ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Белгородский государственный технологический университет
им. В. Г. Шухова

Вычислительные машины, системы и сети

Методические указания к выполнению лабораторных работ
для студентов бакалавриата направлений
220400.62 — Управление и в технических системах,
220700.62 — Автоматизация технологических процессов и
производств

Часть 2

Белгород
2013

УДК 004.3
ББК 32.97
В92

Составители: ст. преп. И. А. Рыбин
ассистент А. В. Шарпатая
ст. преп. А. В. Крюков

Вычислительные машины, системы и сети: методические указания к выполнению лабораторных работ для студентов специальностей 220400.62 — Управление и информатика в
В92 технических системах, 220700.62 — Автоматизация технологических процессов и производств / сост.: И. А. Рыбин, А. В. Шарпатая, А. В. Крюков. — Белгород: Изд-во БГТУ, 2013. — Ч. 2. — 55 с.

В данном издании приводятся методические указания к выполнению второй части курса лабораторных работ по дисциплине «Вычислительные машины, системы и сети». Рассматриваются основные команды действий и ветвлений, входящие в систему команд процессора Intel 8086, способы ввода и вывода данных с использованием функций операционной системы DOS, вопросы организации массивов данных в памяти вычислительной машины. Теоретические знания применяются для углубленного программирования на низкоуровневом языке Ассемблера (Assembler). Содержатся краткие теоретические сведения, контрольные вопросы для подготовки к защите, порядок выполнения и структура отчета по каждой лабораторной работе.

Методические указания предназначены для студентов 3-го курса специальностей 220400.62 — Управление и информатика в технических системах, 220700.62 — Автоматизация технологических процессов и производств, а также могут быть использованы при проведении лабораторных работ по другим дисциплинам, связанным с архитектурой вычислительных систем.

Издание публикуется в авторской редакции.

УДК 004.3
ББК 32.97

Содержание

Введение

Программирование на языках высокого уровня, казалось бы, вытеснило низкоуровневое общение с вычислительной машиной. Однако зачастую это не так, и области, где требуются специалисты, понимающие низкий уровень организации вычислительной машины очень востребованы. Например, программирование контроллерных и микроконтроллерных средств автоматизации и управления требует понимания внутреннего их устройства, которое приходит как следствие изучения принципов низкоуровневого программирования.

С другой стороны, даже с учетом принципиальных различий в гарвардской и принстонской архитектуре, можно заметить сходства в этих архитектурах, которые проявляются в том числе в системах команд, организации ввода или создании массивов данных. Конечно, это не сверхновое веяние в науке и технике, однако понимание таких вещей позволит ликвидировать компьютерную безграмотность и дать базу для освоения последних инноваций в вычислительной технике, способствующих повышению показателей функционирования управляющих устройств при управлении техническими и технологическими объектами промышленности.

Лабораторная работа № 1.
Система команд процессора Intel 8086. Команды действий

Цель работы

Изучение команд действий процессора i8086.

Содержание работы

— команды пересылки данных;
— арифметические команды;
— логические команды.

Теоретические сведения

Команды пересылки данных присваивают значение операнда источника операнду приемнику. Арифметико-логическое устройство (АЛУ) процессора при этом не используется, а операндами могут быть регистры процессора, ячейки памяти или устройства ввода/вывода. К таким командам относятся, например, команды `mov`, `xchg`, `push`, `pop` и другие. Они имеют следующий синтаксис.

```
mov <операнд приемник>, <операнд источник>
xchg <операнд1>, <операнд2>
push <операнд источник>
pop <операнд приемник>
```

Так, командой `mov AX, DX` значение регистра `DX` пересылается в регистр `AX`. А после выполнения команды `mov AX, DS:mem`[1] — значение регистра `AX` станет равным значению ячейки памяти, которая в тексте программы именована как mem и расположена в сегменте, на который указывает регистр `DS`. Команда `xchg AH, AL` обменяет значения старшей и младшей половин регистра `AX`, т. е. оба операнда являются и источником и приемником одновременно. В командах `push` и `pop` вторым операндом неявно задаются ячейки памяти в сегменте стека.

Рассмотрим особенности применения команды `mov` (табл. 1.1), которые также относятся и к другим командам действий.

---

1 Обычно в начале программы прописывается директива assume DS:<имя сегмента>, которая сообщает компилятору, что при обращении к переменной, находящейся в программе в сегменте <имя сегмента>, адрес сегмента будет находится в регистре DS. Тогда вместо `mov AX, DS:mem` можно просто записать `mov AX, mem`.

Таблица 1.1

Особенности работы команды `mov`

№	Особенность	Примеры[2]	
		правильно	**неправильно**
1	Нельзя осуществлять пересылку из одной области памяти в другую. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения.	`mov AX, x` `mov y, AX`	`mov x, y`

2	Нельзя загрузить в сегментный регистр значение непосредственно из памяти. Для выполнения такой загрузки нужно использовать промежуточный объект. Это может быть регистр общего назначения или стек.	`mov AX, data`
		`mov DS, AX`
		
		`и`
		
		`mov AX, x`
		`mov DS, AX`
			`mov DS, data`
			
			`и`
			
			`mov DS, x`

3	Нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр, так как в системе команд процессора нет соответствующего кода операции. Выполнить такую пересылку можно, используя в качестве промежуточных, те же регистры общего назначения или стек.	`mov AX, DS`
		`mov ES, AX`
		
		`или`
		
		`push DS`
		`pop ES`
			`mov ES, DS`

4	Нельзя использовать сегментный регистр CS в качестве операнда назначения. Причина в том, что в архитектуре микропроцессора пара CS:IP всегда содержит адрес команды, которая должна выполняться следующей. Изменение командой mov содержимого регистра CS фактически означало бы операцию перехода, а не пересылки, что недопустимо.	—	`mov CS, AX`

2 В примерах `x` и `y` — двухбайтовые ячейки памяти, объявленные в сегменте данных `data`, на который указывает регистр `DS` (`assume DS:data`):

```
…
data segment
   x dw 2
   y dw 5
data ends
…
```

Еще одна особенность выполнения рассматриваемых команд пересылки в том, что разрядность операндов должна быть одинаковой. Это же присуще арифметическим и логическим командам, приведенным далее.

Рассмотрим организацию *арифметических* операций сложения (табл. 1.2), вычитания (табл. 1.4), умножения и деления.

Таблица 1.2

Примеры арифметических команд сложения целых чисел без знака

№	Команда	Принцип действия	Пример	Примечание

1	`inc` операнд	операнд = операнд + 1	`mov CX, 000Fh`
			`inc CX`
			`;CX=0010h`
				Команда не изменяет флаг CF.

2	`add` операнд1, операнд2	операнд1 = операнд1 + операнд2	`mov BH, 0AAh`
			`add BH, 16h`
			`;AH=0C0h`
				Флаг переноса CF устанавливается в 1 в случае, когда результат сложения больше разрядности операндов.

3	`adc` операнд1, операнд2	операнд1 = операнд1 + операнд2 + CF	`mov AH, 0FEh`
			`mov DL, 05h`
			`add AH, DL`
			`adc DL,02h`
			`;AH=03h`
			`;DL=08h`
				Команда сложения с учетом флага переноса CF. Является средством для сложения длинных двоичных чисел.

Отметим также, что результат сложения влияет и на другие флаги. Так, если результат нулевой, то флаг нуля `ZF` устанавливается в 1, иначе — в 0. Это относится и к другим командам арифметических операций.

Для сложения чисел со знаком в процессоре используются также команды `inc`, `add`, `adc`. Но на самом деле процессор не подозревает о различии между числами со знаком и без знака. Вместо этого у него есть средство фиксации возникновения характерных ситуаций, складывающихся в процессе вычислений, например, флаг переноса `CF`. Для фиксирования случаев, когда при сложении двух положительных чисел получается отрицательный результат, или когда два отрицательных слагаемых дают результат положительный, используется флаг переполнения `OF`. Рассмотрим пример сложения однобайтовых чисел:

```
  0110 0101
+
  0101 0100
-------------
  1011 1001.
```

Старший бит результата указывает на получение отрицательного числа в результате сложения двух положительных, что с точки зрения элементарной математики абсурдно. В этом и подобных случаях флаг `OF` устанавливается в единицу. Логика установки флага `OF` приведена в табл. 1.3.

Таблица 1.3

Значения флага OF в зависимости от знаков операндов и знака результата

Старший бит операнда 1	Старший бит операнда 2	Старший бит результата	Значение флага OF
0	0	0	0
0	0	1	1
1	1	0	1
1	1	1	0

В остальных случаях флаг `OF` не устанавливается, т. е. переполнение при работе с одним положительным, а другим отрицательным операндом, не регистрируется.

Приведенные рассуждения о переносе и переполнении при сложении справедливы и для команд вычитания (табл. 1.4). Под переносом при вычитании подразумевается заем, когда вычитаемое больше уменьшаемого.

Таблица 1.4

Примеры арифметических команд вычитания целых чисел

№	Команда	Принцип действия	Пример	Примечание
1	dec операнд	операнд = операнд – 1	mov DL, 00h dec DL ;DL=0FFh	Команда не изменяет флаг CF.

2	sub операнд1, операнд2	операнд1 = операнд1 – операнд2	mov AH, 0Dh
			sub AH, 11h
			;AH=FCh= –3
			;CF=1
				Флаг переноса CF устанавливается в 1 в случае, когда уменьшаемое меньше вычитаемого.

3	sbb операнд1, операнд2	операнд1 = операнд1 – операнд2 – CF	mov AH, 06h
			mov AL, 04h
			sub AH, 08h
			sbb AL, 01h
			;AH=0FEh= –2
			;AL=02h
				Команда вычитания с учетом флага переноса CF. Является средством для вычитания длинных двоичных чисел.

В системе команд процессора есть команда позволяющая выполнять инвертировать значения операндов: операнд = 0 – операнд:

```
neg <операнд>
```

— отрицание с дополнением до двух. Ее можно применять для смены знака или вычитания из константы (вычитание из константы посредством команд sub и sbb невозможно):

```
neg AX
add AX, 200; AX = 200 – AX
```

Для умножения чисел без знака предназначена команда

```
mul <операнд>
```

В случае если операнд имеет размерность 1 байт, произойдёт перемножение операнда на значение регистра AL, а результат операции запишется в регистр AX. Если операнд является словом, то умножаться на операнд будет значение регистра AX. Произведение при этом запишется в пару регистров DX:AX, причём в DX запишутся старшие 2 байта результата. Более широкие возможности умножения предоставляет команда imul.

Для деления чисел без знака предназначена команда

```
div операнд
```

Операнд в этом случае является делителем числа, находящегося в регистре AX или в паре регистров DX:AX (табл. 1.5).

