Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

Е. А. РУЖИЦКАЯ, Е. Ю. КУЗЬМЕНКОВА

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ASSEMBLER

BCD-числа, цепочечные команды

Практическое пособие

для студентов специальностей
1–40 01 01 «Программное обеспечение
информационных технологий»,
1–40 04 01 «Информатика и технологии программирования»

Гомель
ГГУ им. Ф. Скорины
2017

УДК 004.431.4(076)
ББК 32.973.21я73
Р837

Рецензенты:
кандидат физико-математических наук Т. В. Тихоненко,
кандидат технических наук С. Ф. Маслович

Рекомендовано к изданию научно-методическим советом
учреждения образования «Гомельский государственный
университет имени Франциска Скорины»

Ружицкая, Е. А.

Р837 Программирование на языке Assembler : BCD-числа, цепочечные команды : практическое пособие / Е. А. Ружицкая, Е. Ю. Кузьменкова ; М-во образования Республики Беларусь, Гомельский гос. ун-т им. Ф. Скорины. – Гомель: ГГУ им. Ф. Скорины, 2017. – 47 с.

ISBN 978-985-577-258-4

Практическое пособие предназначено для оказания помощи студентам в обладении машинно-ориентированным языком программирования Assembler. Излагается теоретический материал, даны практические задания и примеры их выполнения по операциям с BCD-числами и цепочечным командам.

Адресовано студентам 1 курса специальностей 1–40 01 01 «Программное обеспечение информационных технологий», 1–40 04 01 «Информатика и технологии программирования».

УДК 004.431.4(076)
ББК 32.973.21я73

ISBN 978-985-577-258-4

Оглавление

Предисловие

Микропроцессор имеет достаточно мощные средства для реализации вычислительных операций. Для этого у него есть блок целочисленных операций и блок операций с плавающей точкой. Команды целочисленных операций работают с данными двух типов: двоичными и двоично-десятичными числами (BCD-числами).

Двоичные данные имеют довольно большой, но ограниченный диапазон значений. Для коммерческих приложений этот диапазон слишком мал, поэтому в архитектуру микропроцессора введены средства для работы с так называемыми двоично-десятичными (BCD) данными. Двоично-десятичные данные представляются в двух форматах: упакованном и неупакованном. Наиболее универсальным является неупакованный формат.

Система команд микропроцессора имеет очень интересную группу команд, позволяющих производить действия над блоками элементов до 64 Кбайт или 4 Гбайт, в зависимости от установленной разрядности адреса use16 или use32. Эти блоки логически могут представлять собой последовательности элементов с любыми значениями, хранящимися в памяти в виде двоичных кодов. Единственное ограничение состоит в том, что размеры элементов этих блоков памяти имеют фиксированный размер 8, 16 или 32 бита. Команды обработки строк предоставляют возможность выполнения семи операций-примитивов, обрабатывающих цепочки поэлементно.

Издание предназначено для оказания помощи студентам в обладении машинно-ориентированным языком программирования Assembler. Излагается теоретический материал, даны практические задания и примеры их выполнения по операциям с BCD-числами и командам обработки цепочек.

Язык программирования Assembler изучается студентами 1 курса специальностей 1–40 01 01 «Программное обеспечение информационных технологий» в рамках дисциплины «Языки программирования» и 1–40 04 01 «Информатика и технологии программирования» в курсе «Программирование».

Тема 1. BCD-числа

Микропроцессор может выполнять целочисленные операции и операции с плавающей точкой. Для этого в его архитектуре есть два отдельных блока: устройство для выполнения целочисленных операций; устройство для выполнения операций с плавающей точкой.

Каждое из этих устройств имеет свою систему команд. В принципе, целочисленное устройство может взять на себя многие функции устройства с плавающей точкой, но это потребует больших вычислительных затрат.

Целочисленное вычислительное устройство поддерживает чуть больше десятка арифметических команд. На рисунке 1.1 приведена классификация команд этой группы. Группа арифметических целочисленных команд работает с двумя типами чисел: целыми двоичными числами (числа могут иметь знаковый разряд или не иметь такового, то есть быть числами со знаком или без знака); целыми десятичными числами.

1.1 Целые двоичные числа

Целое двоичное число с фиксированной точкой – это число, закодированное в двоичной системе счисления. Размерность целого двоичного числа может составлять 8, 16 или 32 бита. Знак двоичного числа определяется тем, как интерпретируется старший бит в представлении числа. Это 7-й, 15-й или 31-й биты для чисел соответствующей размерности. Среди арифметических команд есть всего две команды, которые действительно учитывают этот старший разряд как знаковый, – это команды целочисленного умножения и деления `imul` и `idiv`. В остальных случаях ответственность за действия со знаковыми числами и, соответственно, со знаковым разрядом ложится на программиста. Диапазон значений двоичного числа зависит от его размера и трактовки старшего бита либо как старшего значащего бита числа, либо как бита знака числа.

Числа с фиксированной точкой в программе описываются с использованием директив `db`, `dw` и `dd`. Диапазон значений двоичных чисел представлен в таблице 1.1.

Таблица 1.1 – Диапазон значений двоичных чисел

Размерность поля	Целое число без знака	Целое число со знаком
Байт	0...255	–128...+ 127
Слово	0...65 535	–32 768...+32 767
Двойное слово	0...4 294 967 295	–2 147 483 648... +2 147 483 647

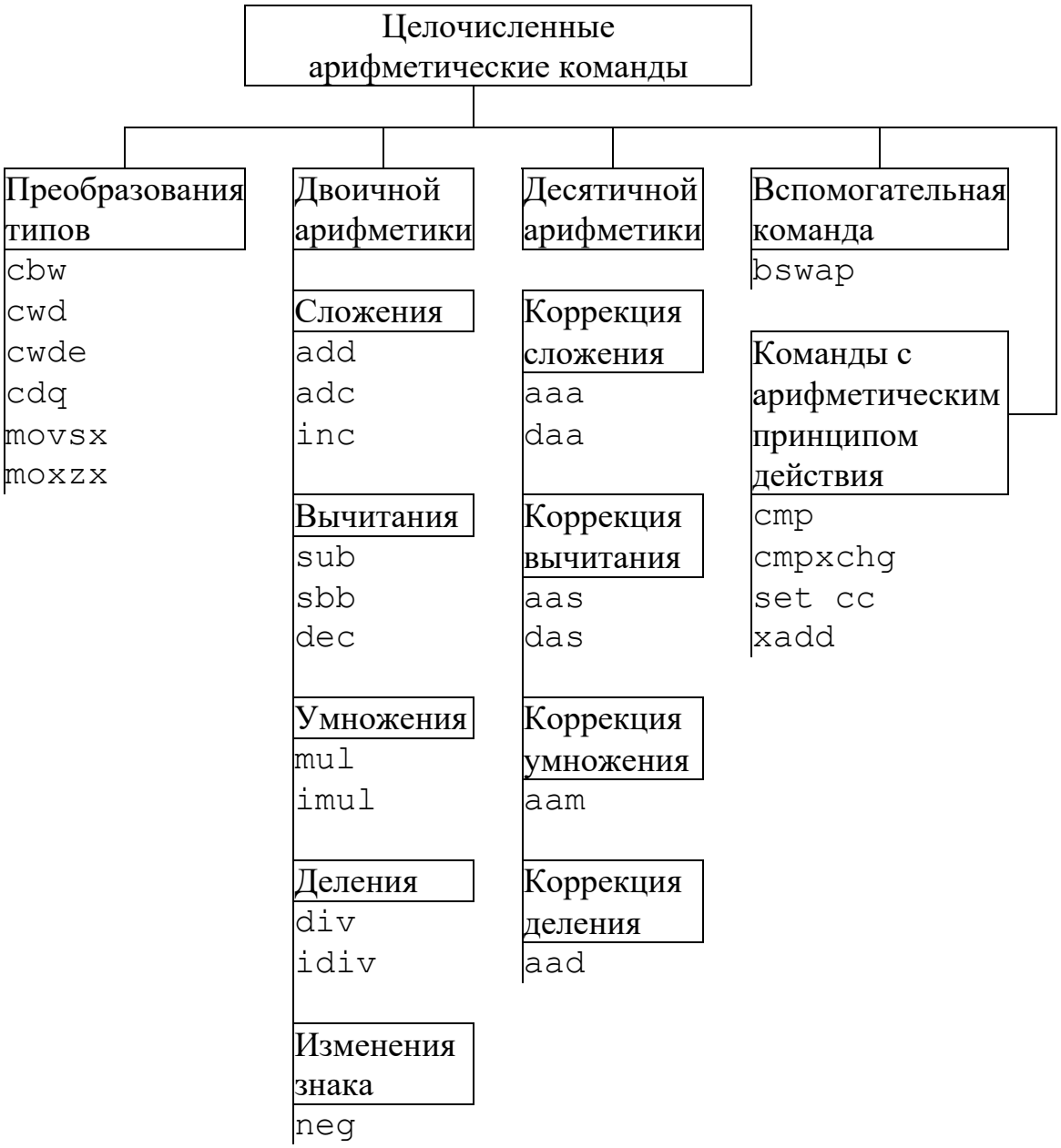

Рисунок 1.1 – Классификация арифметических команд

1.2 Десятичные числа

Десятичные числа – специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырех бит. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом двоично-десятичном коде (BCD – Binary Coded Decimal). Микропроцессор хранит BCD-числа в двух форматах (рисунок 1.2):

– упакованный формат – каждый байт содержит две десятичные цифры. Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 размером 4 бита. При этом код старшей цифры

числа занимает старшие 4 бита. Следовательно, диапазон представления десятичного упакованного числа в одном байте составляет от 00 до 99;

– неупакованный формат – каждый байт содержит одну десятичную цифру в четырех младших битах. Старшие четыре бита имеют нулевое значение. Это так называемая зона. Следовательно, диапазон представления десятичного неупакованного числа в одном байте составляет от 0 до 9.

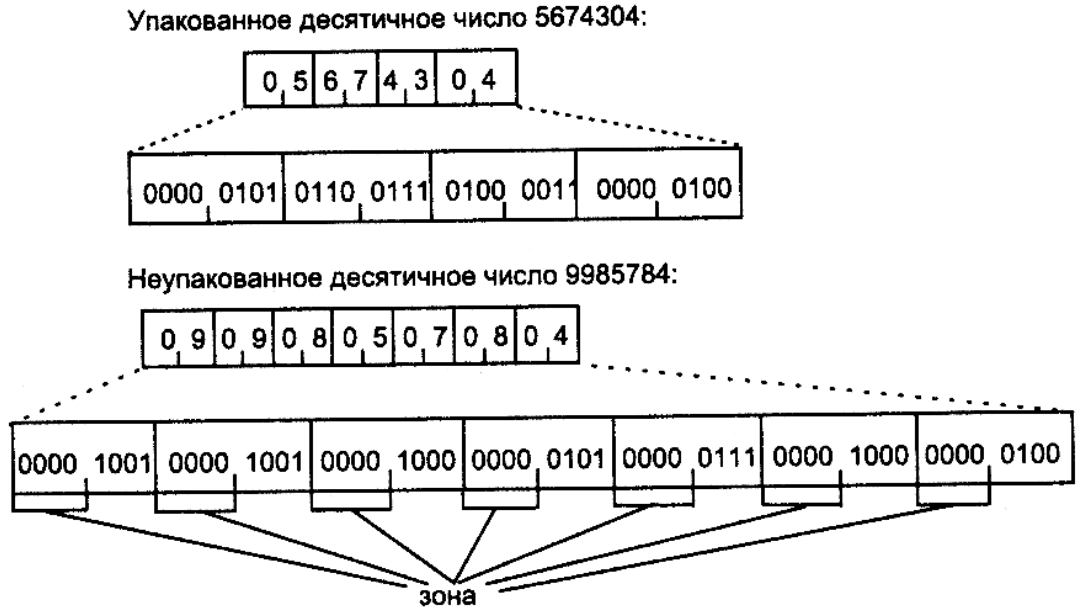

Рисунок 1.2 – Представление BCD-чисел

Для описания двоично-десятичных чисел в программе можно использовать только две директивы описания и инициализации данных – `db` и `dt` (длиной в 10 байт). Возможность применения только этих директив для описания BCD-чисел обусловлена тем, что к таким числам также применим принцип «младший байт по младшему адресу», что очень удобно для их обработки. При использовании BCD-чисел способ описания этих чисел в программе и алгоритм их обработки – это «дело вкуса» программиста.

