Министерство образования и науки Российской Федерации
Муромский институт (филиал)
федерального государственного бюджетного образовательного учреждения
высшего образования
**«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»
(МИ ВлГУ)**

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

Практикум

для студентов образовательной программы
09.03.01 *Информатика и вычислительная техника*

Текстовое электронное издание

Учебно-методический центр МИ ВлГУ
Муром 2016

УДК [004.9 + 004.4] (075.8)
ББК 32.973-018я73

Составители:
Бейлекчи Д.В., ст. преподаватель кафедры ЭиВТ МИ ВлГУ;
Холкина Н.Е., доцент кафедры ЭиВТ МИ ВлГУ.

Ответственный за выпуск:
заведующий кафедрой электроники и вычислительной техники,
доктор технических наук, профессор Кропотов Юрий Анатольевич

Программирование на языке ассемблера: Практикум для студентов образовательной программы 09.03.01 Информатика и вычислительная техника / сост. Бейлекчи Д.В., Холкина Н.Е. [Электронный ресурс]. – Электрон. текстовые дан. (1,7 Мб). - Муром.: МИ ВлГУ, 2016. – 1 электрон. опт. диск (CD-R). – Систем. требования: процессор x86 с тактовой частотой 500 МГц и выше; 512 Мб ОЗУ; Windows XP/7/8; видеокарта SVGA 1280x1024 High Color (32 bit); привод CD-ROM. - Загл. с экрана.
№ госрегистрации 0321602844

В практикуме приведено описание восьми лабораторных работ по курсу «Архитектура микропроцессора и программирование на языке ассемблера». Каждая лабораторная работа содержит краткие теоретические сведения и рассчитана на четыре академических часа самостоятельной работы. После выполнения всего курса лабораторных работ студент должен приобрести базовые знания и навык в программировании на языке ассемблера.

Текстовое электронное издание

Минимальные системные требования:
Компьютер: процессор x86 с тактовой частотой 500 МГц и выше; ОЗУ 512 Мб;
10 Мб на жестком диске; видеокарта SVGA 1280x1024 High Color (32 bit);
привод CD-ROM
Операционная система: Windows XP/7/8
Программное обеспечение: Adobe Acrobat Reader версии 6 и старше.

Содержание

Предисловие

Результат работы самого лучшего оптимизирующего транслятора с языка высокого уровня не может конкурировать по скорости или компактности с кодом, написанным опытным программистом на языке ассемблера. Поэтому на языке ассемблера пишут, когда скорость работы программы или занимаемая ей память имеют решающее значение. Язык ассемблера незаменим для получения полного доступа к некоторым специфическим функциям процессора или периферийных устройств.

Язык ассемблера предоставляет возможность изучения процессора, его возможностей и ограничений, что позволяет лучше понять принципы функционирования как аппаратных, так и программных систем.

Курс лабораторных работ рассчитан на студентов образовательной программы 09.03.01 Информатика и вычислительная техника, но может быть полезен и тем, кто самостоятельно изучает язык ассемблера.

В практикуме приведено описание восьми лабораторных работ по курсу «Архитектура микропроцессора и программирование на языке ассемблера». Каждая лабораторная работа содержит краткие теоретические сведения и рассчитана на четыре академических часа самостоятельной работы. После выполнения всего курса лабораторных работ студент должен приобрести базовые знания и навык программирования на языке ассемблера.

Лабораторная работа № 1
ЗНАКОМСТВО С ПРОГРАММАМИ В МАШИННЫХ КОДАХ

Цель работы: Изучение структуры машинных команд и методов работы с шестнадцатеричным редактором.

ДОМАШНЯЯ ПОДГОТОВКА

- Изучить программную модель микропроцессора i8086:
  - Программно-доступные регистры;
  - Организацию памяти;
  - Формат команд;
  - Режимы адресации данных i8086;
- Подготовить ответы на контрольные вопросы.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Программная модель процессора Intel 8086

Организация памяти

Для хранения программы и данных в микропроцессоре (МП) intel 8086 (i8086) используются одно пространство памяти. Такая организация получила название *архитектуры Джона фон Нейман.*

В i8086 выделяется адресное пространство небольшого объема, которое называется набором программно-доступных *регистров МП*. В отличии от памяти программ и памяти данных регистры располагаются внутри МП, что обеспечивает быстрый доступ к информации, хранящейся в них.

Кроме регистров, в МП существует специальная область называемая стеком. *Стек* – это область памяти, специально выделяемая для временного хранения данных программы. Главная функция стека это организация подпрограмм: в стеке хранятся адреса возврата из подпрограмм (адрес с которого необходимо продолжить программу после завершения подпрограммы), а также параметры, передаваемые в подпрограмму. Принцип работы стека позволяет организовывать вложенные вызовы подпрограмм.

В МП i8086 используется *сегментная* модель памяти. При такой организации основная память (данных, программ и стека) разделяется на блоки – *сегменты*, которые поддерживаются на аппаратном уровне. Сегментная организация обеспечивает существование нескольких независимых адресных пространств как в пределах одной задачи (программы), так и в системе в целом.

При сегментной организации внутри программы используются логические (виртуальные) адреса, которые состоят из адреса сегмента и смещения относительно начала сегмента. Логический адрес принято записывать в формате

«сегмент : смещение». Вторая часть адреса, смещение, называется также эффективным (исполнительным) адресом.

Поскольку для адресации памяти процессор использует 16-разрядные адресные регистры, то это обеспечивает ему доступ к 2^{16}=65536 байт или 64 Кб основной памяти. Однако адресная шина, соединяющая процессор i8086 и память 20-разрядная и МП генерирует 20-битовые физические адреса. (Под физическим адресом понимается адрес памяти, выдаваемый на шину адреса микропроцессора.) Таким образом, МП i8086 может адресовать 2^{20} (1 Мб) памяти.

Рассмотрим порядок формирования физического адреса из логического. В МП хранятся 16-разрядные базовые адреса используемых сегментов. Микропроцессор объединяет 16-разрядный базовый адрес и 16-разрядный исполнительный адрес следующим образом: он расширяет содержимое сегментного регистра (базовый адрес) четырьмя нулевыми битами (в младших разрядах), делая его 20-разрядным (полный адрес сегмента) и прибавляет смещение (исполнительный адрес) (рис. 1). Эта специальная процедура пересчёта адресов поддерживается на аппаратном уровне. Полученный 20-разрядный результат является физическим (абсолютным) адресом ячейки памяти.

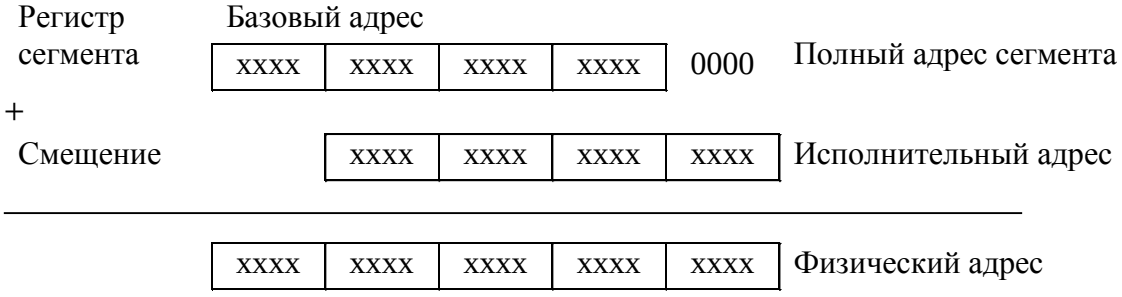


Рис. 1. Формирование физического адреса

Регистры

В программной модели процессора i8086 имеется 14 шестнадцатиразрядных регистров, которые используются для управления исполнением команд, адресации и выполнения арифметических операций.

Регистры общего назначения AX, BX, CX, DX, SI, DI, BP, SP единообразно участвуют в командах манипуляции данными. При этом с каждым регистром связана некоторая специальная функция, что нашло отражение в названиях регистров.

Регистры данных. У 16-разрядных регистров AX, BX, CX, DX отдельно адресуются старшие (High) и младшие (Low) байты: AH и AL в AX, BH и BL в BX, CH и CL в CX, DH и DL в DX, таким образом регистры данных можно рассматривать

как четыре 16-разрядных или восемь 8-разрядных регистров. Перечислим наиболее характерные функции каждого них:

- AX (AL) – аккумулятор (accumulator) служит для хранения промежуточных данных; многие команды оперируют данными в аккумуляторе несколько быстрее, чем в других регистрах;
- BX – базовый регистр (base) применяется для указания базового (начального) адреса объекта данных в памяти;
- CX (CL) – регистр-счётчик (counter) участвует в качестве счётчика в некоторых командах, которые производят повторяющиеся операции;
- DX – регистр данных (data) используется главным образом для хранения промежуточных данных в качестве расширителя аккумулятора.

Регистры указателей и индексов. В микропроцессоре существуют шесть 16-разрядных регистров, которые могут принимать участие в адресации операндов. Один из них относиться к регистрам данных – это регистр базы BX. Кроме BX в явной адресации участвуют указатель базы BP (base pointer), индекс источника SI (source index) и индекс приемника DI (destination index). Основное назначение этих регистров – хранение значений, используемых при формировании адресов операндов. Разнообразные способы сочетания в командах этих регистров и других величин называются способами (режимами) адресации. Эти регистры можно также использовать и для временного хранения данных.

Два оставшихся регистра SP (stack pointer) и IP (instruction pointer) – используются при адресации неявно.

Регистр указателя стека. Указатель стека SP – это 16-разрядный регистр, который определяет смещение текущей вершины стека. Указатель стека SP вместе с сегментным регистром стека SS используются микропроцессором для формирования физического адреса вершины стека. Регистровая пара SS:SP всегда указывает на текущею вершину стека. Стек растет в направлении младших адресов памяти, т.е. перед тем как слово помещается в стек, содержимое SP уменьшается на 2, после того как слово извлекается из стека, микропроцессор увеличивает содержимое регистра SP на 2.

Регистр указателя команд. Микропроцессор использует регистр указателя команд IP совместно с регистром CS для формирования 20-битового физического адреса очередной выполняемой команды, при этом регистр CS определяет сегмент выполняемой программы, а IP – смещение от начала сегмента. По мере того, как микропроцессор загружает команду из памяти и выполняет её, регистр IP увеличивается на длину команды в байтах. Для непосредственного изменения содержимого регистра IP служат команды перехода.

Регистры сегментов. Имеются четыре регистра сегментов, с помощью которых память можно организовать в виде совокупности четырёх различных сегментов (до 64 Кбайт). Эти 4 различные области памяти могут располагаться практически в любом месте физической памяти. Поскольку местоположение каждого сегмента определяется только содержимым соответствующего регистра сегмента, для реорганизации памяти достаточно всего лишь, изменить это содержимое.

- CS – сегментный регистр кода (code segment) указывает на сегмент, содержащий текущую исполняемую программу. Пара CS:IP определяет адрес команды, которая должна быть выбрана для выполнения;
- DS – сегментный регистр данных (data segment) указывает на текущий сегмент данных;
- SS – сегментный регистр стека (stack segment) указывает на текущий сегмент стека;
- ES – дополнительный сегментный регистр (extended segment) указывает на дополнительную область памяти, используемую для хранения данных.

Регистр флагов. Каждый бит (флаг) 16-разрядного регистра флагов (признаков) имеет свое значение. Некоторые из этих бит, *флаги состояния*, отражают особенности результата команд обработки данных. Другие биты, *флаги управления*, показывают текущее состояние микропроцессора. Микропроцессор 8086 использует только 9 флагов, остальные зарезервированы.

Все флаги младшего байта регистра флагов устанавливаются арифметическими или логическими командами микропроцессора и могут быть опрошены командами условного перехода для изменения порядка выполнения программы. За исключением флага переполнения, все флаги старшего байта отражают состояние микропроцессора и влияют на характер выполнения программы. Флаги состояния и CF устанавливаются и сбрасываются специально предназначенными для этого командами.

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

Биты регистра флагов имеют следующее назначение:

× – зарезервированные биты.

OF – флаг переполнения (overflow) равен 1, если возникает арифметическое переполнение, т.е. когда результат превышает диапазон представимых чисел;

DF – флаг направления (direction) устанавливается в 1 для автоматического декремента и в 0 для автоинкремента индексных регистров после выполнения операции над строками;

IF – флаг прерывания (interrupt). Если IF = 1, то прерывания разрешены. Если же IF = 0, то распознаются лишь немаскируемые прерывания;

TF – флаг трассировки (trace). Если TF равен 1, то процессор переводится в режим покомандной работы, т.е. после выполнения каждой команды генерируется внутреннее прерывание с переходом к отладчику;

SF – флаг знака (size) равен 1, когда старший бит результата равен 1. Т.о. SF=0 для положительных чисел, и 1 для отрицательных;

ZF – флаг нуля (zero) равен 1, если результат равен нулю;

AF – флаг вспомогательного переноса (auxiliary carry). Этот флаг устанавливается в 1, если арифметическая операция вызвала перенос или заём из младшей тетрады;

PF – флаг чётности (parity) устанавливается в 1, если младшие 8 бит результата содержат чётное число единичных бит;

CF – флаг переноса (carry) устанавливается в 1, если имеет место перенос или заём из старшего бита результата.

Режимы адресации

Режимом адресации называется способ указания адреса операнда необходимого для выполнения операции. В зависимости от спецификаций и местоположения источников образования полного (абсолютного) адреса в языке ассемблера различают следующие способы адресации операндов: неявная, регистровая, непосредственная, косвенная (прямая, базовая, индексная, базово-индексная).

При неявной адресации операнд или адрес операнда находится в строго определённом месте (обычно в регистре) и поэтому в команде не указывается. Например, в команде LODSB операндами являются регистры AL, SI и флаг DF (в AL записывается байт с адреса DS:SI, затем смещение в SI увеличивается или уменьшается на 1 в зависимости от флага DF).

При регистровой адресации операнд находится в регистре. Например, в команде MOV **AX**,**BX** для обоих операндов используется регистровая адресация.

При непосредственной адресации операнд находится в самой команде, т.е. хранится вместе с командой в сегменте кода, поэтому изменить операнд нельзя и им может быть только константа. Например, в команде MOV AX, **200** для второго операнда используется непосредственная адресация.

При использовании косвенной адресации абсолютный адрес формируется исходя из базового адреса в одном из сегментных регистров и смещения:

```
MOV     AX, [20]          ; База – в DS, смещение в команде – 20
MOV     AL, [BP +10]      ; База – в SS, смещение – сумма = BP +10
MOV     AX, ES:[SI]       ; База – в ES, смещение – в SI
```

Виды косвенной адресации представлены в таблице:

Косвенная адресация	Формат	Сегмент по умолчанию	Примеры
Прямая	метка, смещение (ofs)	DS DS	Mov [lab], AX Mov [1234h], AX
Базовая	[BX+ofs] [BP+ofs]	DS SS	Mov [BX+1], AX Mov [BP+lab], AX
Индексная	[DI+ofs] [SI+ofs]	DS DS	Mov [DI+12h], AX Mov [SI+lab], AX

Базово-
Индексная
	[BX+SI+ofs]
	[BX+DI+ofs]
	[BP+SI+ofs]
	[BP+DI+ofs]
		DS
		DS
		SS
		SS
			Mov [BX+SI-12h], AX
			Mov [BX+DI], AX
			Mov [BP+SI], AX
			Mov [BP+DI+lab], AX

При косвенной прямой адресации адрес (16-разрядное смещение) операнда находится в команде. Например, в команде MOV AX,[**100h**] для второго операнда используется прямая адресация, т.е. данные находятся в памяти по адресу DS:100h (сегментный регистр DS используется по умолчанию для адресации данных).

В случае применения косвенной базовой адресации исполнительный адрес является суммой значения смещения и содержимого регистра BP или BX, например:

```
MOV     AX,[BP+6]          ; База – SS, смещение – BP+6
MOV     DX,[BX][8]         ; База – DS, смещение – BX+8
```

При косвенной индексной адресации исполнительный адрес определяется как сумма значения указанного смещения и содержимого регистра SI или DI так же, как и при базовой адресации, например:

```
MOV     DX,[SI+5]          ;База – DS, смещение – SI+5
```

Косвенная базово-индексная адресация подразумевает использование для вычисления исполнительного адреса суммы содержимого базового регистра и индексного регистра, а также смещения, находящегося в операторе, например:

```
MOV     BX,[BP][SI]        ; База – SS, смещение – BP+SI
MOV     ES:[BX+DI+6],AX    ; База – ES, смещение – BX+DI+6
```

Формат команд i8086

Машинные команды i8086 состоят из необязательных префиксов, одного или двух байт главного кода операции, возможного спецификатора адреса ModR/M, содержащего байт, определяющий форму адреса и, если требуется, смещения в команде и поля непосредственных данных (рис. 2).

Префикс команды	Префикс замены сегмента	КОП	ModR/M	Смещение в команде	Непосредственный операнд
0 или 1 байт	0 или 1 байт	1 или 2 байта	0 или 1 байт	0,1,2 или 4 байта	0,1,2 или 4 байта

Рис. 2. Общий формат команд МП i8086

Поясним назначение полей машинной команды.

Префиксы. Необязательные элементы. В памяти префиксы предшествуют команде. Назначение префиксов – модифицировать операцию, выполняемую командой.

Префиксы команд:

F3	REP	F0	LOCK
F3	REPZ	F2	REPNZ

Префиксы замены сегментов:

2E	CS	3E	DS
36	SS	26	ES

Префикс замены сегмента в явной форме указывает, какой сегментный регистр используется в данной команде для адресации стека или данных. Префикс отменяет выбор сегментного регистра по умолчанию.

Код операции (КОП). Обязательный элемент, описывающий операцию, выполняемую командой. Следующие поля машинной команды определяют местоположение операндов, участвующих в операции, и особенности их использования.

Байт режима адресации (ModR/M) определяет используемую форму адреса операндов и состоит из трёх полей:

7	6	5	4	3	2	1	0
MOD		REG/КОП			R/M		

Поле MOD байта режима адресации ModR/M объединяется с полем R/M образуя 32 возможных значения: 8 регистров и 24 варианта режимов адресации. Например, если mod=01, то смещение в команде присутствует и занимает 1 байт. Если mod =11, то операндов в памяти нет, они находятся в регистрах.

Поле REG/КОП определяет либо номер регистра, либо дополнительные 3 бита кода операции.

Таблица 1.1

Варианты адресации в байте ModR/M

R/m	mod = 00	mod = 01	mod = 10	mod = 11
000	[BX+SI]	[BX+SI+смещ8]	[BX+SI+смещ16]	AL / AX / ES
001	[BX+DI]	[BX+DI+смещ8]	[BX+DI+смещ16]	CL / CX / CS
010	[BP+SI]	[BP+SI+смещ8]	[BP+SI+смещ16]	DL / DX / SS
011	[BP+DI]	[BP+DI+смещ8]	[BP+DI+смещ16]	BL / BX / DS
100	[SI]	[SI+смещ8]	[SI+смещ16]	AH / SP
101	[DI]	[DI+смещ8]	[DI+смещ16]	CH / BP
110	смещ16	[BP+смещ8]	[BP+смещ16]	DH / SI
111	[BX]	[BX+смещ8]	[BX+смещ16]	BH / DI

Примечания:

смещ8 – означает 8 битное смещение после байта ModR/M;

смещ16 – означает 16 битное смещение после байта ModR/M.