Таблица 1.5

Расположение делимого, частного и остатка в зависимости от размера операнда

Делимое	Делитель (операнд)	Частное	Остаток
AX	байт	AL	AH
DX:AX	слово	AX	DX

После выполнения команды деления значение флагов не определено, но возможно возникновение исключительных ситуаций в следующих случаях:

— делитель равен 0;

— делимое в AX больше делителя байта в 256 раз (при этом частное не вмещается в AL);

— делимое в DX:AX больше делителя слова в 65536 раз (частное не вмещается в AX).

Эти случаи приводят к возникновению прерывания с номером 0. Для деления чисел со знаком применяется команда idiv.

В систему команд процессора включен набор команд, поддерживающий работу с *логическими* данными, некоторые из которых приведены в табл. 1.6.

Таблица 1.6

Примеры логических команд процессора

№	Команда	Принцип действия	Пример	Примечание

1	`and <операнд1>, операнд2>`	<операнд1> = <операнд1> И <операнд2>	`mov AH, 00100110b`
			`and AH, 00110011b`
			`;AH=00100010b`
				Выполняется логическая операция И для каждой пары битов (0, если хотя бы один из двух битов равен 0)

2	`or <операнд1>, <операнд2>`	<операнд1> = <операнд1> ИЛИ <операнд2>	`mov AH, 00100111b`
			`mov CL, 10001100b`
			`or AH, CL`
			`;AH=10101111b`
			`;CL=10001100b`
				Выполняется логическая операция ИЛИ для каждой пары битов (1, если хотя бы один из двух битов равен 1)

3	`xor <операнд1>, <операнд2>`	<операнд1> = <операнд1> ИСКЛ. ИЛИ <операнд2>	`mov AH, 10110100b`
			`and AH, 00110011b`
			`;AH=10000111b`
				Выполняется логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ для каждой пары битов (если биты равны 0, иначе 1)

4	`test <операнд1>, <операнд2>`	<операнд1> И <операнд2>	`mov AH, 00100111b`
			`mov CL, 10001100b`
			`test AH, CL`
			`;AH=00100111b`
			`;CL=10001100b`
				Отличие от команды and только в том, что результат никуда не заносится, а лишь устанавливаются соответствующие флаги.

5	`not <операнд>`	НЕ <операнд>	`mov AL, 10100001b`
			`not AL`
			`;AL=01011110b`
				Выполняется поразрядное инвертирование каждого бита операнда.

Команда `xor` часто применяется для очистки регистра, например, после выполнения

```
xor AX, AX
```

значение регистра `AX` станет равным 0, в независимости от того, какое значение регистра `AX` было до выполнения команды.

Команда `test` используется для проверки значений битов в числах. Например,

```
test AX, 00000101b
```

установит флаг нуля в 1, если и нулевой и второй биты числа, хранящегося в регистре AX, одновременно равны 1, иначе ZF=0.

Также к логическим командам относятся команды логического сдвига:

shl <операнд>, <счетчик_сдвигов>

— логический сдвиг влево (Shift Logical Left). Содержимое операнда сдвигается влево на количество битов, определяемое значением <счетчик_сдвигов>. Справа на место сдвинутых младших битов записываются нули.

shr <операнд>, <счетчик_сдвигов>

— логический сдвиг вправо (Shift Logical Right). Аналогично команде shl, но сдвиг осуществляется вправо. Слева на место сдвинутых старших битов записываются нули.

При сдвиге очередного бита влево или вправо он переходит во флаг CF, при этом значение предыдущего сдвинутого бита теряется.

Команды сдвига удобно использовать при делении или умножении на число равное степени двойки. Например, сдвиг операнда влево на 1 бит, равносилен умножению на 2 ($2^1 = 2$), а вправо на 3 бита — делению на 8 ($2^3 = 8$).

В заключении отметим еще одну интересную команду действия — команду nop, код которой 90h. Эта команда делает то, что ничего не делает, то есть не выполняет никаких действий вообще, кроме того, что ее код занимает 1 байт в исполняемом файле и на ее выборку из памяти и дешифрацию процессором тратится некоторое время. Тем не менее, эту команду используют для некоторых практических целей.

Вопросы для подготовки

1. Приведите примеры команд действий, реализованных на языке ассемблера.

2. В сегменте данных объявлены три ячейки памяти:

```
x db 0
y db 1
z dw 2
```

Имеются следующие случаи использования команды mov:

1	**2**	**3**	**4**	**5**
mov AX, x	mov CX, z	mov x, y	mov DS, CX	mov DS, z
6	**7**	**8**	**9**	**10**
mov DS, SS	mov BH, BL	mov AX, CL	mov CS, CS	mov IP, DI

Какие из них приведут к ошибке компиляции и почему?

3. Для чего предназначена команда `xchg` и каковы ограничения на ее применение?

4. Какая из команд `inc AX` и `add AX, 1` занимает меньше места в исполняемом файле (на сколько байт?) и, соответственно, выполняется быстрее?

5. Какие значения примут флаги `ZF`, `SF`, `CF` и `OF` после выполнения команд:

1	2	3	4
`mov AL, 0FFh` `inc AL`	`mov AX, 0` `add AX, 1`	`mov AH, 0F1h` `add AH, 0A0h`	`mov AX, 0BCh` `add AX, 0DEh`
5	**6**	**7**	**8**
`mov AX, 0` `dec AX`	`mov AH, 0` `sub AX, 1`	`mov AL, 98h` `sub AL, 10h`	`mov AX, 0DDh` `sub AX, 0BBh`

6. Для чего пользуются командой `neg`?

7. В регистре `AX` содержится число `1234h`, в регистре `CX` — значение `1000h`. В какие регистры запишется и чему будет равен результат умножения `mul CH`? `mul CX`?

8. Начальное значение регистров следующее: `AX = 0ABCh`, `BX = 1000h`, `DX = 210h`. Определите содержимое регистра `AX` после выполнения команды `div BH`? `div BX`?

9. В каких случаях выполнение команды `div` вызовет ошибку при делении?

10. Приведите примеры логических команд.

11. Для чего предназначена команда `test`?

12. Что окажется в регистре `CX` после выполнения команды `xor CX, CX`?

13. Что делают команды `shl` и `shr`? Как они меняют флаг `CF`?

14. Как команда `nop` изменяет значения различных флагов?

Выполнение работы

1. Составить программу на языке ассемблера, позволяющую вычислять значение выражения, согласно варианту.

2. Определить диапазон изменения значений выражения. Результат, если его размер не превышает 1 байт, должен быть помещен в регистр `AL`; если не превышает 2 байта — в регистр `AX`; при большей разрядности значение выражения должно сохраняться в паре регистров `DX:AX`, причем в `DX` заносится старшая его часть.

3. Произвести отладку программы.

4. Для трех тестовых наборов входных значений переменных выполнить в пошаговом режиме программу. Записать изменяющиеся значения регистров на каждом шаге.

Варианты заданий:

№	Пример	Диапазоны изменения переменных		
		x	y	z
1	$\left(\frac{x+y}{8}\right)\text{and}\ z^2$	0..65535	0..65535	0..255
2	$(x \text{ xor } y \text{ xor } x)\cdot z$	0..255	0..255	0..255
3	$\frac{0\text{FFFFh}-x-y}{z}+255$	0..32767	0..32767	0..255
4	$(x \text{ or } y)\text{and}(\text{not } x)-z+CF$	0..255	0..255	0..255
5	$(4\cdot x)\text{and}\left(\frac{y}{4}\right)-z$	0..255	0..255	0..65535
6	$\frac{-x\cdot y}{5}\text{or}\ z$	0..255	0..255	0..255
7	$\frac{x+65536\cdot y}{16}$	0..65535	0..65535	—
8	$\frac{\text{not}(-x)+1-y}{z}$	0..65535	0..65535	0..255
9	$\frac{x}{(y \text{ and } 0\text{Dh})\text{shr } 3+CF}$	0..65535	0..255	—
10	$\frac{x \text{ xor } 255}{\text{not } x}-y$	0..255	0..255	—
11	$\frac{x+y\cdot z}{65536}$	80..1920	24..1080	80..1920
12	$\frac{x+y}{4}-z$	-128..127	-128..127	-128..127

Содержание отчета

1. Постановка задачи.
2. Краткие теоретические сведения.

3. Листинг программы из пункта 1 и изменяющиеся значения регистров и данных памяти для каждого из трех наборов входных значений из пункта 3 выполнения работы.

4. Вывод.

Лабораторная работа № 2.
Система команд процессора Intel 8086. Команды ветвления

Цель работы

Изучение команд ветвления процессора i8086.

Содержание работы

— команды безусловной передачи управления процессора i8086;
— команды условной передачи управления процессора i8086.

Теоретические сведения

Обычно в программе есть точки, в которых нужно принять решение о том, какая команда или последовательность команд будет выполняться следующей, то есть, то место программы, куда нужно далее передать управление. Это решение может быть:

— безусловным — в данной точке переход осуществляется в определенное место программы, не зависимо ни от чего;

— условным — в зависимости от некоторых условий переход будет осуществляться в ту или иную точку программы.

К командам безусловной передачи управления относятся:

— команды безусловного перехода;

— вызовы подпрограмм и возврат из подпрограмм;

— вызов программных прерываний и возврат из программных прерываний.

Безусловные переходы

Далее рассмотрим безусловные переходы по команде `jmp`, которая имеет следующий синтаксис:

```
jmp [модификатор] адрес,
```

где адрес перехода может быть задан либо в виде метки, либо в виде значения в каком-либо регистре или ячейке памяти, а модификатор, которого может и не быть, указывает на тип перехода.

Система команд процессора содержит несколько вариантов безусловного перехода (рис. 2.1), каждый из которых имеет свой код и свое применение.

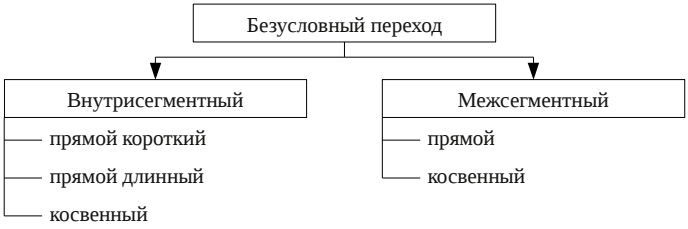


Рис. 2.1. Типы безусловных переходов

В случае *внутрисегментного* перехода адрес следующей выполняемой команды находится в пределах текущего сегмента кода.

Если требуется осуществить *прямой короткий* (ближний) внутрисегментный переход, то адрес перехода должен располагается в пределах –128..+127 байт от команды следующей за jmp. Для указания компилятору того, что переход должен быть именно коротким, записывается модификатор `short`: `jmp short <адрес>`.

В случае *прямого длинного* (дальнего) внутрисегментного перехода адрес перехода может находиться в пределах 64 Кбайт сегмента кода. Для таких переходов модификатор в команде `jmp` указывать не требуется: `jmp <адрес>`. Однако нужно иметь ввиду, что при такой записи компилятор может определить переход как короткий. Интерпретация компилятором перехода, в команде которого не указан модификатор, как ближнего или дальнего зависит от расположения в программе адреса перехода и настроек компилятора, в частности количества проходов. Так при однопроходной схеме переход будет коротким, если команда и адрес перехода располагаются в программе следующим образом:

```
...
адрес ──────────────────────┐
...                         } 0..-128 байт
jmp <адрес>                 }
следующая команда ──────────┘
...
```

т. е. к моменту обработки команды `jmp` компилятор должен уже знать, что адрес, на который эта команда ссылается, находится в пределах короткого перехода.