Пример 1.1. Описание BCD-чисел.

```
masm
model small
stack 256
.data                          ;сегмент данных
per_1 db 2,3,4,6,8,2           ;неупакованное BCD-число 286432
                               ; в памяти 02 03 04 06 08 02
per_3 dt 9875645               ;упакованное BCD-число 9875645
                               ;в памяти 45 56 87 09
.code                          ;сегмент кода
main:                          ;точка входа в программу
```

```
mov ax,@data                ;связываем регистр dx с сегментом
mov ds,ax                   ;данных через регистр ax
; представление чисел можно посмотреть в дампе данных
mov ax,4c00h                ;стандартный выход
int 21h
end main                    ;конец программы
```

На рисунке 1.3 показано представления BCD-чисел из примера 1.1. в дампе данных.

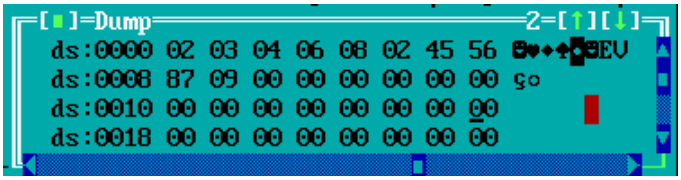

Рисунок 1.3 – Представление BCD-чисел в дампе данных

1.3 Арифметические операции над целыми двоичными числами

1.3.1 Сложение двоичных чисел без знака

Микропроцессор выполняет сложение операндов по правилам сложения двоичных чисел. Проблем не возникает до тех пор, пока значение результата не превышает размерности поля операнда. Например, при сложении операндов размером в байт результат не должен превышать число 255. Если это происходит, то результат оказывается неверен. Рассмотрим, почему так происходит. Например, выполним сложение: 254 + 5 = 259 в двоичном виде:

$1111\ 1110_2 + 0000\ 0101_2 = 1\ 0000\ 0011_2$.

Результат вышел за пределы восьми бит, и правильное его значение укладывается в 9 битов, а в 8-битном поле операнда осталось значение 3, что, конечно, неверно. Для фиксирования ситуации выхода за разрядную сетку результата, как в данном случае, предназначен флаг переноса `cf`. Он располагается в бите 0 регистра флагов `eflags/flags`. Именно установкой этого флага фиксируется факт переноса единицы из старшего разряда операнда. Анализ этого флага можно провести различными способами. Один из них – использовать команду условного перехода `jc` (переход в случае, если в результате работы предыдущей команды флаг `cf` установился в 1).

В системе команд микропроцессора имеются три команды двоичного сложения:

`inc операнд`

операция инкремента, то есть увеличения значения операнда на 1;

`add операнд_1,операнд_2`

команда сложения со следующим принципом действия:

операнд_1 = операнд_1 + операнд_2;

```
adc операнд_1,операнд_2
```

команда сложения с учетом флага переноса `cf`. Принцип действия команды:

операнд_1 = операнд_1 + операнд_2 + значение флага переноса `cf`.

Таким образом, команда `adc` является средством микропроцессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые микропроцессором длины стандартных полей.

Пример 1.2. Пример вычисления суммы чисел с учетом флага переноса.

```
xor ax,ax
add al,17
add al,254
jnc m1      ;если нет переноса, то перейти на m1
adc ah,0    ;в ax сумма с учетом переноса
m1: ...
```

1.3.2 Сложение двоичных чисел со знаком

Микропроцессор не различает числа со знаком и без знака. Вместо этого у него есть средства фиксирования возникновения характерных ситуаций, одно из них – это флаг переноса `cf`, установка которого в 1 говорит о том, что произошел выход за пределы разрядности операндов, и команда `adc`, которая учитывает возможность такого выхода (перенос из младшего разряда). Другое средство – это регистрация состояния старшего (знакового) разряда операнда, которое осуществляется с помощью флага переполнения `of` в регистре `eflags` (бит 11).

В компьютере положительные числа представляются в двоичном коде, а отрицательные – в дополнительном коде. Рассмотрим различные варианты сложения чисел. В примерах показано поведение двух старших битов операндов и правильность результата операции сложения.

Пример 1.3. Сложение чисел 1.

$30\ 566_{10} = 0111\ 0111\ 0110\ 0110_2$

+

$00\ 687_{10} = 0000\ 0010\ 1010\ 1111_2$

=

$31\ 253_{10} = 0111\ 1010\ 0001\ 0101_2$

Следим за переносами из 14-го и 15-го разрядов и правильностью результата: переносов нет, результат правильный.

Пример 1.4. Сложение чисел 2.

$30\ 566_{10} = 0111\ 0111\ 0110\ 0110_2$

+

$30\ 566_{10} = 0111\ 0111\ 0110\ 0110_2$

=

$61\ 132_{10} = 1110\ 1110\ 1100\ 1100_2$

Произошел перенос из 14-го разряда; из 15-го разряда переноса нет. Результат неправильный, так как имеется переполнение – значение числа получилось больше, чем то, которое может иметь 16-битное число со знаком (+32 767).

Пример 1.5. Сложение чисел 3.

$-30\ 566_{10} = 1000\ 1000\ 1001\ 1010_2$

+

$-04\ 875_{10} = 1110\ 1100\ 1111\ 0101_2$

=

$-35\ 441_{10} = 0111\ 0101\ 1000\ 1111_2$

Произошел перенос из 15-го разряда, из 14-го разряда нет переноса. Результат неправильный, так как вместо отрицательного числа получилось положительное (в старшем бите находится 0).

Пример 1.6. Сложение чисел 4.

$-4\ 875_{10} = 1110\ 1100\ 1111\ 0101_2$

+

$-4\ 875_{10} = 1110\ 1100\ 1111\ 0101_2$

=

$-9\ 750_{10} = 1101\ 1001\ 1110\ 1010_2$

Есть переносы из 14-го и 15-го разрядов. Результат правильный. Таким образом, ситуация переполнения (установка флага `of` в 1) происходит при переносе:

– из 14-го разряда (для положительных чисел со знаком);

– из 15-го разряда (для отрицательных чисел).

Переполнения не происходит (то есть флаг `of` сбрасывается в 0), если есть перенос из обоих разрядов или перенос отсутствует в обоих разрядах. Переполнение регистрируется с помощью флага переполнения `of`. Дополнительно к флагу `of` при переносе из старшего разряда устанавливается в 1 и флаг переноса `cf`. Так как микропроцессор не знает о существовании чисел со знаком и без знака, то вся ответственность за правильность действий с получившимися числами ложится на программиста. Таким образом, чтобы правильно огранизовать процесс сложения чисел, нужно анализировать флаги `cf` и `of`. Проанализировать флаги `cf` и `of` можно командами условного перехода `jc\jnc` и `jo\jno`, соответственно. Команды сложения чисел со знаком те же, что и для чисел без знака.

1.3.3 Вычитание двоичных чисел без знака

Как и при анализе операции сложения, рассмотрим процессы, происходящие при выполнении операции вычитания:

– если уменьшаемое больше вычитаемого, то проблем нет, разность положительна, результат верен;

– если уменьшаемое меньше вычитаемого, возникает проблема:

результат меньше 0, а это уже число со знаком. В этом случае результат необходимо «завернуть». Что это означает? При обычном вычитании (в столбик) делают заем 1 из старшего разряда. Микропроцессор поступает аналогично, то есть занимает 1 из разряда, следующего за старшим, в разрядной сетке операнда.

Пример 1.7. Вычитание чисел 1.

$05_{10} = 0000\ 0000\ 0000\ 0101_2$

–

$10_{10} = 0000\ 0000\ 0000\ 1010_2$

Для того чтобы произвести вычитание, произведем воображаемый заем из старшего разряда:

$05_{10} = 1\ 0000\ 0000\ 0000\ 0101_2$

–

$10_{10} = \quad 0000\ 0000\ 0000\ 1010_2$

=

$-5_{10} = \quad 1111\ 1111\ 1111\ 1011_2$

Тем самым, по сути, выполняется действие (65 536 + 5) –10 = = 65 531, 0 здесь как бы эквивалентен числу 65 536. Результат, конечно, неверен, но микропроцессор считает, что все нормально, хотя факт заема единицы он фиксирует установкой флага переноса `cf`. Если внимательно посмотреть на результат операции вычитания, то это же –5 в дополнительном коде! Проведем эксперимент: представим разность в виде суммы 5 + (–10).

Пример 1.8. Вычитание чисел 2.

$05 = 0000\ 0000\ 0000\ 0101_2$

+

$-10 = 1111\ 1111\ 1111\ 0110_2$

=

$-5_{10} = 1111\ 1111\ 1111\ 1011_2$

То есть мы получили тот же результат, что и в предыдущем примере. Таким образом, после команды вычитания чисел без знака нужно анализировать состояние флага `cf`. Если он установлен в 1, то это говорит о том, что произошел заем из старшего разряда и результат получился в дополнительном коде.

К командам вычитания относятся следующие команды:

```
dec операнд
```

операция декремента, то есть уменьшения значения операнда на 1;

```
sub операнд_1,операнд_2
```

команда вычитания; ее принцип действия:

операнд_1 = операнд_1 – операнд_2;

```
sbb операнд_1,операнд_2
```

команда вычитания с учетом заема (флага `cf`):

операнд_1 = операнд_1 – операнд_2 – значение_`cf`

Таким образом, среди команд вычитания есть команда sbb, учитывающая флаг переноса cf. Эта команда подобна adc, но теперь уже флаг cf выполняет роль индикатора заема 1 из старшего разряда при вычитании чисел.

Пример 1.9. Проверка при вычитании чисел без знака.

```
xor ax,ax
mov al,5
sub al,10
jnc ml      ;нет переноса?
neg a1      ;в al модуль результата
ml: ...
```

С указанными для этой команды вычитания исходными данными результат получается в дополнительном коде (отрицательный). Для того чтобы преобразовать результат к нормальном виду (получить его модуль), применяется команда neg, с помощью которой получается дополнение операнда. В нашем случае мы получили дополнение дополнения, или модуль отрицательного результата. А тот факт, что это на самом деле число отрицательное, отражен в состоянии флага cf.

