Лабораторная работа №3

Исследование работы команд пересылки данных

В данной лабораторной работе мы познакомимся с командами обмена данными. Эти команды осуществляют пересылку данных из одного места в другое, запись и чтение информации из портов ввода-вывода, преобразование информации, манипуляции с адресами и указателями, обращение к стеку. Для некоторых из этих команд операция пересылки является только частью алгоритма. Другая его часть выполняет некоторые дополнительные операции над пересылаемой информацией.

Все команды в соответствии с их функциональным назначением делятся на команды:

- пересылки данных;
- ввода-вывода в порт;
- работы с адресами и указателями;
- преобразования данных;
- работы со стеком.

В работе не будет подробно рассмотрена только группа команд работы со стеком, так как этому будет посвящена отдельная лабораторная работа.

Цель работы

Ознакомление с группой команд микропроцессора относящихся к командам пересылки данных:

- пересылки данных общего назначения;
- ввода-вывода в порт;
- работы с адресами и указателями;
- преобразования данных.

Практическое применение данных команд при написании программ.

Исследовать возможности команд пересылки данных. В частности влияние на состояние флагов.

Узнать, какие команды обмена данными нельзя выполнять. Ознакомиться с методами адресации данных.

Теоретическая часть

3.2.1 Команды пересылки данных общего назначения

К этой группе относятся следующие команды:

mov <операнд назначения>,<операнд-источник>

xchg <операнд1>,<операнд2>

mov - это основная команда пересылки данных. Она реализует самые разнообразные варианты пересылки.

Отметим особенности применения этой команды:

- командой mov нельзя осуществить 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.

К примеру, рассмотрим фрагмент программы для пересылки байта из ячейки fls в ячейку fld:

```
masm
model small
.data
fls db 5
fld db ?
.code
start:
...
        mov al,fls
        mov fld,al
...
end start
```

- нельзя загрузить в сегментный регистр значение непосредственно из памяти. Поэтому для выполнения такой загрузки нужно использовать

промежуточный объект. Это может быть регистр общего назначения или стек. Если вы посмотрите листинги программ в первой лабораторной работе, то увидите в начале сегмента кода две команды mov, выполняющие настройку сегментного регистра ds. При этом из-за невозможности загрузить впрямую в сегментный регистр значение адреса сегмента, содержащееся в предопределенной переменной @data, приходится использовать регистр общего назначения ax;

- нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных все те же регистры общего назначения. Вот пример инициализации регистра es значением из регистра ds:

```
mov ax,ds
mov es,ax
```

- Но есть и другой, более красивый способ выполнения данной операции — использование стека и команд push и pop:

```
push ds        ;поместить значение регистра ds в стек
pop es         ;записать в es число из стека
```

- нельзя использовать сегментный регистр cs в качестве операнда назначения. Причина здесь простая. Дело 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.

Для двунаправленной пересылки данных применяют команду xchg. Для этой операции можно, конечно, применить последовательность из нескольких команд mov, но из-за того, что операция обмена используется довольно часто, разработчики системы команд микропроцессора посчитали нужным ввести отдельную команду обмена xchg. Естественно, что операнды должны иметь один тип. Не допускается (как и для всех команд ассемблера) обменивать между

собой содержимое двух ячеек памяти. К примеру,

```
xchg ax,bx              ;обменять содержимое регистров ax и bx
xchg ax,word ptr [si]   ;обменять содержимое регистра ax
                        ;и слова в памяти по адресу в [si]
```

3.2.2 Команды ввода-вывода в порт

Операнды машинной команды могут находиться, в **портах ввода- вывода.**

На рис. 3.1 показана сильно упрощенная, концептуальная схема управления оборудованием компьютера.

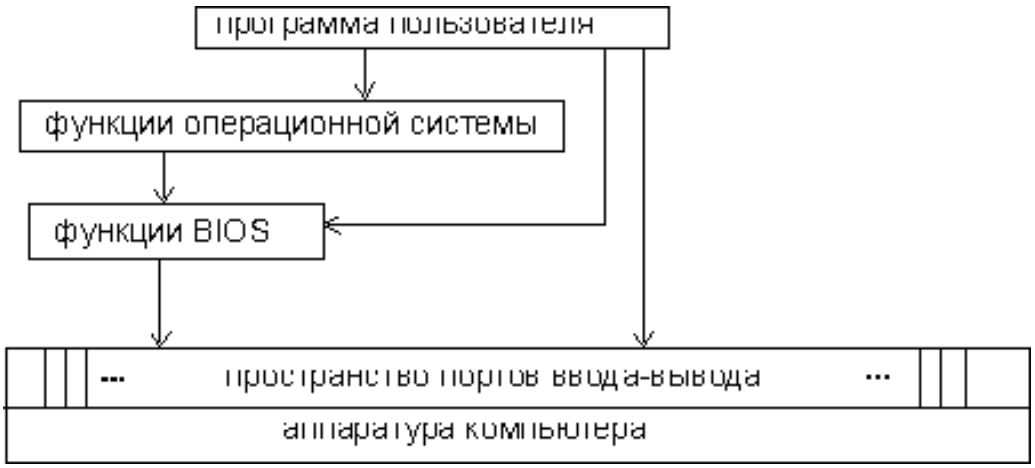

Рисунок 3.1 – Концептуальная схема управления оборудованием компьютера

Как видно из рис. 3.1, самым нижним уровнем является уровень BIOS, на котором работа с оборудованием ведется напрямую через порты. Тем самым реализуется концепция независимости от оборудования. При замене оборудования необходимо будет лишь подправить соответствующие функции BIOS, переориентировав их на новые адреса и логику работы портов.

Принципиально управлять устройствами напрямую через порты несложно. Сведения о номерах портов, их разрядности, формате управляющей информации приводятся в техническом описании устройства. Необходимо знать лишь конечную цель своих действий, алгоритм, в соответствии с которым работает конкретное устройство, и порядок программирования его портов. То есть, фактически, нужно знать, что и в какой последовательности нужно послать в порт (при записи в него) или считать из него (при чтении) и как

следует трактовать эту информацию. Для этого достаточно всего двух команд, присутствующих в системе команд микропроцессора:

in аккумулятор,номер_порта — ввод в аккумулятор из порта с номером номер_порта;

out порт,аккумулятор — вывод содержимого аккумулятора в порт с номером номер_порта.

```
;---------Prg_7_1.asm---------------
;Программа, имитирующая звук сирены.
;Изменение высоты звука от 450 Гц до 2100 Гц.
;Используется макрос delay (задержка).
;При необходимости
;можно поменять значение задержки (по умолчанию - для процессора Pentium).
;-----------------------------------
masm
model small
stack  100h
delay macro time
local  ext,iter
;макрос задержки, его текст ограничивается директивами macro и endm.
;На входе - значение задержки (в мкс)

        push   cx
        mov    cx,time
ext:
        push   cx
        mov    cx,5000
iter:
        loop   iter
        pop    cx
        loop   ext
        pop    cx
endm
.data                       ;сегмент данных
tonelow      dw     2651    ;нижняя граница звучания = 450 Гц
cnt    db     0             ;счётчик для выхода из программы
temp   dw     ?             ;верхняя граница звучания
.code                       ;сегмент кода
main:                       ;точка входа в программу
        mov    ax,@data     ;связываем регистр ds с сегментом
        mov    ds,ax        ;данных через регистр ax
        xor    ax,ax        ;очищаем ax
go:
;заносим слово состояния 10110110b(0B6h) в командный регистр (порт 43h)
        mov    al,0B6h
```

```
        out     43h,al
        in      al,61h          ;получим значение порта 61h в al
        or      al,3            ;инициализируем динамик и подаем ток в порт 61h
        out     61h,al
        mov     cx,2083         ;количество шагов ступенчатого изменения тона
musicup:
                                ;в ax значение нижней границы частоты
        mov     ax,tonelow
        out     42h,al          ;в порт 42h младшее слово ax :al
        xchg    al,ah           ;обмен между al и ah
        out     42h,al          ;в порт 42h старшее слово ax:ah
        add     tonelow,1       ;повышаем тон
        delay 1                 ;задержка на 1 мкс
        mov     dx,tonelow      ;в dx текущее значение высоты
        mov     temp,dx         ;temp - верхнее значение высоты
        loop    musicup         ;повторить цикл повышения
        mov     cx,2083         ; восстановить счетчик цикла
musicdown:
        mov     ax,temp         ;в ax верхнее значение высоты
        out     42h,al          ;в порт 42h младшее слово ax :al
        mov     al,ah           ;обмен между al и ah
        out     42h,al          ;в порт 42h старшее слово ax :ah
        sub     temp,1          ;понижаем высоту
        delay 1                 ;задержка на 1 мкс
        loop musicdown          ;повторить цикл понижения
nosound:
        in      al,61h          ;получим значение порта 61h в AL
        and     al,0FCh         ;выключить динамик
        out     61h,al          ;в порт 61h
        mov     dx,2651         ;для последующих циклов
        mov     tonelow,dx
        inc     cnt             ;увеличиваем счётчик проходов, то есть
                                ;количество звучаний сирены
        cmp     cnt,5           ;5 раз ?
        jne     go              ;если нет, идти на метку go
        exit:
        mov     ax,4c00h        ;стандартный выход
        int     21h
end main                        ;конец программы
```

3.2.3 Команды работы с адресами и указателями памяти

При написании программ на ассемблере производится интенсивная работа с адресами операндов, находящимися в памяти. Для поддержки такого рода операций есть специальная группа команд, в которую входят следующие команды:

lea назначение,источник — загрузка эффективного адреса;

lds назначение,источник — загрузка указателя в регистр сегмента данных ds;

les назначение,источник — загрузка указателя в регистр дополнительного сегмента данных es;

lgs назначение,источник — загрузка указателя в регистр дополнительного сегмента данных gs;

lfs назначение,источник — загрузка указателя в регистр дополнительного сегмента данных fs;

lss назначение,источник — загрузка указателя в регистр сегмента стека ss.

Команда lea похожа на команду mov тем, что она также производит пересылку. Однако, обратите внимание, команда lea производит пересылку не данных, а эффективного адреса данных (то есть смещения данных относительно начала сегмента данных) в регистр, указанный операндом назначение.

Часто для выполнения некоторых действий в программе недостаточно знать значение одного лишь эффективного адреса данных, а необходимо иметь полный указатель на данные. Вы помните, что полный указатель на данные состоит из сегментной составляющей и смещения.

Все остальные команды этой группы позволяют получить в паре регистров такой полный указатель на операнд в памяти. При этом ***имя сегментного регистра***, в который помещается сегментная составляющая адреса, определяется кодом операции.

Соответственно, ***смещение*** помещается в регистр общего назначения, указанный операндом ***назначение.***

Но не все так просто с операндом ***источник***. На самом деле, в команде в качестве источника нельзя указывать непосредственно имя операнда в памяти, на который мы бы хотели получить указатель.

Предварительно необходимо получить само значение полного указателя в некоторой области памяти и указать в команде получения полного адреса имя этой области. Для выполнения этого действия необходимо вспомнить директивы резервирования и инициализации памяти.

При применении этих директив возможен частный случай, когда в поле операндов указывается имя другой директивы определения данных (фактически, имя переменной). В этом случае в памяти формируется адрес этой переменной. Какой адрес будет сформирован (эффективный или полный), зависит от применяемой директивы. Если это dw, то в памяти формируется только 16-битное значение эффективного адреса, если же dd — в память записывается полный адрес. Размещение этого адреса в памяти следующее: в младшем слове находится смещение, в старшем — 16-битная сегментная составляющая адреса.