Рассмотрим пример листинга (табл. 2.1) участка кода программы, в котором используются ближние и дальние переходы.

Таблица 2.1

Использование ближних и дальних внутрисегментных переходов

Листинг программы

```
...
 7      0005             label0:
...
 9      00D3  E9 FF2F              jmp label0       ;дальний переход
10      00D6  33 C0               xor AX, AX
11      00D8  EB 0E               jmp short label1 ;ближний переход
12      00DA  23 C9               and CX, CX
...
14      00E8             label1:
...
```

В строке 8 переход осуществляется в соответствии со значением, загруженным в регистр AX в строке 7. В строке 12 переход осуществляется по адресу, хранимому в ячейке `addr1` в сегменте данных. В отличие от прямых переходов и в ячейке памяти, и в регистре хранится не расстояние перехода от команды следующей за `jmp` до метки, а смещение в сегменте кода к метке (0050r и 003Cr в данном случае).

Организация межсегментных прямых и косвенных безусловных переходов аналогична. Отметим только, что прямые межсегментные переходы занимают 5 байт в памяти в связи с тем, что 1 байт занимает код команды перехода, 2 байта определяют адрес сегмента, к которому требуется перейти, и 2 байта — смещение внутри этого сегмента.

Таким образом, видно, что хотя безусловный переход на языке ассемблера обозначается одной и той же командой `jmp`, но вариантов безусловных переходов в архитектуре команд процессора i8086 несколько, и каждый из них имеет свой код и свое применение. И это следует учитывать при программировании.

Условные переходы

При обработке команды условного перехода, в отличие от безусловного перехода, прыжок по указанному адресу может быть осуществлен, но может быть и нет. Условием, определяющим необходимость перехода, является состояние битов регистра флагов. Так, например, при выполнении команды `jz label1`, переход на метку `label1` будет осуществлен, если флаг нуля `ZF` равен 1, иначе будет выполнен код, следующий за командой `jz label1`. Команды условных переходов, анализирующие состояние одного конкретного флага приведены в табл. 2.2.

Таблица 2.2

Команды условного перехода по значению флага

Название флага	Команда условного перехода	Значение флага для осуществления перехода
Флаг переноса CF	jc	1
	jnc	0
Флаг четности PF	jp	1
	jnp	0
Флаг нуля ZF	jz	1
	jnz	0
Флаг знака SF	js	1
	jns	0
Флаг переполнения OF	jo	1
	jno	0

Заметим, что команды перехода по флагу начинаются с j (jump), затем идет условие (if), которое может начинаться с n (not), а в конце ставится буква, соответствующая флагу. Так команда `jnz` может быть расшифрована как Jump if Not Zero (переход, если не ноль).

Ранее было рассмотрено как некоторые команды действия, например, сложение и вычитание, изменяют значения флагов. Таким образом, можно организовывать структуры, позволяющие выполнять тот или иной участок кода, в зависимости от результата предшествующих операций.

Следующую группу команд условных переходов часто применяют вкупе с командой сравнения `cmp`:

`cmp <операнд1>, <операнд2>`

По сути, эта команда выполняет вычитание `<операнд1>`–`<операнд2>`, устанавливая при этом соответствующие флаги. Ее отличие от команды `sub` только в том, что результат вычитания никуда не записываются и значения операндов не изменяются. С установленными командой `cmp` флагами работают команды условных переходов, приведенные в табл. 2.3, которые позволяют выполнить переход на основании соотношения `<операнда1>` и `<операнда2>`.

Таблица 2.3

Команды условного перехода по результату сравнения

Мнемокод команды условного перехода	Типы операндов	Критерий условного перехода	Значения флагов для осуществления перехода
je	Любые	операнд1 = операнд2	ZF=1
jne	Любые	операнд1 ≠ операнд2	ZF=0
jl (или jnge)	Со знаком	операнд1 < операнд2	SF≠OF
jle (или jng)	Со знаком	операнд1 ≤ операнд2	SF≠OF или ZF=1
jg (или jnle)	Со знаком	операнд1 > операнд2	SF=OF и ZF=0
jge (или jnl)	Со знаком	операнд1 ≥ операнд2	SF=OF
jb (или jnae)	Без знака	операнд1 < операнд2	CF=1
jbe (или jna)	Без знака	операнд1 ≤ операнд2	CF=1 или ZF=1
ja (или jnbe)	Без знака	операнд1 > операнд2	CF=0 и ZF=0
jae (или jnb)	Без знака	операнд1 ≥ операнд2	CF=0

Например, если в регистрах AX и BX находятся целые числа без знака, то код

```
cmp AX, BX
ja label0
```

в случае если значение регистра AX строго больше значения BX, приведет к переходу к метке label0, т. к. вычитание AX–BX при выполнении команды cmp в этом случае произойдет без переноса и результат не будет равен нулю (CF=0 и ZF=0), иначе будет выполнена команда, следующая за `ja label0`.

Мнемокод команд, опять же, состоит из j (jump) и условия (if), в котором присутствуют буквы, соответствующие начальным буквам английских слов (табл. 2.4).

Таблица 2.4

Команды условного перехода по результату сравнения

Буква	Английский	Русский	Типы операндов
E e	equal	равно	Любые
N n	not	не	Любые
G g	greater	больше	Со знаком
L l	less	меньше	Со знаком
A a	above	выше (в смысле «больше»)	Без знака
B b	below	ниже (в смысле «меньше»)	Без знака

Например, `jnae` — Jump if Not Above or Equal (переход если не выше или равно). С логической точки зрения команда `jnae` должна выполняться абсолютно так же, как и команда `jb` — Jump if Below (переход если ниже), что собственно и происходит. Вообще говоря, ассемблер при компиляции командам `jnae` и `jb` сопоставляет один и тот же код.

Следует отметить, что команды условного перехода процессора i8086 позволяют совершать только переходы не более чем на -128..+127 байт относительно команды, следующей за командой перехода. Возможность совершать переходы в пределах сегмента появилась только начиная с процессора 80386. Также нет возможности совершать косвенные условные переходы.

Циклы

В принципе, для организации циклов достаточно знания команд условных переходов. Однако, в связи с тем, что использование циклических структур часто применимо, разработчики внедрили в систему команд процессора специальные команды для их организации.

Рассмотрим пример цикла, которому соответствует блок-схема, изображенная на рис. 2.2.

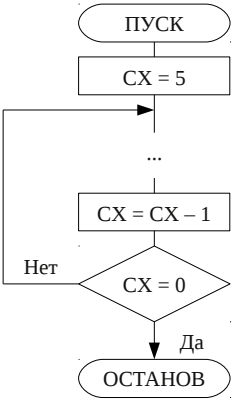


Рис. 2.2. Пример циклической структуры

Данную структуру можно описать следующей последовательностью команд:

```
mov CX, 5
cycle:
...
dec CX
jnz cycle
```

Этот цикл будет выполняться, пока при декременте в регистре `CX` не окажется 0. При этом установится флаг нуля `ZF` и перехода на метку `cycle` по команде `jnz` не будет, а произойдет выполнение следующей за `jnz` команды, т. е. произойдет выход из цикла.

Но правильней в данном случае будет воспользоваться командой `loop` (петля), которая имеет следующий синтаксис:

```
loop <метка>
```

Эта команда уменьшает регистр `CX` на 1 и производит сравнение его с нулем, по результатам которого либо происходит переход на метку, либо происходит выход из цикла и выполняется команда, следующая за loop:

```
mov CX, 5
cycle:
...
loop cycle
```

Отметим, что при выполнении команды `loop` флаг `ZF` не изменяется. Использование команды `loop` предпочтительнее, так как в

отличие от предыдущего способа организации цикла, она выполняется быстрее и занимает меньше места.

Для организации циклов также предназначены команды `loope` (или `loopz`) и `loopne` (или `loopnz`). Они также выполняют операцию декремента регистра `CX`, но отличаются условием выхода из цикла (табл. 2.5).

Таблица 2.5

Условие выхода из цикла для команд организации циклических структур

Мнемокод команды	Условие перехода на метку	Условие выхода из цикла
`loop`	`CX` ≠ 0	`CX` = 0
`loope` (или `loopz`)	`CX` ≠ 0 и `ZF` = 1	`CX` = 0 или `ZF` = 0
`loopne` (или `loopnz`)	`CX` ≠ 0 и `ZF` = 0	`CX` = 0 или `ZF` = 1

Анализ флага нуля командами `loope`/`loopz` и `loopne`/`loopnz` дает возможность организовать досрочный выход из цикла, использую флаг `ZF` в качестве индикатора.

Немаловажным в программировании является организация вложенных циклов:

```
...
mov CX, 2
cycle_out:
...;команды внешнего цикла
push CX
mov CX, 5
cycle_in:
...;команды внутреннего цикла
loop cycle_in
...;команды внешнего цикла
pop CX
loop cycle_out
...
```

Так как для организации внутреннего цикла так же, как и для внешнего, используется значение регистра `CX`, то до выполнения внутреннего цикла это значение должно быть сохранено, например, в стеке, а по окончанию внутреннего цикла должно быть восстановлено.

Недостаток команд работы с циклами такой же, как и у команд условных переходов — метка перехода должна располагаться в

пределах –128..+127 байт от команды, следующей за командой `loop`. Если же метка не располагается в указанных пределах, следует использовать команды условных переходов совместно с командой `jmp`.

Вопросы для подготовки

1. Приведите примеры команд безусловных переходов?
2. Сколько байт занимает команда ближнего внутрисегментного безусловного перехода? дальнего перехода? В каких случаях компилятор ассемблера считает переход дальним, а в каких ближним?
3. Где может находиться адрес перехода при косвенном переходе?
4. Сколько байт занимает команда прямого межсегментного перехода? Что указывается в этих байтах?
5. Приведите примеры команд условных переходов, анализирующих состояние отдельных флагов?
6. Будет ли осуществлен переход на метку `m0` при выполнении следующей последовательности команд:

```
xor BX, BX
dec BX
add BX, 2
jo m0
```

7. Какие флаги и как изменяет команда `cmp`?
8. В чем отличие команд `jl` и `jb` (`jg` и `ja`)?
9. Переходу к какой метке, `m0` или `m1`, приведет выполнение следующей последовательности команд:

```
mov SI, 0FFFFh
mov DI, 0AAAAh
cmp SI, DI
jl m1
m0:
...
m1:
...
```

10. Будет ли совершен переход на метку `m0` при выполнении следующей последовательности команд:

```
mov AX, 5Ah
test AX, 00001000b
je m0
```

11. Какая команда безусловного перехода синонимична и выполняется так же, как и команда `jnae`? `ja`? `jle`?
12. Какие ограничения существуют в применении команд условных переходов для процессора i8086?