1.3.4 Вычитание двоичных чисел со знаком

Микропроцессору незачем иметь два устройства – сложения и вычитания. Достаточно наличия только одного – устройства сложения. Но для вычитания способом сложения чисел со знаком в дополнительном коде необходимо представлять оба операнда – и уменьшаемое, и вычитаемое. Результат тоже нужно рассматривать как значение в дополнительном коде. Но здесь возникают сложности. Прежде всего, они связаны с тем, что старший бит операнда рассматривается как знаковый. Рассмотрим пример вычитания 45 – (–127).

Пример 1.10. Вычитание чисел со знаком 1.

$45_{10} = 0010\ 1101$

–

$–127_{10} = 1000\ 0001_2$

=

$–44_{10} = 1010\ 1100_2$

Судя по знаковому разряду, результат получился отрицательный, что, в свою очередь, говорит о том, что число нужно рассматривать как дополнение, равное –44. Правильный результат должен быть равен 172. Здесь мы, как и в случае знакового сложения, встретились с переполнением мантиссы, когда значащий разряд числа изменил знаковый разряд операнда. Отследить такую ситуацию можно по содержимому флага переполнения of. Его установка в 1 говорит о том, что результат вышел за диапазон представления знаковых чисел (то есть изменился старший бит) для операнда данного размера, и программист должен предусмотреть действия по корректировке результата.

Пример 1.11. Вычитание чисел со знаком 2.

$-45 - 45 = -45 + (-45) = -90.$

$-45_{10} = 1101\ 0011_2$

+

$-45_{10} = 1101\ 0011_2$

=

$-90_{10} = 1010\ 0110_2$

Флаг переполнения `of` сброшен в 0, а 1 в знаковом разряде говорит о том, что значение результата – число в дополнительном коде.

Процесс сложения многобайтных чисел происходит так же, как и при сложении двух чисел «в столбик», с осуществлением при необходимости переноса 1 в старший разряд.

Принцип вычитания чисел с диапазоном представления, превышающим стандартные разрядные сетки операндов, тот же, что и при сложении, то есть используется флат переноса `cf`. Нужно только представлять себе процесс вычитания в столбик и правильно комбинировать команды микропроцессора с командой `sbb`.

Кроме флагов `cf` и `of` в регистре `eflags` есть еще несколько флагов, которые можно использовать с двоичными арифметическими командами:

`zf` – флаг нуля, который устанавливается в 1, если результат операции равен 0, и в 0, если результат не равен 0;

`sf` – флаг знака, значение которого после арифметических операций (и не только) совпадает со значением старшего бита результата, то есть с битом 7, 15 или 31. Таким образом, этот флаг можно использовать для операций над числами со знаком.

1.3.5 Умножение двоичных чисел без знака

Для умножения чисел без знака предназначена команда

```
mul сомножитель_1
```

В команде указан только один операнд-сомножитель. Второй операнд-сомножитель_2 задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Произведение состоит из двух частей и в зависимости от размера операндов размещается в двух местах – на месте сомножителя 2 (младшая часть) и в дополнительных регистрах `ah`, `dx`, `edx` (старшая часть). Чтобы узнать, что результат достаточно мал и уместился в одном регистре, или превысил размерность регистра, и старшая часть оказалась в другом регистре, нужно проанализировать флаги переноса `cf` и переполнения `of`:

– если старшая часть результата нулевая, то после операции произведения флаги `cf` = 0 и `of` = 0;

– если же эти флаги ненулевые, то это означает, что результат вышел за пределы младшей части произведения и состоит из двух частей, что и нужно учитывать при дальнейшей работе.

Таблица 1.2 – Расположение операндов и результата при умножении

Сомножитель_1	Сомножитель_2	Результат
Байт	al	16 бит в ax: al – младшая часть результата; ah – старшая часть результата
Слово	ax	32 бит в паре dx:ax: ax – младшая часть результата; dx – старшая часть результата
Двойное слово	eax	64 бит в паре edx:eax: eax – младшая часть результата; edx – старшая часть результата

Пример 1.12. Умножение операндов с проверкой на корректность результата.

```
masm
model small
.data
  rez_l db 45
  rez_h db 0
.stack
  db 256 dup(0)
.code
main proc near
  mov ax,@data
  mov ds,ax
  xor ax,ax
  mov al,25
  mul rez_l
  jnc ml            ;если нет переполнения, то переход на ml
  mov rez_h,ah      ;старшая часть результата в rez_h
  ml:
  mov rez_l,al
  mov ax,4c00h
  int 21h
main endp
end main
```

Для выяснения размера результата командой условного перехода jnc анализируется состояние флага cf, и если оно не равно 1, то результат остается в рамках регистра al. Если же cf = 1, то выполняется команда, которая формирует в поле rez_h старшее слово результата. Команда mov rez_l,al формирует младшую часть результата.

1.3.6 Умножение двоичных чисел со знаком

Для умножения чисел со знаком предназначена команда

```
imul операнд_1
```

Эта команда выполняется так же, как и команда `mul`. Отличительной особенностью команды `imul` является только формирование знака. Если результат мал и умещается в одном регистре (то есть если `cf` = `of` = 0), то содержимое другого регистра (старшей части) является расширением знака – все его биты равны старшему биту (знаковому разряду) младшей части результата. В противном случае (если `cf` = `of` = 1) знаком результата является знаковый бит старшей части результата, а знаковый бит младшей части является значащим битом двоичного кода результата.

1.3.7 Деление двоичных чисел без знака

Для деления чисел без знака предназначена команда

```
div делитель
```

Делитель может находиться в памяти или в регистре и иметь размер 8, 16 или 32 бит. Местонахождение делимого фиксировано и так же, как в команде умножения, зависит от размера операндов. Результатом команды деления являются значения частного и остатка.

Таблица 1.3 – Расположение операндов и результата при делении

Делимое	Делитель	Частное	Остаток
Слово 16 бит в регистре `ax`	Байт – регистр или ячейка памяти	Байт в регистре `al`	Байт в регистре `ah`
32 бит: `dx` – старшая часть; `ax` – младшая часть	16 бит регистр или ячейка памяти	Слово 16 бит в регистре `ax`	Слово 16 бит в регистре `dx`
64 бит: `edx` – старшая часть; `eax` – младшая часть	Двойное слово 32 бит регистр или ячейка памяти	Двойное слово 32 бит в регистре `eax`	Двойное слово 32 бит в регистре `edx`

После выполнения команды деления содержимое флагов неопределенно, но возможно возникновение прерывания с номером 0, называемого «деление на ноль». Этот вид прерывания относится к так называемым исключениям. Эта разновидность прерываний возникает внутри микропроцессора из-за некоторых аномалий во время вычислительного процесса. Прерывание 0 – «деление на ноль» – при выполнении команды `div` может возникнуть по одной из следующих причин:

– делитель равен нулю;

– частное не входит в отведенную под него разрядную сетку, что может произойти в следующих случаях:

– при делении делимого величиной в слово на делитель величиной в байт, причем значение делимого в более чем 256 раз больше значения делителя;

– при делении делимого величиной в двойное слово на делитель величиной в слово, причем значение делимого в более чем 65 536 раз больше значения делителя;

– при делении делимого величиной в учетверенное слово на делитель величиной в двойное слово, причем значение делимого в более чем 4 294 967 296 раз больше значения делителя.

1.3.8 Деление двоичных чисел со знаком

Для деления чисел со знаком предназначена команда

```
idiv делитель
```

Для этой команды справедливы все рассмотренные положения, касающиеся команд и чисел со знаком. Отметим лишь особенности возникновения исключения 0 «деление на ноль» в случае чисел со знаком. Оно возникает при выполнении команды `idiv` по одной из следующих причин:

– делитель равен нулю;

– частное не входит в отведенную для него разрядную сетку. Последнее, в свою очередь, может произойти:

– при делении делимого величиной в слово со знаком на делитель величиной в байт со знаком, причем значение делимого в более чем 128 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –128 до +127);

– при делении делимого величиной в двойное слово со знаком на делитель величиной в слово со знаком, причем значение делимого в более чем 32 768 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –32 768 до +32 768);

– при делении делимого величиной в учетверенное слово со знаком на делитель величиной в двойное слово со знаком, причем значение делимого в более чем 2 147483648 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от ~2 147 483 648 до + 2147483647).

1.4 Вспомогательные команды для целочисленных операций

Эти команды расширяют байты в слова, слова – в двойные слова и двойные слова – в учетверенные слова (64-разрядные значения). Команды преобразования типа используются при преобразовании целых со знаком, так как они автоматически заполняют старшие биты вновь формируемого операнда значениями знакового бита старого объекта. Эта операция приводит к целым значениям того же знака и той же

величины, что и исходная, но уже в более длинном формате. Подобное преобразование называется операцией распространения знака. Существуют два вида команд преобразования типа:

1. Команды без операндов – эти команды работают с фиксированными регистрами:

`cbw` (Convert Byte to Word) – команда преобразования байта (в регистре `al`) в слово (в регистре `ax`) путем распространения значения старшего бита `al` на все биты регистра `ah`;

`cwd` (Convert Word to Double) – команда преобразования слова (в регистре `ax`) в двойное слово (в регистрах `dx:ax`) путем распространения значения старшего бита `ax` на все биты регистра `dx`;

`cwde` (Convert Word to Double) – команда преобразования слова (в регистре `ax`) в двойное слово (в регистре `eax`) путем распространения значения старшего бита `ax` на все биты старшей половины регистра `eax`;

`cdq` (Convert Double Word to Quarter Word) – команда преобразования двойного слова (в регистре `eax`) в учетверенное слово (в регистрах `edx:eax`) путем распространения значения старшего бита `eax` на все биты регистра `edx`;

2. Команды `movsx` и `movzx`, относящиеся к командам обработки строк. Эти команды обладают полезным свойством:

```
movsx операнд_1,операнд_2
```

переслать с распространением знака. Расширяет 8- или 16-разрядное значение операнд_2, которое может быть регистром или операндом в памяти, до 16- или 32-разрядного значения в одном из регистров, используя значение знакового бита для заполнения старших позиций операнд_1. Данную команду удобно использовать для подготовки операндов со знаками к выполнению арифметических действий;

```
movzx операнд_1,операнд_2
```

переслать с расширением нулем. Расширяет 8- или 16-разрядное значение операнд_2 до 16- или 32-разрядного с очисткой (заполнением) нулями старших позиций операнда. Данную команду удобно использовать для подготовки операндов без знака к выполнению арифметических действий.

В системе команд микропроцессора имеются еще две полезные команды:

```
xadd назначение,источник
```

обмен местами и сложение. Команда позволяет выполнить последовательно два действия: обменять значения операндов назначение и источник, поместить на место операнда назначение сумму:

назначение = назначение + источник.

```
neg операнд
```

отрицание с дополнением до двух. Команда выполняет инвертирование значения операнд. Физически команда выполняет одно действие:

операнд = 0 – операнд,

то есть вычитает операнд из нуля.

