИЕ СТРОКИ - В DX
     MOV  AH,9 ; НОМЕР ФУНКЦИИ DOS - В AH (ВЫВОД СТРОКИ)
     INT  21H       ; ВЫЗОВ ФУНКЦИИ
     MOV  AX, 4C00H; ФУНКЦИЯ "ЗАВЕРШЕНИЕ ПРОГРАММЫ"
     INT  21H       ; ВЫЗОВ ФУНКЦИИ
     .DATA          ; СЕГМЕНТ ДАННЫХ
MESSAGE  DB   'HELLO WORLD!',0DH,0AH,'$'  ; СТРОКА
     END  START     ; КОНЕЦ ПРОГРАММЫ
```

Рассмотрим еще один пример: двоичный поиск в упорядоченном массиве заданного значения Y. Массив, его размер и искомое значение заданы в программе. Ответ выводится на экран.

```
; Двоичный Поиск В Массиве Arr Значения Y
.Model Tiny         ; модель памяти, используемая для Com
.Code               ; начало сегмента кода
Org 100h            ; начальное значение счетчика - 100h
Start:
     Lea Di,Arr     ; в Di адрес начала массива
     Mov Bx, Y      ; в Bx искомое значение
     Xor Cx,Cx      ;  Cx=0 - не нашли
     Cmp Bx,[Di]    ; Y <=Arr[1] ?
     Ja Chk_Last    ; нет, сравнить с последним
     Je Eq_1        ; равно первому?
     Jmp Output     ; нет в массиве - вывод результата
Eq_1:
     Inc Cx         ; Cx=1 -нашли
     Jmp Output

Chk_Last:
     Mov Si,N       ; в Si количество элементов массива
     Dec Si
     Shl Si,1       ; в Si длина массива в байтах
     Add Si,Di      ; в Si адрес последнего элемента
     Cmp Bx,[Si]    ; Arr[N]>=Y?
     Jb Search      ; нет, искать в середине
     Je Eq_N        ; равно последнему
```

```
     Jmp Output    ; нет в массиве, больше последнего

Eq_N:
     Inc Cx        ; нашли = последнему
     Jmp Output;
;Поиск Внутри Массива
Search:
     Mov Si, N     ; в Si смещение на половину массива
Iven_Inx:
     Test Si,1     ; проверка на четность
     Jz Add_Inx
     Inc Si        ; сделать четным
     Add_Inx:
     Add Di,Si     ; в Di адрес очередного элемента
Compare:
     Cmp Bx,[Di]   ; Arr[I] = Y?
     Je All_Done   ; нашли
     Ja Higer      ; выбрать отрезок для продолжения поиска
; искать в левой части
     Cmp Si,2      ; не закончились элементы?
     Jne Inx_Ok

No_Match:
     Jmp Output    ;не нашли
Inx_Ok:
     Shr Si,1      ; разделить индекс на 2
     Test Si,1     ; проверить на четность Si
     Jz Sub_Inx
     Inc Si        ; сделать четным
     Sub_Inx:
     Sub Di,Si     ; в Di  адрес очередного элемента
     Jmp Compare

; искать в правой части

Higer:
     Cmp Si,2      ; не закончились элементы?
     Je No_Match   ; да - выход
     Shr Si,1      ; разделить индекс на 2
     Jmp  Iven_Inx

All_Done:
```

```
     Inc Cx        ;   нашли

; Вывод Результата
Output:
     Mov Ah,9
     Test Cx,1
     Jz Send_No    ;
Send_Yes:
     Mov Dx,Offset Res_Yes  ;
     Jmp Print
Send_No:
     Mov Dx,Offset Res_No  ;
Print:
     Int 21h
Ret

Arr  Dw 1,12,33,41,55
Y         Dw 33
N         Dw 5
Res_Yes   Db 'Value Exist'
          Db 0dh,0ah,'$'
Res_No    Db 'Value Not Found'
          Db 0dh,0ah,'$'