13. Какие команды применяются для организации циклов?

14. Приведите пример организации вложенных циклов.

15. Сколько раз выполнится цикл

```
mov CX, 0h
m0:
loop m0
```

16. Какие ограничения существуют в применении команд для организации циклов?

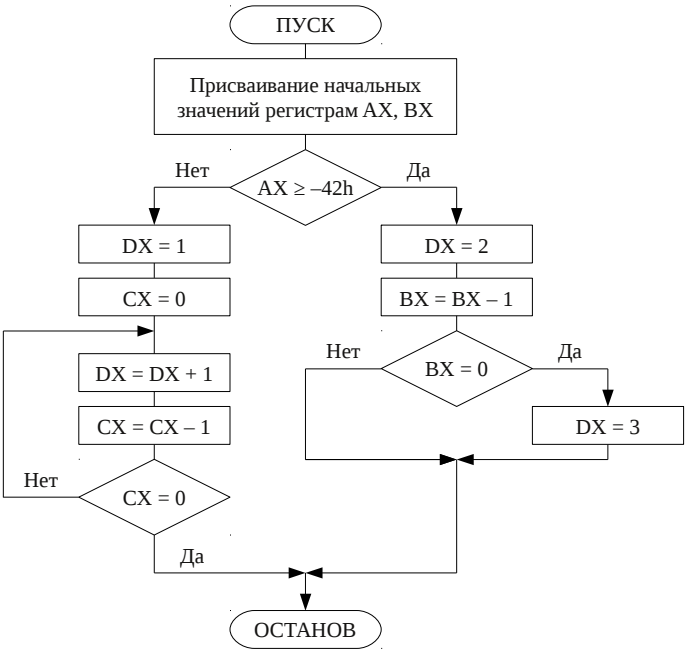
Выполнение работы

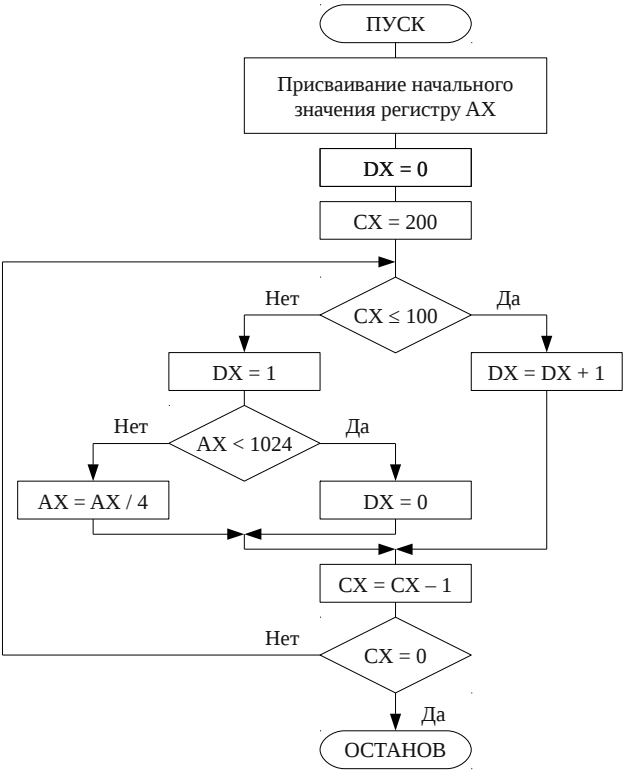
1. Составить программу на языке ассемблера, соответствующую блок-схеме, согласно варианту.

2. Инициализировать командами пересылки (например, командой `mov`) начальные значения используемых регистров перед началом кода, соответствующего блок-схеме.

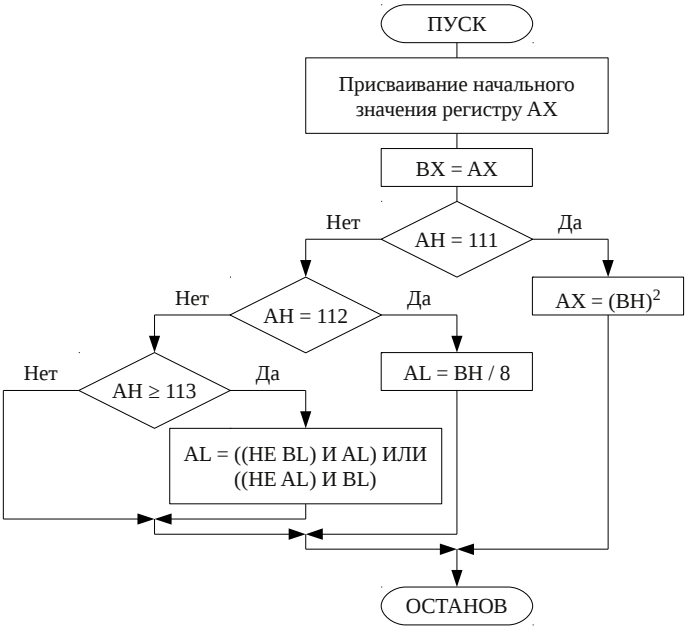
3. Подбирая различные комбинации начальных значений регистров, добиться правильного выполнения всех переходов в программе (или обосновать невозможность перехода). Для проверки правильности использовать отладчик.

Варианты заданий:

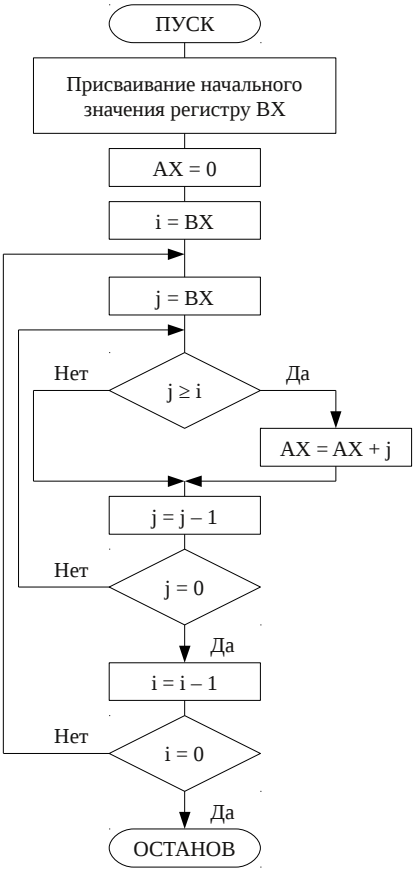
№	Блок-схема
1	

№	Блок-схема
2	

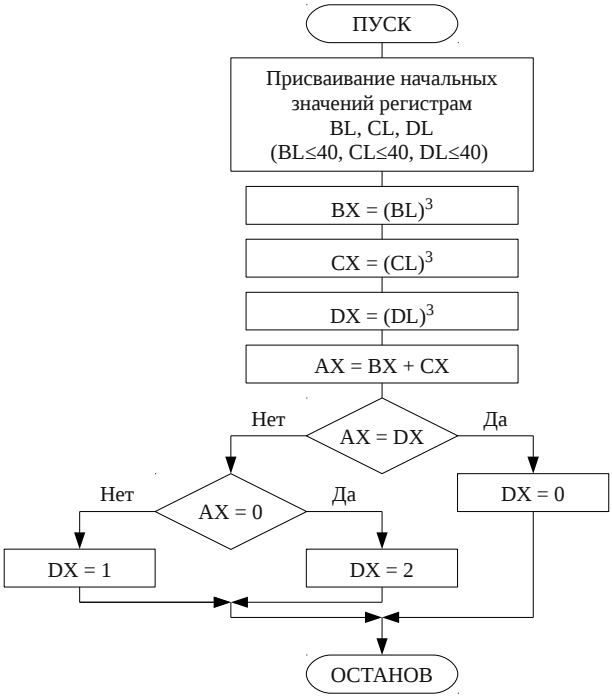
ПУСК
Присваивание начального значения регистру AX
DX = 0
CX = 200
CX ≤ 100
Нет
Да
DX = 1
DX = DX + 1
AX < 1024
Нет
Да
AX = AX / 4
DX = 0
CX = CX – 1
CX = 0
Нет
Да
ОСТАНОВ

№	Блок-схема
3	

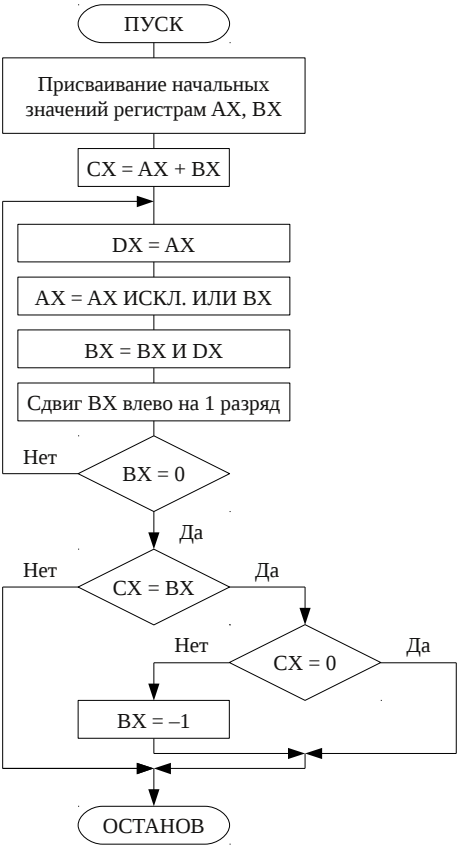
ПУСК
Присваивание начального значения регистру AX
BX = AX
AH = 111
Нет
Да
AX = (BH)2
AH = 112
Нет
Да
AL = BH / 8
AH ≥ 113
Нет
Да
AL = ((НЕ BL) И AL) ИЛИ ((НЕ AL) И BL)
ОСТАНОВ