Команду `neg` операнд можно применять для смены знака; выполнения вычитания из константы. Команды `sub` и `sbb` пе позволяют вычесть что-либо из константы, так как константа не может служить операндом-приемником в этих операциях. Поэтому данную операцию можно выполнить с помощью двух команд:

```
neg ax          ;смена знака (ax)
add ax,340      ;фактически вычитание: (ax) = 340 – (ax)
```

1.5 Арифметические операции над неупакованными BCD-числами

BCD-числа нужны в деловых приложениях, то есть там, где числа должны быть большими и точными. К недостаткам использования двоичных чисел можно отнести следующие: значения величин в формате слова и двойного слова имеют ограниченный диапазон; наличие ошибок округления; представление большого объема результатов в символьном виде (ASCII-коде). Шестнадцатеричное представление неупакованной десятичной цифры и соответствующий ей символ в таблице ASCII отличаются на величину 30h.

Отдельных команд сложения, вычитания, умножения и деления BCD-чисел нет. Складывать и вычитать можно двоично-десятичные числа как в упакованном формате, так и в неупакованном, а вот делить и умножать можно только неупакованные BCD-числа.

1.5.1 Сложение

Рассмотрим два случая сложения.

Пример 1.13. Результат сложения не больше 9.

$6_{10} = 0000\ 0110_2$
+
$3_{10} = 0000\ 0011_2$
=
$9_{10} = 0000\ 1001_2$

Переноса из младшей тетрады в старшую нет. Результат правильный.

Пример 1.14. Результат сложения больше 9.

$06_{10} = 0000\ 0110_2$
+
$07_{10} = 0000\ 0111_2$
=
$13_{10} = 0000\ 1101_2$

То есть получено уже не BCD-число. Результат неправильный. Правильный результат в неупакованном BCD-формате должен быть таким: 0000 0001 0000 0011_2 в двоичном представлении (или 13 в десятичном). Полученный результат нужно корректировать. Для коррекции операции сложения двух однозначных неупакованных BCD-чисел в системе команд микропроцессора существует специальная команда

`aaa` (ASCII Adjust for Addition) – коррекция результата сложения для представления в символьном виде. Эта команда не имеет операндов. Она работает неявно только с регистром `al` и анализирует значение его младшей тетрады. Если это значение меньше 9, то флаг `cf` сбрасывается в 0, и осуществляется переход к следующей команде. Если это значение больше 9, то выполняются следующие действия:

– к содержимому младшей тетрады `al` (но не к содержимому всего регистра!) прибавляется 6, тем самым значение десятичного результата корректируется в правильную сторону;

– флаг `cf` устанавливается в 1, тем самым фиксируется перенос в старший разряд для того, чтобы его можно было учесть в последующих действиях.

Так, в примере 1.14, предполагая, что значение суммы 0000 1101 находится в `al`, после команды `aaa` в регистре будет 1101 + 0110 = = 0011, то есть двоичное 0000 0011 или десятичное 3, а флаг `cf` установится в 1, то есть перенос запомнился в микропроцессоре. Далее программисту нужно будет использовать команду сложения `adc`, которая учтет перенос из предыдущего разряда.

Пример 1.15. Сложение неупакованных BCD-чисел.

```
masm
model small
.data
  len equ 2 ;разрядность числа
  b db 1,7  ;неупакованное число 71
  c db 4,5  ;неупакованное число 54
  sum db 3 dup (0) ;результат сложения формируем в поле sum
.stack
  db 256 dup (0)
.code
main proc near
  mov ax,@data
  mov ds,ax
  xor bx,bx
  mov cx,len
m1: mov al,b[bx]
  adc al,c[bx] ;складываем цифры в очередных разрядах
               ;BCD-чисел, при этом учитываем возможный
```

```
                    ;перенос из младшего разряда
  aaa               ;корректируем результат сложения, формируя
                    ;в al BCD-цифру и, при необходимости,
                     ;устанавливая в 1 флаг cf
  mov sum[bx],al
  inc bx
  loop m1
  adc sum[bx],0 ;учитываем возможность переноса при сложении
                    ;цифр из самых старших разрядов чисел
  mov ax,4c00h
  int 21h
main endp
end main
```

Порядок ввода BCD-чисел обратен нормальному, то есть цифры младших разрядов расположены по меньшему адресу. Такой порядок удовлетворяет общему принципу представления данных для микропроцессоров Intel, и это очень удобно для поразрядной обработки неупакованных BCD-чисел, так как каждое из них занимает один байт.

1.5.2 Вычитание

Рассмотрим два возможных случая.

Пример 1.16. Результат вычитания не больше 9.

$6_{10} = 0000\ 0110_2$

–

$3_{10} = 0000\ 0011_2$

=

$3_{10} = 0000\ 0011_2$

Заема из старшей тетрады нет. Результат верный и корректировки не требует.

Пример 1.17. Результат вычитания больше 9.

$6_{10} = 0000\ 0110_2$

–

$7_{10} = 0000\ 0111_2$

=

$-1_{10} = 1111\ 1111_2$

Вычитание проводится по правилам двоичной арифметики. Поэтому результат не является BCD-числом. Правильный результат в неупакованном BCD-формате должен быть 9 (0000 1001 в двоичной системе счисления). При этом предполагается заем из старшего разряда, как при обычной команде вычитания, то есть в случае с BCD-числами фактически должно быть выполнено вычитание 16 – 7.

Таким образом, результат вычитания нужно корректировать. Для этого существует специальная команда

aas (ASCII Adjust for Substraction) – коррекция результата вычитания для представления в символьном виде. Команда aas не имеет операндов и работает с регистром al, анализируя его младшую тетраду следующим образом: если ее значение меньше 9, то флаг cf сбрасывается в 0, и управление передается следующей команде. Если значение тетрады в al больше 9, то команда aas выполняет следующие действия:

– из содержимого младшей тетрады регистра al вычитает 6;

– обнуляет старшую тетраду регистра al;

– устанавливает флаг cf в 1, тем самым фиксируя воображаемый заем из старшего разряда.

Команда aas применяется вместе с основными командами вычитания sub и sbb. При этом команду sub есть смысл использовать только один раз при вычитании самых младших цифр операндов, далее должна применяться команда sbb, которая будет учитывать возможный заем из старшего разряда.

Пример 1.18. Вычитание неупакованных BCD-чисел.

```
masm
model small
.data
  b db 1,7 ;неупакованное число 71
  c db 4,5 ;неупакованное число 54
  subs db 2 dup (0)
  mes db 'уменьшаемое меньше вычитаемого$'
.stack
  db 256 dup(0)
.code
  main proc near
  mov ax,@data
  mov ds,ax
  xor ax,ax    ;очищаем ax
  len equ 2    ;разрядность чисел
  xor bx,bx
  mov cx,len   ;загрузка в cx счетчика цикла
m1: mov al,b[bx]
  sbb al,c[bx]
  aas
  mov subs[bx],al
  inc bx
  loop m1
  jc m2 ;анализ флага заема. Предусматриваем случай, когда после
        ;вычитания старших цифр чисел был зафиксирован факт
        ;заема. Это говорит о том, что вычитаемое было больше
```

```
            ;уменьшаемого, в результате чего разность будет
            ;неправильной. Командой jc анализируется флаг cf.
  jmp exit
m2: lea dx,mes ;вывод сообщения
  mov ax,0900h
  int 21h
exit:
  mov ax,4c00h
  int 21h
main endp
end main
```

1.5.3 Умножение

В системе команд микропроцессора присутствуют только средства для умножения и деления одноразрядных неупакованных BCD-чисел.

Для того чтобы умножать числа произвольной размерности, нужно реализовать процесс умножения самостоятельно, взяв за основу некоторый алгоритм умножения, например «в столбик».

Для того чтобы перемножить два одноразрядных BCD-числа, необходимо:

– поместить один из сомножителей в регистр `al`;

– поместить второй операнд в регистр или память, отведя байт;

– перемножить сомножители командой `mul` (результат будет в `ax`);

– результат получится в двоичном коде, поэтому его нужно скорректировать.

Для коррекции результата после умножения применяется команда `aam` (ASCII Adjust for Multiplication) – коррекция результата умножения для представления в символьном виде. Команда не имеет операндов и работает с регистром `ax` следующим образом:

– делит `al` на 10;

– результат деления записывается следующим образом: частное – в `al`, остаток – в `ah`.

После выполнения команды `aam` в регистрах `al` и `ah` находятся правильные двоично-десятичные цифры произведения двух цифр.

Пример 1.19. Умножение BCD-числа произвольной размерности на однозначное BCD-число.

```
masm
model small
.data
  b db 6,7    ;неупакованное число 76
  c db 4      ;неупакованное число 4
  prod db 4 dup (0) ;произведение
.stack
```

```
  db 256 dup (0)
.code
main proc near
  mov ax,@data
  mov ds,ax
  xor ax,ax
  len equ 2     ;размерность сомножителя 1
  xor bx,bx
  xor si,si
  xor di,di
  mov cx,len    ;в cx длина наибольшего сомножителя 1
m1: mov al,b[si]
  mul c
  aam           ;коррекция умножения
  adc al,dl ;учли предыдущий перенос
  aaa           ;скорректировали результат сложения с переносом
  mov dl,ah ;запомнили перенос
  mov prod[bx],al
  inc si
  inc bx
  loop m1
  mov prod[bx],dl   ;учли последний перенос
  mov ax,4c00h
  int 21h
main endp
end main
```

Команду `aam` можно применять для преобразования двоичного числа в регистре `al` в неупакованное BCD-число, которое будет размещено в регистре `ax`: старшая цифра результата – в `ah`, младшая – в `al`.

1.5.4. Деление

При делении BCD-чисел также требуются действия по коррекции, но они должны выполняться до основной операции, выполняющей непосредственно деление одного BCD-числа на другое BCD-число. Предварительно в регистре `ax` нужно получить две неупакованные BCD-цифры делимого. Далее нужно использовать команду `aad`:

`aad` (ASCII Adjust, for Division) – коррекция деления для представления в символьном виде. Команда не имеет операндов и преобразует двузначное неупакованное BCD-число в регистре `ax` в двоичное число. Это двоичное число впоследствии будет играть роль делимого в операции деления. Кроме преобразования, команда `aad` помещает полученное двоичное число в регистр `al`. Делимое, естественно, будет двоичным числом из диапазона 0...99. Алгоритм, по которому команда `aad` осуществляет это преобразование, состоит в следующем:

– умножить старшую цифру исходного BCD-числа в ax (содержимое ah) на 10;

– выполнить сложение ah + al, результат которого (двоичное число) занести в al;

– обнулить содержимое ah.

Далее нужно записать обычную команду деления div для выполнения деления содержимого ax на одну BCD-цифру, находящуюся и байтовом регистре или байтовой ячейке памяти.

Пример 1.20. Деление неупакованных BCD-чисел.

```
masm
model small
.data
  b db 1,7 ;неупакованное BCD-число 71
  c db 4
  ten db 10
  rez db 2 dup (0)
.stack
  db 256 dup(0)
.code
main proc near
  mov ax,@data
  mov ds,ax
  xor ax,ax  ;формируем в ах число 71
  mov si,1
  mov al,b[si]
  imul ten   ;умножаем первую цифру числа на 10
  dec si
  add al,b[si] ;прибавляем вторую цифру числа
  div c        ;делим: в al BCD-частное, в ah BCD-остаток
  ;формируем результат в виде BCD-числа
  mov ah,0     ;убираем остаток
  div ten      ;выделяем цифры числа
  mov di,0
  mov rez[di],ah ;запоминаем вторую цифру числа
  inc di
  mov rez[di],al ;запоминаем первую цифру числа
  mov ax,4c00h
  int 21h
main endp
end main
```

Команду aad можно использовать для перевода неупакованных BCD-чисел из диапазона 0...99 в их двоичный эквивалент.