Например, при организации работы с цепочкой символов удобно поместить ее начальный адрес в некоторый регистр и далее в цикле модифицировать это значение для последовательного доступа к элементам цепочки. В листинге 1 производится копирование строки байт str_1 в строку байт str_2.

В строках 12 и 13 в регистры si и di загружаются значения эффективных адресов переменных str_1 и str_2.

В строках 16 и 17 производится пересылка очередного байта из одной строки в другую. Указатели на позиции байтов в строках определяются содержимым регистров si и di. Для пересылки очередного байта необходимо увеличить на единицу регистры si и di, что и делается командами сложения inc (строки 18, 19). После этого программу необходимо зациклить до обработки всех символов строки.

Листинг: Копирование строки

```
<1>;---------Prg_7_2.asm---------------
<2> masm
<3> model small
<4> .data
<5> ...
<6> str_1 db 'Ассемблер — базовый язык компьютера'
<7> str_2 db 50 dup (' ')
<8> full_pnt dd str_1
<9> ...
<10> .code
<11> start:
<12> ...
<13> lea si,str_1
<14> lea di,str_2
<15> les bx,full_pnt        ;полный указатель на str1 в пару es:bx
<16> m1:
<17> mov al,[si]
<18> mov [di],al
<19> inc si
<20> inc di
<21>                ;цикл на метку m1 до пересылки всех символов
<22> ...
<23> end start
```

Необходимость использования команд получения полного указателя данных в памяти, то есть адреса сегмента и значения смещения внутри сегмента, возникает, в частности, при работе с цепочками.

В строке 15 листинга 1 в двойном слове full_pnt формируются сегментная часть адреса и смещение для переменной str_1. При этом 2 байта смещения занимают младшее слово full_pnt, а значение сегментной составляющей адреса — старшее слово full_pnt. В строке 15 командой les эти компоненты адреса помещаются в регистры bx и es.

Команду LEA часто используют для быстрых арифметических

вычислений, например умножения:

```
lea bx,[ebx+ebx*4]        ;BX=EBX*5
```

или сложения:

```
lea ebx,[eax+12]          ;EBX=EAX+12
```

(эти команды меньше, чем соответствующие MOV и ADD, и не изменяют флаги).

3.2.4 Команды преобразования данных

К этой группе можно отнести множество команд микропроцессора, но большинство из них имеют те или иные особенности, которые требуют отнести их к другим функциональным группам.

Поэтому из всей совокупности команд микропроцессора непосредственно к командам преобразования данных можно отнести только одну команду:

xlat [адрес_таблицы_перекодировки]

Это очень интересная и полезная команда. Ее действие заключается в том, что она замещает значение в регистре al другим байтом из таблицы в памяти, расположенной по адресу, указанному операндом адрес_таблицы_перекодировки.

Слово «таблица» весьма условно — по сути это просто строка байт. Адрес байта в строке, которым будет производиться замещение содержимого регистра al, определяется суммой (bx) + (al), то есть содержимое al выполняет роль индекса в байтовом массиве.

При работе с командой xlat обратите внимание на следующий тонкий момент. Несмотря на то, что в команде указывается адрес строки байт, из которой должно быть извлечено новое значение, этот адрес должен быть предварительно загружен (например, с помощью команды lea) в регистр bx. Таким образом, операнд адрес_таблицы_перекодировки на самом деле не нужен (необязательность операнда показана заключением его в квадратные скобки). Что касается строки байт (таблицы перекодировки), то она представляет собой область памяти размером от 1 до 255 байт (диапазон числа

без знака в 8-битном регистре).

В качестве иллюстрации работы данной команды мы рассмотрим программу, которая преобразует двузначное шестнадцатеричное число, вводимое с клавиатуры (то есть в символьном виде), в эквивалентное двоичное представление в регистре al. Ниже приведен вариант этой программы с использованием команды xlat.

Листинг: Использование таблицы перекодировки

```
<1>;---------Prg_7_3.asm----------------------
<2>;Программа преобразования двузначного шестнадцатеричного числа
<3>;в двоичное представление с использованием команды xlat.
<4>;Вход: исходное шестнадцатеричное число; вводится с клавиатуры.
<5>;Выход: результат преобразования в регистре al.
<6>.data                    ;сегмент данных
<7> message db 'Введите две шестнадцатеричные цифры,$'
<8> tabl db 48 dup (0),0,1,2,3,4,5,6,7,8,9, 8 dup (0),
<9> db 0ah,0bh,0ch,odh,0eh,0fh,27 dup (0)
<10> db 0ah,0bh,0ch,odh,0eh,0fh, 153 dup (0)
<11> .stack 256             ;сегмент стека
<12> .code
<13>                        ;начало сегмента кода
<14> proc main              ;начало процедуры main
<15>        mov ax,@data    ;физический адрес сегмента данных в регистр ax
<16>        mov ds,ax       ;ax записываем в ds
<17>        lea bx,tabl     ;загрузка адреса строки байт в регистр bx
<18>        mov ah,9
<19>        mov dx,offset message
<20>        int 21h         ;вывести приглашение к вводу
<21>        xor ax,ax       ;очистить регистр ax
<22>        mov ah,1h       ;значение 1h в регистр ah
<23>        int 21h         ;вводим первую цифру в al
<24>        xlat            ;перекодировка первого введенного символа в al
<25>        mov dl,al
<26>        mov cl,4
```

```
<27>        shl dl,cl    ;сдвиг dl влево для освобождения места для младшей цифры
<28>        int 21h      ;ввод второго символа в al
<29>        xlat         ;перекодировка второго введенного символа в al
<30>        add al,dl    ;складываем для получения результата
<31>        mov ax,4c00h ;пересылка 4c00h в регистр ax
<32>        int 21h      ;завершение программы
<33> endp main           ;конец процедуры main
<34> code ends           ;конец сегмента кода
<35> endmain             ;конец программы с точкой входа main
```

Сама по себе программа проста; сложность вызывает обычно формирование таблицы перекодировки. Рассмотрим подробнее.

Прежде всего нужно определиться с значениями тех байтов, которые вы будете изменять. В нашем случае это символы шестнадцатеричных цифр. Сконструируем в сегменте данных таблицу, в которой на места байтов, соответствующих символам шестнадцатеричных цифр, помещаем их новые значения, то есть двоичные эквиваленты шестнадцатеричных цифр. Строки 8-10 листинга демонстрируют, как это сделать. Байты этой таблицы, смещения которых не совпадают со значением кодов шестнадцатеричных цифр, нулевые. Таковыми являются первые 48 байт таблицы, промежуточные байты и часть в конце таблицы.

Желательно определить все 256 байт таблицы. Дело в том, что если мы ошибочно поместим в al код символа, отличный от символа шестнадцатеричной цифры, то после выполнения команды xlat получим непредсказуемый результат. В случае листинга 2 это будет ноль, что не совсем корректно, так как непонятно, что же в действительности было в al — код символа «0» или что-то другое.

Поэтому, наверное, есть смысл здесь поставить «защиту от дурака», поместив в неиспользуемые байты таблицы какой-нибудь определенный символ. После каждого выполнения xlat нужно будет просто контролировать значение в al на предмет совпадения с этим символом, и если оно произошло, выдавать сообщение об ошибке.

После того как таблица составлена, с ней можно работать. В сегменте

команд строка 18 инициализирует регистр bx значением адреса таблицы tabl. Далее все очень просто. Поочередно вводятся символы двух шестнадцатеричных цифр, и производится их перекодировка в соответствующие двоичные эквиваленты.

Задание на лабораторную работу

1. Создайте программы типа .exe из представленных в теоретической части, выполните программы пошагово, используя отладчик TD.
2. Напишите программу, используя команды пересылки данных общего назначения. Кроме того, программа должна выводить сообщение на экран «Лабораторную работу 3 выполнил студент Булахова А. В.»:

2.1 Непосредственная адресация

Выполнить непосредственную адресацию десятичного числа в регистр AX полученного из выражения:

2*10, где N – номер по списку в журнале занятий.

Например: номер по списку 23, тогда число, которое необходимо записать равно 23*10=230,

MOV AX,230D.

2.2 Непосредственная адресация

Выполнить непосредственную адресацию двоичного числа в регистр BX полученного из выражения:

2*5, где N – номер по списку в журнале занятий.

Например: номер по списку 23, тогда число, которое необходимо записать равно 23*5=115, переведем это число в двоичное используя калькулятор из набора стандартных программ. Установите вид калькулятора Инженерный. Вводим десятичное число (при установке системы Dec), затем переводим это число в двоичное, используя Bin.

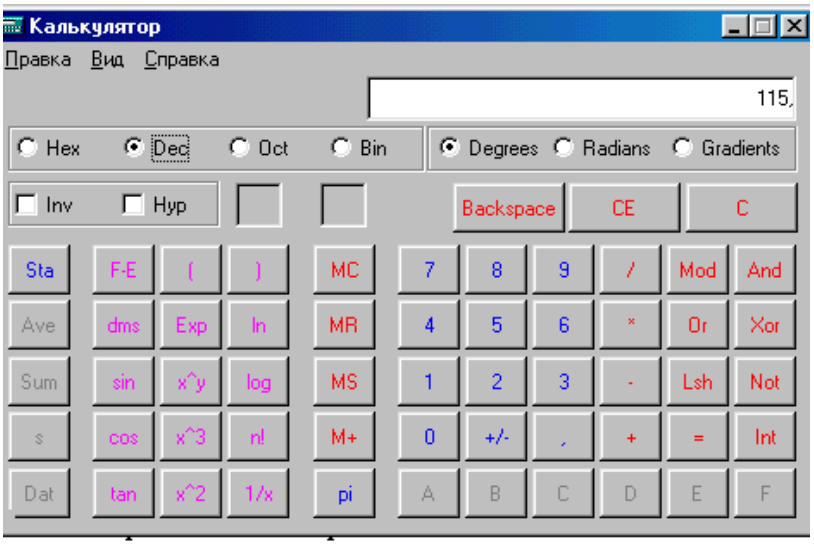

Рисунок 3.2 – Ввод десятичного числа

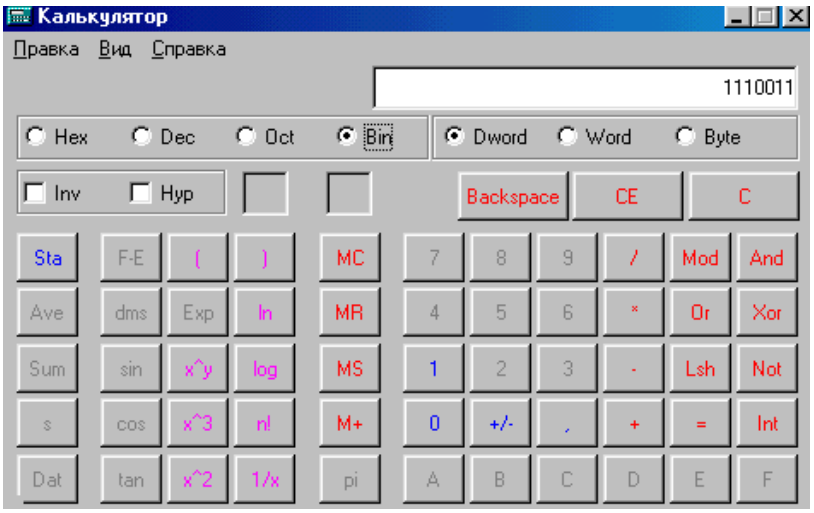

Рисунок 3.3 Преобразование десятичного числа в двоичное

MOV BX,1110011b.

2.3 Регистровая адресация

Выполнить регистровую адресацию из регистра BX в регистр AX.