Справочник машинных команд i8086 приведён в Таблице 1 Приложения 1

Столбец **кода операции** в справочнике содержит код операции, формируемый для каждой формы команды. При описании команд используются следующие обозначения:

/цифра – показывает, что байт ModR/M кодирует только операнд r/m, а поле REG/КОП содержит цифру, являющуюся расширением кода операции.

+rb, +rw – код регистра (см. таблицу 1.1), который прибавляется к базовому коду операции с образованием одного байта кода операции.

Смысл поля R/M определяется КОП команды. Поле R/M может определять регистр как место нахождения операнда или служит частью кодирования режима адресации совместно с полем MOD.

В таблице 1.1 приведены значения полей байта ModR/M для различных вариантов адресации.

Рассмотрим примеры построения машинного кода команд.

Пример 1.

Команда	Режимы адресации аргументов	Шаблон из справочника
Mov ah,bl	Mov r8,r8	Mov r8,r/m8 8A
Mov ah,bl	Mov r8,r8	Mov r/m8,r8 88

Воспользуемся шаблоном из первой строки. Как видим, в шаблоне нет конкретных аргументов, значит, нужен байт ModR/M. Заполняем его. Т.к. все аргументы – регистры, то mod = 11; Т.к. в шаблоне второй аргумент r/m, то код второго аргумента в команде (bl=3) записываем в поле R/M, соответственно код первого аргумента (ah=4) запишем в поле REG/КОП:

1	1	1	0	0	0	1	1
MOD		REG/КОП			R/M		

Получили ModR/M = E3

Полный код команды = 8A E3

Но если посмотреть справочник, то есть ещё один шаблон, по которому можно построить код данной команды: Mov r/m8, r8 = 88. В этом случае в R/M запишем код ah, а в REG/КОП – код bl.

1	1	0	1	1	1	0	0
MOD		REG/КОП			R/M		

Получим ModR/M = DC

Полный код команды = 88 DC

Пример 2.

Команда	Режимы адресации аргументов	Шаблон из справочника
Mov byte ptr [bx+6], 24h	Mov m8,im8	Mov r/m8,im8 C6

Заполняем байт ModR/M. Т.к. один из аргументов – содержимое памяти (m8), а смещение может уместиться в 8-разрядах, то mod = 01. В поле R/M кодируем операнд – [bx+смещ8]. Его код = 7. Т.к. в шаблоне второй аргумент не регистр, то поле REG/КОП может принимать любое значение (заполним нулем). Получили ModR/M = 47. Но верными будут и значения 57, 67,77, 4F, 5F, 6F,7F.

0	1	0	0	0	1	1	1
MOD		REG/КОП			R/M		

Код команды = C6 47. Это не правильно. Построенный код не учитывает величину смещения (= 6) и значение непосредственного операнда (= 24_{16}). Исправим. Полный код команды = C6 47 06 24.

Пример 3.

Команда	Режимы адресации аргументов	Шаблон из справочника
NEG dx	neg r16	neg r/m8 F6 /3

В справочнике нет указанной команды с 16-разрядным операндом. Но команда помечена (*), т.е. 16-разрядный аргумент – допустимый операнд в этой команде. Для получения из кода в справочнике (F6) требуемого кода операции установим младший бит в 1. Получили КОП = F7. Найденный в справочнике код (F6) имеет расширение (/3). Расширение записываем в поле REG/КОП байта ModR/M. Остальные поля мы заполнять уже умеем.

1	1	0	1	1	0	1	0
MOD		REG/КОП			R/M		

Полный код команды = F7 DA.

Пример 4.

Команда	Режимы адресации аргументов	Шаблон из справочника
mov dl,24h	mov r8,im8	mov r8,im8 B0+rb

Такая запись кода (B0+rb) означает, что код команды получается сложением базового кода (B0) и кода регистра (dl=2): B0+2=B2. Байт ModR/M не нужен. Полный код команды = B2 24.

Пример 5.

Команда	Режимы адресации аргументов	Шаблон из справочника
jmp met	jmp m	jmp label E9
jmp met	jmp m	jmp short label EB

Все команды условного перехода и short переход занимают 2 байта. Из них первый байт содержит КОП, а второй – значение, равное разности между адресом метки и адресом байта следующего за командой перехода. Рассмотрим переход вперед. В этом случае метка met транслятору ещё не встречалась. Он не знает близким (near) или коротким (short) окажется этот переход, поэтому, рассчитывая на худшее будет строить близкий переход и зарезервирует под команду 3 байта. Затем вычислит адрес метки. Если расстояние от команды до метки окажется меньше 128 байт, то КОП E9 будет заменён на EB, записан однобайтный операнд, и так как, зарезервировано было 3 байта, то третьим байтом будет КОП NOP = 90 (no operation). При переходе назад код операции сразу будет проставлен правильно, т.к. адрес метки уже

известен. Алгоритм расчёта операнда не измениться, но так как адрес метки меньше, чем адрес команды+2, то разность будет отрицательной и будет записана в дополнительном коде.

Если переход NEAR, то расчёты те же, но операнд займет 2 байта.

Машинный код команды косвенного перехода будет содержать байт ModR/M.

Шестнадцатеричный редактор Hiew.

Hacker's Viewer – это шестнадцатеричный редактор, дизассемблер и ассемблер. Он позволяет просматривать файлы неограниченной длины в текстовом и шестнадцатеричном форматах, а также в режиме дизассемблера процессора 80x86. Основные возможности программы: редактирование файлов в шестнадцатеричном режиме и в режиме дизассемблера; поиск и замена в блоке; встроенный ассемблер; поиск ассемблерных команд по шаблону; поддержка различных форматов исполняемых файлов: MZ, NE, LE, LX, PE.

По умолчанию программа работает в режиме просмотра текста. Воспользуйтесь F4 для смены режима на Text, Hex, Code. Каждый из режимов поддерживает свой спектр возможностей.

ЗАДАНИЕ

1. Записать программу в машинных кодах.
- Изучить последовательность команд в мнемонических обозначениях согласно варианту (Таблица 1.2), написать, какие действия выполняет каждая команда. Указать, какие режимы адресации используются в каждой команде.
- Построить машинный код для команд своего варианта, используя справочник (Таблица 1.3, 1.4), с объяснением хода построения (см. примеры выше).

2. Ввести программу в машинных кодах.
- Создать новый файл с расширением com (в NC / FAR / WinCmd нажать <Shift> + <F4> и ввести имя файла с расширением com).
- Вызвать Hiew.exe с параметром: Hiew.exe имя_файла.com (или в NC / FAR / WinCmd установить курсор на hiew.exe нажать <Ctrl> + <Enter>, затем на com-файл, снова <Ctrl> + <Enter> и потом <Enter>).
- Выбрать режим HEX (<F4>,<F2>).
- Перейти в режим редактирования (<F3>).
- Ввести построенный в п.1 задания машинный код.
- Сохранить результат работы (<F9>).
- Посмотреть дизассемблированные команды (<F4>, <F3>), проверить соответствие полученных команд заданным.

3. Ввести программу в мнемонических обозначениях.

– Используя Hiew.exe в режиме Decode → Asm ввести следующую программу:

1) MOV BX,110
2) MOV AX,[BX]
3) ADD AX,[BX+2]
4) MOV SI,4
5) MOV [BX+SI],AX
6) NOP
7) INT 20

– Перейти в режим HEX и ввести данные: 2301 2500 0000.
– Сохранить программу.
– Написать, что делает эта программа.
– Перечислить использованные в программе режимы адресации.
– Просмотреть машинные коды этой программы и содержимое области данных.

СОДЕРЖАНИЕ ОТЧЁТА

– Тема и цель работы; задание на лабораторную работу (свой вариант).
– Ход выполнения работы:
  – для каждой строки задания своего варианта указать команду, режимы адресации, описание действий выполняемых командой;
  – для каждой команды привести машинные коды с описанием их построения;
  – объяснить назначение двух последних команд задания и привести примеры результата выполнения этих команд для конкретных значений операндов.
– Выводы о проделанной работе.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как формируется физический адрес у МП i8086?
2. Как разделяются регистры МП i8086 по функциональному признаку?
3. Перечислите сегментные регистры и укажите их назначение?
4. Какие регистры могут использоваться при адресации явно (неявно)?
5. Какие режимы адресации существуют в МП i8086?
6. Какие поля может содержать машинная команда?
7. Какие режимы ввода данных доступны в Hiew?
8. Как осуществляется ввод программы в машинных кодах в Hiew?

ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

Таблица 1.2

Задания по вариантам

№ варианта	Последовательность команд	№ варианта	Последовательность команд
1	xchg ax,di not byte ptr [bx+800h] cmp bl,bh rep stosb jnz $+5 shl ax,1 xor ah, 61h	9	dec dl push word ptr [bx+111h] test cl,dl std ja $-20h shr ax,cl and ch,40h
2	inc cx pop word ptr [bx+si+76h] add ah,ch cbw jbe $-5 ror al,cl or cl,5fh	10	push bx neg byte ptr [bp+di+222h] xchg al,bl lodsw jz $+60h rol dl,1 xor dl,21h
3	dec bp mul byte ptr [19h] adc dh,cl clc jmp $+161h rcr bx,1 and bl,5fh	11	pop bp not cl xor [bx+di],dx scasb jnc $-10 ror bh,cl or dl,5Ah
4	push di div byte ptr [bp+si] sub ch,dh cmc jg $+25 rcl dx,cl xor dh,35h	12	mov dh,44h neg ah and [di],cx clc jns $-16 rcl bl,1 or cl,66h

Продолжение таблицы 1.2

№ варианта	Последовательность команд	№ варианта	Последовательность команд
5	pop si imul byte ptr [di+64h] sbb bl,cl cwd loop $-10 shl cx,1 or dh, 20h	13	xchg al,ch mul cx test [bp+41h], di stc jmp [100h] rol ch,1 and cl,7Eh

6	mov dx,20h
	idiv byte ptr es: [bp+di]
	cmp ah,dl
	cld
	jmp [bx]
	shr bx,cl
	and bl,31h
		14	inc ah
			div di
			or [bx+di+0A4Ch],bp
			lodsb
			jle $-30h
			sar ah,cl
			xor dh,27h

7	xchg al,dl inc word ptr [si+324h] mov dh,al xlat jc $-12 sar dl,1 xor dl,55h	15	dec bl imul si xor cx,[si+63h] stc jb $-40h rcl cl,1 and bh,78h
8	inc ch dec word ptr [bp+si] test ch,dl stc jmp $+23h rcl dx,cl or cx,80h	16	inc di pop bx mov cs:[20h],ah xlat jmp [bx+si] shl cx,1 xor dl,73h

Лабораторная работа № 2
ПРОЦЕСС СОЗДАНИЯ И ОТЛАДКИ ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА

Цель работы: Знакомство с методами создания и отладки программ, написанных на языке ассемблера.

ДОМАШНЯЯ ПОДГОТОВКА

- Изучить технологию разработки ассемблерных программ.
- Изучить основные форматы исполняемых файлов DOS, порядок их загрузки на исполнение.
- Подготовить ответы на контрольные вопросы.

ЗАДАЧИ

- Изучить способы ассемблирования и создания исполняемого файла с помощью программ Turbo Assembler (TASM.EXE) и Turbo Link (TLINK.EXE).
- Познакомиться с командами и интерфейсом отладчика Turbo Debbuger (TD.EXE), научиться трассировать и исправлять программы.
- Создать программу на языке ассемблера выполняющую арифметическую операцию и ввод/вывод с консоли.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Технология создания программ на языке ассемблера

Рассмотрим процесс создания программ на языке ассемблера с использованием пакета Turbo Assembler фирмы Borland International.

1. Создание исходного текста программы в операторах языка ассемблер (.asm-файл): на данном этапе можно использовать любой текстовый редактор, позволяющий создать ASCII-файл. Исходный текст программы на языке ассемблера представляет собой обычный текстовый файл, содержащий операторы и директивы языка ассемблер.

Важно понимать, что программы пакета Turbo Assembler – это DOS-приложения, поэтому к именам файлов и каталогов предъявляются требования этой ОС: в имени и расширении не должно быть русских букв и пробелов, длина имени не более 8 символов, расширения не более трёх.

2. Трансляция операторов языка в машинные коды ЭВМ. Для этого используется программа TASM, которая осуществляет трансляцию исходного текста в машинные коды и генерацию объектного модуля. В результате получается модуль (объектный файл, .obj) а, при необходимости, файл с листингом программы (.lst) и файл перекрёстных ссылок (.crf). В процессе трансляции TASM создаёт таблицу идентификаторов, которую можно представить в виде перекрёстных ссылок на метки, идентификаторы и переменные в программе. Посмотрите последние строки (Symbol Table)

вашего файла листинга. Число в формате n# указывает на номер строки в листинге программы, где определён соответствующий идентификатор. Остальные числа в этой строке, указывают на номера строк, в которых используется этот идентификатор.

3. Компоновка, создание исполняемого файла (.exe или .com). Используется программа TLINK. При помощи компоновщика можно объединить несколько отдельно оттранслированных исходных модулей в один исполняемый файл.

4. При необходимости может быть выполнена трассировка полученной программы с целью поиска алгоритмических ошибок при помощи отладчика программ TD.

5. Эксплуатация. Программа загружается на исполнение с помощью стандартных средств операционной системы.

Основные форматы исполняемых файлов в MSDOS

Формат COM

Программы типа .com хранятся на диске в виде точного образа программы в памяти, поэтому .com программы ограничены единственным сегментом, то есть машинный код, данные и стек в сумме могут занимать в памяти не больше 64 Кб. Так как .com файлы не содержат никакой настроечной информации, то они компактнее эквивалентных .exe файлов, и загружаются для выполнения немного быстрее. Отметим, что DOS не пытается выяснить действительно ли файл с расширением .com содержит исполняемую программу. Программы типа .com загружаются непосредственно за префиксом сегмента программы (PSP) и, кроме того, не имеют заголовка, который может задавать другую точку входа, поэтому их начальный адрес всегда составляет 100h, что определено размером PSP = 256 (100h) байт. Максимальная длина .com- программы = 64k минус обязательное слово стека минус 256 байт PSP.

Рассмотрим подробнее, как DOS загружает .com программу:

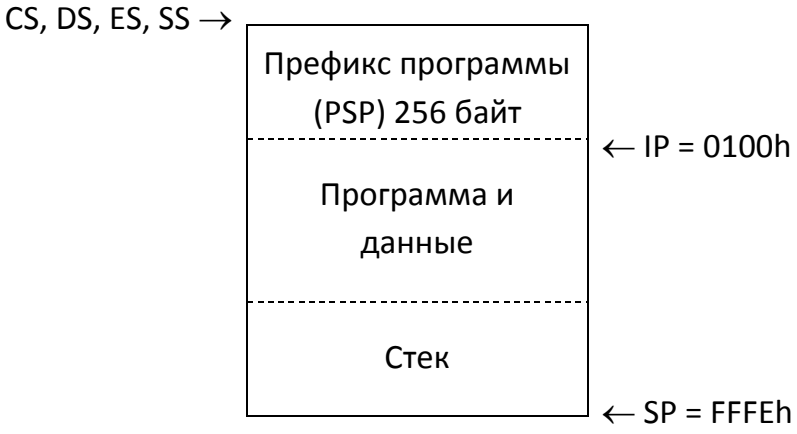


Рис. 3. Образ памяти программы .COM

- создает PSP (256-байтную рабочую область в начале программы, содержащую важную для DOS информацию. Здесь же, со смещением 80h, находится «хвост» командной строки);
- целиком копирует .com-файл в память непосредственно за PSP;
- устанавливает все 4 сегментных регистра на начало PSP;
- устанавливает значение регистра IP в 100h, а регистр SP в FFFEh – на конец текущего сегмента.

Формат EXE

В настоящее время существуют несколько форматов исполняемых файлов с расширением EXE:

MZ – формат программы с 16-разрядным кодом для ОС MSDOS;

LX, NE, LE – форматы программ с 16-разрядным кодом для ОС Windows (в настоящее время практически не применяются);

PE – формат программ с 32-разрядным кодом для ОС Windows.

Программа для DOS формата MZ EXE содержит заголовок, занимающий не менее 512 байт: перемещаемую точку входа, начальные значения CS:IP, SS:SP, объём необходимой для загрузки программы памяти, таблицу перемещений (для настройки абсолютных ссылок после загрузки программы). Exe-программа имеет по крайней мере 2 сегмента: сегмент кода и сегмент стека.

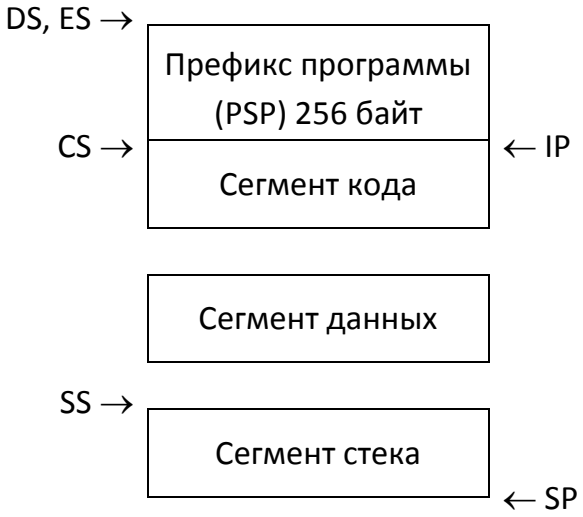


Рис. 4. Образ памяти программы .EXE

Этапы выполняемые DOS для загрузки .exe программы:

- создаёт PSP;
- из заголовка .exe-файла определяет откуда начинается программа и загружает её после PSP;
- находит и исправляет все ссылки в программе, значение которых зависит от физического адреса, начиная с которого будет размещён сегмент;
- устанавливает значения регистров DS и ES так, чтобы они указывали на начало PSP;

– устанавливает CS на начало сегмента кода, значения IP и SS:SP устанавливаются исходя из информации заголовка, и указывают на первую исполняемую команду программы и вершину стека соответственно.

Сервисные функции MS DOS

В любой программе для ЭВМ должен быть предусмотрен ввод исходных данных и вывод результирующих. Для этого используются различные устройства ввода/вывода: клавиатура, видеокарта, диски и т.д. При создании программ на языке ассемблера МП i8086 связь с устройствами ввода/вывода осуществляется через адресное пространство ввода/вывода, а также через области памяти связанные с устройствами ввода/вывода. Так как каждое устройство имеет свой, обычно достаточно сложный, формат (протокол) передачи данных, программирование обмена данными с устройствами на языке ассемблера требует от программиста знание принципа работы устройства и протокола обмена.

Для облегчения задачи обмена данными с устройствами ввода/вывода, а также для выполнения некоторых других задач ОС MSDOS предоставляет специальные подпрограммы – сервисные функции, которые можно вызывать из программы выполняющейся под управлением ОС.

Для того чтобы программисту не требовалось знать конкретные адреса расположения сервисных подпрограмм в памяти (которые могут меняться от версии к версии), вызов сервисных подпрограмм осуществляется через программные прерывания.

Вызов программного прерывания осуществляется командой процессора INT n, где n – номер прерывания. Перед вызовом прерывания, согласно формату вызова сервисной функции, в регистры требуется занести данные необходимые для выполнения запрашиваемой операции.