End Start
```

Задание 1

Обрабатывается числовой массив. Значения и длина массива заданы в самой программе. Используются команды двоичной арифметики. Ответ выводится на экран.

1. Упорядочен ли массив по возрастанию.
2. Упорядочен ли массив по убыванию.
3. Чередуются ли в массиве знаки.
4. Находятся ли все положительные значения раньше отрицательных.
5. Равно ли число нулевых элементов числу ненулевых.
6. Повторяется ли в массиве первое значение.
7. Образуют ли значения арифметическую прогрессию.
8. Является ли массив «палиндромом».
9. Является ли последовательность значений рядом чисел Фибоначчи.
10. Есть ли два равных значения рядом.
11. Повторяется ли максимальное значение.

12. Имеют ли все числа одинаковый знак.
13. Чередуются ли четные и нечетные значения.
14. Есть ли несколько подряд идущих нулей.
15. Упорядочены ли по возрастанию все положительные значения.
16. Равны ли все отрицательные значения.
17. Является ли первая последовательность подряд идущих нулей самой длинной.
18. Превосходит ли какая-нибудь сумма подряд идущих положительных чисел заданное К.
19. Превосходит ли наибольшая длина подряд идущих положительных чисел наибольшую длину подряд идущих отрицательных чисел.
20. Попадают ли все числа в заданный диапазон.
21. Находятся ли все отрицательные значения раньше положительных.
22. Повторяется ли последнее значение.
23. Повторяется ли минимальное значение.
24. Есть ли несколько подряд идущих положительных значений.
25. Упорядочены ли по возрастанию все отрицательные значения.
26. Равны ли все положительные значения.
27. Является ли первая последовательность подряд идущих положительных значений самой длинной.
28. Превосходит ли разница между максимальным и минимальным значениями заданное К.
29. Является ли элемент с заданным номером медианой.
30. Превосходит ли количество упорядоченных по возрастанию серий заданное К.

Организация процедур в ассемблере

Описание процедур выглядит следующим образом:
<*имя процедуры*> **PROC** <*параметр*>
<*тело процедуры*>
<*имя процедуры*> **ENDP**

имя процедуры считается меткой. Оно соответствует первой команде процедуры. Его можно использовать в команде перехода.
<*параметр*>::=**NEAR** | **FAR**
по умолчанию – **NEAR**
К Near-процедуре можно обращаться внутри сегмента, в котором она описана. К Far-процедуре – из любого сегмента.

Вызов процедуры выполняется командой
CALL <*имя процедуры***>**

Если описание процедуры было раньше, то формируется соответствующий тип (Near или Far). Если описание ниже, то формируется Near.

Обращение Near**:** SP:=SP-2
Stack[SP]:=(IP)
IP:=адрес_процедуры

Обращение Far:
SP:=SP-2
Stack[SP]:=(CS)
SP:=SP-2
Stack[SP]:=(IP)
CS:=seg(Proc)
IP:=offset(Proc

Возврат из процедуры
RET
или
RET<*смещение*>
из процедуры Near:
IP:=Stack[SP]
SP:=SP+2
из процедуры Far:
IP:=Stack[SP]
SP:=SP+2
CS:=Stack[SP]
SP:=SP+2
при наличии смещения
SP:=SP+смещение

Ассемблер формирует соответствующие друг другу команды CALL и RET. Имена и метки, описанные в процедуре, *не локализуются* и должны быть уникальны. Чтобы обратиться к Far-процедуре, не описанной ранее, нужно использовать указатель **FAR PTR.** При обращении можно использовать косвенный адрес. Возможно вложенное описание процедур.

Обычно процедуры размещают либо в конце сегмента команд, либо в самом начале этого сегмента перед командой, с которой должно начаться выполнение программы. В больших программах нередко процедуры размещают в отдельном сегменте команд.

Параметры можно передавать различными способами:

- в регистрах;
- в глобальных переменных;
- в стеке;
- в потоке кода;
- в блоке параметров.

При использовании процедуры настоятельно рекомендуется сохранить содержимое всех регистров, которые она изменяет в стеке, а при завершении – восстановить их значения.

Программа, использующая процедуру ввода целого числа с клавиатуры.