№	Блок-схема
4	

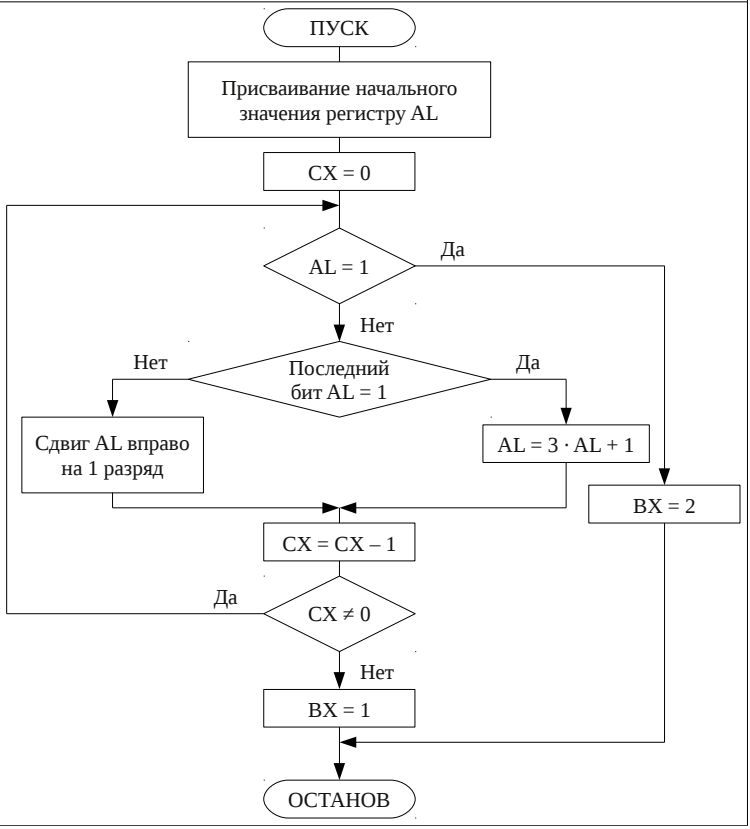
ПУСК
Присваивание начального значения регистру BX
AX = 0
i = BX
j = BX
Нет
j ≥ i
Да
AX = AX + j
j = j – 1
Нет
j = 0
Да
i = i – 1
Нет
i = 0
Да
ОСТАНОВ

№	Блок-схема
5	

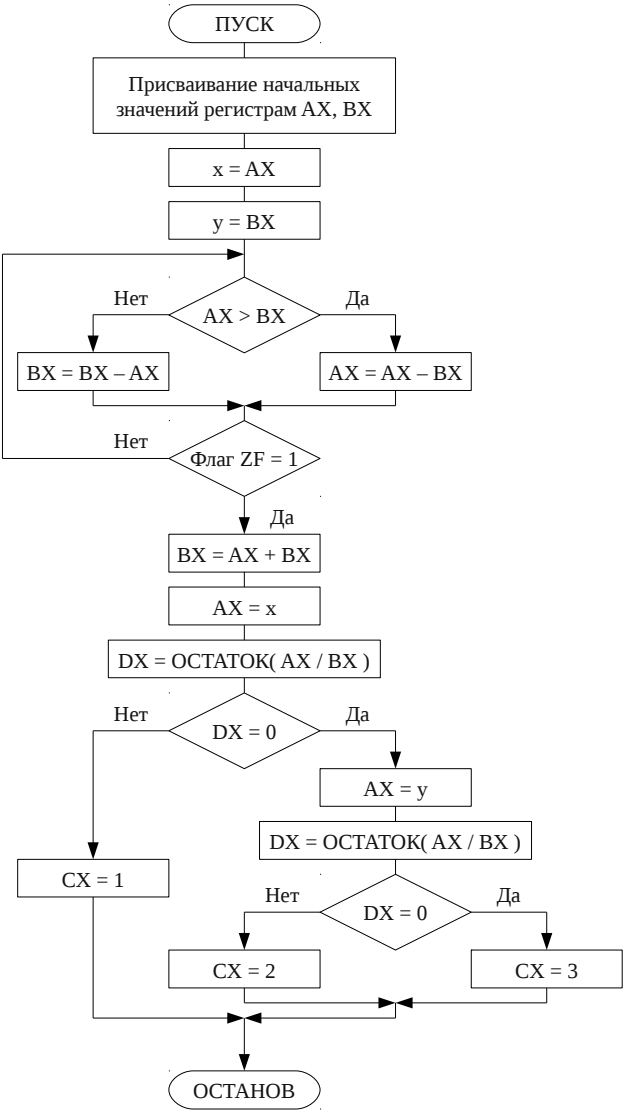
ПУСК
Присваивание начальных значений регистрам BL, CL, DL (BL≤40, CL≤40, DL≤40)
BX = (BL)3
CX = (CL)3
DX = (DL)3
AX = BX + CX
AX = DX
Нет
Да
DX = 0
AX = 0
Нет
Да
DX = 1
DX = 2
ОСТАНОВ

№	Блок-схема
6	

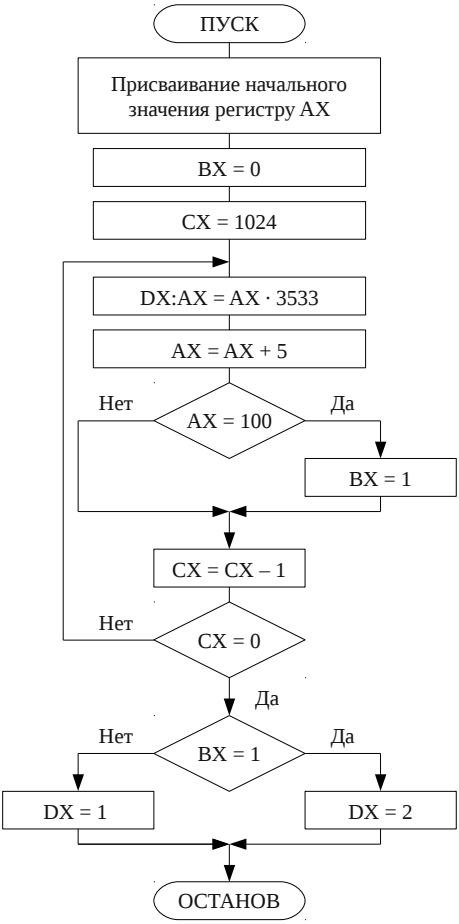
ПУСК
Присваивание начальных значений регистрам AX, BX
CX = AX + BX
DX = AX
AX = AX ИСКЛ. ИЛИ BX
BX = BX И DX
Сдвиг BX влево на 1 разряд
Нет
BX = 0
Да
Нет
CX = BX
Да
Нет
CX = 0
Да
BX = –1
ОСТАНОВ

№	Блок-схема
7	

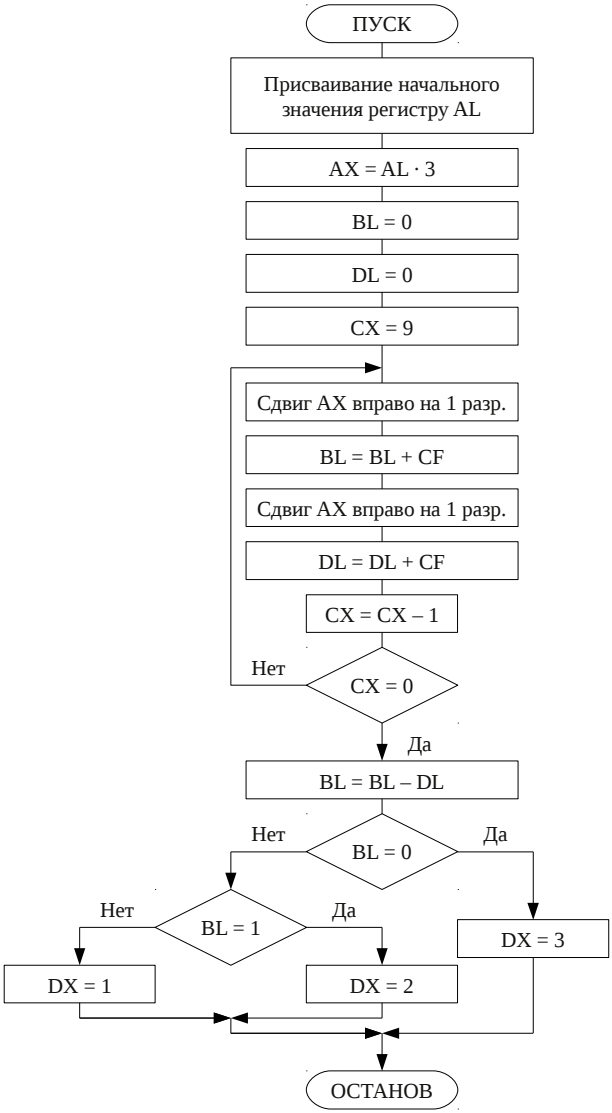
ПУСК
Присваивание начального значения регистру AL
CX = 0
AL = 1
Да
Нет
Последний бит AL = 1
Нет
Да
Сдвиг AL вправо на 1 разряд
AL = 3 · AL + 1
BX = 2
CX = CX – 1
CX ≠ 0
Да
Нет
BX = 1
ОСТАНОВ

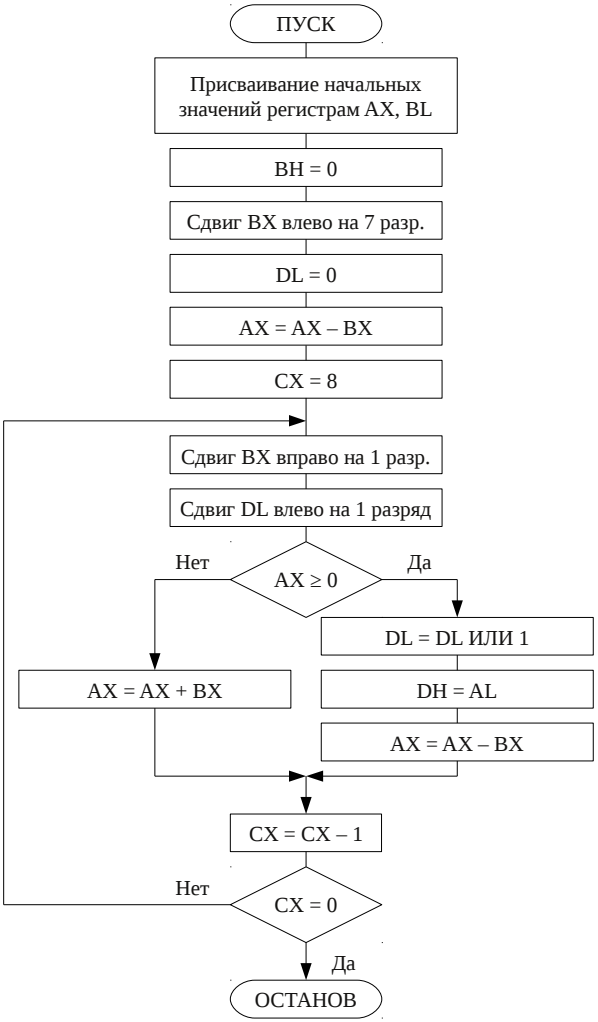
№	Блок-схема
8	

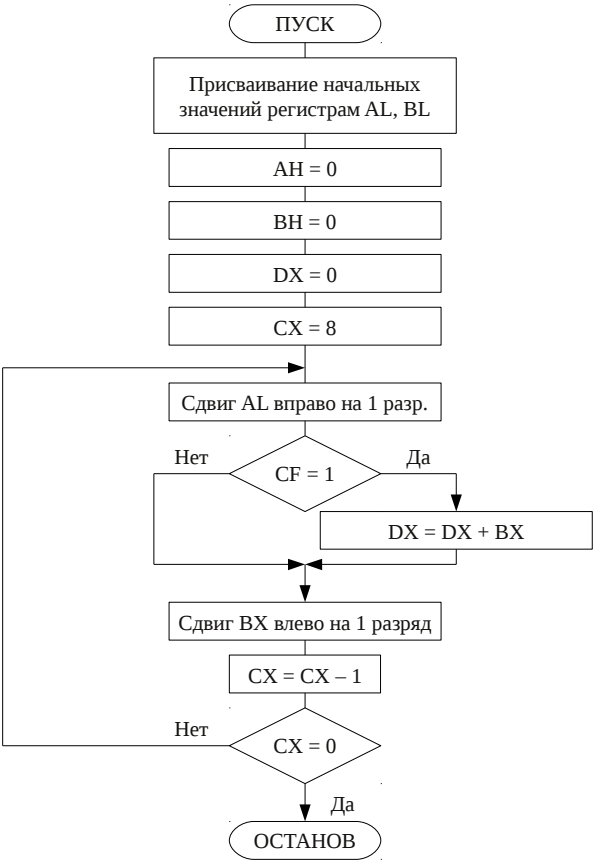
ПУСК
Присваивание начальных значений регистрам AX, BX
x = AX
y = BX
AX > BX
Нет
Да
BX = BX – AX
AX = AX – BX
Флаг ZF = 1
Нет
Да
BX = AX + BX
AX = x
DX = ОСТАТОК(AX / BX)
DX = 0
Нет
Да
AX = y
DX = ОСТАТОК(AX / BX)
CX = 1
DX = 0
Нет
Да
CX = 2
CX = 3
ОСТАНОВ