Ниже приведены некоторые прерывания сервисных функций BIOS и MS DOS:

INT 10h – программы BIOS управления видеосистемой.

INT 13h – программы BIOS управления дисками.

INT 16h – программы BIOS управления клавиатурой.

INT 20h – завершение com-программы DOS.

INT 21h – диспетчер сервисных функций MS DOS (в регистре AH при вызове должен быть номер запрашиваемой функции).

Сервисные функции, необходимые для выполнения лабораторной работы, и формат их вызова приведены в приложении 3.

Транслятор Turbo Assembler.

Программа TASM.EXE осуществляет трансляцию программы на языке ассемблера, представленной в виде текстового файла, в машинные коды. Для того, чтобы отличать просто текстовый файл от программы на языке ассемблера, используется специальное расширение .asm (для текстовых файлов обычно используется расширение .txt).

Результатом работы программы является модуль (объектный файл, .obj) и, если необходимо, файл с листингом программы (.lst) и файл перекрёстных ссылок (.crf).

Формат командной строки транслятора TASM:

TASM [ключи_опций] имя_исходного_файла [, имя_объектного_файла [,имя_листинга [,имя_файла_перекрёстных ссылок]]]

Аргументы в квадратных скобках – необязательные параметры.

Ключи опций программы TASM приведены в Приложении 2.

Примеры использования:

tasm.exe prog1.asm

Транслирует программу prog1.asm, результат – модуль prog1.obj.

tasm prog2, progR.obj, prog2.lst, prog2.crf

Транслирует программу prog2.asm, результат – модуль progR.obj, листинг prog2.lst, файл перекрёстных ссылок prog2.crf.

tasm.exe /zi prog3.txt, prog3.obj, prog3.lst

Транслирует программу prog3.txt, результат – модуль prog3.obj с информацией для отладчика (ключ /zi) и листинг prog3.lst.

Компоновщик Turbo Link.

Программа TLINK.EXE осуществляет связывание модулей, полученных с помощью TASM, и создание исполняемого файла.

Формат командной строки компоновщика TLINK:

TLINK [ключи] список_obj_файлов [,имя_исполняемого_файла [,имя_map_файла [,имя_lib_файла [,имя_def_файла [,имя_res_файла]]]]]

Ключи опций программы TLINK приведены в Приложении 2

Примеры использования:

tlink.exe prog1.obj

Результат – исполняемый файл prog1.exe.

tlink /t prog2

Результат – исполняемый файл prog2.com.

tlink.exe modul1.obj+modul2.obj, prog.exe, prog.map

Результат – исполняемый файл prog.exe, файл карты размещения prog.map.

Отладчик Turbo Debbuger.

Turbo Debugger – отладчик программ в машинных кодах из пакета TASM Borland International. Основные возможности: выполнение трассировки программы в прямом и обратном направлении; просмотр и изменение регистров и содержимого ячеек памяти во время покомандного выполнения программы; поддержка точек останова. TD не имеет встроенного редактора текстов, и не может перекомпилировать вашу программу. Внесённые в код во время работы изменения не будут сохранены на диске. Для внесения изменений в программу в машинных кодах необходимо использовать шестнадцатеричный редактор (например Hiew).

Формат командной строки отладчика TD:

TD [имя_исполняемого_файла]

где необязательный параметр имя_исполняемого_файла – исполняемый файл, который загружается в отладчик для трассировки.

Отладчик TD имеет текстовый псевдографический оконный интерфейс.

Основные элементы интерфейса:

Окно дизассемблера (1) – предназначено для отображения программы в памяти: для каждой команды указан адрес памяти, машинный код и немоническое обозначение. Специальный указатель ► указывает на текущую исполняемую команду. В начале указатель установлен на первую исполняемую команду программы.

Окно шестнадцатеричного просмотра (дампа) (2) – предназначено для отображения участка памяти в шестнадцатеричном виде, обычно его настраивают на сегмент данных программы.

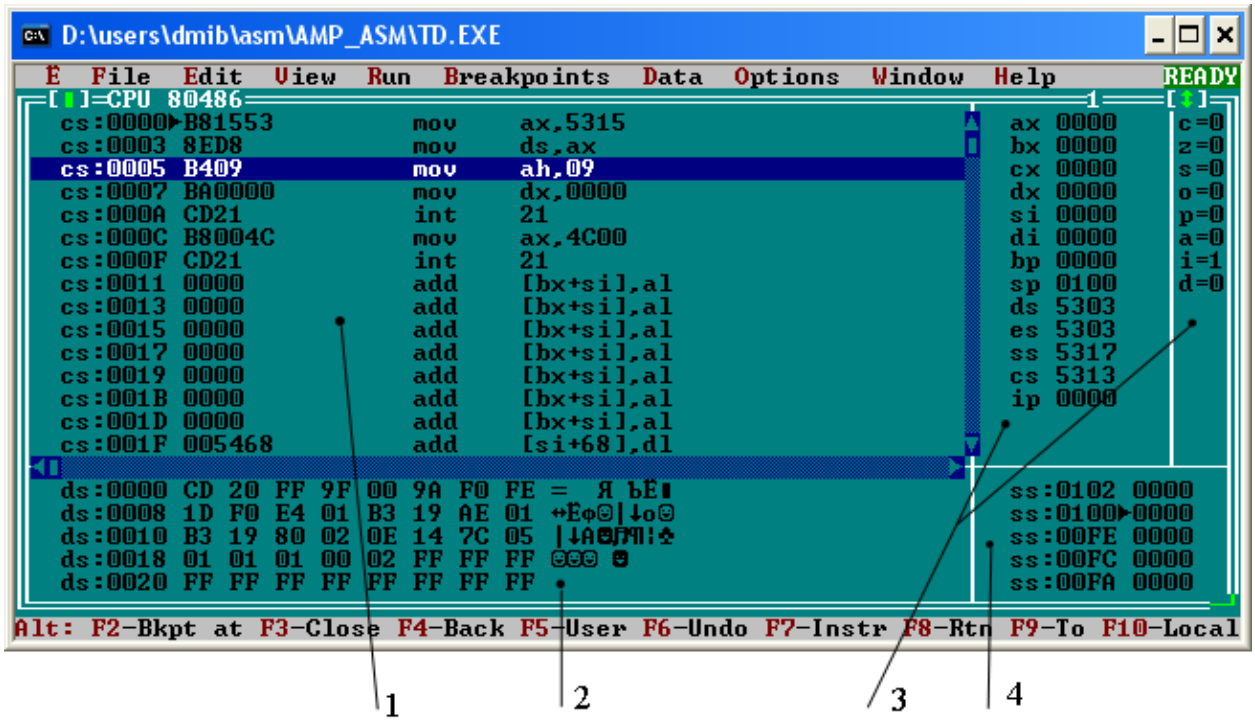


Рис. 5. Окно отладчика TD

Окно регистров и окно регистра флагов (3) – предназначено для просмотра регистров процессора в ходе выполнения программы и, при необходимости, внесения изменений в хранимые значения;

Окно стека (4) – предназначено для отображения стека программы: указывается адрес и значение, хранимое в стеке. Специальный указатель ► указывает на вершину стека.

Основные клавиши управления:

<TAB>, <Shift+TAB> – переключение между окнами отладчика;

<F4> – выполнение программы до курсора (выделенной строки) в окне дизассемблера;

<F5>, <Ctrl>+<F5> – изменение размеров окна

<F7> – выполнение одной инструкции с входом в подпрограммы;

<F8> – выполнение одной инструкции без входа в подпрограммы;

<F9> – запуск программы на исполнение;

<F10> – выход в меню;

<Ctrl>+<F2> – сброс режима трассировки (перезагрузка программы);

<Alt>+<F5> – переключение между отладчиком и окном отлаживаемой программы (пользовательский экран);

<Ctrl>+<G> – переход на заданный адрес памяти в окнах дизассемблера, дампа и стека.

<Ctrl>+<Z> – выход из программы TD;

ЗАДАНИЕ

1. Создать программу формата .exe на языке ассемблера.

В текстовом редакторе наберите приведённый в примере текст программы на языке ассемблера. Сохраните его в файл first.asm;

Посмотрите как набран текст примера: отступы в начале строки облегчают чтение текста, поиск меток. Между мнемоникой и операндами и перед комментариями тоже обычно делают отступ.

Обратите внимание на имена сегментов: они произвольные и смысл в названии – только для пользователя. Назначение сегмента указано в директиве ASSUME.

```
begin segment            ; Сегмент кода программы
assume cs: begin, ds:dates, ss: komod
start:
     mov  ax, dates    ; Настройка DS на начало сегмента данных
     mov  ds, ax
     mov  ah, 9          ; Функция 9 сервиса DOS:
     mov  dx, offset Msg ; вывод строки с указанного
     int  21h            ; в ds:dx адреса до символа '$'
```

```
     mov  ax, 4c00h            ; Завершение программы с кодом 0
     int  21h
beginends
datessegment                   ; Сегмент данных
Msg  db  'Hello! ', 13,10,'$'
datesends
komodsegment   stack           ; Сегмент стека
dw  128  dup (?)               ; под стек отводится 128 слов
komodends
     end  Start
```

Оттранслируйте исходный текст и скомпонуйте из него исполняемый .exe файл: TASM first,,

TLINK first,,

Выполните полученный EXE - файл.

2. Создать программу формата .com на языке ассемблера.

Чтобы получить программу в формате .com необходимо внести в текст программы небольшие изменения (сгруппировать все сегменты в один):

Пример .com программы: Способ 1

```
begin segment
     org 100h
assume cs:begin, ds:begin
start: mov  dx, offset Msg
     mov  ah,9
     int  21h
     int  20h


Msg  db   'Hello!',13,10,'$'
Begin ends
      end start
```

Пример .com программы: Способ 2

```
MyGrp group begin, dates
begin segment
    org 100h
assume cs: MyGrp, ds: MyGrp
start: mov dx, offset mygrp: Msg
    mov  ah,9
    int  21h
    int  20h
begin  ends
dates  segment
Msg    db     'Hello!', 13,10,'$'
Dates  ends
       end start
```

Внесите указанные изменения и сформируйте .com – файлы.

3. Изучить режимы выполнения программы в TD.

В текстовом редакторе наберите следующий исходный текст программы на языке ассемблера. Сформируйте .exe – файл.

```
code    segment
assume cs: code, ds:data, ss: stek
start:  mov     ax, data
        mov     ds, ax
        mov     ax,word ptr [X]         ; первая часть
        mov     bx,[Y]
        add     ax,bx
        mov     [Result],ax
        mov     ax,[Y+2]
        adc     ax,word ptr [X+2]
        mov     [Result+2],ax
        mov     ah,1                    ; вторая часть
        int     21h
        mov     [Chr],al
        mov     ah, 9
        mov     dx, offset Msg
        int     21h
        mov     ax, 4c00h
        int     21h
code    ends
data    Segment
X       dd      123456h             ; число 123456h
Y       dw      0FF88h,0077h        ; число 77FF88h
Result  dw      0,0                 ; результат
Msg     db      13,10,'Enter:'
Chr     db      '0', '$'
data    ends
stek    segment stack
        dw      128 dup (?)
stek    ends
end     Start
```

Запустите TD с именем полученной программы в командной строке.

Выполните трассировку программы под управлением отладчика, отслеживая изменения значений регистров и данных, выясните, что делает программа в первой и во второй части;

4. Согласно варианту (Таблица 2.1) написать программу, по примеру в пункте 3:

- выполняющую заданную арифметическую операцию;
- выполняющую заданную операцию ввода/вывода.

СОДЕРЖАНИЕ ОТЧЁТА

- Тема и цель работы, задание на лабораторную работу (свой вариант)
- Ход выполнения работы, включая:
  - Описание процесса создания программ формата .EXE и .COM при помощи языка ассемблера;
  - Описание процесса трассировки программы-примера (пункт 3 задания), изменения значений регистров и данных в процессе трассировки программы.
  - Листинг (lst-файл) разработанной программы (пункт 4 задания) с комментариями на каждой строке.
  - Выводы о проделанной работе.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Из каких шагов состоит процесс создания программ на языке ассемблера?
2. Чем отличаются .com и .exe программы (структура, размер)?
3. Что собой представляет объектный файл (модуль)?
4. Для чего предназначена программа TASM?
5. Какая программа создает исполняемый файл?
6. Что такое сервисные функции и как они вызываются?
7. Какие действия можно выполнить при помощи программы TD?
8. Какие главные элементы интерфейса программы TD?
9. Какие функции MSDOS осуществляют вывод на экран?
10. Какие функции MSDOS осуществляют ввод с клавиатуры?
11. Сколько байт памяти занимают переменные X и Y из п.3 задания? Как (какими командами) осуществляется обращение к памяти поименованной X и Y в программе? Чем отличаются методы доступа к этой памяти? Какой из методов вам понятнее? Приведите примеры ситуаций (задач), когда может понадобиться первый и второй метод обращений к данным в памяти.

ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

Таблица 2.1

Задания по вариантам

№ варианта	Арифметическая операция	Операция ввода вывода	№ варианта	Арифметическая операция	Операция ввода вывода
1.	Res=X+Y x,y – трехбайтные	Ввод символа	9.	Res=X-Y x,y – трехбайтные	Вывод строки
2.	Res=X-Y x,y –четырехбайтные	Вывод символа	10.	Res=X+Y+Z x,y,z – четырехбайтные	Ввод символа
3.	Res=X+Y+Z x,y – однобайтные, z – двухбайтное	Вывод строки	11.	Res=X+Y+Z x – однобайтное, y,z – двухбайтные	Вывод символа
4.	Res=X+Y+Z x,y – двухбайтные, z – четырехбайтное	Ввод символа	12.	Res=X+Y+Z x – двухбайтное, y,z – четырехбайтные	Вывод строки
5.	Res=X+Y+Z x,y – однобайтные, z – четырехбайтное	Вывод символа	13.	Res=X+Y+Z x – однобайтное, y,z – четырехбайтные	Ввод символа
6.	Res=X-Y+Z x,y – однобайтные, z – двухбайтное	Вывод строки	14.	Res=X-Y+Z x – однобайтное, y,z – двухбайтные	Вывод символа
7.	Res=X-Y+Z x,y – двухбайтные, z – четырехбайтное	Ввод символа	15.	Res=X-Y+Z x – двухбайтное, y,z – четырехбайтные	Вывод строки
8.	Res=X-Y+Z x,y – однобайтные, z – четырехбайтное	Вывод символа	16.	Res=X-Y+Z x – однобайтное, y,z – четырехбайтные	Ввод символа

Лабораторная работа № 3
ОРГАНИЗАЦИЯ ПОДПРОГРАММ НА ЯЗЫКЕ АССЕМБЛЕРА

Цель работы: Изучение модульного программирования на языке ассемблера.

ДОМАШНЯЯ ПОДГОТОВКА

- Изучить способы организации подпрограмм на языке ассемблера.
- Изучить способы передачи аргументов подпрограмме.
- Изучить безусловные и условные команды передачи управления.
- Подготовить ответы на контрольные вопросы.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Организация ассемблерных подпрограмм.

Декомпозиция задачи на отдельные программные модули упрощает её отладку, тестирование и модификацию.

Подпрограмма (процедура или функция) представляет собой группу команд для решения некоторой подзадачи и обладает средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку. Группу команд, образующих подпрограмму, можно никак не выделять в тексте программы. Однако в ассемблере принято оформлять подпрограммы специальным образом:

```
имя_подпрограмы PROC [модификатор]
  тело_подпрограммы
имя_подпрограмы ENDP
```

Директивы PROC и ENDP определяют соответственно начало и конец подпрограммы и должны начинаться одним и тем же именем. Необязательный параметр модификатор может принимать значения FAR или NEAR и характеризует возможность обращения к подпрограмме из другого сегмента кода. Этот параметр определяет тип всех команд RET в блоке. По умолчанию модификатор принимает значение NEAR. Переход к подпрограмме называется вызовом и выполняется командой CALL. Команда вызова подпрограммы CALL передаёт управление с автоматическим сохранением в стеке адреса возврата (адреса команды, находящейся после команды CALL). При внутрисегментном вызове подпрограммы (NEAR) в стеке сохраняется как адрес возврата два байта: только содержимое IP. В случае межсегментного вызова (FAR) в стеке необходимо сохранить четыре байта: содержимое регистровой пары CS:IP. Имя подпрограммы считается меткой и её можно указывать в командах перехода.

Подпрограмма может размещаться в любом месте сегмента кода программы, но так, чтобы на неё случайным образом не попало управление или в другом модуле (файле). Для доступа к подпрограмме из другого модуля её имя должно быть «видимо». Для этого в модуле с подпрограммой имя процедуры должно быть объявлено глобальным:

PUBLIC имя [,имя...].

В модуле с основной программой нужно оповещение об использовании внешнего имени и информация о дальности перехода:

EXTRN имя:тип [,имя:тип...].

Аргументы подпрограммы – это данные, которые требуются для выполнения возложенных на модуль функций и расположенные вне этого модуля. Аргументы подпрограмме можно передать по ссылке или по значению; в регистрах, стеке, глобальных переменных.

При передаче аргументов по ссылке подпрограмме сообщают адрес, по которому можно взять данные. При передаче по значению подпрограмма, в заранее оговоренном месте, получает копию аргумента; оригинал остаётся недоступным и не изменяется.

Старайтесь не передавать аргументы подпрограмме через глобальные переменные. Это делает подпрограмму нереентерабельной.

Если подпрограмма получает небольшое число аргументов, то регистры – идеальное место для их передачи. В регистрах можно вернуть и результаты работы подпрограммы. Примерами служат практически все вызовы прерываний MS DOS и BIOS.

Другой способ – передавать параметры через стек. В этом случае основная программа записывает фактические параметры в стек и вызывает подпрограмму; подпрограмма работает с параметрами и, возвращая управление, очищает стек. Этот способ используется трансляторами с языков высокого уровня. Для чтения параметров из стека обычно используют регистр BP, в который помещают адрес вершины стека после входа в подпрограмму. Удалять параметры из стека должна или подпрограмма командой “RET число_байт” или вызывающая программа. Первый способ короче, более строгий и используется при реализации подпрограмм в языке Pascal. Но если за освобождение стека отвечает вызывающая программа, то становится возможным последовательными командами CALL вызывать несколько подпрограмм с одними и теми же параметрами. Этот способ дает больше возможностей для оптимизации и используется в языке Си [1].

Пример подпрограммы вычисления n! (факториала числа n).

```
public  Fact                ; Дано в BX n
code    segment             ; Результат в AX
assume  cs:code             ; Портит DX
FACT    proc    Near
        CMP     BX,1
        JA      REPEAT
        MOV     AX,1
        RET
REPEAT:
        PUSH    BX
        DEC     BX
        CALL    FACT
        POP     BX
        MUL     BX
        RET
        endp    FACT
code    ends
        End
```

Фрагмент программы, вызывающей подпрограмму Fact

```
EXTRN   Fact:Near
Code    segment
assume  cs:code, ss:stek
start:  ...
        MOV     BX,4
        CALL    FACT            ; Вычислить 4!
        ...
Code    ends
        ...
        End     Start
```

ЗАДАНИЕ

1. Разработать **подпрограмму** проверки условия (по варианту задания из Таблицы 3.1), аргументы подпрограмме передать заданным способом.
2. Разработать **подпрограммы** ввода и вывода 16-ти разрядного числа в десятичной системе счисления (диапазон 0..65535 или -32767..32768 в соответствии с вариантом). Пример приведён в Приложении 4.
3. Разработать COM или EXE-программу, выполняющую ввод чисел, проверку условия и вывод результата, использующую созданные подпрограммы. Подпрограммы и основная программа должны находиться в разных файлах.