```
.MODEL tiny
.CODE
org 100h
; ввод с клавиатуры целого числа и вывод его на экран
_start:

; ввод числа в регистр AX
     call ReadInteger
     mov  A,ax

; перевод строки
     mov  ah,09h
     lea  dx,crlf
     int  21h

; вывод заголовка
     mov  ah,09h
     lea  dx,res
     int  21h

; вывод числа из AX
     mov  ax,A
     call WriteInteger
     ret

; ввод 10-числа в регистр AX
ReadInteger   proc
     push cx        ; сохранение регистров
     push bx
     push dx
     mov  fl,0      ; флаг отрицательного числа
     xor  cx, cx
     mov  bx, 10
     call ReadChar  ; ввод первого символа

     cmp  al,'-'    ; если минус - установить флаг
```

```
     je   nnn
     jmp  nn
nnn:
     mov  fl,1

read:

     call ReadChar  ; ввод очередного символа
nn:  cmp  al, 13    ; Enter ?
     je   done      ; да -  > завершение

     sub  al,'0'    ; нет -> перевод цифры char -> int
     xor  ah, ah
     xor  dx, dx
     xchg cx, ax
     mul  bx
     add  ax, cx
     xchg ax, cx
     jmp  read
     done:
     xchg ax, cx
     cmp  fl,1
     je   eee
     jmp  ee
eee:
     neg  ax
ee:
     pop  dx
     pop  bx
     pop  cx
     ret
ReadInteger   endp

; ввод одного символа
ReadChar proc
     mov  ah,1
     int  21h
     ret
ReadChar endp

; вывод 10-числа
WriteInteger proc near
```

```
     push ax
     push cx
     push bx
     push dx
     xor  cx, cx
     mov  bx, 10
; число отрицательное?
     cmp  ax,0
     jl   ddd  ; если - да
     jmp  divl ; если - нет
; вывести минус и поменять знак
ddd:
     push ax
     mov  dl, '-'
     mov  ah, 2
     int  21h
     pop  ax
     neg  ax

; получить 10-цифры и поместить их в стек,
; в cx - количество полученных цифр
     divl:
     xor  dx, dx
     idiv bx
     push dx
     inc  cx
     cmp  ax,0
     jg   divl

; достать из стека, перевести в код ASSII  и вывести
popl:
     pop  ax
     add  al, '0'

     call WriteChar
     loop popl

     pop  dx
     pop  bx
     pop  cx
     pop  ax
     ret
```

```
WriteInteger endp

; вывод одного символа
WriteChar proc
     push ax
     push dx
     mov  dl, al
     mov  ah, 2
     int  21h
     pop  dx
     pop  ax
     ret
WriteChar endp

A    dw   67
fl   dw   ?
res  db   'res', 0dh,0ah,'$'
crlf db   0dh,0ah,'$'
end  _start
```

Пример процедуры, передающей параметры через стек
Найти сумму элементов массива слов.

```
.MODEL   SMALL
.STACK   100h
.DATA
ARY  dw   100 dup(?)    ;100 слов для массива
COUNT     dw   ?        ;фактическая длина массива
SUM  dw   ?         ;сумма
.CODE
START:
     mov  ax,@DATA
     mov  ds,ax         ;инициализация ds

               ..............................
     mov  bx, offset ARY    ;
     push bx
     mov  bx, offset COUNT   ;
     push bx
     mov  bx, offset SUM    ;
     push bx
     call far ptr PRO_ADD   ;
                         ..............................
```