Например: MOV AX,BX

2.4 Непосредственная адресация

Выполнить непосредственную адресацию восьмеричного числа в регистр

DX полученного из выражения:

2*2, где N – номер по списку в журнале занятий.

Например: номер по списку 23, тогда число, которое необходимо записать равно 23*2=46, переведем это число в восьмеричное используя калькулятор из набора стандартных программ. Установите вид калькулятора Инженерный. Вводим десятичное число (при установке системы Dec), затем переводим это число в восьмеричное, используя Oct.

MOV DX,56O

2.5 Прямая адресация

Выполнить прямую адресацию слова, находящегося в сегменте, на который указывает ES, со смещением от начала сегмента 2*5.

Например: Если номер по списку 7, то ES:0023 и команда MOV AX,ES:0023 поместит это слово в регистр AX.

2.6 Косвенная адресация

Выполнить косвенную адресацию слова, находящегося в сегменте, на который указывает BX.

Например: следующая команда помещает в регистр AX слово из ячейки памяти, селектор сегмента которой находится в DS, а смещение — в BX:

MOV AX,[BX].

2.7 Адресация по базе со сдвигом

Выполнить адресацию по базе со сдвигом, находящегося в сегменте, на который указывает BX со сдвигом 2, где N-номер по списку в журнале занятий.

Например: номер по списку 23

```
mov  ax,[bx+23]
```

помещает в регистр AX слово, находящееся в сегменте, указанном в DS, со смещением на 23 большим, чем число, находящееся в BX.

2.8 Двунаправленная пересылка данных

Выполнить двунаправленную пересылку данных из регистра AX в BX.

Например: для этого применяют команду XCHG.

```
XCHG AX,BX
```

2.9 Пересылка из порта в процессор, из процессора в порт, пересылка в стек и извлечение из стека

```
in    al,60h        ;читаем скан-код
push  ax            ;сохраним его на время
in    al,61h        ;читаем порт 61h
or    al,80h        ;старший бит байта из порта 61h в 1
out   61h,al        ;подтверждаем факт приема скан-кода
pop   ax
out   61h,al        ;восстановили байт в порту 61h
```

3. Результаты работы программ и пошаговое выполнение осуществляем в среде Turbo Debugger. Записываем состояние регистров после каждого шага, кроме тех которые не меняют свое состояние.

Содержание отчета

1. В отчете указать название и порядковый номер лабораторной работы, сформулировать цель работы.
2. Указать имена и типы разработанных программ.
3. Представить листинги всех программ.
4. Распечатать изображения с экрана при выполнении программы.
5. Представить состояние содержимого регистров и ячеек памяти по мере пошагового выполнения программы. Указывать состояние изменившихся регистров, ячеек памяти и флагов состояния процессора.
6. Сделать выводы по каждой группе команд пересылки данных.

Лабораторная работа №4

Исследование работы команд сложения микропроцессора

В данной лабораторной работе мы познакомимся с командами сложения. Любой компьютер, от самого примитивного до супермощного, имеет в своей системе команд команды для выполнения арифметических действий. Работая с компьютером при помощи языков высокого уровня, мы воспринимаем возможность проведения расчетных действий как нечто должное, забывая при этом, что компилятор даже очень развитого языка программирования превращает все самые высокоуровневые действия в унылую последовательность машинных команд. Конечно, мало кому придет в голову писать серьезную расчетную задачу на ассемблере. Но даже в системных программах часто требуется проведение небольших вычислений. Поэтому важно разобраться с этой группой команд. К тому же она, на удивление, очень компактна и не избыточна.

Процессор может выполнять целочисленные операции и операции с плавающей точкой. Для этого в его архитектуре есть два отдельных устройства, каждое из которых имеет свою систему команд. В принципе, целочисленное устройство может взять на себя многие функции устройства с плавающей точкой, но это потребует больших вычислительных затрат. Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.

Цель работы

Ознакомление с группой команд микропроцессора относящихся к командам сложения. Изучить команды:

- двоичной арифметики
    - сложение add, adc, inc;
    - изменение знака neg.
- десятичной арифметики
    - коррекция сложения aaa, daa.

Исследование работы команд целочисленного сложения микропроцессора.

Получить практические навыки по использованию перечисленных команд, научиться работать с двоично-десятичными числами.

Теоретическая часть

Группа арифметических целочисленных команд работает с двумя типами чисел:

4. целыми двоичными числами, которые могут иметь или не иметь знаковый разряд, то есть быть числами со знаком или без знака;
5. целыми десятичными числами.

Рассмотрим форматы данных, с которыми работают арифметические команды.

4.2.1 Целые двоичные числа

Целое двоичное число — это число, закодированное в двоичной системе счисления. В архитектуре IA-32 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. При этом интересно то, что среди арифметических команд есть всего две, которые действительно учитывают этот старший разряд как знаковый, — это команды целочисленного умножения IMUL и деления 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 (табл. 1).

Таблица 4.1 – Диапазон значений двоичных чисел

Размерность поля	Целое без знака	Целое 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

Как описать целые двоичные числа в программе? Это делается с использованием директив описания данных DB, DW и DD. К примеру, последовательность описаний двоичных чисел из сегмента данных листинга 1 (помните о принципе «младший байт по младшему адресу») будет выглядеть в памяти так, как показано на рис. 4.1.

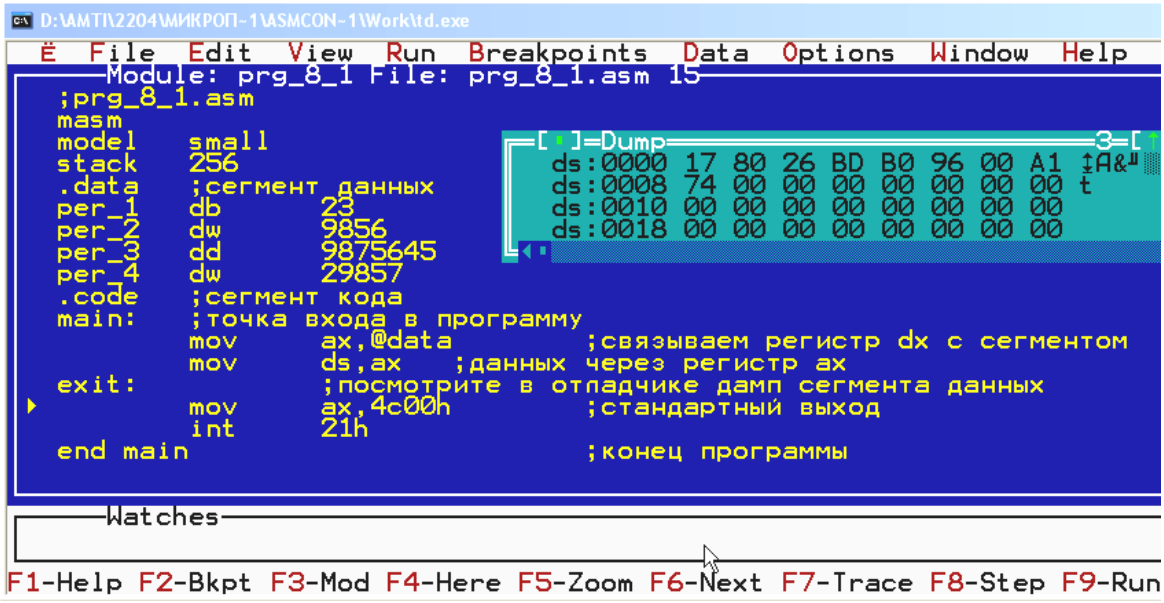

Рисунок 4.1 – Дамп памяти для сегмента данных листинга 8.1

Листинг 8.1. Числа с фиксированной точкой

```
;prg_8_1.asm
masm
model      small
stack 256
.data ;сегмент данных
per_1      db   23
per_2      dw   9856
per_3      dd   9875645
per_4      dw   29857
.code;сегмент кода
```

```
main:     ;точка входа в программу
     mov  ax,@data  ;связываем регистр dx с сегментом
     mov  ds,ax;данных через регистр ax
exit:     ;посмотрите в отладчике дамп сегмента данных
     mov  ax,4c00h  ;стандартный выход
     int  21h
end main            ;конец программы
```

4.2.2 Десятичные числа

Десятичные числа — специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырех битов. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом ***двоично-десятичном коде*** (Binary-Coded Decimal, BCD). Процессор хранит BCD-числа в двух форматах (рис. 2).

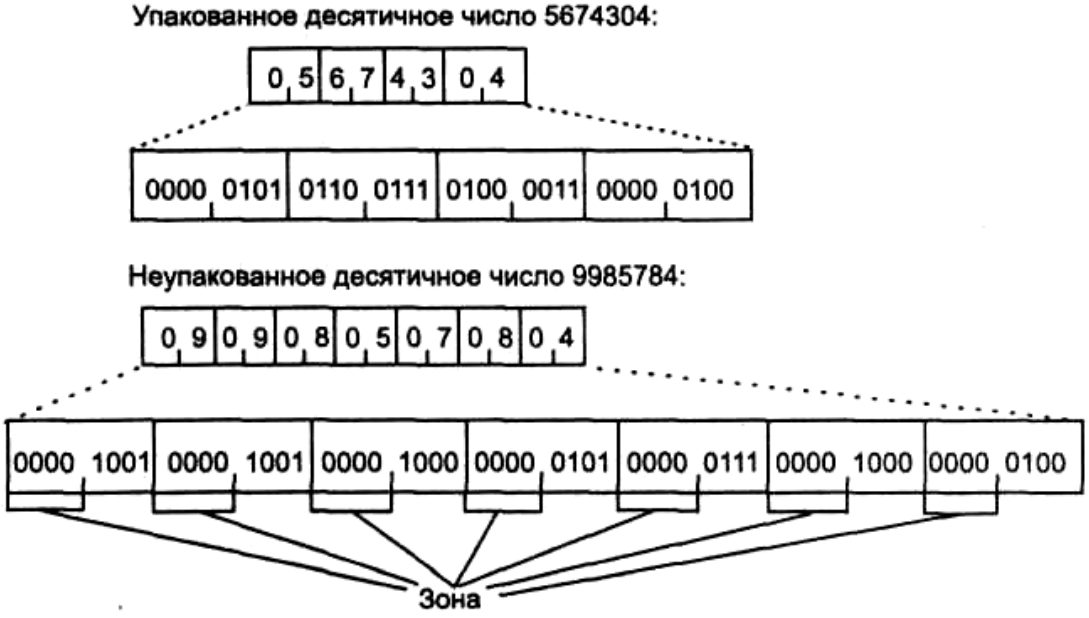

Рисунок 4.2 – Представление BCD-чисел

- В ***упакованном формате*** каждый байт содержит две десятичные цифры. Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 размером четыре бита. При этом код старшей цифры числа занимает старшие четыре бита. Следовательно, диапазон представления десятичного

упакованного числа в одном байте составляет от 00 до 99.

- В ***неупакованном формате*** 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.

Как описать двоично-десятичные числа в программе? Для этого можно использовать только две директивы описания и инициализации данных — DB и DT. Возможность применения только этих директив для описания BCD-чисел обусловлена тем, что к таким числам также применим принцип «младший байт по младшему адресу», что, как мы увидим далее, очень удобно для их обработки. И вообще, при использовании такого типа данных, как BCD-числа, порядок описания этих чисел в программе и алгоритм их обработки — это дело вкуса и личных пристрастий программиста, что станет более ясным после того, как мы далее рассмотрим основы работы с BCD-числами. К примеру, приведенная в сегменте данных листинга 8.2 последовательность описаний BCD-чисел будет выглядеть в памяти так, как показано на рис. 3.