№	Блок-схема
9	

ПУСК
Присваивание начального значения регистру AX
BX = 0
CX = 1024
DX:AX = AX · 3533
AX = AX + 5
Нет
AX = 100
Да
BX = 1
CX = CX – 1
Нет
CX = 0
Да
Нет
BX = 1
Да
DX = 1
DX = 2
ОСТАНОВ

№	Блок-схема
10	

№	Блок-схема
11	

№	Блок-схема
12	

Содержание отчета

1. Постановка задачи.
2. Краткие теоретические сведения согласно содержанию работы.
3. Листинг программы из пункта 1 и наборы начальных значений регистров, позволяющие пройти по каждой ветке блок-схемы, из пункта 3 выполнения работы.
4. Вывод.

Лабораторная работа № 3.
Ввод и вывод с использованием сервиса DOS

Цель работы

Изучение возможностей ввода и вывода символьной информации с использованием сервиса DOS.

Содержание работы

— функции ввода/вывода DOS;
— принципиальная схема клавиатуры и процесс ввода;
— обычный и расширенный код символов;
— ввод/вывод числовой информации.

Теоретические сведения

Ранее уже была рассмотрена функция DOS с номером 09h, предназначенная для вывода текста, которая позволяет вывести строку, начинающуюся по адресу, указанному в паре регистров `DS:DX`, и заканчивающуюся символом `$`. Имеются и другие функции, которые позволяют вводить и выводить как строки, так и отдельные символы (прил. 1). Для вызова функций нужно занести в регистр `AH` номер требуемой функции, заполнить соответствующие регистры и вызвать сервис DOS командой `int 21h`. Например, для вывода одного символа на экран можно использовать функцию DOS с номером 2 следующим образом (табл. 3.1).

Таблица 3.1

Вывод символа на экран

Текст программы

`...`
`mov AH, 02h ;Заносим в AH номер функции DOS вывода символа`
`mov DL, 'Z' ;Заносим в регистр DL ASCII-код выводимого символа`
`int 21h ;Вызываем функцию DOS`
`...`

Для ввода символов с использованием сервиса DOS существует особенность, проявляющаяся в двукратном вызове функции для считывания функциональных клавиш, имеющих расширенный код. Связано это с преобразованием информацией, посылаемой от клавиатуры.

Если рассмотреть сильно упрощенную принципиальную схему клавиатуры (для простоты представлена клавиатура с небольшим количество клавиш) можно заметить, что все клавиши находятся в узлах матрицы (рис. 3.1).

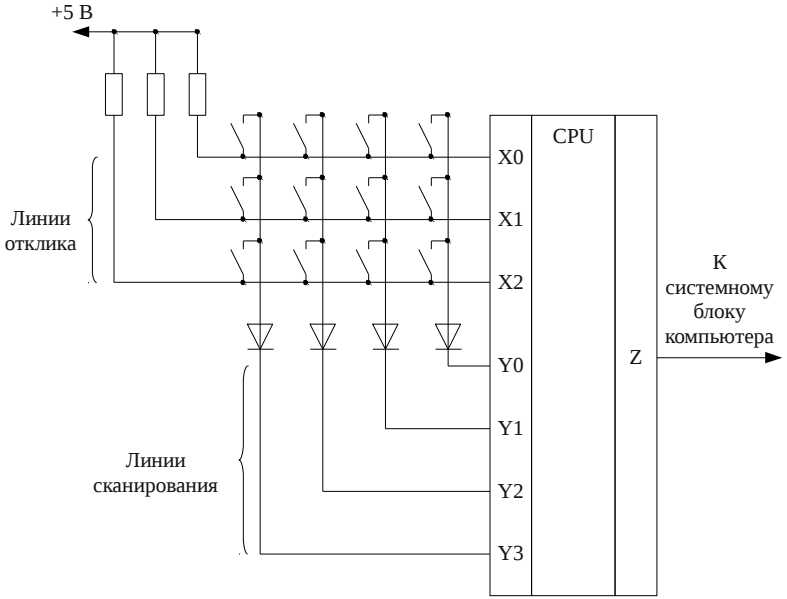


Рис. 3.1 Упрощенная схема клавиатуры

Вертикальные линии матрицы клавиатуры подключены через диоды к выходным линиям Y0...Y3 (линии сканирования) порта Y, который является выходным портом для контроллера в том плане, что контроллер может устанавливать на линиях, связанных с портом, сигналы низкого и высокого напряжения, т. е. логические ноль «0» и единицу «1». Диоды предназначены для предотвращения коротких замыканий при одновременном нажатии нескольких клавиш.

Горизонтальные линии матрицы соединены с линиями X0...X2 (линии отклика) входного порта X, значения каждой линии которого контроллер может считывать, определяя какое установлено на линии напряжение: соответствующее логическому «0» или «1». Кроме этого горизонтальные линии через подтягивающие резисторы соединены с

напряжением питания +5 В, соответствующим логической «1», поэтому, когда ни одна клавиша не нажата и все контакты в узлах матрицы разомкнуты, на входах X0...X2 контроллера установлены «1».

В процессе функционирования контроллер устанавливает поочередно на каждой линии сканирования уровень напряжения логического «0», оставляя на остальных, кроме одной, уровень напряжения логической «1». Таким образом, если нажата клавиша, установка «0» на соответствующей вертикальной линии матрицы, приведет к появления сигнала «0» на соответствующей горизонтальной линии матрицы, что будет обнаружено контроллером, считывающим значения линий порта X. Зная на какой из сканирующих линий установлен в данный момент «0» и на какой линии отклика получен «0», контроллер клавиатуры определяет номер нажатой клавиши в матрицы. Также легко определяется, когда нажатая ранее клавиша отпускается.

Как только контроллер определил нажатие или отпускание клавиши, он посылает в центральный компьютер запрос на прерывание и номер клавиши в матрице, который однозначно зависит от схемы клавиатурной матрицы, но не от обозначений, нанесенных на поверхность клавиш. Этот номер называется скан-кодом (Scan Code). Слово scan «сканирование», подчеркивает тот факт, что клавиатурный компьютер сканирует клавиатуру для поиска нажатой клавиши.

Обычно программе нужен не порядковый номер нажатой клавиши, а код, соответствующий обозначению на этой клавише, то есть код ASCII. Код ASCII не связан напрямую со скан-кодом, так как одной и той же клавише могут соответствовать несколько значений кода ASCII в зависимости от состояния других клавиш. Например, клавиша с обозначением «1» используется для ввода символов '1' и '!' (если она была нажата вместе с клавишей <Shift>). Все преобразования скан-кода в код ASCII выполняются программно.

Преобразование начинается после того, как клавиатурный процессор отошлет центральному процессору запрос на прерывание и скан-код клавиши. При этом центральный процессор прерывает исполняемую программу и переходит на подпрограмму обработки прерывания от клавиатуры. Эта подпрограмма формирует согласно значению скан-кода двухбайтовый код с последующей засылкой его в буфер ввода данных с клавиатуры, заполнение которого происходит по мере нажатия клавиш и никак не связано с выполняемой программой.

Если программе требуется ввести с клавиатуры определенную информацию, она ставит запрос к DOS на ввод с клавиатуры одного

символа или целой строки. Запрошенная функция DOS обращается к буферу ввода и при наличии в нем символов передает первый из поступивших символов в программу. При этом символ изымается из буфера, освобождая там место для последующих символов.

Каждый код представлен в буфере BIOS двумя байтами. Для однобайтных кодов, которые, например, возникают при нажатии алфавитно-цифровых клавиш, первый байт (младший) содержит ASCII-код, а второй байт (старший) — скан-код клавиши, породившей этот ASCII-код. Для расширенных кодов ACSII младший байт содержит 0, а старший байт — ASCII-код.

Таким образом, для проверки на ввод символа 'Q' достаточно одного вызова функции ввода DOS (табл. 3.2).

Таблица 3.2

Проверка введенного алфавитно-цифрового символа

Текст программы

```
...
mov AH, 01h      ;В AH номер функции DOS ввода символа
int 21h          ;Вызов DOS
cmp AL, 'Q'      ;Сравниваем ASCII-код введенного символа
                 ;  в регистре AL с кодом символа 'Q'
jne m1           ;Переход, если код введенного символа в AL
                 ;  не совпадает с ASCII кодом символа 'Q'
...              ;Выполнение действий, если введен символ 'Q'
jmp m2           ;Переход к окончанию обработки клавиши
m1:
...              ;Выполнение действий, если введен символ
                 ;  отличный от 'Q'
m2:              ;Конец проверки
...
```

Для проверки на нажатие клавиши или комбинации клавиш, имеющей расширенный код, например клавиши <F5>, потребуется выполнить два вызова, первый из которых проверит, является ли младший байт кода нулем (табл. 3.3).

Таблица 3.3

Проверка нажатия функциональной клавиши

Текст программы

```
...
mov AH, 01h      ;В AH номер функции DOS ввода символа
int 21h          ;Вызов DOS
cmp AL, 0        ;Сравниваем первый байт с нулем
jne m1           ;Если не 0, то нажата клавиша,
                 ;  не имеющая расширенный код
mov AH, 01h      ;Иначе считываем старший байт расширенного кода
int 21h
cmp AL, 3Fh      ;Сравниваем старший байт с ASCII кодом <F5> – 3Fh
jne m1           ;Переход, если ASCII код не совпал
...              ;Выполнение действий, если нажата клавиша <F5>
jmp m2           ;Переход к окончанию обработки клавиши
m1:
...              ;Выполнение действий, если нажата не <F5>
m2:              ;Конец проверки
```

Заметим, что, хотя вызова функции DOS с номером 01h здесь два, но проверяется на совпадение с клавишей <F5> только одна нажатая клавиша.