СОДЕРЖАНИЕ ОТЧЁТА

Тема и цель работы; задание на лабораторную работу (свой вариант). Текст разработанной программы на языке ассемблера (основной программы и модулей) с комментариями на каждой строке. Выводы из проделанной работы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как оформляется подпрограмма на языке ассемблера?
2. Какая разница между типом FAR и типом NEAR в директиве PROC?
3. Для чего используются директивы EXTRN и PUBLIC?
4. Как осуществляется вызов и возврат из подпрограммы?
5. Какие существуют способы передачи аргументов подпрограмме?
6. Можно ли перейти на подпрограмму, используя команду JMP?
7. Как осуществить сборку многомодульной программы?
8. Какие команды могут использоваться для проверки условий если регистры AX и BX содержат знаковые данные, а CX и DX - беззнаковые:

   а) значение в DX больше, чем в CX?

   б) значение в BX меньше, чем в AX?

   в) CX содержит нуль?

   г) значение в BX равно или больше, чем в AX?

   д) значение в DX равно или меньше, чем в CX?
9. Какие флаги воздействуют следующие события, какое значение этих флагов, и какие команды выполняют переход по данному событию: а) произошло переполнение; б) результат отрицательный; в) результат нулевой.
10. Почему для сравнения чисел со знаком и чисел без знака применяют разные команды?

ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

Таблица 3.1

Задания по вариантам

№ варианта	Операция	Передача параметров	№ варианта	Операция	Передача параметров
1	Ввод: x, y Вывод: (x>y) = (Да/Нет)	Через регистры по ссылке	9	Ввод: x Вывод: x<1000 = (Да/Нет)	Через регистры по ссылке
2	Ввод: x Вывод: (x=0) = (Да/Нет)	Через стек по ссылке	10	Ввод: x Вывод: x>1000 = (Да/Нет)	Через стек по ссылке
3	Ввод: x, y Вывод: Max(x,y)	Через регистры по значению	11	Ввод: x Вывод: x<-1000 = (Да/Нет)	Через регистры по значению
4	Ввод: x, y Вывод: Min(x,y)	Через стек по значению	12	Ввод: x Вывод: x>-1000 = (Да/Нет)	Через стек по значению
5	Ввод: x Вывод: (x<0) = (Да/Нет)	Через регистры по ссылке	13	Ввод: x, y Вывод: x+y>80 = (Да/Нет)	Через регистры по ссылке
6	Ввод: x Вывод: (x>0) = (Да/Нет)	Через стек по ссылке	14	Ввод: x, y Вывод: x+y=100 = (Да/Нет)	Через стек по ссылке
7	Ввод: x, y Вывод: (x and y) = 0 = (Да/Нет)	Через регистры по значению	15	Ввод: x, y Вывод: x-y=0 = (Да/Нет)	Через регистры по значению
8	Ввод: x, y Вывод: (x or y) =0 = (Да/Нет)	Через стек по значению	16	Ввод: X, Y Вывод: (x xor y) =0 = (Да/Нет)	Через стек по значению

Лабораторная работа № 4
ИНСТРУКЦИИ ОБРАБОТКИ ЦЕПОЧЕК

Цель работы: Изучить команды обработки цепочек процессора i8086.

ДОМАШНЯЯ ПОДГОТОВКА

- Изучить назначение и особенности применения команд обработки цепочек.
- Изучить префиксы повторений.
- Подготовить ответы на контрольные вопросы.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Команды обработки цепочек позволяют производить действия над областью памяти длинной до 64 Кбайт и предоставляют возможность выполнения пяти основных операций, которые обрабатывают строку по одному элементу (байту или слову) за прием.

К командам обработки цепочек относятся следующие команды:

MOVS – копировать строку байт (слов).

LODS – загрузить (прочитать) байт (слово) из строки в AL (AX).

STOS – сохранить (записать) байт (слово) из AL (AX) в строку

CMPS – сравнить строки байт (слов).

SCAS – найти байт (слово) в строке .

Все перечисленные команды, выполняя различные действия, подчиняются одинаковым правилам.

По умолчанию строка-приёмник находится в дополнительном сегменте и адресуется через ES:DI, а строка-источник – в сегменте данных и адресуется через DS:SI. С помощью префикса замены сегмента в операнде-источнике можно использовать другой сегмент, но нельзя заменить сегмент, к которому адресуется регистр DI.

Так как команды обработки цепочек предназначены для действий над группой элементов, то они автоматически модифицируют регистры SI и DI для адресации следующего элемента строки. Например, команда MOVS увеличивает или уменьшает значение индексных регистров SI и DI после каждого цикла своего исполнения на 1 (MOVSB) или на 2 (MOVSW).

В регистре флагов флаг направления DF определяет, будут значения регистров SI и DI увеличены или уменьшены по завершении выполнения команды манипулирования строками. Если флаг DF равен 0, то значения регистров SI и DI увеличиваются после исполнения каждой команды; иначе они уменьшаются. Состоянием флага DF можно управлять с помощью команды **CLD** (clear direction flag – сбросить флаг направления), которая полагает его равным

нулю, и **STD** (установить флаг направления), которая присваивает ему значение 1 [4].

Префиксы повторения.

Можно сделать так, чтобы одна команда обработки строк работала с группой последовательных элементов памяти. Для этого перед ней надо указать префикс повторения. Это не команда, а однобайтный модификатор, который приведет к аппаратному повторению команды обработки строк, что сокращает время на обработку длинных строк по сравнению с программно-организованными циклами. Префикс повторения указывается перед цепочечной командой в поле метки. Число повторений извлекается из регистра CX. На каждом шаге значение CX уменьшается на 1.

REP	повторять строковую операцию пока CX≠0
REPE/REPZ	операция повторяется пока (CX≠0 и ZF=1)
REPNE/REPNZ	операция повторяется пока (CX≠0 и ZF=0).

С помощью префикса REP даётся указание повторять команду, пока значение регистра CX не станет равным нулю. Но вначале выполняется команда, затем вычитается 1 и только потом – выполняется проверка (CX=0)?. Таким образом если CX=0 до выполнения строковой команды с префиксом, то команда будет выполнена не 0 а 65536 раз. Если этого надо избежать, то проверьте CX на равенство нулю (jcxz).

Например, скопируем 500 элементов из source в dest:

```
MOV CX, 500
REP MOVS dest, source
```

Префикс REPZ/REPE (repeat while zero/equal – повторять пока нуль/ равно), повторяет команду, пока флаг ZF равен 1 и значение регистра CX не равно 0. Если приписать префикс REPE команде сравнения строк CMPS, то операция сравнения будет повторяться до первого несовпадения или пока не будут просмотрены все CX элементов. Например, последовательность команд:

```
MOV  CX, 100
REPE CMPS dest, source
```

поэлементно сравнивает строки source и dest до тех пор, пока не будет просмотрено 100 пар элементов или пока не будет найден в строке dest элемент, не совпадающий с соответствующим элементом строки source.

Действия префикса REPNE (repeat while not equal – повторять, пока не равно), имеющего синоним REPNZ (repeat while not zero – повторять пока не нуль), противоположно действию префикса REPE. Иначе говоря, префикс REPNE

обеспечивает повторение модифицированной им команды, пока флаг ZF равен 0 и значение регистра CX не равно 0.

При использовании префиксов REPE и REPNE причину выхода из цикла (CX=0 или ZF=?) надо определять, анализируя значение флага ZF а не значение CX.

Команда пересылки строки MOVS (MOVSB, MOVSW).

MOVS dest, src ; Move string – переслать строку байт или слов

MOVSB ; ES:[DI]:=DS:[SI]; DI=DI±1; SI=SI±1

MOVSW ; ES:[DI]:=DS:[SI]; DI=DI±2; SI=SI±2

Команда MOVS пересылает элемент строки, адресуемый DS:SI в элемент строки, адресуемый ES:DI и настраивает значения индексных регистров на следующие элементы строк.

Обратите внимание на то, что каждая команда обработки цепочек представлена тремя разными форматами. Первый из них имеет один или два операнда (например, MOVS имеет два операнда), а два других формата не имеют операндов. Форматы обозначаются символами B (обработка байта) и W (обработка слова). Например:

MOVSB – копировать байт из DS:[SI] в ES:[DI];

MOVSW – копировать слово из DS:[SI] в ES:[DI].

Применение команд без операндов предпочтительнее, поскольку в этом случае ассемблеру нет необходимости выяснять размер операндов. При трансляции программы ассемблер всегда преобразует команду с операндами в одну из команд без операндов. Формат с операндами используют, если необходимо **переопределить** сегмент источника. К сожалению, переопределить сегмент приёмника невозможно.

Например, MOVS:

1. MOVS строка_приёмник, строка_источник
2. MOVS строка_приёмник, ES: строка_источник.

В обоих случаях байт или слово из строки-источника переписывается в строку-приёмник. Операнд-приёмник адресуется через ES:DI, операнд-источник в первом примере через **DS:SI**, а во втором через **ES:SI**. Размер пересылаемого элемента определяется описанием строк (с помощью директив db или dw). Формат с операндами **не избавляет** от необходимости инициализировать регистры ES:DI и DS:SI (или ES:SI) адресами строки-источника и строки-приёмника.

Каждая групповая пересылка с помощью команды MOVS осуществляется с помощью пяти шагов:

- обнулить флаг DF командой CLD или установить его командой STD в зависимости от того, будет ли пересылка осуществляться от младших адресов к старшим или наоборот;
- загрузить адрес строки-источника в регистровую пару DS:SI;
- загрузить адрес строки-приёмника в регистровую пару ES:DI;
- загрузить счётчик элементов (количество пересылаемых элементов: байт или слов) в регистр CX;
- выполнить команду MOVS с префиксом REP.

Команда сравнения строк CMPS (CMPSB, CMPSW).

```
CMPS    dest, src       ; Compare sring – сравнить строки
CMPSB                   ; флаги:=CMP DS:[SI],ES:[DI]; DI±=1; SI±=1
CMPSW                   ; флаги:=CMP DS:[SI],ES:[DI]; DI±=2; SI±=2
```

Команда CMPS сравнивает значение элемента одной строки (DS:SI) со значением элемента второй строки (ES:DI) и настраивает значения регистров на следующие элементы строк в соответствии с флагом направления DF. Результат – сформированные флаги. Обратите внимание, что CMP вычитает операнд-источник из операнда-приёмника, а CMPS операнд-приёмник из операнда-источника. Это означает, что указываемые после команды CMPS команды условной передачи управления должны отличаться от тех, что в аналогичной ситуации следовали бы за командой CMP.

Команда сканирования строки SCAS (SCASB, SCASW).

```
SCAS        dest        ; Scaning string- сканировать строку
SCASB                   ; флаги:=(результат CMP AL, ES:[DI]); DI±=1
SCASW                   ; флаги:=(результат CMP AX, ES:[DI]); DI±=2
```

Команда SCAS сравнивает содержимое регистра AL (AX) с байтом (словом) по адресу ES:DI, после чего регистр DI устанавливается на следующий элемент памяти (байт или слово) в соответствии с флагом DF. Устанавливает флаги, аналогично команде CMP. Команда удобна для поиска заданного AL/AX элемента в цепочке или для поиска элемента последовательности отличного от заданного.

Команда загрузки строки LODS (LODSB, LODSW).

```
LODS    src             ; Load string – загрузить строку
LODSB                   ; AL: = DS:[SI]; SI±=1;
LODSW                   ; AX: = DS:[SI]; SI±=2;
```

Команда LODS пересылает элемент строки, начинающийся с адреса DS:SI в регистр AL (при пересылки байта) или в регистр AX (при пересылки слова), а затем изменяет регистр SI так, чтобы он указывал на следующий элемент строки. Команда удобна для переписывания данных из массива в массив с промежуточным их изменением.

Команда сохранения строки STOS (STOSB, STOSW).

```
STOS        dest        ; Store string – сохранить строку
STOSB                   ; ES:[DI]:=AL; DI±=1
STOSW                   ; ES:[DI]:=AX; DI±=2
```

Команда STOS пересылает байт (слово) из AL (AX) по адресу ES:DI, а затем, в зависимости от DF, меняет значение DI так, чтобы оно указывало на следующий элемент строки.

ЗАДАНИЕ

- Изучить пример программы №2 из Приложения 4.
- Составить **подпрограмму** решения поставленной задачи (см. индивидуальные задания), использующую **инструкции обработки цепочек**.
- Составить программу, осуществляющую ввод необходимых данных, вызов подпрограммы (п. 2) и вывод результата.

СОДЕРЖАНИЕ ОТЧЁТА

Тема и цель работы; задание на лабораторную работу (свой вариант). Текст подпрограммы с комментариями, текст программы. Выводы из проделанной работы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Перечислите команды, относящиеся к группе команд обработки цепочек?
2. Какие действия позволяют производить команды обработки цепочек?
3. Опишите действия выполняемые командами SCAS и CMPS.
4. Опишите действия выполняемые командами MOVS, LODS и STOS.
5. Как адресуются строка-приёмник (источник) в командах обработки цепочек
6. Какие бывают форматы записи команд обработки цепочек?
7. От чего зависит то, как будет изменяться (уменьшаться или увеличиваться) содержимое индексных регистров SI (DI) при выполнении строковых команд?
8. Что такое префикс повторения? Какие префиксы повторения используются для команд обработки цепочек?
9. Какой префикс повторений надо использовать для поиска заданного элемента в строке?
10. Можно ли переопределять сегмент строки-приёмника (источника).

ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. Зашифровать строку по таблице. Таблица считается известной.
2. Подсчитать количество элементов=x в массиве integer[N].
3. Подсчитать количество равных пар (x_i,y_i) в массивах слов.
4. Заменить символ на группу.
5. Сформировать строку, содержащую данный символ указанное число раз.
6. Определить позицию последнего вхождения символа в ASCIIZ-строку.
7. Преобразовать маленькие латинские буквы строки в большие.
8. Вычислить длину ASCIIZ (ограниченной нулем) строки.
9. Удалить пробелы слева.
10. Вернуть указанное число символов с правого конца строки.
11. Удалить подстроку из строки.
12. Вставить подстроку.
13. Проверить сбалансированность скобок в математическом выражении.
14. Подсчитать количество элементов, отличных от последнего в массиве двухбайтных чисел размерности N.
15. Вычислить количество слов в строке.
16. Найти самую длинную группу одинаковых подряд идущих символов строки.

Лабораторная работа № 5
ОРГАНИЗАЦИЯ ФАЙЛОВЫХ ОБМЕНОВ.

Цель работы: Изучение методов работы с файлами.

ЗАДАЧИ

- Закрепить навыки программирования на языке ассемблера.
- Научиться создавать, открывать, читать, записывать, закрывать файлы и обрабатывать возникающие ошибки.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Если в ассемблерной программе возникает необходимость в файловом вводе-выводе, то программа должна содержать фрагменты кода, в которых производится обращение к средствам ОС, осуществляющим взаимодействие с файловой системой. Мы будем изучать средства для работы с файлами в ОС MS DOS. Сервис файловой системы доступен программе через прерывание MS-DOS INT 21h.

Дескриптор файла (handle) это 16-разрядное целое без знака, используемое ОС и программами для обращения к файлам (на логическом уровне). Слово «файл» может относиться как к дисковому файлу, так и к устройству, например, такому как клавиатура, дисплей, принтер... Когда DOS загружает и исполняет программу, инициализируются некоторые стандартные дескрипторы:

Handle	Устройство	Описание
0	CON	Стандартное устройство ввода (обычно клавиатура)
1	CON	Стандартное устройство вывода (обычно экран)
2	CON	Стандартное устройство вывода сообщений об ошибках (всегда экран)
3	AUX	Устройство последовательного ввода-вывода (асинхронный адаптер; обычно – COM1)
4	PRN	Стандартное устройство печати (принтер; обычно 1 параллельный порт принтера – LPT1)

Зарезервированные файловые дескрипторы всегда доступны программе. Для устройств, соответствующих этим дескрипторам, не требуется выполнять операцию открытия.

В MS DOS используются короткие имена файлов (в формате 8.3).

Дисковый ввод-вывод в ОС MS-DOS буферизован. Это означает, что данные не сразу записываются на диск, а накапливаются в специальном массиве (буфере). По мере заполнения буфер сбрасывается на диск. При чтении информация заполняет весь входной буфер, независимо от количества байт, которые программа читает из файла. В дальнейшем, если программе потребуются данные, которые уже были считаны с диска и записаны во входной буфер, она получит данные непосредственно из этого буфера. Обращения к диску при этом не будет.

При закрытии файла все буфера, связанные с ним, сбрасываются на диск. Если вам надо сбросить буфер, не закрывая файл, это можно сделать с помощью функции 68h прерывания INT 21h. Дополнительно обновляется дескриптор файла в каталоге, а именно поля времени, даты и размера файла.

Обратите также внимание на функцию расширенного открытия файлов 6Ch, входящую в состав MS-DOS с версии 4.0. Эта функция позволяет при открытии файла отменить буферизацию.

Важно понимать что:

- Для всех файловых операций строки должны иметь формат ASCIIZ (строка заканчивающаяся нулём (кодом нуля)).
- Прежде чем использовать файл в программе его необходимо открыть. При открытии существующих файлов информация, записанная в файле, не теряется.
- Если файл не существует, то его нужно создать. При создании нового файла он открывается, но если на диске уже существует файл с таким именем, то он открывается с нулевой длиной, при этом его содержимое теряется.
- Текущая позиция – это указатель на место в файле, откуда будет считана (куда будет записана) следующая запись файла. Указатель в файле автоматически перемещается после выполнения очередной команды чтения-записи. Так что при чтении он всегда указывает на фрагмент, расположенный непосредственно за последним прочитанным.
- Данные, которые вы записываете в файл, временно хранятся в буфере. Не думайте, что операция записи сразу записывает данные на диск. При закрытии файла данные из буфера сбрасываются на диск, обновляется запись в каталоге, а файловый дескриптор освобождается для дальнейшего использования.
- Если после файловой операции флаг CF сброшен, то она завершилась успешно. В противном случае в регистре AX находится код ошибки. В программе обязательно надо контролировать результат завершения файловых операций.

Функции работы с файлами приведены в справочнике (Приложение 3).

В Windows 9x появилась поддержка длинных имён файлов. Приложения MS DOS получают доступ к длинным именам файлов с помощью дополнительных функций прерывания 21h. Функции MS DOS, поддерживающие длинные имена файлов, имеют двухбайтные номера. Старший байт номера = 71h, младший равен номеру аналогичной старой функции MS DOS. В программах старые и новые функции применяются вместе по принципу: там, где функция работает с дескриптором файла, используют старую функцию; там, где функция должна работать с длинными именами файлов и каталогов, используются новые функции.

Установить факт того, что система поддерживает длинные имена можно вызовом функции 71A0h прерывания 21h – получить информацию о томе. Если она возвращает ошибку, то текущая файловая система не поддерживает длинных имён. Для вызова этой функции необходимо указать ASCIIZ имя корневого каталога тома (DS:SI), о котором запрашиваем информацию. ES:DI - буфер для имени файловой системы, CX - размер буфера (32 байта).