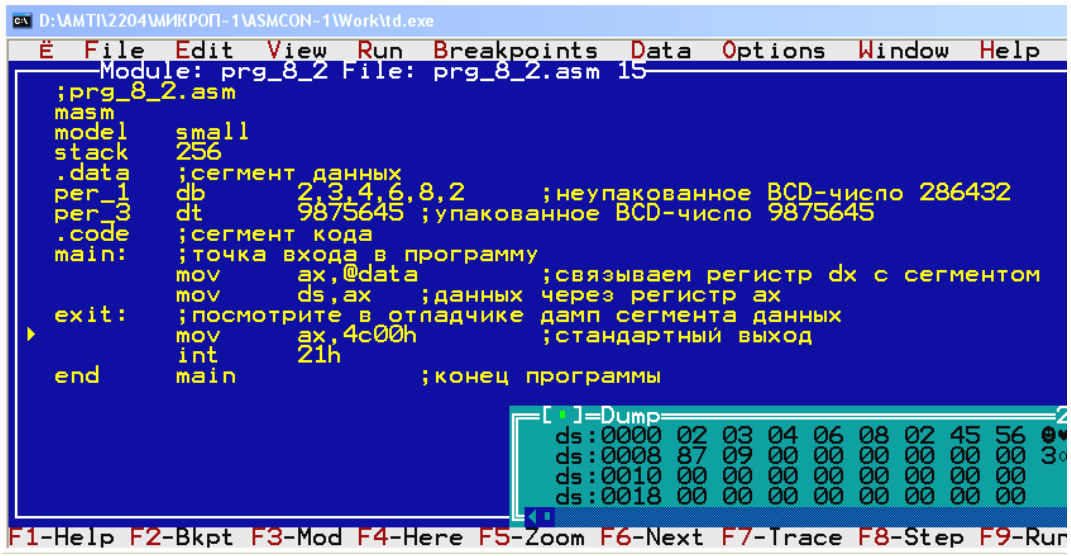

Рисунок 4.3 – Дамп памяти для сегмента данных листинга 8.2

Листинг 8.2. BCD-числа

```
;prg_8_2.asm
masm
model      small
```

```
stack 256
.data ;сегмент данных
per_1     db   2,3,4,6,8,2     ;неупакованное BCD-число 286432
per_3     dt   9875645         ;упакованное BCD-число 9875645
.code                          ;сегмент кода
main:                          ;точка входа в программу
     mov  ax,@data             ;связываем регистр dx с сегментом
     mov  ds,ax                ;данных через регистр ax
exit:                ;посмотрите в отладчике дамп сегмента данных
     mov  ax,4c00h             ;стандартный выход
     int  21h
end  main                      ;конец программы
```

После обсуждения форматов данных, с которыми работают арифметические операции, можно приступить к рассмотрению средств их обработки на уровне системы команд процессора.

4.2.3 Арифметические операции над целыми двоичными числами

Рассмотрим особенности каждого из двух основных арифметических действий для целых двоичных чисел со знаком и без знака.

4.2.3.1 Сложение двоичных чисел без знака

Процессор выполняет сложение операндов по правилам сложения двоичных чисел. Проблем не возникает до тех пор, пока значение результата не превышает размерности поля операнда (см. табл. 1). Например, при сложении операндов размером в байт результат не должен превышать 255. Если это происходит, то результат оказывается неверен. Рассмотрим, почему. К примеру, выполним сложение: 254 + 5 = 259 в двоичном виде:

11111110 + 0000101 = 1 00000011.

Результат вышел за пределы восьми битов, и правильное его значение укладывается в 9 битов, а в 8-разрядном поле операнда осталось значение 3, что, конечно, неверно. В процессоре подобный исход сложения прогнозируется, и предусмотрены специальные средства для фиксации подобных ситуаций и их

обработки. Так, для фиксации 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. Естественно, что программист должен предусматривать возможность такого исхода операции сложения и средства для корректировки. Это предполагает включение фрагментов кода после операции сложения, в которых анализируется состояние флага CF. Этот анализ можно провести различными способами. Самый простой и доступный — использовать команду условного перехода JC. Эта команда в качестве операнда имеет имя метки в текущем сегменте кода. Переход на метку осуществляется в случае, если в результате работы предыдущей команды флаг CF установился в 1.

В системе команд процессора имеются три команды двоичного сложения:

- команда ***инкремента***, то есть увеличения значения операнда на 1:

  ```
  inc операнд
  ```

- команда сложения (операнд_1 = операнд_1 + операнд_2):

  ```
  add операнд_1,операнд_2
  ```

- команда сложения с учетом флага переноса CF (операнд_1 = операнд_1 + операнд_2 + значение_CF):

  ```
  adc операнд_1,операнд_2
  ```

Обратите внимание на последнюю команду — это команда сложения, учитывающая перенос единицы из старшего разряда. Механизм появления такой единицы мы уже рассмотрели. Таким образом, команда ADC является средством процессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые процессором размеры стандартных полей.

Рассмотрим пример вычисления суммы чисел (листинг 8.3).

Листинг 8.3. Вычисление суммы чисел

```
<1>  ;prg_8_3.asm
<2>  masm
```

```
<3>  model small
<4>  stack  256
<5>  .data
<6>  a      db     254
<7>  .code                          ;сегмент кода
<8>  main:
<9>         mov    ax,@data
<10>        mov    ds,ax
<11>        ;...
<12>        xor    ax,ax
<13>        add    al,17
<14>        add    al,a
<15>        jnc    m1           ;если нет переноса, то перейти на m1
<16>        adc    ah,0         ;в ax сумма с учетом переноса
<17> m1:    ;...
<18>        exit:
<19>        mov    ax,4c00h     ;стандартный выход
<20>        int    21h
<21> end main                    ;конец программы
```

В строках 13-14 создана ситуация, когда результат сложения выходит за границы операнда. Эта возможность учитывается строкой 15, где команда JNC (хотя можно было обойтись и без нее) проверяет состояние флага CF. Если он установлен в 1, то результат операции получился большим по размеру, чем операнд, и для его корректировки необходимо выполнить некоторые действия. В данном случае мы просто полагаем, что границы операнда расширяются до размера AX, для чего учитываем перенос в старший разряд командой ADC (строка 16). Исследовать работу команд сложения без учета знака вы можете в отладчике. Для этого введите в текстовом редакторе текст листинга 8.3, получите исполняемый модуль, запустите отладчик и откройте в нем окна командами View ► Dump и View ► Registers. Далее, в пошаговом режиме отладки можно более наглядно проследить за всеми процессами,

происходящими в процессоре во время работы программы.

4.2.3.2 Сложение двоичных чисел со знаком

Процессор не подозревает о различии между числами со знаком и числами без знака. Вместо этого у него есть средства фиксации возникновения характерных ситуаций, складывающихся в процессе вычислений. Некоторые из них мы рассмотрели ранее при обсуждении команд сложения чисел без знака — это учет флага переноса CF. Установка этого флага в 1 говорит о том, что произошел выход за пределы разрядности операндов. Далее с помощью команды ADC можно учесть возможность такого выхода (переноса из младшего разряда) во время работы программы. Другое средство фиксации характерных ситуаций в процессе арифметических вычислений — регистрация состояния старшего (знакового) разряда операнда, которое осуществляется с помощью флага переполнения OF в регистре EFLAGS (бит 11).

Положительные числа представляются в двоичном коде, а отрицательные — в дополнительном. Рассмотрим различные варианты сложения чисел. Примеры призваны показать поведение двух старших битов операндов и правильность результата операции сложения.

Первый вариант сложения чисел:

30566 = 0111011101100110

+

00687 = 0000001010101111

=

31253 = 01111010 00010101.

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

30566 = 0111 0111 0110 0110

+

30566 = 0111 0111 0110 0110

=

61132 = 1110 1110 1100 1100.

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-разрядное число со знаком (+32 767).

Третий вариант сложения чисел:

-30566 = 1000 1000 1001 1010

+

-04875 = 1110 1100 1111 0101

=

-35441 = 0111 0101 1000 1111.

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).

Четвертый вариант сложения чисел:

-4875 = 1110 1100 1111 0101

+

-4875 = 1110 1100 1111 0101

-9750 = 1101 1001 1110 1010.

Есть переносы из 14-го и 15-го разрядов. Результат правильный.

Таким образом, мы исследовали все случаи и выяснили, что 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).

И, наоборот, переполнения не происходит (то есть флаг OF сбрасывается в 0), если есть перенос из обоих разрядов или перенос отсутствует в обоих разрядах.

Таким образом, ситуация переполнение регистрируется процессором с помощью флага переполнения OF. Дополнительно к флагу OF при переносе из

старшего разряда устанавливается в 1 и флаг переноса CF. Так как процессор не знает о существовании чисел со знаком и без знака, то вся ответственность за правильность действий с получившимися числами ложится на программиста. Теперь, наверное, понятно, почему мы столько внимания уделили тонкостям сложения чисел со знаком. Учтя все это, мы сможем организовать правильный процесс сложения чисел — будем анализировать флаги CF и OF и принимать правильное решение! Проанализировать флаги CF и OF можно командами условного перехода JC\JNC и JO\JNO соответственно.

Что же касается команд сложения чисел со знаком, то из изложенного ранее понятно, что в архитектуре IA-32 сами команды сложения чисел со знаком те же, что и для чисел без знака.

4.2.4 Арифметические операции над двоично-десятичными числами

BCD-числа нужны в коммерческих приложениях, то есть там, где числа должны быть большими и точными. Как мы уже убедились, выполнение операций с двоичными числами для языка ассемблера довольно проблематично:

- Значения величин в формате слова и двойного слова имеют ограниченный диапазон. Если программа предназначена для работы в области финансов, то ограничение суммы в рублях величиной 65 536 (для слова) или даже 4 294 967 296 (для двойного слова) существенно сужает сферу ее применения (да еще в наших экономических условиях — тут уж никакая деноминация не поможет).
- Двоичные числа дают ошибки округления. Представляете себе программу, работающую где-нибудь в банке, которая не учитывает величину остатка при действиях с целыми двоичными числами и оперирует при этом миллиардами. Не хотелось бы быть автором такой программы. Применение чисел с плавающей точкой не спасет — там существует та же проблема округления.
- Большой объем результатов в коммерческих программах требуется представлять в символьном виде (ASCII-коде). Коммерческие программы не

просто выполняют вычисления; одной из целей их применения является оперативная выдача информации пользователю. Для этого, естественно, информация должна быть представлена в символьном виде. Перевод чисел из двоичного кода в ASCII-код, как мы уже видели, требует определенных вычислительных затрат. Число с плавающей точкой еще труднее перевести в символьный вид. А вот если посмотреть на шестнадцатеричное представление неупакованной десятичной цифры (в начале данной главы) и на соответствующий ей символ в таблице ASCII, то видно, что они различаются на величину 30h. Таким образом, преобразование в символьный вид и обратно получается намного проще и быстрее.

Эти доводы убеждают в важности овладения хотя бы основами действий с десятичными числами. Далее рассмотрим особенности выполнения основных арифметических операций с такими числами. Для предупреждения возможных вопросов сразу отметим тот факт, что отдельных команд сложения, вычитания, умножения и деления BCD-чисел нет. Сделано это по вполне понятным причинам — размерность таких чисел может быть сколь угодно большой. 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.

4.2.4.1 Неупакованные BCD-числа

Сложение

Рассмотрим два случая сложения. Результат сложения не больше 9:

6 = 0000 0110

+

3 = 0000 0011

=

9 = 0000 1001.

Переноса из младшей тетрады в старшую нет. Результат правильный.

Результат сложения больше 9:

6 = 0000 0110

+

7 = 0000 0111

=

13 = 00001101.