Для деления чисел большей разрядности нужно реализовывать алгоритм деления в «столбик».

1.6 Упакованные BCD-числа

Упакованные BCD-числа можно только складывать и вычитать. Для выполнения других действий над ними их нужно дополнительно преобразовывать либо в неупакованный формат, либо в двоичное представление.

1.6.1 Сложение

Пример 1.21. Сложение упакованных BCD-чисел.

$67_{10} = 0110\ 0111_2$

+

$75_{10} = 0111\ 0101_2$

=

$142_{10} = 1101\ 1100_2 = 220_{10}$

В двоичном виде результат равен 1101 1100 (или 220 в десятичном представлении), что неверно. Это происходит потому, что микропроцессор «не подозревает» о существовании BCD-чисел и складывает их по правилам сложения двоичных чисел. Правильный результат в двоично-десятичном виде должен быть равен 0001 0100 0010 (или 142 в десятичном представлении). Для упакованных BCD-чисел также нужно корректировать результаты арифметических операций. У микропроцессора есть для этого команда `daa`:

`daa` (Decimal Adjust for Addition) – коррекция результата сложения для представления в десятичном виде. Команда `daa` преобразует содержимое регистра `al` в две упакованные десятичные цифры.

Получившаяся в результате сложения единица (если результат сложения больше 99) запоминается во флаге `cf`, тем самым учитывается перенос в старший разряд.

Пример 1.22. Сложение упакованных BCD-чисел.

```
masm
model small
.data       ;сегмент данных
  b db 17h ;упакованное число 17h
  c db 45h ;упакованное число 45h
  sum db 2 dup (0)
.stack
  db 256 dup(0)
.code
  main proc near
  mov ax,@data
  mov ds,ax
  xor ax,ax
  mov al,b
  add al,c
  daa
```

```
  jnc m1         ;переход через команду, если результат ≤ 99
  mov sum+1,ah   ;учет переноса при сложении (результат > 99)
m1: mov sum,al   ;младшие упакованные цифры результата
  mov ax,4c00h
  int 21h
main endp
end main
```

Результат формируется в соответствии с основным принципом работы микропроцессоров Intel: младший байт по младшему адресу.

1.6.2 Вычитание

Аналогично сложению, микропроцессор рассматривает упакованные BCD-числа как двоичные, и, соответственно, выполняет вычитание BCD-чисел как двоичное. Выполним вычитание 67 – 75. При вычитании чисел нужно помнить, что микропроцессор выполняет вычитание способом сложения.

Пример 1.23. Вычитание упакованных BCD-чисел.

$67_{10} = 0110\ 0111_2$

+

$-75_{10} = 1011\ 0101_2$

=

$-8_{10} = 0001\ 1100_2 = 28_{10}$?

Результат равен 28 в десятичной системе счисления, что является неверным. В двоично-десятичном коде результат должен быть равен $0000\ 1000_2$ (или 8 в десятичной системе счисления). При программировании вычитания упакованных BCD-чисел программист должен сам осуществлять контроль за знаком. Это делается с помощью флага `cf`, который фиксирует заем из старших разрядов. Само вычитание BCD-чисел осуществляется простыми командами вычитания `sub` или `sbb`. Коррекция результата осуществляется командой `das`:

`das` (Decimal Adjust for Substraction) – коррекция результата вычитания для представления в десятичном виде.

1.7 Практическое задание

Составить и отладить программу на языке ассемблера для вычисления заданного арифметического выражения. Исходные данные являются двузначными неупакованными двоично-десятичными числами. При написании программы предусмотреть возможность переполнения и некорректного результата (использовать команды коррекции). Варианты заданий даны в таблице 1.4.

Пример 1.24. Вычислить значение выражения $y = \dfrac{7a + 2b - c}{3}$, используя неупакованные BCD-числа.

```
masm
model small
.data
  len equ 2   ;разрядность числа
  a db 2,1    ;неупакованное число 12
  b db 1,2    ;неупакованное число 21
  c db 4,5    ;неупакованное число 54
  seven db 7
  two db 2
  three db 3
  ten db 10
  prod1 db 3 dup (0) ;произведение 7a = 84
  prod2 db 3 dup (0) ;произведение 2b = 42
  sum1 db 3 dup (0)  ;сумма 7a + 2b = 126
  sub1 db 3 dup (0)  ;разность 7a + 2b – c = 72
  rez db 2 dup (0)   ;частное (7a + 2b – c) / 3 = 24
.stack
  db 256 dup (0)
.code
main proc near
  mov ax,@data
  mov ds, ax
  xor ax,ax    ;произведение 7a = 84
  len equ 2    ;размерность сомножителя 1
  xor bx,bx
  xor si,si
  xor di,di
  mov cx,len ;в cx длина наибольшего сомножителя 1
m1: mov al,a[si]
  mul seven
  aam          ;коррекция умножения
  adc al,dl    ;учли предыдущий перенос
  aaa          ;скорректировали результат сложения с переносом
  mov dl,ah    ;запомнили перенос
  mov prod1[bx],al
  inc si
  inc bx
  loop m1
  mov prod1[bx],dl    ;учли последний перенос
  xor ax,ax    ;произведение 2b = 84
  len equ 2    ;размерность сомножителя 1
  xor bx,bx
  xor si,si
```

```
  xor di,di
  mov x,len   ;в cx длина наибольшего сомножителя 1
m2: mov al,b[si]
  mul two
  aam          ;коррекция умножения
  adc al,dl    ;учли предыдущий перенос
  aaa ;скорректировали результат сложения с переносом
  mov dl,ah    ;запомнили перенос
  mov prod2[bx],al
  inc si
  inc bx
  loop m2
  mov prod2[bx],dl     ;учли последний перенос
  xor bx,bx    ;сложение 7a + 2b = 126
  mov cx,3
m3: mov al,prod1[bx]
  adc al,prod2[bx]
  aaa
  mov sum1[bx],al
  inc bx
  loop m3
  adc sum1[bx],0
  xor ax,ax    ;вычитание 7a + 2b – c = 72
  xor bx,bx
  mov cx,2     ;загрузка в cx счетчика цикла
m4: mov al,sum1[bx]
  sbb al,c[bx]
  aas
  mov sub1[bx],al
  inc bx
  loop m4
  xor ax,ax    ;деление (7a + 2b – c) / 3 = 24
  mov si,1
  mov al,sub1[si]
  imul ten   ;умножаем первую цифру числа на 10
  dec si
  add al,sub1[si]  ;прибавляем вторую цифру числа
  div three    ;делим: в al BCD-частное, в ah BCD-остаток
  ;формируем результат в виде BCD-числа
  mov bl,al    ;запоминаем частное
  mov ah,0     ;убираем остаток
  div ten      ;выделяем цифры числа
  mov di,0
```

```
  mov rez[di],ah   ;запоминаем вторую цифру числа
  inc di
  mov rez[di],al   ;запоминаем первую цифру числа
  mov ax,4c00h
  int 21h
main endp
end main
```

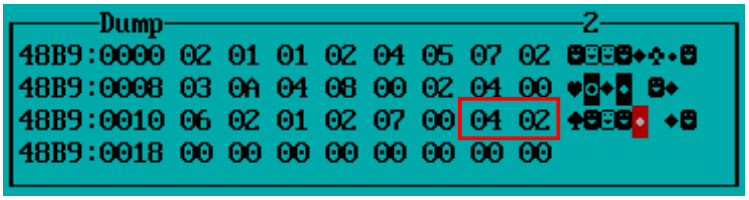

Рисунок 1.4 – Результат выполнения программы

Таблица 1.4 – Варианты заданий

№	Выражение	№	Выражение	№	Выражение
1	$y=\frac{5a-b+2c}{2}$	11	$y=\frac{3a+4b-2c}{5}$	21	$y=\frac{3b+2a-c}{5}$
2	$y=\frac{4a+b-3c}{2}$	12	$y=\frac{4a-2b+7c}{4}$	22	$y=\frac{3(a-b)+c}{2}$
3	$y=\frac{2a+b-3c}{8}$	13	$y=\frac{3a-2b+c}{3}$	23	$y=\frac{a-5b+2c}{7}$
4	$y=\frac{2a-c+4b}{7}$	14	$y=\frac{2(a+7b)-c}{4}$	24	$y=\frac{5(a-3b)+2c}{6}$
5	$y=\frac{2a+5b-3c}{2}$	15	$y=\frac{3(a-b)+c}{5}$	25	$y=\frac{4(a-2b)+3c}{5}$
6	$y=\frac{3a+2b-c}{6}$	16	$y=\frac{2a+3b-c}{5}$	26	$y=\frac{7(a-b)+2c}{5}$
7	$y=\frac{6a+4b-2c}{7}$	17	$y=\frac{5(a+b)-2c}{3}$	27	$y=\frac{4(a+2b)-3c}{5}$
8	$y=\frac{5a-2b+c}{3}$	18	$y=\frac{2a-3b+5c}{4}$	28	$y=\frac{7(b+2a)-3c}{4}$
9	$y=\frac{3a-b+4c}{4}$	19	$y=\frac{a-5b+2c}{3}$	29	$y=\frac{2(a+b)-4c}{7}$
10	$y=\frac{5a-2b+c}{3}$	20	$y=\frac{a+4b-2c}{3}$	30	$y=\frac{2(a+3b)-5c}{4}$

Тема 2. Цепочечные команды

Цепочечные команды также называют *командами обработки строк символов*. Под строкой символов понимается последовательность байт, а цепочка – это более общее название для случаев, когда элементы последовательности имеют размер слово или двойное слово. То есть цепочечные команды позволяют проводить действия над блоками памяти, представляющими собой последовательности элементов следующего размера: 8 бит (байт); 16 бит (слово); 32 бита (двойное слово).

Содержимое этих блоков для микропроцессора не имеет никакого значения. Это могут быть символы, числа и все что угодно. Главное, чтобы размерность элементов совпадала с одной из вышеперечисленных, и эти элементы находились в соседних ячейках памяти.

В системе команд микропроцессора имеются семь операций-примитивов обработки цепочек. Каждая из них реализуется в микропроцессоре тремя командами, в свою очередь, каждая из этих команд работает с соответствующим размером элемента – байтом, словом или двойным словом. Особенность всех цепочечных команд в том, что они, кроме обработки текущего элемента цепочки, осуществляют еще и автоматическое продвижение к следующему элементу данной цепочки.

Операции-примитивы и команды, с помощью которых они реализуются:

1. Пересылка цепочки:

`movs адрес_приемника,адрес_источника` (MOVe String) – переслать цепочку;

`movsb` (MOVe String Byte) – переслать цепочку байтов;

`movsw` (MOVe String Word) – переслать цепочку слов;

`movsd` (MOVe String Double word) – переслать цепочку двойных слов.

Команды производят копирование элементов из одной области памяти (цепочки) в другую.

2. Сравнение цепочек:

`cmps адрес_приемника,адрес_источника` (CoMPare String) – сравнить строки;

`cmpsb` (CoMPare String Byte) – сравнить строку байт;

`cmpsw` (CoMPare String Word) – сравнить строку слов;

`cmpsd` (CeMPare String Double word) – сравнить строку двойных слов.

Команды производят сравнение элементов цепочки-источника с элементами цепочки-приемника.