Нужно сказать, что кроме средств ввода-вывода символьной информации, операционная система DOS другими возможностями, в том числе по вводу-выводу числовых данных, не обладает. Введенное и считанное одной из функций DOS в виде строки символов число для дальнейших вычислений требует программного преобразования в собственно числовую форму. А полученный в ходе вычислений результат для вывода на экран предварительно должен быть преобразован в последовательность ASCII кодов символов, соответствующих цифрам числа результата.

Рассмотрим метод ввода чисел на примере числа 365 в десятичной системе счисления. Это число может быть считано функциями DOS как последовательность ASCII кодов символов '3', '6' и '5'. Вычитая из кода считанной цифры код нуля, в силу последовательного расположения кодов цифр в кодовой таблице, получаем саму цифру, т. е. $'3'-'0'=33-30=3$, $'6'-'0'=36-30=6$, $'5'-'0'=35-30=5$. Далее, умножая каждую цифру на вес позиции, в которой она находится, и складывая полученные результаты, получим введенное данное в числовом виде: $3\cdot10^2+6\cdot10^1+5\cdot10^0=3\cdot100+6\cdot10+5\cdot1=365$.

В общем виде строка из m символов $s_m s_{m-1} s_1 s_0$, где s_i — символ в позиции i ($i=0,1,2...m$), представляющая собой число x, записанное в

некоторой системе счисления, преобразовывается в числовую форму следующим образом:

1) из строки выделяется каждый символ s_i;

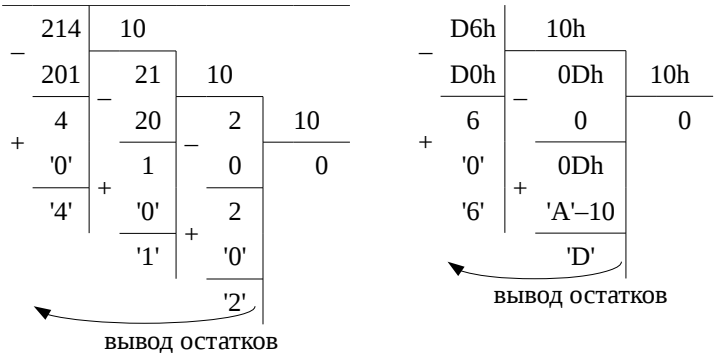
2) выделенный символ преобразуется в число $s_i \rightarrow n_i$;

3) каждое значение n_i умножается на вес i-й позиции $\omega_i = B^i$, где B — основание системы счисления;

4) введенное число представляет собой сумму произведений $x = \sum_{i=0}^{m} \omega_i n_i = \sum_{i=0}^{m} B^i n_i$.

Заметим, что в шестнадцатеричной системе счисления символом s_i могут быть и цифра и буква от 'A' до 'F' или от 'a' до 'f', что нужно учитывать при преобразовании по пункту 2. Умножение по пункту 3 для систем счисления с основание кратным степени двойки удобно осуществлять сдвигом. Суммирование произведений по пункт 4 легко реализовать с использованием схемы Горнера: $x = \left(\left(\ldots \left(\left(0 + n_m \right) \cdot B + n_{m-1} \right) \cdot B + \ldots \right) \cdot B + n_1 \right) \cdot B + n_0$, т. е. в рассмотренном примере с числом 365 последнее действие представимо в виде $\left(\left(0 + 3 \right) \cdot 10 + 6 \right) \cdot 10 + 5 = 365$.

Преобразование числа в строку символов при выводе выполняется в обратном порядке: сначала нужно получить значение позиции n_i, а затем преобразовать эти значения в символы $n_i \rightarrow s_i$. Приведем пример представления одного и того же данного, но в различных системах счисления.



Процесс деления заканчивается, когда частное от деления оказывается равным нулю, после чего получившиеся остатки,

преобразованные в символьный вид, выводятся в порядке обратном их получению (этого просто реализовать, занося остаток каждого деления в стек, из которого затем выбирать значения для вывода). Разница в приведенных примерах не в форме записи чисел (214 или D6h), а в самом преобразовании. Это касается деления на 10 (0Ah) в первом случае и на 16 (10h) — во втором, а также преобразования букв в выводимые символы для шестнадцатеричной системы счисления.

В заключении скажем, что для ввода чисел со знаком достаточно проверить, введен ли перед числом символ '–', и, если да, то после преобразования беззнаковой части выполнить команду `neg`. Аналогично при выводе — если первый бит установлен в единицу, то сначала выполнить команду `neg` над выводимым число, при этом перед собственно выводом числа вывести символ '–'.

Вопросы для подготовки

1. Как можно вывести отдельный символ на экран?
2. Каким образом контроллер клавиатуры определяет нажатую клавишу?
3. Зачем нужны диоды в упрощенной схеме клавиатуры?
4. Что первично: скан-код или ASCII-код?
5. В чем разница между расширенным и не расширенным кодом?
6. Как программно получить код нажатой клавиши?
7. Как ввести или вывести числовое данное целого типа, если в распоряжении имеются только функции для ввода символов?
8. Что выведется на экран при выполнении следующего фрагмента программы, если пользователь может набирать на клавиатуре только большие и маленькие символы латинского алфавита?

```
mov CX, 10
m0:
mov AH, 1
int 21h
and AL, 00010000b
mov AH, 2
mov DL, AL
int 21h
loop m0
```

9. Число 10 записано в 13-ричной системе счисления. Чему равно число в 6-ричной системе счисления?

Выполнение работы

Написать программу, по средствам которой введенное с клавиатуры целое число без знака в указанной системе счисления будет преобразовано и выведено на монитор в другой системе счисления согласно варианту. При этом максимальное количество знаков вводимого числа для простоты реализации ограничено.

№	Основание системы вводимого числа	Максимальное количество символов вводимого числа	Основание системы выводимого числа
1	14	4	3
2	5	6	11
3	12	4	6
4	9	5	14
5	15	4	7
6	6	6	1
7	13	4	9
8	11	4	13
9	3	10	5
10	1	32	4
11	4	8	12
12	7	5	15
13	35	3	2

Содержание отчета

1. Постановка задачи выполнения работы.
2. Краткие теоретические сведения согласно содержанию работы.
3. Листинг программы выполнения работы.
4. Вывод.

Лабораторная работа № 4.
Организация одномерных и многомерных массивов

Цель работы

Изучение организации одномерных и многомерных массивов и работа с ними.

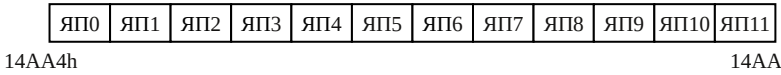
Содержание работы

— элементы массива;
— описание массива;
— получение адреса элемента в массиве;
— режимы адресации: базовая, индексная.

Теоретические сведения

С физической точки зрения массив на низком уровне представляется как последовательность ячеек памяти. То, как трактуется это набор ячеек, определяет программист: одномерный ли это массив или двухмерная таблица, являются ли элементы однобайтовыми или многобайтовыми данными. Следующим важным вопросом является доступ к конкретному элементу. На языках высокого уровня элемент массива определяется его индексом, на низком же уровне процессор оперирует только адресами и, нужно каким-то образом указывать адрес ячейки или ячеек памяти, соответствующих элементу.

Допустим имеется последовательность из двенадцати однобайтовых ячеек памяти (ЯП), начиная с адреса 14AA4h.



Эти ячейки могут рассматриваться, например, как одномерный массив двухбайтовых элементов Э*i*.

Заметим, что индексы элементов начинаются с нуля, что удобно при вычислениях, при этом нулевой элемент имеет нулевое смещение относительно адреса начала массива, что логично. Обратиться к элементу с индексом *i* можно рассчитав адрес соответствующей ячейки памяти как адрес начала массива плюс смещение в нем с учетом размера элементов:

$$\text{base} + i \cdot \text{size},$$

где base — адрес начала массива в памяти; size — размер элементов массива. Так, для элемента Э3 получаем: 14AA4h+3·2=14AAAh, что соответствует адресу ЯП6.

При нумерации элементов, начинающейся с единицы, перед вычислением по формуле нужно предварительно произвести декремент индекса *i*.

Эту же последовательность ячеек можно трактовать как двумерную таблицу (3×4) однобайтовых элементов.

ЯП0	ЯП1	ЯП2	ЯП3	ЯП4	ЯП5	ЯП6	ЯП7	ЯП8	ЯП9	ЯП10	ЯП11

14AA4h 14AA8h 14AACh 14AAFh

Э00	Э01	Э02	Э03
Э10	Э11	Э12	Э13
Э20	Э21	Э22	Э23

Элементы таблицы располагаются в ячейках памяти следующим образом: сначала располагаются элементы первой строки, затем следуют элементы второй строки, и, наконец, третьей. Другой вариант, который обычно не применяют, это располагать сначала элементы первого столбца, затем второго и т. д.

Для индексации элементов двухмерных массивов применяется пара значений (i, j), где i — номер строки, в которой находится элемент, j — номер столбца. Строки и столбцы, опять же для удобства, нумеруются с нуля. Смещение к ячейке памяти, соответствующей элементу Э*ij* в массиве размерностью $(m \times n)$, где m — количество строк, n — количество столбцов, находится как:

$$\text{base} + (i \cdot n + j) \cdot \text{size}.$$

В рассматриваемом примере количество столбцов равно 4 и для элемента Э21, который находится в ячейке памяти ЯП9 по адресу 14AADh, получаем $14AA4h + (2 \cdot 4 + 1) \cdot 1 = 14AADh$.

В соответствии с пониманием массива на низком уровне программирования, описать массив в тексте программы на языке ассемблера можно как последовательность значений ячеек памяти:

```
array0 db 5, –10, 225, 0, 1
array1 dw 64, 2, 11
```

Т. е. массив описывается так же как и любые данные. При этом можно использовать директиву `dup`:

```
array2 dw 16 dup (0)
```

Размер массива совпадает с размером места отводимого под описанные данные: для `array0` — 5 элементов, всего 5 байт; для `array1` — 3 элемента, всего 6 байт; для `array2` — 16 элементов, всего 32 байта, первоначальное значение которых равно 0. Описанное сообщение

```
message db 'Hello!$'
```

также можно трактовать как массив ASCII кодов символов 'H', 'e' и т. д.

При таком объявлении массива нельзя говорить о его размерности, является ли он одномерным или многомерным. С другой стороны, есть некоторая информация о размерности элементов, хотя, как будет показано ниже, к последовательности двухбайтовых ячеек можно обращаться побайтово, и наоборот, последовательность однобайтовых элементов можно адресовать по словам. Важно понимать, что директивами `db` и `dw` только лишь отводится место в программе под массив и инициализируются начальные значения ячеек.