То есть мы получили уже не BCD-число. Результат неправильный. Правильный результат в неупакованном BCD-формате в двоичном представлении должен быть таким: 0000 0001 0000 0011 (или 13 в десятичном). Проанализировав данную проблему при сложении BCD-чисел (и подобные проблемы при выполнении других арифметических действий), а также возможные пути ее решения, разработчики системы команд процессора решили не вводить специальные команды для работы с BCD-числами, а ввести несколько корректировочных команд. Назначение этих команд — корректировка результата работы обычных арифметических команд для случаев, когда операнды в них являются BCD-числами. В случае сложения во втором примере видно, что полученный результат нужно корректировать. Для коррекции операции сложения двух однозначных неупакованных BCD-чисел и представления результата сложения в символьном виде в системе команд процессора существует специальная команда AAA (ASCII Adjust for Addition).

Команда AAA 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.

1. К содержимому младшей тетрады AL (но не к содержимому всего регистра!) прибавляется 6, тем самым значение десятичного результата корректируется в правильную сторону.
2. Флаг CF устанавливается в 1, тем самым фиксируется перенос в старший разряд для того, чтобы его можно было учесть в последующих действиях.

Так, во втором примере сложения, предполагая, что значение суммы 0000 1101 находится в AL, после выполнения команды AAA в регистре будет 1101 +

0110 = 0011, то есть двоичное значение 0000 0011 или десятичное значение 3, а флаг CF установится в 1, то есть перенос запоминается в процессоре. Далее программисту нужно использовать команду сложения ADC, которая учтет перенос из предыдущего разряда. Приведем пример программы сложения двух неупакованных BCD-чисел (листинг 8.8).

Листинг 8.8. Сложение неупакованных BCD-чисел

```
<1>  ;prg_8_8.asm
<2>  masm
<3>  model small
<4>  stack  256
<5>  .data
<6>  len    equ   2       ;разрядность числа
<7>  b      db    1,7     ;неупакованное число 71
<8>  c      db    4,5     ;неупакованное число 54
<9>  sum    db    3 dup (0)
<10> .code
<11> main: ;точка входа в программу
<12>       mov   ax,@data
<13>       mov   ds,ax
<14> ;...
<15>       xor   bx,bx
<16>       mov   cx,len
<17> m1:
<18>       mov   al,b[bx]
<19>       adc   al,c[bx]
<20>       aaa
<21>       mov   sum[bx],al
<22>       inc   bx
<23>       loop  m1
<24>       adc   sum[bx],0
<25> ;...
<26>       exit:
<27>       mov   ax,4c00h      ;стандартный выход
```

```
<28>        int     21h
<29> end main               ;конец программы
```

Начнем с описания BCD-чисел. Из строк 7 и 8 видно, что порядок их ввода обратен нормальному, то есть цифры младших разрядов расположены по меньшему адресу. Это вполне логично по нескольким причинам: во-первых, такой порядок удовлетворяет общему принципу представления данных для процессоров Intel, во-вторых, это очень удобно для поразрядной обработки неупакованных BCD-чисел, так как каждое из них занимает один байт. Хотя, повторюсь, программист сам волен выбирать способ описания BCD-чисел в сегменте данных. Строки 18 и 19 содержат команды, которые складывают цифры в очередных разрядах BCD-чисел, при этом учитывается возможный перенос из младшего разряда. Команда AAA в строке 20 корректирует результат сложения, формируя в AL BCD-цифру и при необходимости устанавливая в 1 флаг CF. Строка 24 учитывает возможность переноса при сложении цифр из самых старших разрядов чисел. Результат сложения формируется в поле sum, описанном в строке 9.

4.2.4.2 Упакованные BCD-числа

Как уже отмечалось ранее, 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. Таким образом, сами по себе упакованные BCD-числа представляют не слишком большой интерес для программиста, поэтому мы их рассмотрим кратко.

Сложение

Попытаемся сложить два двузначных упакованных BCD-числа:

```
67 = 0110 0111
+
75 = 0111 0101
```

142 = 1101 1100 = 220

Как видим, в двоичном виде результат равен 1101 1100 (или 220 в десятичном представлении), что неверно. Это происходит по той причине, что процессор не подозревает о существовании BCD-чисел и складывает их по правилам сложения двоичных чисел. На самом деле результат в двоично-десятичном виде должен быть равен 0001 0100 0010 (или 142 в десятичном представлении). Этот пример иллюстрирует необходимость корректировки результатов арифметических операций с упакованными BCD-числами, так же как это было в случае неупакованных BCD-чисел. Для корректировки результата сложения упакованных чисел в целях представления его в десятичном виде процессор предоставляет команду DAA (Decimal Adjust for Addition).

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.

Проиллюстрируем сказанное на примере сложения двух двузначных BCD-чисел в упакованном формате (листинг 8.12).

Листинг 8.12. Сложение упакованных BCD-чисел

```
<1> ;prg8_12.asm
<2> masm
<3> model     small
<4> stack     256
<5> .data;сегмент данных
<6> b    db   17h            ;упакованное число 17h
<7> c    db   45h            ;упакованное число 45h
<8> sum  db   2 dup (0)
<9> .code;сегмент кода
<10>     main:     ;точка входа в программу
<11>     mov ax,@data
```

```
<12>      mov  ds,ax
<13>      xor  ax,ax
<14>      mov  al,b
<15>      add  al,c
<16>      daa
<17>      jnc  $+6 ;переход через команду, если результат <= 99
<18>mov sum+1,ah ;учет переноса при сложении (результат > 99)
<19>      mov  sum,al    ;младшие упакованные цифры результата
<20>exit:
<21>      mov  ax,4c00h
<22>      int  21h
<23>      end  main
```

В приведенном примере следует обратить внимание на описание упакованных BCD-чисел и порядок формирования результата. Результат формируется в соответствии с основным принципом работы процессоров Intel: младший байт по младшему адресу.

Задание на лабораторную работу

1 Создайте программы типа .exe, выполняющие сложение чисел указанного размера, двоичных и BCD, со знаком и без, выполните программы пошагово, используя отладчик TD.

2 Варианты заданий соответствуют номеру по журналу занятий:

Таблица 4.2 – Сложение двоичных чисел

	Порядковый номер фамилии обучающегося по списку									
	1	2	3	4	5	6	7	8	9	10
Числа размером 1 байт без знака	127 + 136	**58 + 231**	29 + 98	26 + 223	234 + 67	134 + 68	209 + 54	54 + 198	90 + 200	19 + 128

Числа размером N байт без учета знака	Числа размером 1 байт со знаком
34fcafdch + 78acdfh	- 120 +(- 7)
0af4d57h + 83654adfh	**- 89 + (- 55)**
9aacdfe3h + 0b91dafh	- 60 + (- 101)
758466h + 0af98ffcbh	- 45 + 127
0fdacb9e3h + 5fd98dh	- 100 + (- 100)
1754dah + 0ffad65fdh	125 + (- 45)
1291fdaah + 439fd4h	- 120 + (- 22)
0dfac6799h + 81a8c9h	58 + (- 128)
0bdc3a5f4h + 3459afh	- 19 + (- 99)
0fd87cbh + 0ad9874h	100 + (- 57)

Таблица 4.3 – Сложение двоичных чисел

	Порядковый номер фамилии обучающегося по списку									
	11	12	13	14	15	16	17	18	19	20
Числа размером 1 байт	102 + 145	64 + 95	111 + 155	87 + 255	92 + 245	122 + 47	98 + 180	106 + 155	164 + 88	206 + 99
Числа размером 1 байт со знаком	- 114 + (- 34)	124 + (- 78)	- 90 + (- 24)	88 + (- 93)	- 123 + (- 55)	- 68 + (- 99)	- 20 + (- 79)	- 64 + 127	- 103 + (- 27)	98 + (- 99)
Числа размером N байт без учета знака	0afdc34fch + 98acdfh	0f4d57ah + 654adf83h	93aacdfeh + 0dafb91h	0cbf9aah + 3421dae8h	5346dcabh + 0fc438a3fh	0cb321dh + 534adc49h	876644cdh + 0fa65c3h	0b8723da4h + 7391fbcah	2008a3dbh + 0bc5df5h	7af0129fh + 0dacc44h

Таблица 4.4 Сложение BCD чисел

	Порядковый номер фамилии обучающегося по списку									
	1	**2**	3	4	5	6	7	8	9	10
Неупакованные BCD числа	67 + 45	**64 + 95**	141 + 155	534 + 255	92 + 25	622 + 847	198 + 580	196 + 455	164 + 988	206 + 998
Упакованные BCD числа	1234 + 3299	**9812 + 7546**	3321 + 9087	8765 + 4567	6553 + 1239	4343 + 9100	8090 + 1112	5643 + 9127	9103 + 6531	8798 + 5556

Таблица 4.5 Сложение BCD чисел

	Порядковый номер фамилии обучающегося по списку									
	11	12	13	14	15	16	17	18	19	20
Неупакованные BCD числа	87 + 85	54 + 97	341 + 435	644 + 215	72 + 79	921 + 247	898 + 489	396 + 652	662 + 788	506 + 999
Упакованные BCD числа	5224 + 9232	1298 + 4675	2133 + 8790	6587 + 6745	5365 + 3912	4343 + 9108	9080 + 1211	4356 + 2791	3910 + 3165	9887 + 5655

Содержание отчета

- В отчете указать название и порядковый номер лабораторной работы, сформулировать цель работы.
- Указать имена и типы разработанных программ.
- Представить листинги всех программ.
- Распечатать изображения с экрана при выполнении программы.
- Представить состояние содержимого регистров и ячеек памяти по мере пошагового выполнения программы. Указывать состояние изменившихся регистров, ячеек памяти и флагов состояния процессора.
- Сделать выводы по каждой команде сложения, как двоичных так и BCD-чисел.

Лабораторная работа №5

Исследование работы команд вычитания микропроцессора

В данной лабораторной работе мы познакомимся с командами вычитания. Команды вычитания используются одинаково часто с другими арифметическими командами. Алгоритм работы команд своеобразный. Есть особенности работы с двоично-десятичными числами и числами со знаком.

Цель работы

Ознакомление с группой команд микропроцессора относящихся к командам вычитания. Изучить команды:

- двоичной арифметики
    - вычитание sub, sbb, dec.
- десятичной арифметики
    - коррекция вычитания aas, das.

Исследование работы команд целочисленного вычитания микропроцессора. Выработать навыки написания программ с использованием команд вычитания.

Теоретическая часть

5.2.1 Арифметические операции над целыми двоичными числами

Рассмотрим особенности одного из двух основных арифметических действий, вычитания, для целых двоичных чисел со знаком и без знака.

5.2.1.1 Вычитание двоичных чисел без знака

Аналогично анализу операции сложения, рассмотрим что происходит при выполнении вычитания.

6. Если уменьшаемое больше вычитаемого, то проблем нет, — разность положительна, результат верен.

7. 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. Процессор поступает аналогично, то есть занимает 1 из разряда, следующего за старшим в разрядной сетке операнда. Поясним на примере.

Первый вариант вычитания чисел:

05 = 0000 0000 0000 0101

-10 = 0000 0000 0000 1010.

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

1 0000 0000 0000 0101

-

0000 0000 0000 1010

=

1111 1111 1111 1011.

Тем самым, по сути, выполняется действие (65 536 + 5) - 10 = 65 531, 0 здесь как бы эквивалентен числу 65 536. Результат, конечно, неверен, но процессор считает, что все нормально, тем не менее, факт заема единицы он фиксирует, устанавливая флаг переноса CF. Посмотрите еще раз внимательно на результат операции вычитания. Это же число -5 в дополнительном коде! Проведем эксперимент: представим разность в виде суммы 5 + (-10).