Чтобы создать (открыть) файл с длинным именем вместо функции 6Ch воспользуйтесь функцией 716Ch (номер функции передайте в AX, остальные аргументы те же). После того, как файл открыт или создан, с ним можно работать, используя старые функции чтения, записи и позиционирования.

Для получения параметров командной строки надо знать, что command.com загружает исполняемый файл в память за PSP. Хвост командной строки копируется в PSP со смещением 80h. Имейте ввиду, что эту же область используют некоторые функции DOS в качестве временного буфера DTA (disk transfer area).

ЗАДАНИЕ

Используя справочный материал (Приложение 3), написать программу, выполняющую заданное действие с файлом (см. индивидуальное задание).

Имя файла ввести с клавиатуры или передать как параметр командной строки. Чтение из файла организовать блоком. Размер блока **согласовать с преподавателем**.

СОДЕРЖАНИЕ ОТЧЁТА

Тема и цель работы; задание на лабораторную работу (свой вариант). Текст программы с комментариями. Выводы из проделанной работы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как осуществляется доступ к файлам из программы на ассемблере при помощи MSDOS?
2. Что такое дескриптор файла, какие существуют стандартные дескрипторы?
3. Что такое буферизированный ввод/вывод?

4. Зависит ли от размера читаемого/записываемого блока время обработки файла?
5. Обязательно ли закрывать файл, открытый для чтения или записи?
6. Как создать файл, используя сервисные функции MSDOS?
7. Как осуществляется ввод/вывод из файла при помощи сервисных функций MSDOS?
8. Что такое указатель чтения/записи в файле?
9. Что такое длинные имена файлов?
10. Какие функции MSDOS поддерживают длинные имена файлов?
11. Можно ли, используя файловые операции, прочитать данные с клавиатуры (вывести данные на экран)

ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. Зашифровать файл по таблице. Таблицу считать заданной.
2. Зашифровать файл: букву на следующую (последнюю на первую).
3. Сообщить о количестве строк в файле.
4. Склеить 2 файла в один.
5. Подсчитать количество запятых в файле.
6. Подсчитать количество чисел файла, делящихся без остатка на 4.
7. Преобразовать в файле маленькие латинские буквы в большие.
8. Преобразовать в файле большие русские буквы в маленькие.
9. Найти сумму компонент файла of integer.
10. Найти последнюю компоненту файла of longint.
11. Заменить на пробелы все управляющие символы в файле (кроме #9,#10,#13).
12. Сравнить два файла игнорируя различия с разделителях (32,9).
13. Сравнить два файла игнорируя различия с разделителях (CR, 32).
14. Сравнить два файла игнорируя различия в регистре букв.
15. Подсчитать количество строк файла, начинающихся с 'd'.
16. Подсчитать количество строк файла, заканчивающихся на 'd'.

Лабораторная работа № 6
СВЯЗЬ ПОДПРОГРАММ НА АССЕМБЛЕРЕ С ПРОГРАММАМИ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ (PASCAL).

Цель работы: Изучение методов использования ассемблерных подпрограмм в программах на языках высокого уровня.

ДОМАШНЯЯ ПОДГОТОВКА

- Изучить организацию подпрограмм на встроенном ассемблере.
- Изучить способы подключения ассемблерных подпрограмм к программам на языке высокого уровня.
- Изучить способы передачи аргументов подпрограмме и методы доступа к ним из подпрограммы.
- Подготовить ответы на контрольные вопросы.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Существуют следующие формы комбинирования программ на языках высокого уровня (ЯВУ) с программами на языке ассемблера:

- Использование операторов Inline или ассемблерных вставок (в том числе и подпрограмм написанных на встроенном ассемблере). Эта форма сильно зависит от синтаксиса ЯВУ и конкретного компилятора. Она предполагает, что машинные коды или команды ассемблера вставляются в текст программы на ЯВУ.
- Использование внешних подпрограмм. Это наиболее универсальная форма комбинирования.

ВНИМАНИЕ! В своих подпрограммах вы не должны изменять регистры DS и BP или не забудьте сохранить их в стеке, а потом извлечь. Никогда не изменяйте содержимое регистра SS и будьте очень аккуратны при изменении SP.

Использование встроенного ассемблера

Вы можете писать ассемблерный код прямо в программах на языке Pascal, заключив его в операторные скобки ASM END. Встроенный ассемблер реализует подмножество синтаксиса TASM и MASM и поддерживает все коды операций 8086 и почти все операторы выражений TASM.

Если вы в подпрограмме на встроенном ассемблере используете метки, то их надо описать в разделе label или имя метки должно начинаться с признака локального имени “@”.

В подпрограмме на встроенном ассемблере вы можете использовать специальные переменные:

- @Result – результат, возвращаемый функцией (только внутри функции). Это удобно если результат типа string, так как в этом случае в @Result находится адрес строки.
- Для обращения к текущему сегменту кода и данных можно использовать переменные @Code и @Data соответственно.

Использование внешних ассемблерных подпрограмм

- Напишите подпрограмму на языке ассемблера с явным указанием типа (FAR или NEAR). Объявите имя подпрограммы внешним (директивой PUBLIC).
- Постройте объектный модуль (TASM).
- В Pascal-программе, которая будет вызывать внешнюю подпрограмму, вставьте директиву {$L имя_объектного_модуля}. Это требование транслятору с языка Pascal при компоновке объединить указанный модуль с тем кодом, который он формирует сам.
- Напишите в Pascal-программе заголовок подпрограммы, используя синтаксис и типы языка Pascal, после объявления укажите явно тип (FAR или NEAR) и служебное слово EXTERNAL.
- Теперь можете вызывать внешнюю подпрограмму так, как будто она написана на языке Pascal.

Передача аргументов и возврат результатов работы подпрограмм

Входные параметры подпрограмме на языке Pascal передаются через стек процессора или сопроцессора. Результаты работы остаются в регистрах, стеке процессора или в стеке сопроцессора. Компилятор Pascal генерирует команды подготовки аргументов в стеке при обработке вызова подпрограммы.

Параметры-значения передаются следующим образом:

- 1 байтовые параметры передаются как двухбайтовые, дополненные старшим байтом с неопределённым значением.
- 2, 4, 6 байтные в 1, 2, 3 словах в стеке.
- параметры типов single, double, extended, comp передаются на вершине стека сопроцессора.
- параметр типа множество передаётся указателем на его 32-байтное соответствие.
- все остальные параметры-значения передаются своими полными адресами.

Параметры-переменные передаются по ссылке, т.е. своими полными адресами (сегмент:смещение).

По соглашениям, принятым в компиляторах с языка Pascal, вызываемая подпрограмма должна перед возвратом управления очистить стек от аргументов подпрограммы. Для этого можно использовать команду RET n, где n — это размер переданных параметров в байтах.

Результат функции может быть простого, ссылочного или строкового типа. Результаты первых двух типов возвращаются через регистры:

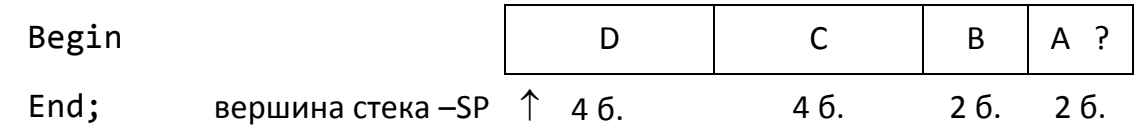
1 байт	– AL	4 байта	– DX:AX
2 байта	– AX	6 байт	– DX:BX:AX,

Вещественные, кроме real, – на вершине стека сопроцессора.

Под строки выделяется память, и указатель на неё находится в стеке выше (адрес старше) всех передаваемых параметров. Pascal ожидает, что при выходе из подпрограммы указатель на строку-результат из стека удалён не будет, в отличие от всех остальных аргументов подпрограммы, забота об удалении которых из стека возложена на подпрограмму.

Посмотрим, как будет подготовлен стек перед вызовом подпрограммы Ex_1.

```
Procedure Ex_1 (A:byte; B:word; C:longint; var D:byte); far;
Var E,F,G:byte;
 Begin
 End;
```



Организация доступа к аргументам из ассемблерных подпрограмм

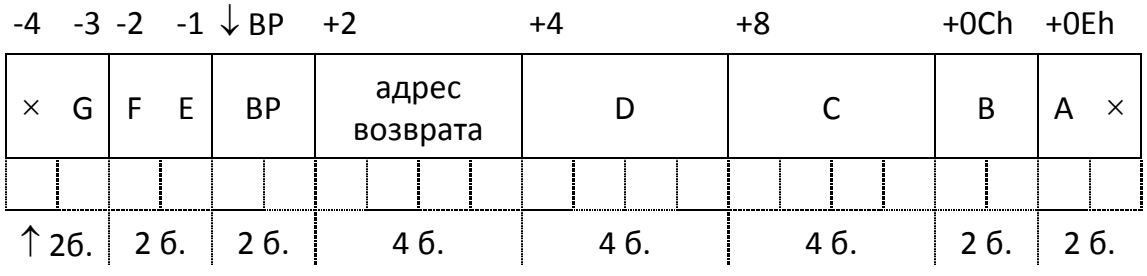
Вызов подпрограммы осуществляется командой CALL, которая перед передачей управления на подпрограмму сохраняет в стеке адрес возврата (одно слово для NEAR или два для FAR) из подпрограммы.

Перед первыми строками подпрограммы транслятор генерирует код пролога, выполняя настройку указателя BP на текущую вершину стека:

```
 PUSH BP
 MOV  BP,SP
```

Под локальные переменные подпрограммы (E, F, G в Example_1) память будет выделена в стеке, командой: `SUB SP,4`

Таким образом, перед выполнением первого оператора подпрограммы Example_1 стек будет заполнен следующим образом:



SP – вершина стека

Если не изменять значение BP, то относительно него можно рассчитать местоположение всех аргументов. Ниже приведены два способа выполнения одной и той же операции копирования G:=A.

```
1)   MOV    AL,[BP+0Eh]        |  2)   MOV    AL,A
     MOV    [BP][-3],AL        |       MOV    G,AL
```

В первом способе для доступа к аргументам A и G, находящимся в стеке, использованы выполненные расчёты. Второй способ понятнее, но воспользоваться им можно только в подпрограмме на встроенном ассемблере.

А теперь рассмотрим примеры организации доступа к данным одного типа, но переданным в подпрограмму различными способами. Аргумент A передан в подпрограмму по значению,а D по ссылке:

```
1)   MOV    AL,A          ; AL:=A
2)   LES    BX,D          ; теперь в ES:BX адрес 1 байта D.
     MOV    CL,ES:[BX]    ; CL:=D
```

После последней строки подпрограммы генерируется код эпилога – команд освобождения стека от аргументов подпрограммы:

```
     MOV    SP, BP              ; Эти действия стоят
     POP    BP                  ; за END подпрограммы
     RET    12                  ; Для моего примера.
```

Придерживаясь этих правил можно писать и «чисто» ассемблерные подпрограммы, получающие данные через стек.

Проиллюстрируем всё вышеизложенное примером.

Пишем программу вычисления минимального значения из двух двухбайтных чисел со знаком.

```
{ ---------- Example.pas ----------        }
{ Подключаем две внешние подпрограммы (Min_a, Min_ b), а одну
подпрограмму (Min) пишем на встроенном ассемблере }
{$L Min_a.obj}
function Min_a(a:integer; var b:integer):integer; near;
external;
{$L Min_b.obj}
function Min_b(a:integer; var b:integer):integer; near;
external;
function Min   (a:integer; var b:integer):integer; near;
assembler;
asm  MOV       AX,A
     LES       SI,B
     CMP       AX,[ES:SI]
     JLE       @M1       {A ≤ B => в AX уже и так минимум}
     MOV       AX,[ES:SI]{A > B => в AX – меньшее число}
```

```
@M1: end;
var x:integer;
begin     { Вызываем три разные подпрограммы поиска минимума}
    x:=6; writeln(Min(2,x));
          writeln(Min_a(7,x));
          writeln(Min_b(3,x));
end.
```

В тексте первой внешней подпрограммы используем директиву MODEL, чтобы учесть соглашения языка высокого уровня о вызове подпрограмм. Применение этой директивы позволяет:

- описать аргументы подпрограммы непосредственно в директиве PROC;
- автоматически сгенерировать код пролога и эпилога в ассемблерной подпрограмме (см. в листинге к примеру или к своей программе);
- осуществлять доступ к аргументам подпрограммы по их именам;
- использовать упрощенные директивы сегментации.

;---------- min_a.asm ----------

```
.MODEL Large, PASCAL
.CODE
PUBLIC  Min_a           ; (a:integer; var b:integer):integer
Min_a   PROC   NEAR a:Word, B:Dword
        MOV    AX,A
        LES    SI,B     ; Посмотрите в листинге, какие
        CMP    AX,      ; параметры используются по умолчанию
[ES:SI]                 ; с данными упрощёнными директивами
        JLE    @M1      ; сегментации?
        MOV    AX,      ; Какие строки (пролога и эпилога) были
[ES:SI]                 ; добавлены транслятором? Почему?
@M1:    RET
Min_a   ENDP
        END
```

Недостаток рассмотренного способа организации подпрограмм – невозможность воспрепятствовать TASM, сгенерировать инструкции, подготавливающие BP для адресации переменных в стеке. Пролог и эпилог генерируются, даже если у подпрограммы нет аргументов. Решение проблемы – всё делать вручную.

;---------- min_b.asm ----------

```
Code SEGMENT byte public
ASSUME    cs:code
PUBLIC Min_b ;(a,b: integer) : integer;
Min_b   PROC    NEAR
```

```
A       EQU    word ptr  [BP+8]
B       EQU    dword ptr [BP+4]
        PUSH   BP          ; Вместо расчётов и использования
        MOV    BP, SP      ; директивы EQU можно воспользоваться
        MOV    AX, A       ; директивой ARG. Но помните,
        LES    SI, B       ; что порядок следования аргументов
        CMP    AX,[ES:SI]  ; в директиве ARG является обратным
        JLE    @M1         ; их следованию в описании PASCAL
        MOV    AX,         ; подпрограммы. Для приведённого
[ES:SI]                    ; примера вместо выделенных строк
@M1:    POP    BP          ; можно поместить строку:
        RET    6           ; ARG  b : dword, a : word = siz.
Min_b   ENDP               ; В этом случае команду RET 6 можно
Code    ENDS               ; заменить на RET siz.
        END
```

У директивы ARG есть ещё один интересный аргумент: RETURNS за которым следует список параметров, не подлежащих удалению из стека по завершении подпрограммы. Это может потребоваться, например, когда результат функции – строка.

ARG x : WORD, y : DWORD = siz RETURNS s : DWORD

ВНИМАНИЕ! Контроль за соответствием типов подпрограмм, объявленных директивой PROC (NEAR или FAR) типам подпрограмм, объявленных директивой EXTERNAL в PASCAL-программе отсутствует. Программист должен сам принимать необходимые меры, чтобы объявления подпрограмм в разных модулях были согласованы.

ЗАДАНИЕ

Используя ассемблер и встроенный ассемблер, написать подпрограмму (задание по вариантам) и продемонстрировать её использование в программе на языке высокого уровня (Pascal). Дополнить код ещё двумя вариантами заданной подпрограммы.

СОДЕРЖАНИЕ ОТЧЁТА

Тема и цель работы; задание на лабораторную работу (свой вариант). **Три** варианта заданной подпрограммы (на встроенном ассемблере, с использованием модели памяти PASCAL (**листинг**) и без использования директивы MODEL (**листинг**)). Программу, вызывающую эти подпрограммы с указанием автора и варианта задания. Выводы из проделанной работы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие существуют способы использования ассемблера в программах на языках высокого уровня?
2. Какие особенности использования встроенного ассемблера в языке Turbo Pascal ?
3. Какие действия необходимо выполнить для подключения внешней ассемблерной подпрограммы к программе на языке Turbo Pascal?
4. Как происходит передача параметров подпрограмме в языке Turbo Pascal на низком уровне (где и как передаются аргументы)?
5. Как осуществляется возврат результата функции подпрограмме в языке Turbo Pascal на низком уровне?
6. Какие команды процессора используются для организации стандартного входа в подпрограмму и выхода из подпрограммы и для чего они используются?
7. Для чего применяется в ассемблере директива MODEL?
8. Как оформляется внешняя ассемблерная подпрограмма с использованием директивы SEGMENT?
9. Как оформляется внешняя ассемблерная подпрограмма с использованием директивы MODEL?
10. Для чего применяется в ассемблере директива ARG?

ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. Найти позицию первого вхождения символа в строку (string).
2. Найти позицию последнего вхождения символа в строку (string).
3. Поиск максимального значения в цепочке байт со знаком.
4. Поиск минимального значения в цепочке слов без знака.
5. Поиск максимального значения в цепочке двойных слов со знаком.
6. Подсчитать количество элементов = x в массиве byte[N].
7. Подсчитать количество элементов $\leq$ x в массиве byte[N].
8. Подсчитать количество элементов > x в массиве integer[N].
9. Подсчитать количество элементов < x в массиве longint[N].
10. Подсчитать количество отрицательных элементов в массиве shortint[N].
11. Подсчитать количество положительных элементов в массиве integer[N].
12. Найти максимум среди k последних элементов в массиве integer [N].
13. Найти минимум среди k последних элементов в массиве byte [N].
14. Вычислить сумму квадратов всех отрицательных элементов массива shortint[N].
15. Вычислить сумму квадратов всех положительных элементов массива integer [N].
16. Вычислить НОД(a,b).

Лабораторная работа № 7
СВЯЗЬ ПОДПРОГРАММ НА АССЕМБЛЕРЕ IA-32 С ПРОГРАММАМИ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ (OBJECT PASCAL)

Цель работы: Изучение методов использования ассемблера и ассемблерных подпрограмм в языках высокого уровня.

ДОМАШНЯЯ ПОДГОТОВКА

- Изучить программную модель микропроцессоров архитектуры IA-32.
- Изучить организацию подпрограмм на встроенном ассемблере.
- Изучить способы подключения ассемблерных подпрограмм к программам на языке высокого уровня.
- Изучить способы передачи аргументов подпрограмме и методы доступа к ним из подпрограммы.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Архитектура IA-32

Начало семейства современных процессоров архитектуры IA-32 (32-bit Intel Architecture) положил процессор i80386 (1985 г.). Основные его отличия от предыдущих процессоров (8086 и 80286):

- 32-разрядные шины адреса и данных;
- 32-разрядные регистры данных и указателей;
- три режима работы процессора:
  - реальный (режим процессора 8086),
  - защищенный,
  - виртуальный 8086 (V86);
- аппаратная поддержка страничной организации памяти и многозадачности в защищенном режиме.

Процессоры i80486 (1989 г.), Pentium (1993 г.), Pentium Pro (1995 г.), Pentium II (1997 г.), Pentium III (2000 г.), Pentium IV (2002 г.), Pentium D (2005 г.), а также Intel-совместимые процессоры фирм AMD и Cyrix основаны на архитектуре IA-32. Основные отличия между ними заключаются в усовершенствовании аппаратной реализации, введении новых вычислительных блоков (FPU, MMX, SSE) и увеличения разрядности шин адреса и данных с целью повышения производительности.