```
PROG_SEG ends

 ;Процедура нахождения суммы элементов массива слов
PRO_ADD  proc far
     push bp        ; сохранить регистры
     mov  bp,sp
     push ax
     push cx        ;
     push si        ;
     push di        ;
; в стеке
;    di        <-- sp
;    si
;    cx
;    ax
;    bp        <--bp
;    ip
;    cs
;    адрес результата
;    адрес COUNT
;    адрес ARY
     mov  si,[bp+10]    ; si:=адрес ARY
     mov  di,[bp+8];
     mov  cx,[di]  ; cx:=COUNT
     mov  di,[bp+6]; di:=адрес результата
     xor  ax,ax
NEXT:    add  ax,[si]  ; ax:=ax+a[i]
     add  si,2
     loop NEXT
     mov  [di],ax  ; SUM:=результат
     pop  di        ; восстановить
     pop  si        ; регистры
     pop  cx        ;
     pop  ax        ;
     pop  bp        ;
     ret  6
PRO_ADD  endp
.....
```

Задание 2

Выполнить **задание 1** (свой вариант), изменив условие следующим образом:

- Размер массива ввести с клавиатуры.
- Массив ввести с клавиатуры.
- Введенный массив вывести на экран.
- Обработку массива оформить в виде процедуры.
- Массив и другие входные данные (если они есть) передать параметрами через стек.

Команды обработки строк

Команды обработки строк позволяют производить действия над блоками байтов или слов памяти. Эти блоки (строки) могут иметь длину до 64 Кбайт и состоять из числовых значений, символов или любых других типов.

Команды работы со строками предполагают, что адрес строки-источника DS:SI (DS:ESI), а адрес строки-приемника ES:DI (ES:EDI). Команды работы со строками выполняют операцию для одного элемента за один раз. Для того чтобы команда выполнялась над всей строкой, необходим один из префиксов повторения операций:

REP – повторять, пока CX ≠ 0;
REPE / REPZ – повторять, пока CX ≠ 0 и ZF = 1;
REPNE / REPNZ – повторять, пока CX ≠ 0 и ZF = 0.

Префикс повторения позволяет выполнить операцию над группой элементов. Любой из префиксов обеспечивает повторение следующей за ним команды столько раз, сколько указано в регистре CX (ECX), уменьшая его при каждом выполнении команды на 1. Регистры SI (ESI) и DI (EDI) автоматически увеличиваются, если флаг направления DF = 0, или уменьшаются, если DF = 1.

Префикс REP обычно используется с командами INS, OUTS, MOVS, LODS, STOS. Префиксы REPE, REPNE, REPZ, REPNZ – с командами CMPS и SCAS.

Команда **CLD** устанавливает DF=0, команда **STD** устанавливает DF = 1.

Пересылка строки:

MOVS *приемник, источник* – копирование строки;
MOVSB – копирование строки байтов;
MOVSW – копирование строки слов;
MOVSD – копирование строки двойных слов.

При использовании формы записи MOVS ассемблер по типу операндов заменит ее на соответствующую команду MOVSB, MOVSW или 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).

Используя MOVS с операндами, разрешается заменить регистр DS другим с помощью префикса замены сегмента (ES:, GS:, FS:, CS:, SS:), регистр ES заменить нельзя. После выполнения команды регистры ESI (или SI) и EDI (или DI) увеличиваются на 1, 2 или 4 (если копируются байты, слова или двойные слова), когда флаг DF = 0, и уменьшаются, когда DF = 1. Команда MOVS с префиксом REP выполняет копирование строки длиной в ECX (или CX) байтов, слов или двойных слов.

Сравнение строк:

CMPS *приемник, источник* – сравнение строк;
CMPSB – сравнение строк байтов;
CMPSW – сравнение строк слов;
CMPSD – сравнение строк двойных слов.

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. При использовании формы записи CMPS ассемблер сам определяет по типу указанных операндов, какую из трех форм этой команды (CMPSB, CMPSW или CMPSD) выбрать. Применяя CMPS с операндами, можно заменить регистр DS другим, воспользовавшись префиксом замены сегмента (ES:, GS:, FS:, CS:, SS:), регистр ES заменить нельзя. После выполнения команды регистры ESI (или SI) и EDI (или DI) увеличиваются на 1, 2 или 4 (если сравниваются байты, слова или двойные слова), когда флаг DF = 0, и уменьшаются, когда DF = 1. Команда CMPS с префиксами REPNE/REPNZ или REPE/REPZ выполняет сравнение строки длиной в ECX (или CX) байтов, слов или двойных слов. В первом случае сравнение продолжается до первого совпадения в строках, а во втором – до первого несовпадения.

Сканирование строки:

SCAS *приемник* – сканирование строки;
SCASB – сканирование строки байтов;
SCASW – сканирование строки слов;
SCASD – сканирование строки двойных слов.

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. При использовании формы записи SCAS ассемблер сам определяет по типу указанного операнда, какую из трех форм этой команды (SCASE, SCASW или SCASD) выбрать. После выполнения команды регистр EDI (или DI) увеличивается на 1, 2 или 4 (если ска-

нируются байты, слова или двойные слова), когда флаг DF = 0, и уменьшается, когда DF = 1. Команда SCAS с префиксами REPNE/REPNZ или REPE/REPZ выполняет сканирование строки длиной в ECX (или CX) байтов, слов или двойных слов. В первом случае сканирование продолжается до первого элемента строки, совпадающего с содержимым аккумулятора, а во втором – до первого отличного.

Чтение из строки:
LODS *источник* – чтение из строки;
LODSB – чтение байта из строки;
LODSW – чтение слова из строки;
LODSD – чтение двойного слова из строки.

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 формы записи LODS ассемблер сам определяет по типу указанного операнда, какую из трех форм этой команды (LODSB, LODSW или LODSD) выбрать. Применяя LODS с операндом, можно заменить регистр DS на другой с помощью префикса замены сегмента (ES:, GS:, FS:, CS:, SS:). После выполнения команды регистр ESI (или SI) увеличивается на 1, 2 или 4 (если считывается байт, слово или двойное слово), когда флаг DF = 0, и уменьшается, когда DF = 1. Команда LODS с префиксом REP выполнит копирование строки длиной в ECX (или CX), и в аккумуляторе окажется последний элемент строки. На самом деле LODS 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 на прежнее место в памяти.

Запись в строку:
STOS *приемник* – запись в строку;
STOSB – запись байта в строку;
STOSW – запись слова в строку;
STOSD – запись двойного слова в строку.

Копирует регистр AL (STOSB), AX (STOSW) или EAX (STOSD) в память по адресу ES:EDI (или ES:DI, в зависимости от разрядности адреса). При использовании формы записи STOS ассемблер сам определяет по типу указанного операнда, какую из трех форм этой команды (STOSB, STOSW или STOSD) выбрать. После выполнения команды регистр EDI (или DI) увеличивается на 1, 2 или 4 (если копируется байт, слово или двойное слово), когда флаг DF = 0, и уменьшается, когда DF = 1. Команда STOS с префиксом REP заполнит строку длиной в ECX (или CX) числом, находящимся в аккумуляторе.

Примеры использования строковых команд