Второй вариант вычитания чисел:

5 = 0000 0000 0000 0101

+

(-10) = 1111 1111 1111 0110

=

1111 1111 1111 1011.

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, это говорит о том, что произошел заем из старшего разряда и результат получился в дополнительном коде.

Аналогично командам сложения группа команд вычитания состоит из минимально возможного набора. Эти команды выполняют вычитание по алгоритмам, которые мы сейчас рассматриваем, а учет особых ситуаций должен производиться самим программистом.

- Команда декремента выполняет уменьшения значения операнда на 1:

  dec операнд

- Команда вычитания (операнд_1 = операнд_1 - операнд_2):

  sub операнд_1,операнд_2

- Команда вычитания с учетом заема, то есть флага CF (операнд_1 = операнд_1 - операнд_2 - значение_CF):

  sbb операнд_1,операнд_2

Как видите, среди команд вычитания есть команда SBB, учитывающая флаг переноса CF. Эта команда подобна ADC, но теперь уже флаг CF играет роль индикатора заема 1 из старшего разряда при вычитании чисел.

Рассмотрим пример (листинг 8.4) программной обработки ситуации, рассмотренной ранее для второго варианта вычитания чисел.

Листинг 8.4. Проверка при вычитании чисел без знака

```
<1>  ;prg_8_4.asm
<2>  masm
<3>  model small
<4>  stack  256
<5>  .data
<6>  .code                    ;сегмент кода
<7>  main:                    ;точка входа в программу
<8>          mov   ax,@data
<9>          mov   ds,ax
<10>         ;...
<11>         xor   ax,ax
<12>         mov   al,5
```

```
<13>        sub    al,10
<14>        jnc    m1          ;нет переноса?
<15>        neg al             ;в al модуль результата
<16> m1:                       ;...
<17> exit:
<18>        mov    ax,4c00h    ;стандартный выход
<19>        int    21h
<20>        end main                ;конец программы
```

В этом примере в строке 13 выполняется вычитание. С указанными для этой команды вычитания исходными данными результат получается в дополнительном коде (отрицательный). Для того чтобы преобразовать результат к нормальному 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. Дальше все зависит от алгоритма обработки. Исследуйте программу в отладчике.

5.2.1.2 Вычитание двоичных чисел со знаком

Вычитание двоичных чисел со знаком выполнять несколько сложнее. Последний пример показал то, что процессору незачем иметь два устройства — сложения и вычитания. Достаточно наличия только одного — устройства сложения. Но для вычитания способом сложения чисел со знаком оба операнда (и уменьшаемое, и вычитаемое) необходимо представлять в дополнительном коде. 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). Первый вариант вычитания чисел со знаком:

45 = 0010 1101

-

- 127 = 1000 0001

- 44 = 1010 1100.

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.

Следующее вычитание чисел со знаком выполним способом сложения:

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

-45 = 1101 0011

+

-45 = 1101 0011

-90 = 1010 0110.

Здесь все нормально, флаг переполнения OF сброшен в 0, а 1 в знаковом разряде говорит о том, что значение результата — число в дополнительном коде.

5.2.1.3 Вычитание и сложение операндов большой размерности

Если вы заметили, команды сложения и вычитания работают с операндами фиксированной размерности: 8, 16, 32 бита. А что делать, если нужно сложить числа большей размерности, например 48 битов, используя 16-разрядные операнды? К примеру, сложим два 48-разрядных числа (рис. 4).

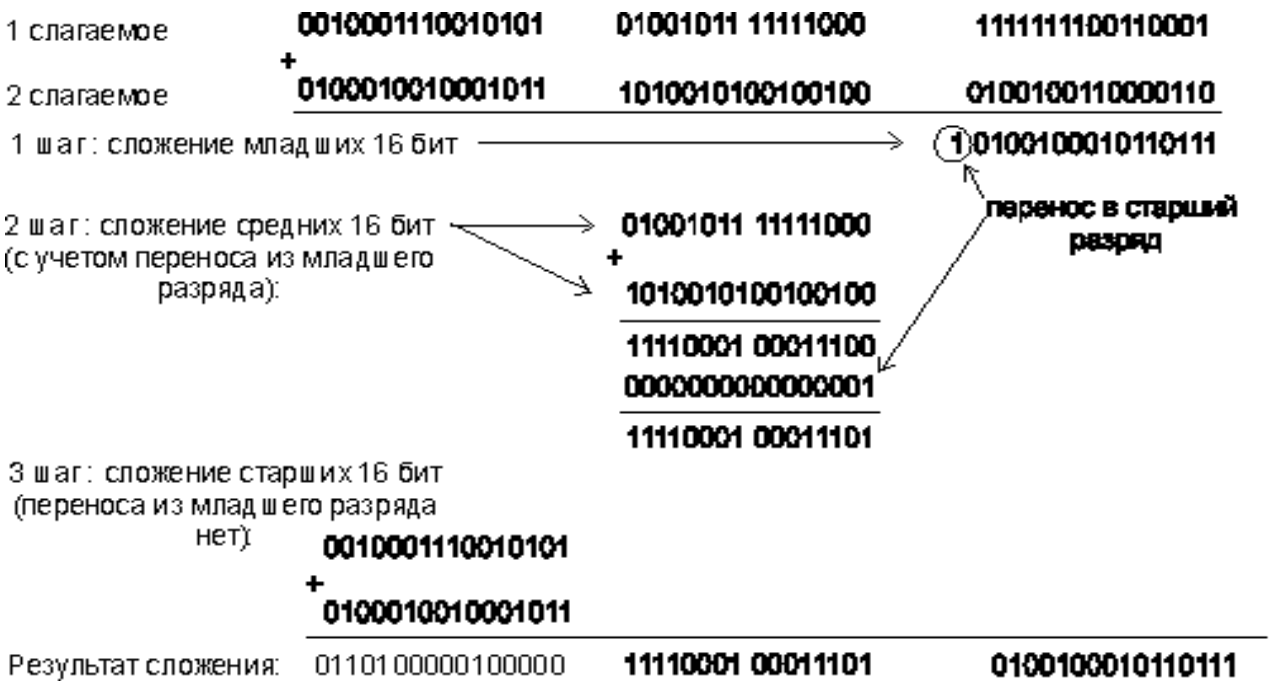

Рисунок 5.1 – Сложение операндов большой размерности

Рисунок по шагам иллюстрирует технологию сложения длинных чисел. Видно, что процесс сложения многобайтных чисел происходит так же, как и при сложении двух чисел «в столбик», — с осуществлением при необходимости переноса 1 в старший разряд. Если нам удастся запрограммировать этот процесс, то мы значительно расширим диапазон двоичных чисел, над которыми можно выполнять операции сложения и вычитания.

Принцип вычитания чисел с диапазоном представления, превышающим стандартные разрядные сетки операндов, тот же, что и при сложении, то есть используется флаг переноса CF. Нужно только представлять себе процесс вычитания в столбик и правильно комбинировать команды процессора с командой SBB.

В завершение обсуждения команд сложения и вычитания отметим, что кроме флагов CF и OF в регистре EFLAGS есть еще несколько флагов, которые можно использовать с двоичными арифметическими командами. Речь идет о следующих флагах:

- 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).

5.2.2 Арифметические операции над двоично-десятичными числами

5.2.2.1 Вычитание неупакованных BCD-чисел

Ситуация при вычитании вполне аналогична сложению. Рассмотрим те же случаи.

Результат вычитания не больше 9:

6 = 0000 0110

-

3 = 0000 0011

=

3 = 0000 0011.

Как видим, заема из старшей тетрады нет. Результат верный и корректировки не требует.

Результат вычитания больше 9:

6 = 0000 0110

-

7 = 0000 0111

=

-1 = 1111 1111.

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, как и в случае сложения, результат вычитания нужно корректировать. Для этого существует специальная команда

AAS (ASCII Adjust for Subs traction), выполняющая коррекцию результата вычитания для представления в символьном виде.

Команда AAS также 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 (заметьте, не из содержимого всего регистра) вычитается 6.
- Старшая тетрада регистра AL обнуляется.
- Флаг CF устанавливается в 1, тем самым фиксируется воображаемый заем из старшего разряда.

Понятно, что 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. В листинге 8.9 мы обходимся одной командой SBB, которая в цикле производит поразрядное вычитание двух BCD-чисел.

Листинг 8.9. Вычитание неупакованных BCD-чисел

```
<1>  ;prg_8_9.asm
<2>  masm
<3>  model small
<4>  stack  256
<5>  .data                      ;сегмент данных
<6>  b      db     1,7          ;неупакованное число 71
<7>  c      db     4,5          ;неупакованное число 54
<8>  subs   db     2 dup (0)
<9>. code
<10> main:              ;точка входа в программу
<11>        mov    ax,@data     ;связываем регистр dx с сегментом
<12>        mov    ds,ax        ;данных через регистр ax
```

```
<13>          xor    ax,ax          ;очищаем ax
<14> len      equ    2              ;разрядность чисел
<15>          xor    bx,bx
<16>          mov    cx,len ;загрузка в cx счетчика цикла
<17> m1:
<18>          mov    al,b[bx]
<19>          sbb    al,c[bx]
<20>          aas
<21>          mov    subs[bx],al
<22>          inc    bx
<23>          loop   m1
<24>          jc     m2             ;анализ флага заема
<25>          jmp    exit
<26> m2:      ;...
<27> exit:
<28>          mov    ax,4c00h       ;стандартный выход
<29>          int    21h
<30> end      main                  ;конец программы
```

Данная программа не требует особых пояснений в случае, когда уменьшаемое больше вычитаемого. Поэтому обратите внимание на строку 24. С ее помощью мы 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 оказалась неправильной. Эту ситуацию нужно как-то обработать. С этой целью в строке 24 командой JC анализируется флаг CF. По результату этого анализа мы уходим на ветку программы, обозначенную меткой m2, где и выполняются нужные действия. Набор этих действий зависит от конкретного алгоритма обработки, поэтому поясним только их суть. Для этого посмотрите в отладчике, как наша программа выполняет вычитание 50 - 74 (правильный ответ - 24). То, что вы увидите в окне Dump отладчика (в поле, соответствующем адресу subs), окажется далеким от истины. Что делает в этом случае человек? Он просто мысленно меняет

местами вычитаемое и уменьшаемое (74 - 50 = 24), а результат рассматривает со знаком «минус». Так и фрагмент программы, обозначенный меткой m2, может, поменяв уменьшаемое и вычитаемое местами и выполнив вычитание, где-то отметить тот факт, что разность, на самом деле, нужно рассматривать как отрицательное число.

5.2.2.2 Вычитание упакованных BCD-чисел

Аналогично сложению, при вычитании процессор рассматривает упакованные BCD-числа как двоичные. Выполним вычитание 67 - 75. Так как процессор выполняет вычитание способом сложения, то и мы последуем этому:

67 = 0110 0111

+

-75 = 1011 0101

=

- 8 = 0001 1100 = 28.

Как видим, результат равен 28 в десятичной системе счисления, что является абсурдом. В двоично-десятичном коде результат должен быть равен 0000 1000 (или 8 в десятичной системе счисления). При программировании вычитания упакованных BCD-чисел программист, как и при вычитании неупакованных BCD-чисел, должен сам осуществлять контроль за знаком. Это делается с помощью флага CF, который фиксирует заем из старших разрядов. Само вычитание BCD-чисел осуществляется обычной командой вычитания SUB или SBB. Коррекция результата вычитания для его представления в десятичном виде осуществляется командой DAS (Decimal Adjust for Subs traction).

Команда DAS выполняет следующие действия:

3. Если AF = 1 или значение младшей тетрады AL > 9, то (AL) = (AL) – 6; AF = 1; в случае заема при вычитании установить флаг CF.