3. Сканирование цепочки:

`scas адрес_приемника` (SCAning String) – сканировать цепочку;

`scasb` (SCAning String Byte) – сканировать цепочку байт;

`scasw` (SCAning String Word) – сканировать цепочку слов;

`scasd` (SCAning String Double Word) – сканировать цепочку двойных слов.

Команды производят поиск некоторого значения в области памяти.

4. Загрузка элемента из цепочки:

`lods адрес_источника` (LOaD String) – загрузить элемент из цепочки в регистр-аккумулятор `al`/`ax`/`eax`;

`lodsb` (LOaD String Byte) – загрузить байт из цепочки в регистр `al`;

`lodsw` (LOaD String Word) – загрузить слово из цепочки в регистр `ax`;

`lodsd` (LOaD String Double Word) – загрузить двойное слово из цепочки в регистр `eax`.

Эта операция позволяет извлечь элемент цепочки и поместить его в регистр-аккумулятор `al`, `ax` или `eax`.

5. Сохранение элемента в цепочке:

`stos адрес_приемника` (STOre String) – сохранить элемент из регистра-аккумулятора `al`/`ax`/`eax` в цепочке;

`stosb` (STOre String Byte) – сохранить байт из регистра `al` в цепочке;

`stosw` (STOre String Word) – сохранить слово из регистра `ax` в цепочке;

`stosd` (STOre String Double Word) – сохранить двойное слово из регистра `eax` в цепочке.

Эта операция позволяет произвести действие, обратное команде `lods`, то есть сохранить значение из регистра-аккумулятора в элементе цепочки.

6. Получение элементов цепочки из порта ввода-вывода:

`ins адрес_приемника,номер_порта` (INput String) – ввести элементы из порта ввода-вывода в цепочку;

`insb` (INput String Byte) – ввести из порта цепочку байтов;

`insw` (INput String Word) – ввести из порта цепочку слов;

`insd` (INput String Double Word) – ввести из порта цепочку двойных слов.

7. Вывод элементов цепочки в порт ввода-вывода:

`outs номер_порта,адрес_источника` (OUTput String) – вывести элементы из цепочки в порт ввода-вывода;

`outsb` (OUTput String Byte) – вывести цепочку байтов в порт ввода-вывода;

`outsw` (OUTtput String Word) – вывести цепочку слов в порт ввода-вывода;

`outsd` (OUTput String Double Word) – вывести цепочку двойных слов в порт ввода-вывода.

К цепочечным командам нужно отнести и префиксы повторения:

`rep`

`repe` или `repz`

`repne` или `repnz`

Префиксы повторения указываются перед нужной цепочечной командой в поле метки. Цепочечная команда без префикса выполняется один раз. Размещение префикса перед цепочечной командой заставляет ее выполняться в цикле. Отличия префиксов в том, на каком основании принимается решение о циклическом выполнении цепочечной команды: по состоянию регистра ecx/cx или по флагу нуля zf:

– префикс повторения rep (REPeat) используется с командами, реализующими операции-примитивы пересылки и сохранения элементов цепочек movs и stos. Префикс rep заставляет данные команды выполняться, пока содержимое в ecx/cx не станет равным 0. При этом цепочечная команда, перед которой стоит префикс, автоматически уменьшает содержимое ecx/cx на единицу. Та же команда, но без префикса, этого не делает;

– префиксы повторения repe или repz (REPeat while Equal or Zero) являются синонимами. Цепочечная команда выполняется до тех пор, пока содержимое ecx/cx не равно нулю или флаг zf равен 1. Как только одно из этих условий нарушается, управление передается следующей команде программы. Благодаря возможности анализа флага zf наиболее эффективно эти префиксы можно использовать с командами cmps и scas для поиска отличающихся элементов цепочек;

– префиксы повторения repne или repnz (REPeat while Not Equal or Zero) также являются синонимами. Префиксы repne/repnz заставляют цепочечную команду циклически выполняться до тех пор, пока содержимое ecx/cx не равно нулю или флаг zf равен нулю. При невыполнении одного из этих условий работа команды прекращается. Данные префиксы также можно использовать с командами cmps и scas, но для поиска совпадающих элементов цепочек.

Особенность формирования физического адреса операндов адрес_источника и адрес_приемника. Цепочка-источник, адресуемая операндом адрес_источника, может находиться в текущем сегменте данных, определяемом регистром ds. Цепочка-приемник, адресуемая операндом адрес_приемника, должна быть в дополнительном сегменте данных, адресуемом сегментным регистром es. Допускается замена (с помощью префикса замены сегмента) только регистра ds, регистр es заменять нельзя. Вторые части адресов – смещения цепочек должны находиться:

– для цепочки-источника это регистр esi/si (Source Index register – индексный регистр источника);

– для цепочки-получателя это регистр edi/di (Destination Index register – индексный регистр приемника). Таким образом, полные физические адреса для операндов цепочечных команд следующие:

адрес_источника – пара ds:esi/si;

адрес_приемника – пара es:edi/di.

Команды lds и les позволяют получить полный указатель (сегмент:смещение) на ячейку памяти.

Семь групп команд, реализующих цепочечные операции-примитивы, имеют похожий по структуре набор команд. В каждом из этих наборов присутствуют одна команда с явным указанием операндов и три команды, не имеющие операндов. На самом деле набор команд микропроцессора имеет соответствующие машинные команды только для цепочечных команд ассемблера без операндов. Команды с операндами транслятор ассемблера использует только для определения типов операндов. После того как выяснен тип элементов цепочек по их описанию в памяти, генерируется одна из трех машинных команд для каждой из цепочечных операций. Поэтому все регистры, содержащие адреса цепочек, должны быть инициализированы заранее, в том числе и для команд, допускающих явное указание операндов. Так как цепочки адресуются однозначно, нет особого смысла применять команды с операндами. Главное – правильная загрузка регистров указателями.

Есть две возможности задания направления обработки цепочки: от начала цепочки к её концу, то сеть в направлении возрастания адресов; от конца цепочки к началу, то есть в направлении убывания адресов.

Направление определяется значением флага направления df (Direction Flag) в регистре eflags/flags:

– если df = 0, то значения индексных регистров esi/si и edi/di будут автоматически увеличиваться (операция инкремента) цепочечными командами, то есть обработка будет осуществляться в направлении возрастания адресов;

– если df = 1, то значения индексных регистров esi/si и edi/di будут автоматически уменьшаться (операция декремента) цепочечными командами, то есть обработка будет идти в направлении убывания адресов.

Состоянием флага df можно управлять с помощью двух команд, не имеющих операндов:

cld (Clear Direction Flag) – очистить флаг направления. Команда сбрасывает флаг направления df в 0.

std (Set Direction Flag) – установить флаг направления. Команда устанавливает флаг направления df в 1.

2.1 Пересылка цепочек

Формат команды movs:

```
movs адрес_приемника,адрес_источника
```

Команда копирует байт, слово или двойное слово из цепочки, адресуемой операндом адрес_источника, в цепочку, адресуемую операндом адрес_приемника.

Команда movs пересылает только один элемент, исходя из его типа, и модифицирует значения регистров esi/si и edi/di. Если перед командой написать префикс rep, то одной командой можно переслать несколько элементов данных. Число пересылаемых элементов должно быть загружено в счетчик – регистры cx или ecx.

Порядок пересылки последовательности элементов из одной области памяти в другую с помощью команды movs. Этот набор действий можно рассматривать как типовой для выполнения любой цепочечной команды:

1. Установить значение флага df в зависимости от того, в каком направлении будут обрабатываться элементы цепочки – в направлении возрастания или убывания адресов.

2. Загрузить указатели на адреса цепочек в памяти в пары регистров ds:(e)si и es:(e)di.

3. Загрузить в регистр ecx/cx количество элементов, подлежащих обработке.

4. Записать команду movs с префиксом rep.

Пример 2.1. Пересылка символов из одной строки в другую. Строки находятся в одном сегменте памяти. Для пересылки используется команда-примитив movs с префиксом повторения rep.

```
masm
model small
.data
  source db 'Тестируемая строка$' ;строка-источник
  dest db 20 dup (' ')            ;строка-приёмник
.stack
  db 256 dup (0)
.code
  assume ds:@data,es:@data
  main proc near
  mov ax,@data   ;загрузка сегментных регистров
  mov ds,ax      ;настройка регистров ds и es
  mov es,ax      ;на адрес сегмента данных
  cld       ;сброс флага df – обработка строки от начала к концу
  lea si,source ;загрузка в si смещения строки-источника
  lea di,dest    ;загрузка в ds смещения строки-приёмника
  mov cx,20 ;для префикса rep счетчик повторений (длина строки)
rep movs dest,source ;пересылка строки
  lea dx,dest
  mov ah,09h     ;вывод на экран строки-приёмника
  int 21h
  mov ax,4c00h
  int 21h
```

```
main endp
end main
```

Пересылка байт, слов и двойных слов производится командами movsb, movsw и movsd. Единственной отличительной особенностью этих команд от команды movs является то, что последняя может работать с элементами цепочек любого размера – 8, 16 или 32 бита. При трансляции команда movs преобразуется в одну из трех команд: movsb, movsw или movsd. В какую конкретно команду будет произведено преобразование, определяет транслятор исходя из размеров элементов цепочек, адреса которых указаны в качестве операндов команды movs. Адреса цепочек для любой из четырех команд должны формироваться заранее в регистрах esi/si и edi/di.

При использовании команды movsb изменится только строка с командой пересылки:

```
lea si,source ;загрузка в si смещения строки-источника
lea di,dest   ;загрузка в ds смещения строки-приёмника
mov cx,20 ;для префикса rep – счетчик повторений (длина строки)
rep movsb     ;пересылка строки
```

Отличие в том, что программа из примера 2.1 может работать с цепочками элементов любой из трех размерностей: 8, 16 или 32 бита, а последний фрагмент – только с цепочками байтов.

2.2 Сравнение цепочек

Формат команды cmps:

```
cmps адрес_приемника,адрес_ источника
```

адрес_источника определяет цепочку-источник в сегменте данных. Адрес цепочки должен быть заранее загружен в пару ds:esi/si;

адрес_приемника определяет цепочку-приемник. Цепочка должна находиться в дополнительном сегменте, и ее адрес должен быть заранее загружен в пару es:edi/di.

Алгоритм работы команды cmps заключается в последовательном выполнении вычитания (элемент цепочки-источника – элемент цепочки-получателя) над очередными элементами обеих цепочек. Команда cmps производит вычитание элементов, не записывая при этом результат, и устанавливает флаги zf, sf и of. После выполнения вычитания очередных элементов цепочек командой cmps индексные регистры esi/si и edi/di автоматически изменяются в соответствии со значением флага df на значение, равное размеру элемента сравниваемых цепочек. Чтобы команда cmps выполнялась несколько раз, то есть производилось последовательное сравнение элементов цепочек, необходимо перед командой cmps определить префикс повторения.

С командой cmps можно использовать префиксы повторения repe/repz или repne/repnz:

– repe или repz – если необходимо организовать сравнение до тех пор, пока не будет выполнено одно из двух условий: достигнут конец цепочки (содержимое ecx/cx равно нулю); в цепочках встретились разные элементы (флаг zf стал равен нулю);

– repne или repnz – если нужно проводить сравнение до тех пор, пока не будет достигнут конец цепочки (содержимое ecx/cx равно нулю); в цепочках встретились одинаковые элементы (флаг zf стал равен единице).