Программная модель процессоров IA-32

Организация памяти

Организация памяти процессоров IA-32 зависит от режима работы процессора:

Реальный режим – в данном режиме используется программная модель процессора 8086 и соответствующая организация памяти. То есть в данном режиме процессор работает просто, как быстрый МП 8086. Исключением является возможность работы с 32-разрядными регистрами данных, с дополнительными регистрами сегментов, а также с регистрами управления позволяющими перевести процессор в защищённый режим. При включении процессор переходит в данный режим.

Защищённый режим – реализует полную функциональность программной модели процессора IA-32. Переключение в данный режим обычно осуществляется при загрузки ОС защищенного режима (например Windows).

В данном режиме используется 32-разрядная адресация памяти и сегментная модель с использованием таблиц описателей (дескрипторов) сегментов. В отличие от сегментной модели МП 8086 данная модель позволяет:

- расположить сегменты с произвольного адреса в памяти;
- организовать сегменты произвольного размера;
- задать назначение сегмента (код, данные или стек);
- организовать защиту памяти - установить доступ к сегментам согласно уровню привилегий выполняемой программы (задачи);
- организовать защиту команд - установить возможность выполнения системных (привилегированных) команд процессора внутри сегмента кода согласно уровню привилегий задачи.

Также в данном режиме процессор поддерживает многозадачность и дополнительно к сегментной организации страничную организацию памяти.

Режим виртуального 8086 (V86) – позволяет выполнять программы предназначенные для МП 8086 в защищенном режиме, то есть без перехода в реальный режим. При этом ОС и остальные программы для IA-32 работают в защищённом режиме. Данный режим позволяет одновременно выполнять несколько программ для МП 8086, так как в защищённом режиме поддерживается многозадачность. Переключение в данный режим обычно осуществляется ОС работающей в защищённом режиме при запуске программы для реального режима.

Регистры

Набор регистров процессоров IA-32 включает все регистры процессора 8086. Кроме этого процессоры IA-32 включают следующие регистры:

Регистры данных. EAX, EBX, ECX, EDX – 32-разрядные регистры данных. Функциональное назначение регистров такое же, как и в МП 8086. В процессорах IA-32 все регистры данных (а не только BX как в МП 8086) можно использовать в качестве регистров указателей при 32-разрядной косвенной адресации (в защищённом режиме).

Регистры AX (AH, AL), BX (BH, BL), CX (CH, CL), DX (DH, DL) являются младшими частями соответствующих 32-разрядных регистров.

Регистры указателей. ESI, EDI, EBP, ESP – 32-разрядные регистры указателей. Функциональное назначение регистров такое же, как и в МП 8086. Регистры SI, DI, BP, SP, IP являются младшими частями соответствующих 32-разрядных регистров и используются для адресации в реальном или V86 режимах.

EIP – 32-разрядный регистр указателя команд. Так же как и регистр IP в МП 8086, содержит смещение относительно содержимого регистра CS (сегмента кода) следующей подлежащей выполнению команды процессора. Регистр IP является младшей частью регистра EIP и используется в реальном или V86 режимах.

Регистр состояния. EFLAGS – 32-разрядный регистр флагов. Младшие 16 разрядов данного регистра соответствуют регистру флагов МП8086. Остальные разряды регистра содержат дополнительные флаги, предназначенные для работы в защищённом режиме.

Регистры сегментов CS, SS, DS, ES, FS, GS – 16-разрядные регистры. CS (сегмент кода), SS (сегмент стека), DS (сегмент данных), ES (дополнительный сегмент данных) в реальном или V86 режиме, также как и МП 8086, используются для хранения номеров сегментов. В защищённом режиме в них хранятся селекторы соответствующих сегментов (индекс в таблице дескрипторов сегментов).

Регистры FS, GS также как и ES используются в качестве дополнительных сегментных регистров данных.

Системные регистры. Данные регистры предназначены для организации работы защищённого режима и многозадачности.

Регистр GDTR – определяет местонахождение в памяти глобальной (главной) таблицы дескрипторов сегментов.

Регистр LDTR – определяет местонахождение в памяти локальной таблицы дескрипторов сегментов текущей задачи.

Регистр IDTR – определяет местонахождение в памяти таблицы дескрипторов векторов (обработчиков) прерываний.

Регистр TR – определяет местонахождение в памяти информации о текущей задаче.

Регистры управления CR0, CR1, CR2, CR3, CR4 – предназначены для управления режимом работы процессора и выполнения задач в многозадачной среде.

Регистры отладки DR0, DR1, DR2, DR3, DR4, DR5, DR6, DR7 – предназначены для управления и мониторинга аппаратной отладки программ.

Регистр типов областей памяти MTTR используется для управления кэш-памятью.

Машинно-зависимые регистры MSR используются для управления процессором, контроля производительности, получения информации о модели и возможностях процессора. Количество данных регистров зависит от модели процессора

Наборы дополнительных регистров данных предназначены для работы с дополнительными вычислительными блоками процессора, которые появились в новых процессорах IA-32.

Регистры математического сопроцессора (FPU): 80-разрядные регистры ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7);

Регистры MMX-расширения: 64-разрядные регистры MMX0, MMX1, MMX2, MMX3, MMX4, MMX5, MMX6, MMX7;

Регистры SSE-расширения: 128-разрядные регистры XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7;

Система команд

Система команд процессоров IA-32 включает все команды процессоров 8086 и 80286. Дополнительные команды процессоров IA-32 позволяют:

- работать с 32-разрядными регистрами процессора;
- работать с дополнительными регистрами сегментов;
- работать с системными регистрами, регистрами управления, регистрами отладки и т.д.;
- использовать дополнительные способы адресации операндов (косвенная адресация по регистру данных, косвенная адресация с масштабированием);
- выполнять дополнительные арифметические и логические операции.

В новых процессорах IA-32 присутствуют команды, позволяющие работать с дополнительными вычислительными и управляющими блоками процессора.

Связь подпрограмм на языке ассемблера IA-32 с программами на ЯВУ.

Программы на ЯВУ выполняются под управлением операционной системы. При создании ассемблерных подпрограмм для ОС Windows необходимо учитывать, что программа выполняется в защищённом режиме процессора. Это означает, что:

- используется 32-разрядная адресация памяти, то есть смещение адреса является 32-разрядным;
- используется защита памяти, то есть ОС выделяет программе определенные сегменты и другие сегменты использовать запрещено (нет необходимости модифицировать сегментные регистры);
- используется защита команд процессора, поэтому в прикладной программе запрещается использовать привилегированные команды процессора;
- по умолчанию используются 32-разрядные операции с данными.

Таким образом, в ассемблерной подпрограмме для адресации памяти необходимо указывать только смещение ячейки памяти. Сегмент данных является для программы на ЯВУ и подпрограммы на ассемблере общим и изменять содержимое сегмента DS не рекомендуется.

В остальном, методика использования ассемблера в программах на ЯВУ не отличается от методики для МП 8086.

Использование внешних ассемблерных подпрограмм.

Для трансляции внешних ассемблерных подпрограмм должен использоваться 32-разрядный транслятор (например TASM32).

При оформлении подпрограмм необходимо указать, что используется 32-разрядная адресация.

Для использования внешних ассемблерных подпрограмм в программе на языке высокого уровня Object Pascal необходимо выполнить следующие действия:

- Написать подпрограмму на языке ассемблера с явным указанием типа процессора (.386) и 32-разрядной адресации (.MODEL FLAT,PASCAL или code segment public use32). Объявить имя подпрограммы внешним (директивой PUBLIC).
- Получить объектный модуль (при помощи TASM32).
- В Pascal-программе, которая будет вызывать внешнюю подпрограмму, вставить директиву {$L имя_объектного_модуля}. Это требование транслятору с языка Pascal при компоновке объединить указанный модуль с тем кодом, который он формирует сам.
- Написать в Pascal-программе заголовок подпрограммы, используя синтаксис и типы языка Pascal, после объявления указать явно тип вызова (PASCAL) и служебное слово EXTERNAL.

– Теперь можно вызывать внешнюю подпрограмму так, как будто она написана на языке Pascal.

При расчётах местоположения аргументов в необходимо учитывать, что указатели стека (ESP, EBP), а также адрес возврата из подпрограммы сохраняемый в стеке являются 32-разрядными.

ЗАДАНИЕ

Изучить пример программы из приложения 4. Используя ассемблер и встроенный ассемблер, написать подпрограмму (задание по вариантам) и продемонстрировать её использование в программе на языке высокого уровня (Object Pascal).

СОДЕРЖАНИЕ ОТЧЁТА

Тема и цель работы; задание на лабораторную работу (свой вариант). Три варианта разработанной программы на языке ассемблера (на встроенном ассемблере, с использованием директивы SEGMENT (листинг) и с использованием директивы MODEL (листинг)), а также программы, вызывающей эти подпрограммы. Выводы из проделанной работы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие основные отличия процессоров IA-32 от предыдущих Intel-совместимых процессоров?
2. Какие существуют режимы работы процессора IA-32?
3. Какие особенности работы процессора в защищённом режиме?
4. Как организована память в защищённом режиме процессора?
5. Какие изменения произошли со стандартным набором регистров МП 8086 в IA-32?
6. Какие регистры появились в IA-32 по сравнению с МП 8086?
7. Какие изменения присутствуют в системе команд процессоров IA-32 по сравнению с предыдущими Intel-совместимыми процессоров?
8. Что необходимо учитывать при разработке подпрограмм на ассемблере IA-32, предназначенных для подключения к программе на ЯВУ?
9. Какие действия необходимо выполнить для подключения внешней ассемблерной подпрограммы в программу на языке ObjectPascal?
10. Как оформляется внешняя ассемблерная подпрограмма с использованием директивы SEGMENT для IA-32?
11. Как оформляется внешняя ассемблерная подпрограмма с использованием директивы MODEL для IA-32?

Лабораторная работа № 8
МАКРОСРЕДСТВА ЯЗЫКА АССЕМБЛЕР.

Цель работы: Изучение макросредств языка ассемблер.

ДОМАШНЯЯ ПОДГОТОВКА

- Изучить макрокоманды.
- Изучить директивы условной компиляции.
- Подготовить ответы на контрольные вопросы.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Ассемблер состоит из двух частей – непосредственно транслятора, формирующего объектный модуль, и макроассемблера. Таким образом, обработка программы на ассемблере с использованием макросредств неявно осуществляется транслятором в две фазы. На первой фазе работает часть компилятора, называемая макроассемблером. На второй фазе работает непосредственно ассемблер, задачей которого является формирование объектного кода, содержащего текст исходной программы в машинном виде. К простейшим макросредствам языка ассемблера можно отнести псевдооператоры EQU и "=".

Макрокоманды

Макрокоманда – это однострочное сокращение группы команд, которую называют *макроопределением*. *Макрогенерацией* называют процесс замены макрокоманд макроопределениями. Результат макрогенерации называют *макрорасширением*.

Синтаксис макроопределения:

```
имя_макрокомандыmacro список_формальных_аргументов
                    тело_макроопределения
                endm
```

Где должны располагаться макроопределения? Есть два варианта [2]:

- В начале исходного текста программы до сегмента кода и данных с тем, чтобы не ухудшать читабельность программы.
- В отдельном файле. Чтобы сделать доступными эти макроопределения в конкретной программе, необходимо в начале исходного текста этой программы записать директиву include имя_файла, к примеру: include show.inc ;в это место будет вставлен текст файла show.inc

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

- в отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с набором фактических параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды;
- при каждом вызове макрокоманды её текст в виде макрорасширения вставляется в программу. При вызове процедуры микропроцессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре.

Для того чтобы использовать макроопределение, оно должно быть вызвано с помощью макрокоманды указанием следующей синтаксической конструкции: имя_макрокоманды список_фактических_аргументов

В результате в исходном тексте программы макрокоманда будет замещена строками из тела макроопределения, в котором формальные аргументы заменены фактическими. Процесс такого замещения называется макрогенерацией, а результатом этого процесса является макрорасширение.

Пример макроопределения – clear_r. Результаты работы макроассемблера можно узнать, просмотрев файл листинга после трансляции. Рассмотрим несколько фрагментов листинга, которые демонстрируют, как был описан текст макроопределения clear_r (строки 24-27), как был осуществлён вызов макрокоманды clear_r с фактическим параметром ax (строка 74) и как выглядит результат работы макрогенератора: команда xor ax,ax (строка 75);

```
24          clear_r   macro  rg
25          ;;очистка регистра rg
26          xor  rg, rg
27          endm
...
74                    clear_r  ax
75  000E 33 C0 xor   ax, ax
```

В итоге мы получили то, что и требовалось – команду очистки заданного регистра, в данном случае ax. В другом месте программы мы можем выдать ту же макрокоманду, но уже с другим именем регистра.

Каждый фактический аргумент представляет собой строку символов, для формирования которой применяются следующие правила:

- строка может состоять из: (1) последовательности символов без пробелов, точек, запятых, точек с запятой; (2) последовательности любых символов, заключённых в угловые скобки: <...>;
- для того чтобы указать, что некоторый символ внутри строки, представляющей фактический параметр, является собственно символом, а не чем-то иным, например некоторым разделителем или ограничивающей скобкой, применяется специальный оператор “!”. Этот оператор ставится

непосредственно перед описанным выше символом, и его действие эквивалентно заключению данного символа в угловые скобки;

- если требуется вычисление в строке некоторого константного выражения, то вначале этого выражения нужно поставить знак "%":

  % константное_выражение – значение константное_выражение вычисляется и подставляется в текстовом виде в соответствии с текущей системой счисления.

Если макроопределение содержит метки, то необходимо указать ассемблеру на необходимость изменять метки при каждой макроподстановке. В противном случае вы столкнетесь с ошибкой: «Символ многократно определён». Чтобы создавать автоматически нумерующиеся локальные метки, используйте внутри макроса директиву LOCAL. Эта директива должна находиться после строки открывающей макрос:

```
MyMacros        MACRO
                LOCAL  met1, met2
```

Макродирективы

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

- директивы повторения WHILE, REPT, IRP и IRPC (предназначены для создания макросов, содержащих несколько идущих подряд одинаковых последовательностей строк);
- директивы управления процессом генерации макрорасширения: EXITM и GOTO.

Директивы WHILE и REPT применяют для повторения определённого количества раз некоторой последовательности строк.

Эти директивы имеют следующий синтаксис:

```
WHILE  конст_выражение          REPT  конст_выражение
       последоват_строк                 последоват_строк
ENDM                            ENDM
```

При использовании директивы WHILE макрогенератор транслятора будет повторять последоват_строк до тех пор, пока значение конст_выражение не станет равно нулю. Это значение вычисляется каждый раз перед очередной итерацией цикла повторения (то есть значение конст_выражение должно подвергаться изменению внутри последоват_строк в процессе макрогенерации).

Директива REPT, в отличие от WHILE автоматически уменьшает на единицу значение конст_выражение после каждой итерации.

Директива IRP имеет следующий синтаксис:

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

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

К примеру, рассмотрим результат определения в программе следующей конструкции:

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

Макрогенератором будет сгенерировано следующее макрорасширение:

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

Директива IRPC имеет следующий синтаксис:

```
IRPC формальный_аргумент, строка_символов
      последовательность_строк
ENDM
```

Действие данной директивы подобно IRP, но отличается тем, что она на каждой очередной итерации заменяет формальный_аргумент очередным символом из строка_символов. Понятно, что количество повторений последовательность_строк будет определяться количеством символов в строка_символов.

Например директива:

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

развернётся в следующую последовательность строк:

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

Директивы условной компиляции

Последний тип макросредств – директивы условной компиляции. Существует два вида этих директив:

- директивы компиляции по условию позволяют проанализировать определённые условия в ходе генерации макрорасширения и, при необходимости, изменить этот процесс;
- директивы генерации ошибок по условию также контролируют ход генерации макрорасширения с целью генерации или обнаружения определённых ситуаций, которые могут интерпретироваться как ошибочные.

С этими директивами применяются директивы управления процессом генерации макрорасширений EXITM и GOTO.

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

Директива GOTO имя_метки переводит процесс генерации макроопределения в другое место, прекращая тем самым последовательное разворачивание строк макроопределения. Метка, на которую передаётся управление, имеет специальный формат: :имя_метки

Директивы компиляции по условию предназначены для организации выборочной трансляции фрагментов программного кода. Такая выборочная компиляция означает, что в макрорасширение включаются не все строки макроопределения, а только те, которые удовлетворяют определённым условиям. Всего имеется 10 типов условных директив компиляции. Все они имеют общий синтаксис и применяются в составе следующей синтаксической конструкции:

```
IFxxx логич_выраж_или_аргументы
     фрагмент_программы_1
ELSE
     фрагмент_программы_2
ENDIF
```

Заключение некоторых фрагментов текста программы между директивами IFxxx, ELSE и ENDIF приводит к их выборочному включению в объектный модуль.

Какой именно из этих фрагментов – фрагмент_программы_1 или фрагмент_программы_2 – будет включён в объектный модуль, зависит

от конкретного типа условной директивы, задаваемого значением xxx, и значения условия, определяемого операндом (операндами) условной директивы логич_выраж_или_аргументы.

Действие директивы IFNxxx обратно IFxxx.

Директивы IF и IFE – условная трансляция по результату вычисления логического выражения. Удобны, если необходимо изменить текст программы в зависимости от некоторых условий.

Директивы IFDEF и IFNDEF – условная трансляция по факту определения символического имени. Один из ключей командной строки TASM: /dидентификатор=значение даёт возможность управлять значением идентификатора прямо из командной строки транслятора, не изменяя при этом текста программы.

Директивы IFB и IFNB – условная трансляция по факту определения фактического аргумента при вызове макрокоманды (проверяет равенство аргумента пробелу. В качестве аргумента могут выступать имя или число. Если его значение равно пробелу (то есть фактический аргумент при вызове макрокоманды не был задан), то транслируется и помещается в объектный модуль фрагмент_программы_1).

Директивы IFIDN, IFIDNI, IFDIF и IFDIFI – условная трансляция по результату сравнения строк символов. IFIDNI и IFDIFI игнорируют различие строчных и прописных букв, а IFIDN и IFDIF – учитывают.

Синтаксические конструкции, соответствующие директивам условной компиляции, могут быть вложенными друг в друга.

Примеры:

```
Show  MACRO  reg
IFB
DISPLAY   'не задан регистр'
EXITM
ENDIF
...
ENDM
```

Если теперь в сегменте кода вызвать макрос show без аргументов, то будет выведено сообщение о том, что не задан регистр и генерация макрорасширения будет прекращена директивой exitm.

Директивы генерации ошибок. В языке TASM есть ряд директив, называемых директивами генерации пользовательской ошибки. Их можно рассматривать и как самостоятельное средство, и как метод, расширяющий возможности директив условной компиляции. Они предназначены для обнаружения различных ошибок в программе, таких как неопределённые метки или пропуск параметров макроса. Большинство директив генерации ошибок

имеют два обозначения, хотя принцип их работы одинаков. Второе название отражает их сходство с директивами условной компиляции. При дальнейшем обсуждении такие парные директивы будут приводиться в скобках.

К безусловным директивам генерации пользовательской ошибки относится только одна директива – это ERR (.ERR). Данная директива безусловно приводит к генерации ошибки на этапе трансляции и удалению объектного модуля.

Остальные директивы являются условными, так как их поведение определяют некоторые условия. Набор условий, на которые реагируют директивы условной генерации пользовательской ошибки, такой же, как и у директив условной компиляции. Поэтому и количество этих директив такое же.