```
; Замена в символьной строке каждого знака '$' на знак '_'
     mov  cx, length ASCII_STG        ; инициализация
     mov  al, '$'
     mov  di, offset ASCII_STG
     cld
LOOP1:   repnz scas    ASCII_STG; искать '$'
     jnz       DONE
     mov  byte ptr[di-1], '_'    ; заменить
     jmp  short LOOP1
DONE:
……….
ASCII_STG:    db   'abc$$aaaaa$aaaa$'
……….

;  Поиск  последовательности  символов  SEQUENCE  в  строке
START_DATA.
char proc near
     lea  bx, START_DATA         ; адрес текста
; число символов в искомой последовательности загрузить в
счетчик
     mov  cx, NO_CHARACTER
     lea  si, SEQUENCE  ; загрузить в si адрес искомой по-
следовательности
     call SEARCH
     ret
START_DATA    db 'In word processing anod'
          db 'text editing application'
          db 0Dh, 0Ah        ; 0Dh – CR, 0Ah – LF
          db 'it is often'
          db 0Dh, 0Ah, 1Ah
SEQUENCE db   'locate'
NO_CHARACTER dw    6
CHAR      endp
;
SEARCH    proc near
          mov  dx, cx    ; сохранить количество символов
          mov  bp, si         ; сохранить искомый перемещаемый
адрес
MATCH:    mov  di, bx         ; сохранить текущий символ тек-
ста
```