Вместе с командой сравнения используется команда условного перехода jcxz. Ее работа заключается в анализе содержимого регистра ecx/cx, и если он равен нулю, то управление передается на метку, указанную в качестве операнда jcxz. Так как в регистре ecx/cx содержится счетчик повторений для цепочечной команды, имеющей любой из префиксов повторения, то, анализируя ecx/cx, можно определить причину выхода из зацикливания цепочечной команды. Если значение в ecx/cx не равно нулю, то это означает, что выход произошел по причине совпадения либо несовпадения очередных элементов цепочек.

В таблице 2.1 представлены команды условной передачи управления, которые используются с командой сравнения cmps (для чисел без знака).

Таблица 2.1 – Сочетание команд условной передачи управления с результатами команды cmps (для чисел без знака)

Причина прекращения операции сравнения	Команда условного перехода, реализующая переход
операнд_источник > операнд_приемника	ja
операнд_источник = операнду_приемнику	jc
операнд_источник <> операнду_приемнику	jnc
операнд_источник < операнда_приемника	jb
операнд_источник <= операнда_приемника	jbe
операнд_источник >= операнда_приемника	jac

Как определить местоположение очередных совпавших или несовпавших элементов в цепочках? После каждой итерации цепочечная команда автоматически осуществляет инкремент/декремент значения адреса в соответствующих индексных регистрах. Поэтому после выхода из цикла в этих регистрах будут находиться адреса элементов, находящихся в цепочке после (!) элементов, которые послужили причиной выхода из цикла. Для получения истинного адреса этих элементов

необходимо скорректировать содержимое индексных регистров, увеличив либо уменьшив значение в них на длину элемента цепочки.

Пример 2.2. Программа, которая сравнивает две строки, находящиеся в одном сегменте.

```
masm
model small
.data
  match db 0ah,0dh,'Строки совпадают$'
  failed db 0ah,0dh,'Строки не совпадают$'
  string1 db '0123456789',0ah,0dh,'$';исследуемые строки
  string2 db '0123406789$'
.stack
  db 256 dup (0)
.code
  assume ds:@data,es:@data ;привязка ds и es к сегменту данных
  main proc near
  mov ax,@data    ;загрузка сегментных регистров
  mov ds,ax
  mov es,ax       ;настройка es на ds
;вывод на экран исходных строк string1 и string2
  mov ah,09h
  lea dx,string1
  int 21h
  lea dx,string2
  int 21h
;сброс флага df – сравнение в направлении возрастания адресов
  cld
  lea si,string1  ;загрузка в si смещения string1
  lea di,string2  ;загрузка в di смещения string2
  mov cx,10       ;длина строки для префикса repe
;сравнение строк (пока сравниваемые элементы строк равны)
;выход при обнаружении не совпавшего элемента
  repe cmps string1,string2
  jcxz equal      ;cx = 0, то есть строки совпадают
  jne not_match   ;если не равны – переход на not_match
equal:            ;иначе, если совпадают, то
  mov ah,09h      ;вывод сообщения
  lea dx,match
  int 21h
  jmp exit        ;выход
not_match:        ;не совпали
  mov ah,09h
  lea dx,failed
```

```
  int 21h              ;вывод сообщения
;теперь, чтобы обработать не совпавший элемент в строке,
;необходимо  уменьшить значения регистров si  и  di
  dec si  ;скорректировали адреса очередных элементов для получения
  dec di  ;адресов несовпавших элементов
;сейчас в ds:si и es:di адреса несовпавших элементов
;здесь вставить код по обработке несовпавшего элемента
;после этого продолжить поиск в строке:
  inc si
;для просмотра оставшейся части строк необходимо установить
;указатели на следующие элементы строк за последними
;несовпавшими. После этого можно повторить весь процесс просмотра
;и обработки несовпавших элементов в оставшихся частях строк
  inc di
exit:              ;выход
  mov ax,4c00h
  int 21h
main endp
end main
```

Если сравниваются цепочки с элементами слов или двойных слов, то корректировать содержимое esi/si и edi/di нужно на 2 и 4 байта, соответственно.

2.3. Сканирование цепочек

Команды производят поиск некоторого значения в области памяти. Логически эта область памяти рассматривается как последовательность (цепочка) элементов фиксированной длины размером 8, 16 или 32 бита. Искомое значение предварительно должно быть помещено в регистр al/ax/eax. Выбор конкретного регистра из этих трех должен быть согласован с размером элементов цепочки, в которой осуществляется поиск.

Формат команды scas:

```
scas адрес_приемника
```

Команда имеет один операнд, обозначающий местонахождение цепочки в дополнительном сегменте (адрес цепочки должен быть заранее сформирован в es:edi/di). Транслятор анализирует тип идентификатора адрес_приемника, который обозначает цепочку в сегменте данных и формирует одну из трех машинных команд, scasb, scasw или scasd. Условие поиска для каждой из этих трех команд находится в строго определенном месте: если цепочка описана с помощью директивы db, то искомый элемент должен быть байтом и находиться в al, а сканирование цепочки осуществляется командой scasb; если

цепочка описана с помощью директивы dw, то это – слово в ax, и поиск ведется командой scasw; если цепочка описана с помощью директивы dd, то это двойное слово в eax, и поиск ведется командой scasd. Принцип поиска тот же, что и в команде сравнения cmps, то есть последовательное выполнение вычитания (содержимое_регистра_аккумулятора – содержимое_очередного_элемента_цепочки). В зависимости от результатов вычитания производится установка флагов, при этом сами операнды не изменяются. С командой scas удобно использовать префиксы repe/repz или repne/repnz:

– repe или repz – если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий: достигнут конец цепочки (содержимое ecx/cx равно 0); в цепочке встретился элемент, отличный от элемента в регистре al/ax/eax;

– repne или repnz – если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий: достигнут конец цепочки (содержимое ecx/cx равно 0); в цепочке встретился элемент, совпадающий с элементом в регистре al/ax/eax.

Таким образом, команда scas с префиксом repe/repz позволяет найти элемент цепочки, отличающийся по значению от заданного в аккумуляторе. Команда scas с префиксом repne/repnz позволяет найти элемент цепочки, совпадающий по значению с элементом в аккумуляторе.

Пример 2.3. Поиск символа в строке.

```
masm
model small
.data
  find   db 0ah,0dh,'Символ найден! ','$'
  nofind db 0ah,0dh,'Символ не найден.','$'
  mes db 'String for search.',0ah,0dh,'$'
  nom dw ? ;номер совпавшего символа должен быть 13₁₀ = D₁₆
.stack
  db 256 dup(0)
.code
  assume ds:@data,es:@data
  main proc near
  mov ax,@data
  mov ds,ax
  mov es,ax    ;настройка es на ds
  mov ah,09h
  lea dx,mes
  int 21h      ;вывод сообщения mes
  mov al,'a'   ;символ для поиска
  cld          ;сброс флага df
```

```
  lea di,mes    ;загрузка в es:di смещения строки
  mov si,di     ;запоминаем адрес начала строки
  mov cx,18     ;для префикса repne длина строки
;поиск в строке (пока искомый символ и символ в строке не совпадут)
;выход при первом совпадении
  repne scas mes
  je found      ;если равны – переход на обработку,
  mov ah,09h    ;вывод сообщения о том, что символ не найден
  lea dx,nofind
  int 21h       ;вывод сообщения nofind
  jmp exit      ;переход на завершение программы
found:          ;совпали
  mov ah,09h
  lea dx,find
  int 21h       ;вывод сообщения  find
;чтобы узнать место, где совпал элемент в строке,
;необходимо уменьшить значение в регистре  di
  dec di
  sub di,si     ;находим номер совпавшего символа
  mov nom,di
exit: mov ax,4c00h
  int 21h
main endp
end main
```

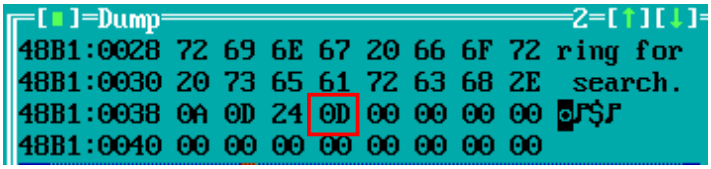

Рисунок 2.2 – Результат работы программы

2.4 Загрузка элемента цепочки в аккумулятор

Эта операция-примитив позволяет извлечь элемент цепочки и поместить его в регистр-аккумулятор `al`, `ax` или `eax`. Эту операцию удобно использовать вместе с поиском (сканированием) с тем, чтобы, найдя нужный элемент, извлечь его (например, для изменения). Размер извлекаемого элемента определяется применяемой командой.

Формат команды `lods`:

`lods адрес_источника` (LOaD String) – загрузить элемент из цепочки в аккумулятор `al`/`ax`/`eax`.

Команда имеет один операнд, обозначающий строку в основном сегменте данных. Работа команды заключается в том, чтобы извлечь

элемент из цепочки по адресу, соответствующему содержимому пары регистров ds:esi/si, и поместить его в регистр eax/ax/al. При этом содержимое esi/si подвергается инкременту или декременту (в зависимости от состояния флага df) на значение, равное размеру элемента. Эту команду удобно использовать после команды scas, локализующей местоположение искомого элемента в цепочке.

Пример 2.4. Программа сравнивает командой cmps две цепочки байт в памяти string1 и string2 и помешает первый несовпавший байт из string2 в регистр al, номер несовпавшего символа – в переменную nom. Для загрузки этого байта в регистр-аккумулятор al используется команда lods. Префикса повторения в команде lods нет, так как он просто не нужен.

```
masm
model small
.data
  string1 db 'String one.',0ah,0dh,'$'
  string2 db 'String two.',0ah,0dh,'$'
  mes_eq  db 'The strings are equal',0ah,0dh,'$'
 find db 'Not equal in the al register',0ah,0dh,'$'
  nom dw ? ;номер несовпавшего элемента
.stack
  db 256 dup (0)
.code
  assume ds:@data,es:@data
  main proc near
  mov ax,@data;загрузка сегментных регистров
  mov ds,ax
  mov es,ax    ;настройка es на ds
  mov ah,09h
  lea dx,string1
  int 21h      ;вывод string1
  lea dx,string2
  int 21h      ;вывод string2
  cld          ;сброс флага df
  lea di,string1 ;загрузка в es:di смещения строки string1
  lea si,string2 ;загрузка в ds:si смещения строки string2
  mov nom,si
  mov cx,11    ;для префикса repe – длина строки
;поиск в строке (пока нужный символ и символ в строке не равны)
;выход – при первом несовпавшем
  repe cmps string1,string2
  jcxz eql     ;если равны – переход на eql
  jmp no_eq    ;если не равны – переход на no_eq
eql: mov ah,09h    ;выводим сообщение о совпадении строк
```

```
   lea dx,mes_eq
   int 21h       ;вывод сообщения mes_eq
   jmp exit      ;переход на конец
no_eq: mov ah,09h ;обработка несовпадения элементов
   lea dx,find
   int 21h       ;вывод сообщения fnd
;чтобы извлечь несовпавший элемент из строки
;в регистр-аккумулятор, уменьшаем значение регистра si  и тем самым
;перемещаемся к действительной позиции элемента в строке
   dec si        ;команда lods использует ds:si-адресацию
                 ;ds:si  указывает на позицию в string2
   lods string2 ;загрузим элемент из строки в  al,  это символ  't'
   sub si,nom
   dec si
   mov nom,si   ;номер несовпавшего символа = 7
exit: mov ax,4c00h
   int 21h
main endp
end main
```