К их числу относятся следующие директивы:

.ERRB (ERRIFB) и .ERRNB (ERRIFNB)

.ERRDEF (ERRIFDEF) и .ERRNDEF (ERRIFNDEF)

.ERRDIF (ERRIFDIF) и .ERRIDN (ERRIFIDN)

.ERRE (ERRIFE) и .ERRNZ (ERRIF)

Если принцип их работы не понятен, см. [2, С. 302-305].

ЗАДАНИЕ

Решить задачу (см. Индивидуальные задания) с помощью макросредств. Макроопределения поместить в отдельный inc-файл. Написать программу на ассемблере для демонстрации работы полученных макросов.

СОДЕРЖАНИЕ ОТЧЁТА

Тема и цель работы; задание на лабораторную работу (свой вариант). Текст разработанной программы и макросов на языке ассемблера. Выводы из проделанной работы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое макроассемблер?
2. Какие существуют виды макросредств?
3. Как осуществляется вызов макрокоманд (пример)?
4. Как передаются параметры в макроопределение?
5. Что такое и для чего предназначены макродирективы?
6. В чем заключается действие макродиректив WHILE и REPT?
7. В чем заключается действие макродиректив IRP и IRPC?
8. Что такое и для чего предназначены директивы условной компиляции?
9. Что такое и для чего предназначены директивы генерации ошибок?
10. Каковы преимущества и недостатки использования макросов?
11. Как определяют типы параметров в макросах?
12. Как проверить был ли передан параметр в макрокоманде?
13. Как можно выйти из макроса.

14. Как указать макропроцессору, что надо загрузить макроопределения из текстового файла?
15. Можно ли использовать метки в макроопределениях? Какая проблема возникает при использовании меток в макроопределениях? Как она решается? Почему подобной проблемы нет, при использовании меток в подпрограммах?

ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. Вывод на экран байта в двоичном виде.
2. Вывод на экран байта в восьмеричном виде.
3. Вывод на экран байта в шестнадцатеричном виде.
4. Вывод на экран слова в двоичном виде.
5. Вывод на экран слова в восьмеричном виде.
6. Вывод на экран слова в шестнадцатеричном виде.
7. Ввод строки.
8. Вывод строки.
9. Перевести курсор в заданную позицию.
10. Узнать текущую позицию курсора.
11. Установить цвет фона и рисования.
12. Прочитать символ с клавиатуры не отображая его на экране.
13. Узнать есть ли символы в буфере клавиатуры.
14. Очистить экран (+атрибут).
15. Скрыть курсор.
16. Вывести на экран символ с указанным атрибутом (вывод с форматированием (a:4)).

Библиографический список

1. Пильщиков В.Н. Программирование на языке ассемблера IBM PC. – М.: Диалог-МИФИ, 2014 – 288 с.
2. Юров В. Assembler: учебник.– СПб: Питер, 2010. – 637 с.
3. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование (в 4-х книгах). – М.: ГРАНАЛ, 1993.– 382 с.
4. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера: пер. с англ. – М.: Радио и связь, 1991. – 336 с.

Приложение 1
Система команд МП i8086

Таблица 1

Система команд МП i8086 [3,4]

Мнемоника	КОП	*	oditszapc	Комментарии
AAA	37		?---??*?*	AL:=(AL исправленный для ASCII-сложения)
AAD	D50A		?---**?*?	AH:AL:=(AH:AL подготовить для деления BCD)
AAM	D40A		?---**?*?	AH:AL:=(AH:AL испр. для ASCII-умножения)
AAS	3F		?---??*?*	AL:=(AL исправленный для ASCII-вычитания)
ADC ac,im	14 im	*	*---*****	dst:=(src + dst + CF) сложить, учитывая перенос, результат в dst
ADC r/m8,im8	80 /2 im	*		
ADC r/m8 ,r8	10	*		
ADC r8,r/m8	12	*		
ADD ac,im	04 im	*	*---*****	dst:=(src + dst); сложить два операнда, результат в dst
ADD r/m8,im8	80 /0 im	*		
ADD r/m8 ,r8	00	*		
ADD r8,r/m8	02	*		
AND ac,im	24 im	*	0---**?*0	dst:= dst & src сброс битов dst, равных 0 в src
AND r/m8,im8	80 /4 im	*		
AND r/m8 ,r8	20	*		
AND r8,r/m8	22	*		
CALL label			---------	запомнить адрес возврата в стеке, передать управление
CALL far im	9A			вызов межсегментный прямой
CALL far r/m	FF /3			вызов межсегментный косвенный
CALL near r/m	FF/2			вызов близкий косвенный

Продолжение таблицы 1

Мнемоника	КОП	*	oditszapc	Комментарии
CALL near im	E8			вызов близкий, смещение относительно следующей команды
CBW	98		---------	Преобразовать байт в слово AH:=(заполнен битом 7 из AL)
CLC	F8		--------0	CF:=0 очистить флаг переноса
CLD	FC		-0-------	DF:=0 очистить флаг направления
CLI	FA		--0------	IF:=0 запретить маскируемые аппаратные прерывания
CMC	F5		--------*	CF:=~CF инвертировать флаг переноса
CMP ac,im	3C im	*	*---*****	сравнение выполняется как
CMP r/m8,im8	80 /7 im	*		неразрушающее вычитание:
CMP r/m8, r8	38	*		флаги:=(dst - src).
CMP r8,r/m8	3A	*		Влияет только на флаги.
CMPSB	A6		*---*****	[byte ptr DS:SI]-[byte ptr ES:DI], si,di+=□Δ; □ □ =±1
CMPSW	A7			[word ptr DS:SI]-[word ptr ES:DI], si,di+=□Δ; Δ =±2
CWD	99		---------	word ->dword; DX:=(заполнен 15-м битом из AX)
DAA	27		?---*****	AL:=(AL исправленное для BCD-сложения)
DAS	2F		?---*****	AL:=(AL исправленное для BCD-вычитания)
DEC r/m8	FE /1	*	*---****-	dst:=(dst - 1)
DEC r16	48+RW			вычесть 1 из dst

Продолжение таблицы 1

Мнемоника	КОП	*	oditszapc	Комментарии
DIV r/m8	F6 /6		?---?????	разделить аккумулятор на байт без знака (со знаком) AL:=(AX div src8); AH:=(AX MOD src8)
IDIV r/m8	F6 /7		?---?????	
DIV r/m16	F7 /6			AX:=(DX:AX div src16); DX:=(DX:AX MOD src16)
IDIV r/m16	F7 /7			
IMUL r8/m8	F6 /5		*---????*	умножить AL на целое со знаком. Результат в AX.
IMUL r16/m16	F7 /5			DX:AX:=(AX * src16)
IN ac,im8	E4 im	*		чтение в AL/AX из порта I/O
IN ac,DX	EC im	*		AL:=[порт]; AH:=[порт+1]
INC r/m8	FE /0	*	*---****-	dst:=(dst+1)
INC r16	40+rw		*---****-	прибавить 1 к dst
INSB / INSW	6C / 6D		---------	ES:[DI]:=(байт/ слово из порта DX); DI±=Δ;
INT type			--00-----	выполнить программное прерывание: pushf; IF=0; tf=0; push CS; push IP; IP:= 0000:[type * 4]; CS := 0000:[(type * 4) + 2]
INT 3	CC		--00-----	ловушка отладчика
INTO	CE		--**-----	прерывание по переполнению (если OF=1, то INT 4)
INT im8	CD		--00-----	см. INT type, INT 3
IRET	CF		*********	возврат из прерывания. (POP IP; POP CS; POPF)

Продолжение таблицы 1

Мнемоника	КОП	*	oditszapc	Комментарии
Jcond s_lab	см. табл. 1.4		---------	переход по условию: IP=IP+(8-битное смещение, расширенное со знаком до 16 бит)
JMP label			---------	безусловная передача управления на метку
JMP short_im8	EB			short: IP:=IP+(смещение цели, расширенное со знаком)
JMP near_im16	E9			near: IP:= near_im16
JMP near r/m16	FF /4			near: IP:= r/m16
JMP far im16:16	EA			far: CS:=целевой_сегмент; IP:=целевое_смещение
JMP far m	FF /5			
LAHF	9F		---------	загрузить флаги в AH
LEA r16,m	8D		---------	загрузить адрес в регистр reg:=(результат вычисления исполнительного адреса)
LDS r16,m32	C5		---------	загрузить DS и reg16 из поля памяти: r:=[m16]; DS:=[m16+2]
LES r16,m32	C4		---------	загрузить ES и reg16 из поля памяти: r:=[m16]; ES:=[m16+2]
LODSB	AC		---------	загрузка байта из строки в AL(AX); AL := DS:[SI]; SI+=1;
LODSW	AD		---------	AX := DS:[SI]; SI+=2;
LOOP s_lab	E2		---------	организация цикла: CX:=(CX-1) и short переход если CX≠0
LOOPE/LOOPZ	E1		---------	организация цикла: CX:=(CX-1) и short переход если CX≠0 && ZF=1
LOOPNE / LOOPNZ	E0		---------	организация цикла: CX:=(CX-1) и short переход если CX≠0 && ZF=0

Продолжение таблицы 1

Мнемоника	КОП	*	oditszapc	Комментарии
MOV dst,src			---------	переслать из src в dst
MOV r/m8,r8	88	*		
MOV r8,r/m8	8A	*		
MOV r/m16,seg	8C			
MOV seg,r/m16	8E			
MOV ac,m	A0	*		
MOV m,ac	A2	*		
MOV r8,im8	B0+rb			
MOV r16,im16	B8+rw			
MOV r/m8,im8	C6 im	*		
MOVS dst,src			---------	копировать строку байт (слов) (байт: Δ=1, слово: Δ=2) es:[di]:=ds:[si]; di+=Δ; si+=Δ
MOVSB	A4			
MOVSW	A5			
MUL r/m8	F6 /4		*---????	умножить AL на значение без знака: AX:=(AL * src8)
MUL r/m16	F7 /4			умножить AX на значение без знака: DX:AX:=(AX * src16)
NEG r/m8	F6 /3	*	*---*****	dst:=(0 - dst); изменить знак
NOP	90		---------	отсутствие операции
NOT r/m8	F6 /2	*	---------	dst:=~dst (инверсия всех бит dst)
OR ac,im	24 im	*	0---**?*0	dst:=dst | src (установка битов dst, равных 1 в src)
OR r/m8,im8	80 /1 im	*		
OR r/m8 ,r8	08	*		
OR r8,r/m8	0A	*		
OUT DX,ac	EE	*	---------	вывод из AL/ AX в порт ввода-вывода
OUT im,ac	E6	*		

Продолжение таблицы 1

Мнемоника	КОП	*	oditszapc	Комментарии
OUTSB	6E		---------	вывод из DS:[SI] в порт ввода-вывода
OUTSW	6F		---------	
POP m16	8F /0		---------	r/m:=SS:[SP]; SP+=2 извлечь из стека в r/m16
POP r16	58+rw			
POP ds/es/ss	1F/ 07/17			sreg := SS:[SP]; SP+=2; cs – недопустим
POPF	9D		*********	POP Flags: flags:=SS:[SP]; SP+=2
PUSH r16	50+rw		---------	SP-=2; SS:[SP]:=r/m/im
PUSH m16	FF /6		---------	
PUSH im16	68		---------	
PUSH CS/DS	0E/1E		---------	
PUSH SS/ES	16/06		---------	
PUSHF	9C		---------	PUSH Flags: переслать регистр флагов в стек; SP-=2; SS:[SP]:=флаги
RCL r/m8,1	D0 /2	*	*-------*	← CF ← [7..0] ← CF циклический сдвиг влево через перенос
RCL r/m8,CL	D2 /2	*		
RCR r/m8,1	D0 /3	*	*-------*	CF → [7..0] → CF циклический сдвиг вправо через перенос
RCR r/m8,CL	D2 /3	*		
ROL r/m8,1	D0 /0	*	*-------*	← CF ← [7..0] ← CF циклический сдвиг влево
ROL r/m8,CL	D2 /0	*		
ROR r/m8,1	D0 /1	*	*-------*	CF → [7..0] → CF циклический сдвиг вправо
ROR r/m8,CL	D2 /1	*		
REP без LODS	F3		---------	(префикс) CX:=(CX-1); повторять строковую операцию пока CX ≠ 0
REP перед LODS	F2		---------	

Продолжение таблицы 1

Мнемоника	КОП	*	oditszapc	Комментарии
REPE/REPNE	F3/F2		---------	(префикс) CX:=(CX-1); повторять пока (CX≠0 и ZF ? 0)
RET near im	C2		---------	выход из подпрограммы с удалением im байт из стека
RETF im	CA		---------	
RET near	C3		---------	выход из подпрограммы
RETF	CB		---------	
SAHF	9E			flags:=AH
SAL r/m8,1	D0 /4	*	*-------*	арифметический сдвиг влево
SAL r/m8,CL	D2 /4	*		CF← [7 .. 0] ← 0
SAR r/m8,1	D0 /7	*	*-------*	[7]+[6 .. 0] → CF
SAR r/m8,CL	D2 /7	*		арифметический сдвиг вправо
SHL r/m8,1	D0 /4	*	*-------*	← CF ← [7..0] ← 0
SHL r/m8,CL	D2 /4	*		логический сдвиг влево
SHR r/m8,1	D0 /5	*	*-------*	0 → [7..0] → CF
SHR r/m8,CL	D2 /5	*		логический сдвиг вправо
SBB ac,im	1C im	*	*---*****	dst:=((dst - src) - CF)
SBB r/m8,im8	80 /3 im	*		вычесть, учитывая заём
SBB r/m8 ,r8	18	*		
SBB r8,r/m8	1A	*		
SCASB/SCASW	AE/AF		---------	флаги:=(рез-тат CMP DS:[DI],AL/AX); DI+=1 или 2
STC/STD	F9/FD		-*------*	установить флаг CF:=1 (переноса) / DF:=1 (направления)
STI	F1		--1------	IF:=1 разрешить маскируемые аппаратные прерывания

Окончание таблицы 1

Мнемоника	КОП	*	oditszapc	Комментарии
STOSB	AA		---------	ES:[DI]:=AL; DI+=1;
STOSW	AB		---------	ES:[DI]:=AX; DI+=2;
SUB al/ax,im	2C/2D		*---*****	dst:=(src + dst)
SUB r/m8,im8	80 /5	*		сложить два операнда,
SUB r/m8 ,r8	28	*		результат в dst
SUB r8,r/m8	2A	*		
TEST ac,im	A8 im	*	0---**?*0	неразрушающее И
TEST r/m8,im8	F6 /0 im	*		флаги:=(как для dst & src)
TEST r/m8 ,r8	84	*		
XCHG ax,r16	90+rw		---------	dst ⇔ src
XCHG r/m,r8	86	*	---------	
XLAT	D7		---------	AL:=ES:[BX+(AL)]
XOR ac,im	34 im	*	0---**?*0	dst:=(dst ^ src);
XOR r/m8,im8	80 /6 im	*		инверсия битов dst,
XOR r/m8 ,r8	30	*		равных 1 в src
XOR r8,r/m8	32	*		

Используемые обозначения:

ac	операнд в аккумуляторе – AL / AX
r	операнд в регистре – AL, AH, BL, BH, CL, CH, DL, DH, AX, BX, DX, CX, SI, DI, BP, SP
m	операнд в памяти – метка, символьное имя, переменная
seg	операнд в сегментном регистре: CS, DS, ES, SS
r/m	операнд в регистре или в памяти
im	операнд – непосредственное значение (const, имя)
scr	источник
dst	приемник
a+=b	a:=a+b
a-=b	a:=a+b
a±=b	a:=a±b
Δ	обозначено приращение на 1 или 2 (величина зависит от команды или операнда)
/цифра	показывает, что байт ModR/M кодирует только операнд r/m, а поле REG/КОП содержит цифру, являющуюся расширением кода операции.
+rb, +rw	код регистра (см. таблицу 1.1), который прибавляется к базовому коду операции с образованием одного байта кода операции.

В четвёртом столбце показано, какое влияние команда оказывает на флаги:

?	флаг не определён;
-	флаг не изменился;
*	флаг выставлен этой командой в соответствии с результатом.

Команды, отмеченные звёздочкой в 3 столбце, могут оперировать не только 8 но и 16-разрядными операндами. Код операции для команды с 16-разрядными операндами получается из приведённого установкой младшего бита.

Таблица 2

Команды условной передачи управления

Команда	КОП	переход если...	Условие перехода
JA	77	выше (больше) для без знаковых	CF=0 и ZF=0
JAE	73	выше или равно для без знаковых	CF=0
JB	72	ниже для без знаковых	CF=1
JBE	76	ниже или равно для без знаковых	CF=1 или ZF=1
JC	72	Carry – перенос	CF=1
JCXZ	E3	CX=0	CX=0
JE/JZ	74	равно	ZF=1
JG	7F	Больше для знаковых	SF=OF & ZF=0
JGE	7D	больше или равно для знаковых	SF=OF
JL	7C	Меньше для знаковых	ZF≠OF
JLE	7E	меньше или равно для знаковых	SF≠OF | ZF=1
JNB	73	не ниже для без знаковых	CF=0
JNBE	77	не ниже или равно для без знаковых	CF=0 и ZF=0
JNC	73	не устновлен флаг переноса	CF=0
JNE/JNZ	75	не равно	ZF=0
JNG	7E	не больше для знаковых	SF≠OF или ZF=1
JNGE	7C	не больше или равно для знаковых	SF≠OF
JNL	7D	не меньше для знаковых	SF=OF
JNLE	7F	не меньше или равно для знаковых	ZF=0 и SF=OF
JNO	71	нет переполнения	OF=0
JNP	7B	сумма битов нечётна	PF=0
JNS	79	знаковый разряд нулевой	SF=0
JO	70	переполнение	OF=1
JP/ JPE	7A	число единичных битов чётно	PF=1
JPO/JNP	7B	сумма битов нечётна	PF=0
JS	78	знак	SF=1

Приложение 2
Ключи опций программ TASM и TLINK

Ключи TASM

/a	Сегменты в объектном файле размещать в алфавитном порядке;
/s	Сегменты в объектном файле следуют в порядке их описания в программе;
/c	Указание на включение в файл листинга информации о перекрёстных ссылках;
/dимя[= знач.]	Определяет идентификатор. Это эквивалент директивы ассемблера =, как если бы она была записана в начале исходного текста программы;
/e	Генерация инструкций эмуляции операций с плавающей точкой;
/r	Разрешение трансляции действительных инструкций с плавающей точкой, которые должны выполняться реальным арифметическим сопроцессором;
/h, /?	Вывод на экран справочной информации;
/iпуть	Задаёт путь к включаемому по директиве INCLUDE файлу. Синтаксис аргумента “путь” такой же, как для команды PATH файла autoexec.bat;
/jдиректива	Определяет директивы, которые будут транслироваться перед началом трансляции исходного файла программы на ассемблере. В директиве не должно быть аргументов;
/khn	Задаёт максимальное количество идентификаторов, которое может содержать исходная программа, то есть задаёт размер таблицы символов транслятора. По умолчанию может быть до 16384 идентификаторов. Это значение можно увеличить до 32 768 или уменьшить до n. Сигналом к тому, что необходимо использовать данный параметр, служит появление сообщения “Out of hash space” (“Буферное пространство исчерпано”);
/l, /la	/l - создать файла листинга, даже если он не “заказывается” в командной строке; /la - показать в листинге код, вставляемый транслятором для организации интерфейса с языком высокого уровня по директиве MODEL;
/ml	Различать во всех идентификаторах прописные и строчные буквы;
/mx	Различать строчные и прописные символы во внешних и общих идентификаторах. Это важно при компоновке с программами на тех языках высокого уровня, в которых строчные и прописные символы в идентификаторах различаются;

/mu	Воспринимать все символы идентификаторов как прописные;
/mvn	Определение максимальной длины идентификаторов. Минимальное значение n = 12;
/mn	Установка количества (n) проходов транслятора TASM. По умолчанию транслятор выполняет один проход. Максимально при необходимости можно задать выполнение до 5 проходов;
/n	Не выдавать в файле листинга таблицы;
/os, /o,	/op, /oi - генерация оверлейного кода;
/p	Проверять наличие кода с побочными эффектами при работе в защищенном режиме;
/q	Удаление из объектной программы лишней информации, ненужной на этапе компоновки;
/t	Подавление вывода всех сообщений при условном ассемблировании, кроме сообщений об ошибках;
/w0, /w1, /w2	Генерация предупреждающих сообщений разного уровня полноты: w0 – сообщения не генерируются; w1, w2 – сообщения генерируются;
/w-xxx, /w+xxx	Генерация предупреждающих сообщений класса xxx (эти же функции выполняют директивы WARN и NOWARN). Знак “-” означает “запрещена”, а “+” - “разрешена”.
/x	Включить в листинг все блоки условного ассемблирования для директив IF, IFNDEF, IFDEF и т. п., в том числе и не выполняющиеся;
/z	При возникновении ошибок наряду с сообщением о них выводить соответствующие строки текста;
/zi	Включить в объектный файл информацию для отладки;
/zd	Поместить в объектный файл информацию о номерах строк, что необходимо для работы отладчика на уровне исходного текста программы;
/zn	Запретить помещение в объектный файл отладочной информации.