   Иначе – AF = 0.

4. Если CF = 1 или значение старшей тетрады AL > 9, то (AL) = (AL) – 60; CF = 1.

   Иначе – CF = 0.

Команда DAS корректирует результат вычитания в регистре AL двух упакованных двоично-десятичных (BCD) чисел (по одной цифре в каждом полубайте), чтобы получить пару правильных упакованных десятичных цифр. Команда используется вслед за операцией вычитания упакованных двоично-десятичных чисел. Если для вычитания требовался заем, устанавливается флаг CF. Команда воздействует на флаги SF, ZF, AF, PF и CF.

Пример 1

```
mov AL,55h      ;Упакованное BCD 55

sub AL,19h      ;После вычитания AL=3Ch

das             ;AL=36h, т.е. упакованное BCD 36
```

Пример 2

```
mov AL,55h      ;Упакованное BCD 55

sub AL,15h      ;После вычитания AL=40h

das             ;AL=40h, т.е. упакованное

                ;BCD 40 (в данном случае команда das ничего не делает)
```

Задание на лабораторную работу

3 Создайте программы типа .exe, выполняющие сложение чисел указанного размера, двоичных и BCD, со знаком и без, выполните программы пошагово, используя отладчик TD.

4 Варианты заданий соответствуют номеру по журналу занятий:

Таблица 5.1 – Вычитание двоичных чисел

	Порядковый номер фамилии обучающегося по списку									
	1	2	3	4	5	6	7	8	9	10
Числа размером 1 байт без знака	127 - 136	**58 - 231**	29 - 98	26 - 223	234 - 67	134 - 68	209 - 54	54 - 198	90 - 200	19 - 128
Числа размером 1 байт со знаком	- 120 -(- 7)	**- 89 - (- 55)**	- 60 - (- 101)	- 45 - 127	- 100 - (- 100)	125 - (- 45)	- 120 - (- 22)	58 - (- 128)	- 19 - (- 99)	100 - (- 57)
Числа размером N байт без учета знака	034fcafdch - 78acdfh	**0af4d57h - 83654adfh**	9aacdfe3h - 0b91dafh	758466h - 0af98ffcbh	0fdacb9e3h - 5fd98dh	1754dah - 0ffad65fdh	1291fdaah - 439fd4h	0dfac6799h - 81a8c9h	0bdc3a5f4h - 3459afh	0fd87cbh - 0ad9874h

Числа размером N байт с учетом знака	- 39aacdfeh - (- 0afb91dh)	**- 2911fdaah - 4fd439h**	0ac67df99h - (-8c981ah)	- 0af4d57h - (- 83654adfh)	- 0cbfd87h - (- 0ad9874h)	- 667584h - 0ffcbaf98h	- 7584cbh - 0af98932bh	44cd8766h - (- 0c3fa65h)	- 3aacdf9eh - (- 0b91dafh)	- 0f4d57ah - (- 83654adfh)

Таблица 5.2 – Вычитание двоичных чисел

	Порядковый номер фамилии обучающегося по списку									
	11	12	13	14	15	16	17	18	19	20
Числа размером 1 байт	102 - 145	64 - 95	111 - 155	87 - 255	92 - 245	122 - 47	98 - 180	106 - 155	164 - 88	206 - 99
Числа размером 1 байт со знаком	- 114 - (- 34)	124 - (- 78)	- 90 - (- 24)	88 - (- 93)	- 123 - (- 55)	- 68 - (- 99)	- 20 - (- 79)	- 64 - 127	- 103 - (- 27)	98 - (- 99)
Числа размером N байт без учета знака	0afdc34fch - 98acdfh	0f4d57ah - 654adf83h	93aacdfeh - 0dafb91h	0cbf9aah - 3421dae8h	5346dcabh - 0fc438a3fh	0cb321dh - 534adc49h	876644cdh - 0fa65c3h	0b8723da4h - 7391fbcah	2008a3dbh - 0bc5df5h	7af0129fh - 0dacc44h

Числа размером N байт с учетом знака	- 034fcac54h - (- 8acdf7h)	- 91fd12aah - (- 9fd443h)	- 0acb9e3fdh - 98d5fdh	0fa4d57h - (- 64adf853h)	- 39cdfaaeh - (- 0ab91dfh)	- 3a5caf9eh - (- 0afb91dh)	- 21fdaa91h - (- 4394fdh)	- 0f9acbah - (- 321da4e8h)	- 0af4d61h - 836544dfh	- 0dfac67h - (- 85afc9h)

Таблица 5.3 – Вычитание BCD чисел

	Порядковый номер фамилии обучающегося по списку									
	1	2	3	4	5	6	7	8	9	10
Неупакованные BCD числа	87 - 85	**54 - 97**	341 - 435	644 - 215	72 - 79	921 - 247	898 - 489	396 - 652	662 - 788	506 - 999
Упакованные BCD числа	5224 - 9232	**1298 - 4675**	2133 - 8790	6587 - 6745	5365 - 3912	4343 - 9108	9080 - 1211	4356 - 2791	3910 - 3165	9887 - 5655

Таблица 5.4 – Вычитание BCD чисел

	Порядковый номер фамилии обучающегося по списку									
	1	2	3	4	5	6	7	8	9	10
Неупакованные BCD числа	67 - 45	**64 - 95**	141 - 155	534 - 255	92 - 25	622 - 847	198 - 580	196 - 455	164 - 988	206 - 998
Упакованные BCD числа	1234 - 3299	**9812 - 7546**	3321 - 9087	8765 - 4567	6553 - 1239	4343 - 9100	8090 - 1112	5643 - 9127	9103 - 6531	8798 - 5556

Содержание отчета

- В отчете указать название и порядковый номер лабораторной работы, сформулировать цель работы.
- Указать имена и типы разработанных программ.
- Представить листинги всех программ.
- Распечатать изображения с экрана при выполнении программы.
- Представить состояние содержимого регистров и ячеек памяти по

мере пошагового выполнения программы. Указывать состояние изменившихся регистров, ячеек памяти и флагов состояния процессора.

- Сделать выводы по каждой команде вычитания, как двоичных так и BCD-чисел.

Лабораторная работа №6

Исследование работы команд умножения микропроцессора

В данной лабораторной работе мы познакомимся с командами умножения. Команды умножения используются одинаково часто с другими арифметическими командами. Алгоритм работы команд своеобразный. Есть особенности работы с двоично-десятичными числами и числами со знаком.

Цель работы

Ознакомление с группой команд микропроцессора относящихся к командам умножения. Изучить команды:

- умножения
    - умножение без знака mul;
    - умножение со знаком imul;
    - коррекция умножения aam.

Исследование работы команд целочисленного умножения микропроцессора. Выработать навыки написания программ с использованием команд умножения.

Теоретическая часть

6.2.1 Умножение двоичных чисел

Для умножения чисел предназначена команда

```
mul сомножитель_1
```

Как видите, в команде указан всего лишь один операнд-сомножитель. Второй операнд-сомножитель задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Так как в общем случае результат умножения больше, чем любой из его сомножителей, то его размер и местоположение должны быть тоже определены однозначно. Варианты размеров сомножителей и мест размещения второго операнда и результата

приведены в табл. 6.1.

Таблица 6.1 – Расположение операндов и результата при умножении

Первый сомножитель	**Второй сомножитель**	**Результат**
Байт	AL	16 битов в AX: AL — младшая часть результата; AH — старшая часть результата
Слово	AX	32 бита в паре DX:AX: AX — младшая часть результата; DX — старшая часть результата
Двойное слово	EAX	64 бита в паре EDX:EAX: EAX — младшая часть результата; EDX — старшая часть результата

Из таблицы видно, что произведение состоит из двух частей и в зависимости от размера операндов размещается в двух местах — на месте второго сомножителя (младшая часть) и в дополнительных регистрах AH, DX, EDX (старшая часть). Как же динамически (то есть во время выполнения программы) узнать, что результат достаточно мал и уместился в одном регистре или что он превысил размерность регистра и старшая часть оказалась в другом регистре? Для этого привлекаются уже известные нам флаги переноса CF и переполнения OF:

8. если старшая часть результата нулевая, то после завершения операции CF = 0 и OF = 0;
9. если же флаги CF и OF ненулевые, это означает, что результат вышел за пределы младшей части произведения и состоит из двух частей, что и нужно учитывать при дальнейшей работе.

Рассмотрим следующий пример программы (листинг 8.5).

Листинг 8.5. Умножение

```
<1>  ;prg_8_5.asm
<2>  masm
<3>  model small
<4>  stack  256
<5>  .data                          ;сегмент данных
<6>  rez    label   word
```

```
<7>  rez_l        db     45
<8>  rez_h db     0
<9>  .code                              ;сегмент кода
<10> main:                  ;точка входа в программу
<11>        mov   ax,@data
<12>        mov   ds,ax
<13> ;...
<14>        xor   ax,ax
<15>        mov   al,25
<16>        mul   rez_l
<17>        jnc   m1              ;если нет переполнения, то на м1
<18>        mov   rez_h,ah        ;старшую часть результата в rez_h
<19> m1:
<20>        mov   rez_l,al        ;младшую часть результата в rez_l
<21>        ;...
<22>        exit:
<23>        mov   ax,4c00h        ;стандартный выход
<24>        int   21h
<25> end    main          ;конец программы
```

В строке 16 производится умножение значения в rez_l на число в регистре AL. Согласно информации из табл. 6.1, результат умножения будет располагаться в регистре AL (младшая часть) и в регистре AH (старшая часть). Для выяснения размера результата в строке 17 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 в строке 18, которая формирует в поле rez_h старшее слово результата. Команда в строке 19 формирует младшую часть результата. Теперь обратите внимание на сегмент данных, а именно на строку 6. В этой строке содержится директива label. Мы еще не раз будем сталкиваться с этой директивой. В данном случае она назначает еще одно символическое имя rez

адресу, на который уже указывает другой идентификатор rez_l. Различие заключается в типах этих идентификаторов — rez имеет тип слова, который ему назначается директивой label (имя типа указано в качестве операнда label). Введя эту директиву в программе, мы подготовились к тому, что, возможно, результат операции умножения будет занимать в памяти целое слово. Обратите внимание на то, что мы не нарушили принципа: младший байт по младшему адресу. Далее, используя имя rez, можно обращаться к значению в этой области как к слову.

В заключение можно исследовать в отладчике программу на разных наборах сомножителей.