```
NEXT:         lodsb                ; выбрать символ текста
RESTART:
          cmp  byte ptr[bx], 0Dh       ; CR?
          je   CONTROL_CHAR
          cmp  byte ptr[bx], 0Ah       ; LF?
          je   CONTROL_CHAR
          cmp  byte ptr[bx], 1Ah       ; конец файла?
          je   END_FILE
          cmp  al, [bx]
          pushf          ; сохранение флага нуля
          inc       bx
          popf
          loope     NEXT
          jne       SKIP
          clc            ; найдена последовательность
          ret
SKIP:          mov  si, bp
          mov  cx, dx
          jmp  short MATCH
CONTROL_CHAR: inc  bx
          jmp  short RESTART
END_FILE: std            ; последовательность не обнаружена
SEARCH    endp
```

Задание 3

Вводится текст. Текст состоит из слов. Между словами один или несколько пробелов. Перед первым словом и после последнего слова могут быть пробелы. В конце текста – точка.

Требования к реализации приложения:

- ввод данных с клавиатуры одной строкой;
- вывод введенного текста и ответа на экран;
- решение задачи оформить процедурой с параметрами, параметры передаются через стек;
- использовать строковые команды с префиксом повторения.

1. Дано предложение. Определить, сколько раз два соседних слова начинаются на одну букву.
2. Дано предложение. Удвоить последнюю букву каждого слова.
3. Дано предложение. Наибольшее число пробелов между словами уменьшить на единицу, если это число больше одного.
4. Дано предложение. Между словами добавить по одному пробелу.
5. Дано предложение. Определить, сколько слов содержат два одинаковых символа рядом.

6. Дано предложение. Определить количество слов, содержащих хотя бы одну букву из двух, заданных вводом.
7. Дано предложение. Определить, сколько раз два соседних слова заканчиваются на одну букву.
8. Дано предложение. Определить, сколько раз два соседних слова имеют одинаковую длину.
9. Дано предложение. Определить, встречаются ли два одинаковых слова рядом.
10. Дано предложение. Определить, сколько слов имеют такую же длину, что и первое.
11. Дано предложение. Между словами исключить по одному пробелу, если их число два или больше.
12. Дано предложение. В слове с заданным номером после заданной буквы вставить заданную букву.
13. Дано предложение. Определить, сколько слов содержат две заданные буквы.
14. Дано предложение. Определить, сколько слов начинаются или заканчиваются на заданную букву.
15. Дано предложение. После слова с заданным номером вставить заданное слово.
16. Дано предложение. Удалить слово с заданным номером.
17. Дано предложение. Удалить слова с заданной длиной.
18. Дано предложение. Заменить латинские строчные буквы в начале слова на прописные.
19. Дано предложение. Найти количество слов с максимальной длиной.
20. Дано предложение. Определить, сколько слов начинаются и заканчиваются на те же буквы, что и первое слово.
21. Дано предложение. Определить, сколько слов начинается на последнюю букву предыдущего слова.

Литература

1. Юров В.И. Assembler / В.И. Юров. – 2-е изд. – СПб. : Питер, 2010. – 636 с.
2. Зубков С.В. Assembler для DOS, Windows и UNIX / С.В. Зубков. – 3-е изд., стер. – М. ; СПб. : ДМК Пресс : Питер, 2004. – 608 с.
3. Абель П. Язык ассемблера для IBM PC и программирование / П. Абель ; пер. с англ. – М. : Высшая школа, 1992. – 448 с.
4. Таненбаум Э. Архитектура компьютера / Э. Таненбаум ; пер. с англ. (под ред. А.В. Гордеева). – 4-е изд. – СПб. : Питер, 2002. – 698 с.
5. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера / Л. Скэнлон ; пер. с англ. – М. : Радио и связь, 1989. – 336 с.
6. Турбо Ассемблер 3.0. Руководство пользователя [Электронный ресурс]. – Режим доступа: http://citforum.ru/programming/tasm3/index.shtml

Учебное издание

ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ

Учебно-методическое пособие

Вощинская Гильда Эдгаровна,
Михайлова Елена Евгеньевна,
Лещенко Елена Михайловна

Редактор М. С. Исаева
Электронная верстка О. В. Нагаевой

Подписано в печать 19.01.2016. Формат 60×84/16.
Уч.- изд. л. 3,0. Усл. п. л. 2,5. Тираж 30 экз. Заказ 848

Издательский дом ВГУ
394000 Воронеж, пл. Ленина, 10
Отпечатано в типографии Издательского дома ВГУ
394000 Воронеж, ул. Пушкинская, 3