Ключи TLINK (чувствительны к регистру!)

/x	Не создавать файл с картой памяти (map)
/m	Создать файл карты
/s	Создать файл карты памяти с дополнительной информацией о сегментах (адрес, длина в байтах, класс, имя сегмента и т. д.)
/l	Создать раздел в файле карты с номерами строк
/L	Спецификация пути поиска библиотеки
/n	Не использовать библиотеки
/c	Различать строчные и прописные буквы в идентификаторах
/v	Включить отладочную информацию в исполняемый файл
/3	Поддержка 32-битного кода
/d	Предупреждать о дублировании идентификаторов в подключаемых библиотеках
/t	Создать файл типа .com (по умолчанию .exe)
/i	Инициализировать все сегменты
/o	Ключ для оверлейных перекрытий
/Txx	Выбрать тип выходного файла: /Tdx - DOS (по умолчанию); /Twx – Windows; (третья буква может быть c=COM, e=EXE, d=DLL)
/P	[=NNNNN] Упаковка кодового сегмента
/A=	NNNN Установить выравнивание адресов в новом EXE сегменте

Приложение 3
Справочник прерываний IBM PC

Таблица 3

Функции ввода-вывода MS DOS

№	Функция	Вход	Выход
1	Ввод символа с эхом с клавиатуры	-----	AL=символ
2	Вывод символа на экран	DL=символ	-----
6	Ввод/вывод символа на CON без контроля	DL=символ	при AL=FFh ввод: -> AL = символ + ZF=0 =успех, иначе вывод. Если символа нет функция завершается
7	Ввод символа с CON без эха и контроля	-----	AL=символ
8	Ввод символа без эха с CON	-----	AL=символ
9	Вывод строки на CON (до '$')	DS:DX = адрес строки	-----
A	Чтение строки в буфер	DS:DX=адрес буфера	заполненный буфер

Формат	max	кол	s1,s2,s3,...	0D	.....
буфера:			Прочитанные символы	CR	не определено

MAX - число символов, на которое рассчитан буфер;

КОЛ - количество введенных символов без учёта CR;

s1,s2,s3,... - прочитанные символы, включая завершающий CR (признак конца введенной строки).

4C	Завершить процесс с кодом возврата	AL=код возврата	-----

Функция вызывается через 21h прерывание. Номер функции должен при вызове находиться в регистре AH.

Таблица 4

Функции файлового ввода-вывода MS DOS

№	Функция	Вход	Выход
3C	Создать/усечь файл	CX=атрибут; DS:DX=ASCIIZ имя файла	CF=0 ⇒ AX=дескриптор иначе {ошибка 3,4,5}
3D	Открыть существующий файл	AL=доступ/разделение; DS:DX = ASCIIZ имя файла; CL=атрибуты для сервера	CF=0 ⇒ AX=дескриптор файла иначе ={ошибка 1,2,3,4,5,0c}
3E	Закрытие файл	BX=дескриптор файла	CF=0 или {AX=ошибка 6}
3F	Чтение из файла или устройства	BX=дескриптор файла; CX=число считанных байт; DS:DX=адрес буфера	AX=кол-во считанных байт; AX=0 при чтении за концом файла или {ошибка 5, 6}
40	Запись данных в файл или на устройство	BX=дескриптор файла; CX=число записыв. байт; DS:DX=данные для записи Если CX=0, то файл усекается до текущей позиции файлового указателя	CF=0 ⇒ AX=кол-во реально записанных байт; иначе {AX=ошибка 5,6}
41	Удалить файл	DS:DX= ASCIZ имя файла; CL=атрибуты для сервера	если CF=1 то в AX=код ошибки 2, 3, 5
42	Установить указатель в файле	BX=дескриптор файла; CX:DX=смещение указателя; AL=точка отсчёта: = 0 - от начала; = 1 - от текущей позиции; = 2 - от конца файла.	CF=0 ⇒ DX:AX= значение новой позиции в байтах, относительно начала файла иначе {AX=ошибка 1, 6};
5B	Создать новый файл с сохранением существующего	CX=атрибуты; DS:DX=ASCIIZ имя файла	CF=0 ⇒ AX=дескриптор; иначе {AX=ошибка 50h}
68	Принудительно сбросить в файл	BX=дескриптор файла;	CF=0 ⇒ успех

Продолжение таблицы 4

№	Функция	Вход	Выход
6C	Расширенное открытие файла	AL=00H; BX=режимы доступа/ разделения; CX=атрибуты файла; DX=флаг открытия; DS:SI=ASCIIZ имя файла; DI=порядковый номер – добавка к псевдониму (только для 716Ch), если установлен 10 бит в BX.	CF=0 ⇒ AX=дескриптор; CX=действие: 1=файл существовал, открыт; 2=файл не существовал, создан; 3=файл существовал, усечен.

Функция вызывается через 21h прерывание. Номер функции должен при вызове находиться в регистре AH.

Режимы доступа и разделения для функций 3D и 6C.

Бит(ы)	Описание	Значение
0-2	тип доступа	000=R/O 001=W/O 010=R/W
3=0	резерв	
4-6	режим разделения	000=совместимость 100=разрешено всё 001=запрет чтения/записи другими 010=запрет записи 011=запрет чтения
7	наследование	=1 – if файл не наследуется дочерним процессом
8-12 = 0	резерв	
13	обработка критической ошибки	0=использовать стандартный обработчик ошибок (Int 24h) 1=вернуть процессу ошибку
14	сквозная запись	0=использовать буферизацию; 1=без буферизации (физическая запись в момент запроса)
15=0	резерв	

Атрибуты для функций 3C, 3D, 41,43, 56, 5A, 5B.

Биты	Если бит установлен.
0	Read only – только чтение
1	Hidden – скрытый;
2	System – системный;
3	Volume – метка тома;
4	Directory – элемент каталога;
5	Arhiv – архивный;
6-15	= 0 зарезервированы

Флаг открытия для функции 6C:

Бит(ы)	Описание	Значение
0-3	если файл существует	0000=вернуть ошибку 0001=открыть файл 0010=заменить файл (открыть / усечь)
4-7	если файл не существует	0000=вернуть ошибку 0001=создать файла
8-15=0	резерв	

Коды некоторых ошибок:

1h – неверное значение в AL.

2h – файл не найден.

3h – путь не найден.

4h – нет свободного дескриптора.

5h – доступ запрещен.

6h – недопустимый дескриптор.

0Ch – недопустимый код доступа.

50h – файл существует.

7100h – функция не поддерживается.

Таблица 5

Функции ПЗУ BIOS (int 10h)

№	Функция	Вход	Выход
0	Выбрать видеорежим	AL=видеорежим	–
1	Установить тип курсора	CH / CL биты 0-4=начальная / конечная строка курсора;	–
2	Установить поз. курсора	BH=страница DH=строка (OY) DL=столбец (OX)	–
3	Получить позицию курсора	BH=страница	CH / CL =первая / последняя строка развёртки курсора DH / DL =строка (OY) / столбец (OX)
5	Установить видеостраницу	AH=05h; AL=страница: 0-7 для режимов 0, 1, 2, 3, 7 и 0Dh; 0-3 для режимов 2, 3 и 0Eh; 0-1 для режимов 0Fh и 10h	–
6	Инициализировать или прокрутить окно вверх	AL=число строк прокрутки (если 0, всё окно очищается); BH=атрибут, используемый для очищаемой области; CL, CH = X, Y-координаты, верхнего левого угла окна; DL, DH = X, Y-координаты, нижнего правого угла окна.	–
7	Инициализировать или прокрутить окно вниз		–
8	Прочитать символ и атрибут в позиции курсора	BH=страница	AH=атрибут; AL=символ
9	Записать символ в позицию курсора	AL=символ; BH=страница; BL=атрибут (текст. реж.) или цвет (граф.реж); CX=число записываемых символов	–
0a	Запись символа в позицию курсора	AL=символ; BH=страница CX=число записываемых символов	–

Приложение 4
Примеры программ

Пример 1. Многомодульная программа

Модуль ввода/вывода десятичных чисел

```
public InputInt, OutputInt
code segment public
assume cs:code, ds:data
InputInt proc near       ; Подпрограмма ввода числа
     push bx             ; из диапазона 0-65535 с клавиатуры
     push cx             ; в десятичной системе счисления
     push dx             ; На выходе в регистре AX находится
     push si             ; введённое число
     mov dx,offset strdsc
     mov ah,0Ah
     int 21h
     mov dl,0Ah
     mov ah,2
     int 21h

     xor ax,ax
     xor cx,cx
     mov cl,[strdsc+1]
     mov si,offset strbuf
     mov bx,10

s1:  mul bx
     mov dl,[si]
     inc si
     sub dl,30h
     add ax,dx
     loop s1
     pop si
     pop dx
     pop cx
     pop bx
     ret
InputInt  endp
```

```
OutputInt proc near      ; Подпрограмма вывода числа
     push ax             ; из диапазона 0-65535 на экран
     push bx             ; в десятичной системе счисления
     push cx             ; На входе в регистре AX должно
     push dx             ; находиться выводимое число
     mov bx,10
     xor cx,cx

n1:  xor dx,dx
     div bx
     inc cx
     push dx
     or ax,ax
     jnz n1

     mov bx,offset strbuf
n2:  pop dx
     add dl,30h
     mov [bx],dl
     inc bx
     loop n2
     mov byte ptr [bx],'$'

     mov dx,offset strbuf
     mov ah,9
     int 21h
     pop dx
     pop cx
     pop bx
     pop ax
     ret
     endp OutputInt
code ends
data segment public
strdsc db 6,0
strbuf db 6 dup (?)
data ends
end
```

Пример программы, использующей модуль:

```
extrn InputInt:near, OutputInt:near

code segment public
assume cs:code, ds:data, ss: stek
start:
        mov ax,data
        mov ds,ax
        call InputInt
        call OutputInt
        mov dx,offset strend
        mov ah,9
        int 21h

        mov ax,4c00h
        int 21h
code ends

data segment public
strend  db 13,10,'$'
data ends

Stek segment stack
        dw      128 dup (?)
Stek ends
end start
```

Пример 2. Поиск первого вхождения числа в массив двухбайтных чисел.

```
extrn InputInt:near      ; подпрограмма ввода числа
extrn OutputInt:near     ; подпрограмма вывода числа

code segment public
assume cs:code, ds:data, ss: stek
start:
     mov ax,data         ; настроим на начало сегмента данных
     mov ds,ax           ; DS
     mov es,ax           ; и ES (для команд обработки цепочек)

     mov dx,offset strA  ; вывод строки 'Введите массив'
     mov ah,9
     int 21h
                         ; ввод массива
     mov di,offset Arr   ; di <- адрес первого элемента массива
     mov cx,5            ; cx <- количество элементов
     cld                 ; установить направление увеличения
linput:                  ; адреса (для строковых команд)
     call InputInt       ; ввод - ax <- элемент массива
     stosw               ; сохранить элемент
     loop linput         ; цикл пока cx<>0

     mov dx,offset strC  ; вывод строки 'Введите число'
     mov ah,9
     int 21h
     call InputInt       ; ввод числа

     mov cx,5            ; cx <- количество элементов
     mov di,offset Arr   ; di <- адрес первого элемента массива
     call FindNum        ; вызов п/п поиска
     or  bx,bx           ; проверка BX
     jz  no_pos          ; переход, если ноль

     mov dx,offset strP  ; вывод строки 'Позиция:'
     mov ah,9
     int 21h
     mov ax,bx           ; вывод позиции
     call OutputInt
     jmp endprg
no_pos:
```

```
     mov dx,offset strN       ; вывод строки 'Не найдено'
     mov ah,9
     int 21h
endprg:
     mov dx,offset strend     ; перевод строки
     mov ah,9
     int 21h

     mov ax,4c00h             ; завершение EXE-программы
     int 21h
;подпрограмма поиска числа в массиве
     ;вход:    AX - искомое число,
     ;         DS:DI - адрес массива,
     ;         CX - количество элементов
     ;выход:   BX - позиция числа в массиве, 0 если не найден
FindNum   proc near
          repne scasw         ; поиск вхождения в массив
          jne no_find         ; переход, если не найдено
          mov bx,5            ; вычисление позиции BX=5-CX
          sub bx,cx
          ret
no_find:  xor bx,bx           ; BX=0 (элемент не найден)
          ret
FindNum endp
code ends

data segment public
Arr    dw 5 dup(0)            ; массив из 5-ти двухбайтных эл-
тов
strA   db 'Input Array',13,10,'$'
strC   db 'Input Number',13,10,'$'
strP   db 'Position:$'
strN   db 'No Position$'
strend db 13,10,'$'
data ends

Stek segment   stack
     dw   128 dup (?)
Stek ends

end start
```

Пример 3. Консольное приложение, использующее ассемблерные подпрограммы.

```
program lab7_c;
//Программа обработки массива 32-разрядных целых со знаком
{$APPTYPE CONSOLE}

type    TArray=array [1..20] of integer;
var     Arr,Arr1            : TArray;
        i,n,n1  : integer;

//Решение на Object Pascal
function AbsArrPAS(var A:TArray; N:integer):integer;
var i,c:integer;
begin
c:=0;
for i:=1 to N do
      if A[i]<0 then begin inc(c); A[i]:=-A[i]; end;
AbsArrPAS:=c;
end;

//Решение на встроенном ассемблере
function   AbsArrPASASM   (var   A:TArray;   N:integer):integer;
assembler;
asm
        mov     esi,A
        mov     ecx,N
        xor     bx,bx
@povt:  mov     eax,[esi]
        or      eax,eax
        jns     @next
        inc     bx
        neg     eax
        mov     [esi],eax
@next:  add     esi,4
        loop    @povt
        movzx   eax,bx
end;
```

```
// Решение на ассемблере. Подключить можно любую из двух
// подпрограмм: AbsArr2.obj или AbsArr1.obj

{$L AbsArr2.obj}
function AbsArrASM(var A:TArray; N:integer):integer;
pascal; external;
begin
writeln('Input N:');  readln(n);
if N>20 then N:=20;
for i:=1 to N do Arr[i]:=random(100)-50;

writeln('Array:');
for i:=1 to N-1 do write(Arr[i]:3,',');
writeln(Arr[N]:3);

for i:=1 to N do Arr1[i]:=Arr[i];
n1:=AbsArrPAS(Arr1,N);
writeln('AbsArray(PAS):');
for i:=1 to N-1 do  write(Arr1[i]:3,',');
writeln(Arr1[N]:3);
writeln('Less0=',n1);

for i:=1 to N do Arr1[i]:=Arr[i];
n1:=AbsArrPASASM(Arr1,N);
writeln('AbsArray(PASASM):');
for i:=1 to N-1 do  write(Arr1[i]:3,',');
writeln(Arr1[N]:3);
writeln('Less0=',n1);

for i:=1 to N do Arr1[i]:=Arr[i];
n1:=AbsArrASM(Arr1,N);
writeln('AbsArray(ASM):');
for i:=1 to N-1 do  write(Arr1[i]:3,',');
writeln(Arr1[N]:3);
writeln('Less0=',n1);
end.
```

```
; Файл AbsArr1.asm
.386
public AbsArrASM
code segment public use32
assume cs:code
AbsArrASM proc
        push ebp            ; сохранить ebp
        mov ebp,esp         ; запомнить в ebp указатель на
                            ; параметры в стеке
        push esi
        push ebx
        mov esi,[ebp+0Ch]   ; esi  <- адрес массива
        mov ecx,[ebp+8]     ; ecx  <- количество элементов
        xor bx,bx           ; bx <- 0 (счётчик отрицательных
                            ; элементов)
povt:   mov eax,[esi]       ; загрузить значение из массива
        or  eax,eax         ; установить флаги по значению
        jns short next      ; переход если не отрицательное
        inc bx              ; увеличить счётчик отрицательных
        neg eax             ; преобразовать в положительное
        mov [esi],eax       ; сохранить в массив

next:   add esi,4           ;получить адрес следующего элемента
        loop povt           ; цикл ecx=ecx-1 пока ecx<>0
        movzx eax,bx        ; возвратить значение счётчика
        pop ebx             ; отрицательных чисел в eax
        pop esi
        pop ebp             ; восстановить ebp
        ret 8               ; выход из п/п с удалением
; параметров из стека,
; 8 байт = 4(адрес массива)+4(количество элементов)
AbsArrASM endp
code ends
end
```

```
; Файл AbsArr2.asm

.386
.MODEL FLAT,PASCAL
.CODE
public AbsArrASM
AbsArrASM proc A:DWORD, N:DWORD
        push esi
        push ebx
        mov esi,[ebp+0Ch]   ; esi <- адрес массива
        mov ecx,[ebp+8]     ; ecx <- количество элементов
        xor bx,bx           ;bx  <- 0 (счётчик отрицательных
povt:                       ; элементов)
        mov eax,[esi]       ; загрузить значение из массива
        or eax,eax          ; установить флаги по значению
        jns short next      ; переход если не отрицательное
        inc bx              ;увеличить счётчик отрицательных
        neg eax             ; преобразовать в положительное
        mov [esi],eax       ; сохранить в массив
next:
        add esi,4           ;получить адрес следующего элемента
        loop povt           ; цикл ecx=ecx-1 пока ecx<>0
        movzx eax,bx        ; возвратить значение счётчика
        pop ebx             ; отрицательных чисел в eax
        pop esi
        ret                 ; выход из п/п
AbsArrASM endp
end
```