6.2.1.1 Умножение двоичных чисел размером 1 байт без учета знака

Листинг mul_unsign.asm

```
<1>  ;Вход: multiplier1 и multiplier2 - множители размером 1 байт.
<2>  ;Выход: product - значение произведения.
<3>  MASM
<4>  MODEL small
<5>  STACK 256
<6>  .data
<7>  ;значения в multiplier1 и multiplier2 нужно внести
<8>  product     label  word
<9>  product_l   label  byte
<10> multiplier1 db     ?       ;множитель 1 (младшая часть произведения)
<11> product_h   db     0       ;старшая часть произведения
<12> multiplier2 db     ?       ;множитель 2
<13> .code
<14> mul_unsign proc
<15>        mov   al,multiplier1
<16>        mul   multiplier2
<17> ;оценить результат:
<18> jnc   no_carry      ;нет переполнения - на no_carry
```

```
<19>                    ;обрабатываем ситуацию переполнения
<20>        mov   product_h,ah;старшая часть результата
<21> no_carry:  mov   product_l,al  ;младшая часть результата
<22>        ret
<23> mul_unsign endp
<24> main:
<25>        mov   dx,@data
<26>        mov   ds,dx
<27>        call    mul_unsign
<28>        mov   ax,4c00h
<29>        int     21h
<30> end   main
```

6.2.1.2 Умножение чисел размером N байт без учета знака

Листинг mul_unsign_NM_I.asm.

```
;-------------------------------------------------------
 ;mul_unsign_NM_I.asm - программа mul_unsign_NM_I умножения N-байтного числа на число размером M байт
 ;(порядок - младший байт по младшему адресу (Intel))
 ;Вход: U и V - множители размерностью N и M байт соответственно; b=256 - размерность ;машинного слова.
 ;Выход: W - произведение размерностью N+M байт.
 ;-------------------------------------------------------
 MASM
 MODEL small
 STACK       256
 .data
 ;значения в U и V нужно внести
 U      db      0dfh,4ah,65h,83h            ;множитель 1 размерностью N байт 83654adfh
 i=$-U
 V      db      57h,4dh,0afh          ;множитель 2 размерностью M байт 0af4d57h
 j=$-V
 len_product=$-u
```

```
W       db      len_product dup (0);len_product=N+M
k       db      0       ;перенос 0 г k < 255
b       dw      100h    ;размер машинного слова
.code
mul_unsign_NM_I  proc
;НАЧ_ПРОГ
;m1
;ДЛЯ j:= 0 ДО M-1 //J изменяется в диапазоне 0..M-1
        xor     bx,bx
        mov     cx,j
m2:
;НАЧ_БЛОК_1
        push    cx      ;вложенные циклы
;//проверка на равенство нулю очередного элемента множителя (не обязательно)
;ЕСЛИ v[j]==0 ТО ПЕРЕЙТИ_НА m6
        cmp     v[bx],0
        je      m6
;m3
;k:=0; i:=n-1 //i изменяется в диапазоне 0..N-1
;ДЛЯ i:= 0 ДО N-1
        xor     si,si
        mov     cx,i
        mov     k,0
m4:
;НАЧ_БЛОК_2
;//перемножаем очередные элементы множителей
;temp_word:=u[i]*v[j]+w[i+j]+k
        mov     al,u[si]
        mul     byte ptr v[bx]
        xor     dx,dx
        mov     dl,w[bx+si]
        add     ax,dx
        xor     dx,dx
```

```
        mov    dl,k
        add    ax,dx  ;t=(ax) - временная переменная
;w[i+j]:=temp_word MOD b //остаток от деления temp_word\b -> w[i+j]
;k:=temp_word\b //целая часть частного temp_word\b -> k
        push   dx
        xor    dx,dx
        div    b       ;t mod b
        mov    ah,dl
        pop    dx
        mov    k,al
        mov    w[bx+si],ah
;m5
        inc    si
        loop   m4
;КОН_БЛОК_2
;w[i+j]:=k
        mov    al,k
        mov    w[bx+si],al
;m6:
m6:
        inc    bx
        pop    cx
        loop   m2
;КОН_БЛОК_1
        ret
;КОН_ПРОГ
mul_unsign_NM_I  endp
main:
        mov    dx,@data
        mov    ds,dx
        call   mul_unsign_NM_I
        mov    ax,4c00h
        int    21h
```

```
end   main
```

6.2.2 Умножение двоичных чисел со знаком

Для умножения чисел со знаком предназначена команда

```
imul  операнд_1[,операнд_2,операнд_3]
```

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. Если вы найдете в приложении команду IMUL, то увидите, что у нее имеются более широкие возможности по заданию местоположения операндов. Это сделано для удобства использования.

6.2.2.1 Умножение чисел размером 1 байт с учетом знака

Листинг mul_sign.asm

```
<1>  ;-------------------------------------------------------
<2>  ;mul_sign.asm - программа умножения чисел размером 1 байт с учетом знака
<3>  ;Вход: multiplier1 и multiplier2 - множители со знаком размерностью 1 байт.
<4>  ;Выход: product - значение произведения.
<5>  ;-------------------------------------------------------
<6>  MASM
<7>  MODEL small
<8>  STACK 256
<9>  .data
<10>  ;значения в multiplier1 и multiplier2 нужно внести
<11>  product      label   word
<12>  product_l    label   byte
```

```
<13> multiplier1  db     ?       ;множитель 1 (младшая часть произведения)
<14> product_h    db     0       ;старшая часть произведения
<15> multiplier2  db     ?       ;множитель 2
<16> .code
<17> mul_sign     proc
<18>         mov    al,multiplier1
<19>         imul   multiplier2
<20>;оценить результат:
<21> jnc    no_carry      ;нет переполнения - на no_carry
<22> ;обрабатываем ситуацию переполнения
<23>         mov    product_h,ah        ;старшая часть результата,
<24>                                    ;знак результата - старший бит product_h
<25> no_carry:    mov    product_l,al   ;младшая часть результата,
<26>                                    ;product_h - расширение знака
<27>         ret
<28> mul_sign     endp
<29> main:
<30>         mov    dx,@data
<31>         mov    ds,dx
<32>         call   mul_sign
<33>         mov    ax,4c00h
<34>         int    21h
<35> end   main
```

6.2.3 Умножение BCD - чисел

На примере сложения и вычитания неупакованных чисел мы выяснили, что стандартных алгоритмов для выполнения этих действий над BCD-числами нет и программист должен сам, исходя из требований к своей программе, реализовать эти операции. Реализация двух оставшихся операций — умножения и деления — еще сложнее. В системе команд процессора присутствуют только средства для умножения и деления одноразрядных неупакованных BCD-чисел. Для их умножения необходимо воспроизвести

описанную далее процедуру.

- Поместить один из сомножителей в регистр AL (как того требует команда MUL).
- Поместить второй сомножитель в регистр или память, отведя для него байт.
- Перемножить сомножители командой MUL (результат, как и положено, окажется в регистре AX).
- Скорректировать результат, который, конечно, будет представлен в двоичном коде.

Для коррекции результата после умножения в целях представления его в символьном виде применяется специальная команда AAM (ASCII Adjust for Multiplication). Она не имеет операндов и работает с регистром AX следующим образом.

- Делит AL на 10.
- Результат деления записывается так: частное — в AH, остаток — в AL.

В результате после выполнения команды AAM в регистрах AL и AH находятся правильные двоично-десятичные цифры произведения двух цифр.

В листинге 8.10 приведен пример умножения BCD-числа произвольной размерности на однозначное BCD-число.

Листинг 8.10. Умножение неупакованных BCD-чисел

```
<1> ;prg8_10.asm
<2> masm
<3> model     small
<4> stack 256
<5> .data
<6> b     db   6,7  ;неупакованное число 76
<7> c     db   4    ;неупакованное число 4
<8> proizv     db   4 dup (0)
<9> .code
<10>main:     ;точка входа в программу
<11>          mov  ax,@data
```

```
<12>          mov ds,ax
<13>          xor  ax,ax
<14>     len  equ  2    ;размерность сомножителя 1
<15>          xor  bx,bx
<16>          xor  si,si
<17>          xor  di,di
<18>          mov  cx,len;в cx длина наибольшего сомножителя 1
<19>     m1:
<20>          mov  al,b[si]
<21>          mul  c
<22>          aam           ;коррекция умножения
<23>          adc  al,dl      ;учли предыдущий перенос
<24>     aaa  ;скорректировали результат сложения с переносом
<25>          mov  dl,ah      ; запомнили перенос
<26>          mov  proizv[bx],al
<27>          inc  si
<28>          inc  bx
<29>          loop m1
<30>          mov  proizv[bx],dl    ;учли последний перенос
<31>          exit:
<32>          mov  ax,4c00h
<33>          int  21h
<34>     end  main
```

Данную программу можно легко модифицировать для умножения BCD-чисел произвольной длины. Для этого достаточно представить алгоритм умножения в «столбик». Листинг 8.10 можно использовать для получения частичных произведений в этом алгоритме. После их сложения со сдвигом получится искомый результат. Попробуйте написать эту программу самостоятельно.

Перед окончанием обсуждения команды AAM необходимо отметить, что ее можно применять для преобразования двоичного числа в регистре AL в

неупакованное BCD-число, которое окажется в регистре AX: старшая цифра результата — в AH, младшая — в AL. Понятно, что двоичное число должно быть в диапазоне 0...99.

Индивидуальные задания

- Создайте программы типа .exe, выполняющие сложение чисел указанного размера, двоичных и BCD, со знаком и без, выполните программы пошагово, используя отладчик TD.
- Варианты заданий соответствуют номеру по журналу занятий:

Таблица 6.2 – Умножение двоичных чисел

	Порядковый номер фамилии обучающегося по списку									
	1	2	3	4	5	6	7	8	9	10
Числа размером 1 байт без знака	127 - 136	58 - 231	29 - 98	26 - 223	234 - 67	134 - 68	209 - 54	54 - 198	90 - 200	19 - 128
Числа размером 1 байт со знаком	- 120 -(- 7)	- 89 - (- 55)	- 60 - (- 101)	- 45 - 127	- 100 - (- 100)	125 - (- 45)	- 120 - (- 22)	58 - (- 128)	- 19 - (- 99)	100 - (- 57)
Числа размером N байт без учета знака	034fcafdch - 78acdfh	0af4d57h - 83654adfh	9aacdfe3h - 0b91dafh	758466h - 0af98ffcbh	0fdacb9e3h - 5fd98dh	1754dah - 0ffad65fdh	1291fdaah - 439fd4h	0dfac6799h - 81a8c9h	0bdc3a5f4h - 3459afh	0fd87cbh - 0ad9874h

Таблица 6.3 – Умножение двоичных чисел

	Порядковый номер фамилии обучающегося по списку									
	11	12	13	14	15	16	17	18	19	20

Числа размером 1 байт	102 - 145	64 - 95	111 - 155	87 - 255	92 - 245	122 - 47	98 - 180	106 - 155	164 - 88	206 - 99
Числа размером 1 байт со знаком	- 114 - (- 34)	124 - (- 78)	- 90 - (- 24)	88 - (- 93)	- 123 - (- 55)	- 68 - (- 99)	- 20 - (- 79)	- 64 - 127	- 103 - (- 27)	98 - (- 99)
Числа размером N байт без учета знака	0afdc34fch - 98acdfh	0f4d57ah - 654adf83h	93aacdfeh - 0dafb91h	0cbf9aah - 3421dae8h	5346dcabh - 0fc438a3fh	0cb321dh - 534adc49h	876644cdh - 0fa65c3h	0b8723da4h - 7391fbcah	2008a3dbh - 0bc5df5h	7af0129fh - 0dacc44h

Таблица 6.4 – Умножение BCD чисел

	Порядковый номер фамилии обучающегося по списку									
	1	2	3	4	5	6	7	8	9	10
Неупакованные BCD числа	87 - 85	**54 - 97**	341 - 435	644 - 215	72 - 79	921 - 247	898 - 489	396 - 652	662 - 788	506 - 999

Таблица 6.5 – Умножение BCD чисел

	Порядковый номер фамилии обучающегося по списку									
	11	12	13	14	15	16	17	18	19	20
Неупакованные BCD числа	67 - 45	64 - 95	141 - 155	534 - 255	92 - 25	622 - 847	198 - 580	196 - 455	164 - 988	206 - 998

Содержание отчета

- В отчете указать название и порядковый номер лабораторной работы, сформулировать цель работы.
- Указать имена и типы разработанных программ.
- Представить листинги всех программ.
- Распечатать изображения с экрана при выполнении программы.
- Представить состояние содержимого регистров и ячеек памяти по мере пошагового выполнения программы. Указывать состояние изменившихся регистров, ячеек памяти и флагов состояния процессора.
- Сделать выводы по каждой команде умножения, как двоичных так и BCD-чисел.