Следующий вопрос касается доступа к элементам массива. Пусть имеется следующая последовательность данных, расположенная в сегменте, на который ссылается регистр `DS`, которая трактуется как одномерный массив двухбайтовых элементов:

```
array dw 0Ah, 1123h, 0F580h, 85h, 3h, 14D6h
```

Нумерация элементов начинается с нуля, элементы объявлены как двухбайтовые.

Рассмотрим как можно обращаться к элементам массива, т. е. режимы адресации. При *базовой адресации* для описания адреса используется только один регистр (табл. 4.1).

Таблица 4.1

Считывание значения из массива при базовой адресации

Текст программы

…
mov SI, offset array ;В SI заносим адрес начала массива
add SI, 3 ; и прибавляем смещение к требуемому элементу
;После этого, в паре DS:SI будет находится полный адрес элемента
mov AL, [SI] ;Пересылка байта с адресом DS:SI в AL
...

В приведенном отрывке программы для описания адреса элемента массива используется только один регистр SI, который будет содержать смещение в сегменте данных. Этот регистр является базовым. Адрес начала сегмента данных находится в регистре, который был объявлен в директиве `assume`. Далее будем считать, что массив `array` находится в сегменте с именем `data`, и было определено:

```
assume DS: data
```

Таким образом, последней командой в регистр `AL` занесется 3-й, начиная с нуля, элемент массива `array`, то есть значение 85h. Заметим, что хоть изначально элементы массива были объявлены как двухбайтовые, в регистр `AL` занеслось однобайтовое значение, о чем говорилось немного ранее.

При *индексной адресации* адрес элемента получается исходя из идентификатора и регистра (табл. 4.2).

Таблица 4.2

Считывание значения из массива при базово-индексной адресации

Текст программы

...
mov SI, 2 ;Загружаем в SI номер элемента массива
mov AL, array[SI] ;Пересылаем байт с адресом в DS:(array + SI)
; в регистр AL
...

Смещение в сегменте данных к элементу массива в данном случае получается как сумма адреса идентификатора и индексного регистра. Регистр `DS`, как и при предыдущем способе адресации элементов, не указывается явно.

Для старших моделей процессоров существуют и другие режимы, например, *базово-индексная адресации*, которую можно эффективно применять при работе с матрицами, но они оставляются на самостоятельное изучение, т. к. в общем случае приведенных

рассуждений достаточно для организации массивов любой размерности.

Вопросы для подготовки

1. В сегменте данных объявлен массив mass:

```
mass db 1, 2, 'A', 4, 5, 6
```

Каким образом можно изменить 3-й элемент этого массива (символ 'A') на число 3?

2. В сегменте данных описан массив данных `mass` и ячейка памяти `x`:

```
mas db 0, 1, 2, 3, ...
x db 0
```

Имеются следующие случаи работы с массивом:

1	**2**	**3**	**4**
`mov AL, mas(SI)`	`mov mas[DI], BH`	`mov CX, mas(DI)`	`mov CL, mas[AX]`
5	**6**	**7**	**8**
`mov CL, mas[BP]`	`mov AL, mas[SP]`	`mov mas(SI), 10h`	`mov mas(SI), x`

Какие из этих случаев приведут к ошибке компиляции?

3. Каким образом осуществляется работа с многомерными массивами?

4. Допустим, в сегменте данных описан массив, который трактуется как двухмерная матрица (5×7). Требуется присвоить значение 0 элементу матрицы, номер строки которого находится в регистре SI, номер столбца — в регистре DI. Как это будет реализовано программно?

Выполнение работы

Требуется написать программу на языке ассемблера согласно варианту задания. Обеспечить:

— ввод и вывод данных в той системе счисления, в которой приводятся диапазоны их изменения в задании (учесть при необходимости знак);

— фильтрацию недопустимых символов при вводе;

— формирование массива чисел.

№	Задание
1	Массив чисел из диапазона –0FFFh..0FFFh размерностью 0..2·N вводится с клавиатуры, где N – число из диапазона 1..9, вводимое с клавиатуры. Вывести на экран N/2 наибольших элементов массива умноженные на 2.
2	С клавиатуры вводятся два массива чисел из диапазона –0FFFh..0FFFh, признак окончания ввода каждого массива — нажатие клавиши <F1> (при этом массивы должны содержать не менее 1-го и не более 255-ти элементов). Вставить второй массив после первого наибольшего элемента первого массива, а получившийся массив вывести на экран.
3	Массив чисел из диапазона –99..99 вводится с клавиатуры, признак окончания ввода массива — нажатие клавиши <Enter> (при этом массив должен содержать не менее 1-го, но не более 255-ти элементов). Вывести на экран наименьшее значение элемента в массиве, общее количество и индексы элементов, имеющих это значение.
4	Массив чисел из диапазона –99..99 вводится с клавиатуры, признак окончания ввода массива — нажатие клавиши <End> (при этом массив должен содержать не менее 1-го, но не более 255-ти элементов). Удалить из массива три элемента, имеющих наименьшее значение. Вывести на экран значения удаленных элементов и полученный после удаления массив.
5	Массив чисел из диапазона –0FFFh..0FFFh, вводится с клавиатуры, признак окончания ввода массива — нажатие клавиши <Esc> (при этом массив должен содержать не менее 1-го, но не более 255-ти элементов). Поменять в массиве местами любой элемент, имеющий наименьшее значение, и любой элемент, имеющий наибольшее значение. Получившийся массив вывести на экран.
6	Массив чисел из диапазона –0FFFh..0FFFh, вводится с клавиатуры, признак окончания ввода массива — ввод символа <Q> или <q> (при этом массив должен содержать не менее 1-го, но не более 255-ти элементов). Элементы массива, которые больше среднего арифметического по всем элементам, уменьшить на значение среднего арифметического и вывести на экран.
7	Квадратная матрица чисел из диапазона 0..9999 размерностью [1..N; 1..N] вводится с клавиатуры по строкам, где N — число из диапазона 1..9, вводимое с клавиатуры . Вывести на экран результат сложения введенной матрицы с транспонированной введенной матрицей.
8	Квадратная матрица чисел из диапазона 0..9999 размерностью [1..5; 1..5] вводится с клавиатуры по строкам. Произвести сглаживание матрицы, заменив каждый элемент (кроме элементов первой и последней строки, а также первого и последнего столбца) целой частью от среднего арифметического 8-ми соседних элементов, умноженных на 1, и текущего элемента, умноженного на 8. Вывести сглаженную матрицу на экран.
9	Матрица чисел из диапазона 0h..0FFh размерностью [1..3; 1..4] вводится с клавиатуры по строкам. Вывести на экран номер столбца с максимальной суммой элементов.

№	Задание
10	Матрица чисел из диапазона 0..9999 размерностью [1..4; 1..3] вводится с клавиатуры по строкам. Найти сумму элементов столбца, содержащего максимальный элемент.
11	Квадратная матрица чисел из диапазона 0h..0FFh размерностью [1..4; 1..4] вводится с клавиатуры по строкам. Определить, является ли данная матрица треугольной, и если является, то вывести ее определитель на экран.
12	Две матрицы чисел из диапазона 0..99 размерностью [1..4; 1..4] вводятся с клавиатуры по строкам. Вывести на экран матрицу, являющуюся произведением введенных.

Содержание отчета

1. Постановка задачи выполнения работы.
2. Краткие теоретические сведения согласно содержанию работы.
3. Листинг программы выполнения работы.
4. Вывод.

Приложение
Функции DOS для ввода/вывода символьных данных

Функция 01h
Вход: AH = 01h
Выход: AL = код символа

Описание: Ожидает поступления на стандартный ввод символа и выводит этот символ на стандартный вывод. При нажатии <Ctrl> + <Break> вызывает прерывание, завершающее программу и передающее управление DOS.

Функция 02h
Вход: AH = 02h
DL = код символа
Выход: –

Описание: Выводит символ, код которого указан в DL, на стандартный вывод. При выводе кода <Backspace> (ASCII код 08h) перемещает курсор в начало строки. Также отслеживает нажатие <Ctrl> + <Break>.

Функция 06h
Вход: AH = 06h
DL = код символа (0..0FEh)
Выход: –
Вход: AH = 06h
DL = 0FFh
Выход: ZF = 1, если есть символ на стандартном входе и ZF = 0, если нет
AL = код символа, если ZF=1

Описание: Если DL = 0FFh, то выполняется ввод символа из буфера клавиатуры «без ожидания». Если символ в буфере есть, то его код записывается в AL и флаг ZF устанавливается в 1. Если DL ≠ 0FFh, то код символа, помещенный в DL, посылается на стандартный вывод. Нажатие <Ctrl> + <Break> не отслеживается.

Функция 07h
Вход: AH = 07h
Выход: AL = код символа

Описание: Ожидает поступления на стандартный ввод символа и выводит этот символ на стандартный вывод. Не проверяет на нажатие <Ctrl> + <Break>, <Backspace> и т.д.

Функция 08h
Вход: AH = 08h
Выход: AL = код символа
Описание: Ожидает поступления на стандартный ввод символа. Проверяет на нажатие <Ctrl> + <Break>. Символ вводится без эха, то есть без вывода введенного символа на стандартный вывод.

Функция 09h
Вход: AH = 09h
Выход: DS:DX = адрес начала строки
AL = 24h
Описание: Посылает на стандартный вывод строку символов, исключая завершающий символ ‘$’.

Функция 0Ah
Вход: AH = 0Ah
DS:DX = адрес начала буфера ввода
Выход: –
Описание: До вызова функции буфер ввода по адресу DS:DX должен быть оформлен следующим образом:
max

…

max – максимальная допустимая длина ввода (от 1 до 254).
По окончанию ввода буфер будет заполнен следующим образом:
max
len

…
0Dh
len – количество введенных символов без учета завершающего символа перевода строки 0Dh.

Символы считываются со стандартного ввода, пока на ввод не поступит символ 0Dh (при нажатии <Enter>), или пока не будет

введено max-1 символов. Во втором случае включается консольный звонок при попытке ввести еще один символ. В процессе ввода действительны клавиши редактирования: <Esc> выдает ‘\’ и ввод начинается заново; <F5> – выдает ‘@’ и следующие символы вводятся в начало буфера (после байта len); и т.д.

Распознается нажатие <Ctrl>+<Break>.

Функция 0Bh

Вход: AH = 0Bh

Выход: AL = 0FFh – если на стандартном входе есть символ для ввода,

AL = 0h – если символа на стандартном входе нет

Описание: Используется перед функциями ввода 01h, 07h и 08h, чтобы избежать нажатия клавиш. Функция дает возможность прекращать выполнение длинных вычислений или другой обработки, обычно не требующей ввода, по нажатию клавиш <Ctrl>+<Break>.

Функция 0Ch

Вход: AH = 0Ch

AL = номер функции ввода (01h, 06h, 07h, 08h или 0Ah) остальные регистры настраиваются в зависимости от номера функции в AL

Выход: –

Описание: Очищает буфер ввода стандартного входа, а затем вызывает функцию ввода, указанную в AL.