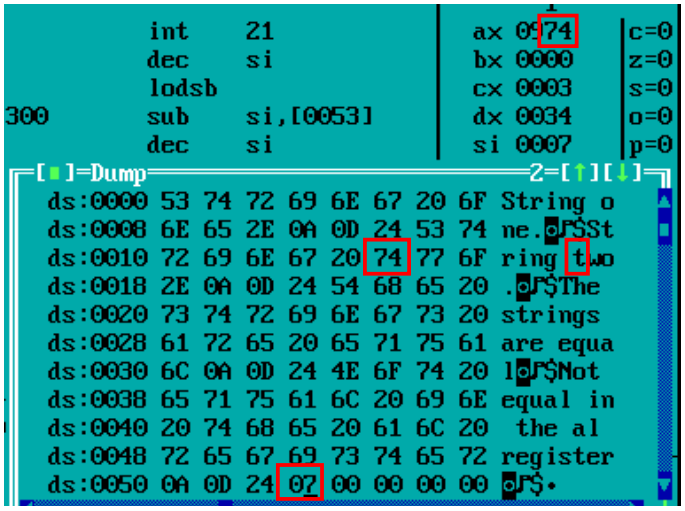

Рисунок 2.3 – Результат выполнения программы

2.5 Перенос элемента из аккумулятора в цепочку

Эта операция-примитив позволяет сохранить значение из регистра-аккумулятора в элементе цепочки. Операцию удобно использовать вместе с операциями поиска (сканирования) `scans` и загрузки `lods` с тем, чтобы, найдя нужный элемент, извлечь его в регистр и записать на его место новое значение.

Формат команды `stos`:

`stos адрес_приемника` (STOrage String) – сохранить элемент из регистра-аккумулятора `al`/`ax`/`eax` в цепочке.

Команда имеет один операнд адрес_приемника, адресующий цепочку в дополнительном сегменте данных. Команда пересылает элемент из аккумулятора (регистра eax/ax/al) в элемент цепочки по адресу, соответствующему содержимому пары регистров es:edi/di. При этом содержимое edi/di подвергается инкременту или декременту (в зависимости от состояния флага df) на значение, равное размеру элемента цепочки.

Пример 2.5. Программа производит замену в строке всех символов ‘a’ на другой символ, вводимый с клавиатуры.

```
masm
model small
.data
  find db 'Character is found',0ah,0dh,'$'
  nochar db 'Character not found',0ah,0dh,'$'
  mes1 db 'Source string:',0ah,0dh,'$'
  string db 'Search character in this string',
            0ah,0dh,'$' ;строка для поиска
  mes2 db 'Enter character-->$'
  mes3 db 0ah,0dh,'New string: ',0ah,0dh,'$'
.stack
  db 256 dup(0)
.code
  assume ds:@data,es:@data   ;привязка ds и es
                             ;к сегменту данных
  main proc near
  mov ax,@data     ;загрузка сегментных регистров
  mov ds,ax
  mov es,ax        ;настройка es на ds
        mov ax,0002h      ;очистка экрана
        int 10h
  mov ah,09h
  lea dx,mes1
  int 21h  ;вывод сообщения mes1
  lea dx,string
  int 21h      ;вывод string
  mov al,'a'   ;символ для поиска
  cld          ;сброс флага df
  lea di,string   ;загрузка в di смещения string
  mov cx,31    ;для префикса repne - длина строки
;поиск в строке string до тех пор, пока символ в al и очередной
;символ в строке не равны: выход – при первом совпадении
repne scas string
  je found ;если элемент найден, то переход на found
```

```
                ;иначе, если не найден, то вывод сообщения nochar
  mov ah,09h
  lea dx,nochar
  int 21h
  jmp exit ;переход на выход
found: mov ah,09h
  lea dx,find
  int 21h       ;вывод сообщения об обнаружении символа
  mov ah,09h    ;ввод символа с клавиатуры
  lea dx,mes2
  int 21h       ;вывод сообщения mes2
  mov ah,01h
  int 21h       ;в al - введённый символ
  mov bl,al     ;сохраняем введенный символ
;устанавливаем начальные данные для замены
  cld           ;сброс флага df
  lea di,string    ;загрузка в di смещения string
  mov cx,31        ;для префикса repne - длина строки
;поиск в строке string до тех пор, пока символ в al и очередной
;символ в строке не равны: выход – при первом совпадении
cycl: mov al,'a'  ;символ для поиска
  repne scas string
  jne exit  ;если элемент не найден, переход на конец программы
;корректируем di для получения значения действительной позиции
;совпавшего элемента в строке и регистре al
  dec di
  new_char:     ;блок замены символа
  mov al,bl
  stos string ;сохраним введённый символ (из al) в строке
                ;string в позиции старого символа
;переход на поиск следующего символа 'a' в строке
  inc di        ;указатель в строке string на следующий,
                ;после совпавшего, символ
  jmp cycl      ;на продолжение просмотра string
exit: mov ah,09h
  lea dx,mes3
  int 21h       ;вывод сообщения mes3
  lea dx,string
  int 21h       ;вывод сообщения string
  mov ax,4c00h
  int 21h
main endp
end main
```

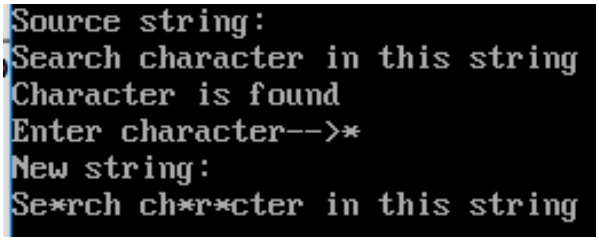

Рисунок 2.4 – Результат работы программы

2.6 Ввод элемента цепочки из порта ввода-вывода

Данная операция вводит цепочки элементов из порта ввода-вывода и реализуется командой `ins`, имеющей следующий формат:

`ins адрес_приемника,номер_порта` (Input String) – ввести элементы из порта ввода-вывода в цепочку.

Команда вводит элемент из порта, номер которого находится в регистре `dx`, в элемент цепочки, адрес которого определяется операндом адрес_приемника. Адрес цепочки должен быть явно сформирован в паре регистров `es:edi/ di`. Размер элементов цепочки должен быть согласован с размерностью порта; он определяется директивой резервирования памяти, с помощью которой выделяется память для размещения элементов цепочки. После пересылки команда `ins` производит коррекцию содержимого `edi/di` на величину, равную размеру элемента, участвовавшего в операции пересылки, при этом учитывается состояние флага `df`.

2.7 Вывод элемента цепочки в порт ввода-вывода

Данная операция выводит элементов цепочки в порт ввода-вывода. Она реализуется командой `outs`, имеющей следующий формат:

`outs номер_порта, адрес_источника` (OUTput String) – вывести элементы из цепочки в порт ввода-вывода.

Эта команда выводит элемент цепочки в порт, номер которого находится в регистре `dx`. Адрес элемента цепочки определяется операндом адрес_источника. Адрес цепочки должен быть явно сформирован в паре регистров `ds:esi/si`. Размер структурных элементов цепочки должен быть согласован с размерностью порта. Он определяется директивой резервирования памяти, с помощью которой выделяется память для размещения элементов цепочки. После пересылки команда `outs` производит коррекцию содержимого `esi/si` на величину, равную размеру элемента цепочки, участвовавшего в операции пересылки, при этом учитывается состояние флага `df`.

2.8 Практическое задание

Написать и отладить программу на языке Assembler по обработке строк. Исходная строка задается в программе. Вывести на экран исходную и результирующую строки.

Варианты заданий:

1. Заменить в строке все пробелы знаками подчеркивания.
2. Определить количество букв в первом слове.
3. Заменить в строке символы '*' на '+'.
4. Заменить первую букву каждого слова на символ '*'.
5. Заменить последнюю букву каждого слова на символ '*'.
6. Определить количество букв во втором слове.
7. Определить количество пробелов в строке.
8. Определить количество букв в последнем слове.
9. Определить количество букв 'a' в строке.
10. Определить количество слов, начинающихся с буквы 'a'.
11. Определить количество слов, заканчивающихся буквой 's'.
12. Символ, стоящий после буквы 'a' заменить на '+'.
13. Символ, стоящий перед буквой 'a' заменить на '–'.
14. Определить количество букв 'o' в последнем слове.
15. Определить количество букв 'e' в последнем слове.
16. Заменить вторую букву каждого слова символом '+'.
17. Определить количество символов 'f', стоящих после '+'
18. Определить количество символов 'f', стоящих перед '–'.
19. Определить количество слов, которые заканчиваются буквой 'e'.
20. Определить количество слов, которые начинаются буквой 'e'.
21. Распечатать первое слово строки.
22. Определить количество цифр '1' в строке.
23. Определить количество букв 'e' в первом слове.
24. Определить количество запятых в строке.
25. Заменить цифру '1' знаком подчеркивания.
26. Определить количество символов ',', стоящих перед пробелом.
27. Определить, сколько букв 'o' стоят после пробела.
28. Определить, сколько букв 'i' стоят перед пробелом.
29. Заменить все пробелы запятыми.
30. Распечатать последнее слово строки.

Литература

1. Гук, М. Процессоры Pentium III, Athlon и другие / М. Гук, В. Юров. – СПб. : Питер, 2000. – 379 с.

2. Зубков, С. В. Ассемблер для DOS, Windows и UNIX / С. В. Зубков. – М. : ДМК Пресс, 2000. – 534 с.

3. Программирование на языке ассемблера для персональных ЭВМ : учебное пособие / А. Ф. Каморников [и др.]. – Гомель : ГГУ, 1995. – 95 с.

4. Пустоваров, В. И. Ассемблер : программирование и анализ корректности машинных программ / В. И. Пустоваров. – К. : Издательская группа BHV, 2000. – 480 с.

5. Сван, Т. Освоение Turbo Assembler / Т. Сван. – Киев : Диалектика, 1996. – 540 с.

6. Юров, В. Assembler / В. Юров. – СПб. : Питер, 2001. – 624 с.

7. Юров, В. Assembler : практикум / В. Юров. – СПб. : Питер, 2002. – 400 с.

8. Юров, В. Assembler : специальный справочник / В. Юров. – СПб. : Питер, 2000. – 496 с.

Производственно-практическое издание

Ружицкая Елена Адольфовна
Кузьменкова Екатерина Юрьевна

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ASSEMBLER

BCD-числа, цепочечные команды

Практическое пособие

Редактор *В. И. Шкредова*
Корректор *В. В. Калугина*

Подписано в печать 22.02.2017. Формат 60x84 1/16.
Бумага офсетная. Ризография. Усл. печ. л. 2,8.
Уч-изд. л. 3,1. Тираж 25 экз. Заказ 142.

Издатель и полиграфическое исполнение:
учреждение образования
«Гомельский государственный университет
имени Франциска Скорины».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 1/87 от 18.11.2013.
Специальное разрешение (лицензия) № 02330 / 450 от 18.12.2013.
Ул. Советская, 104, 246019, Гомель.