Создание программы на ассемблере TASM для ПК.

До начала работы на диске c:\ создадим папку: C:\ASM.

В этой папке расположим файлы TLINK.EXE, TASM.EXE, TD.EXE и файл исходного текста *.asm. Затем нажмём кнопку «Пуск» выберем пункт «Выполнить». Операция показана на рис. 1.

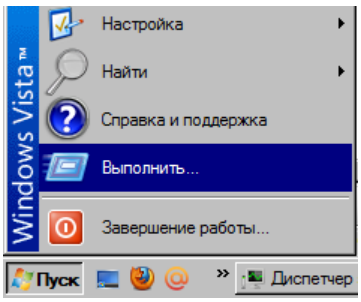

Рис. 1. Пункт Выполнить из меню Пуск.

В появившемся окне наберём CMD – это запустит командную строку (рис. 2).

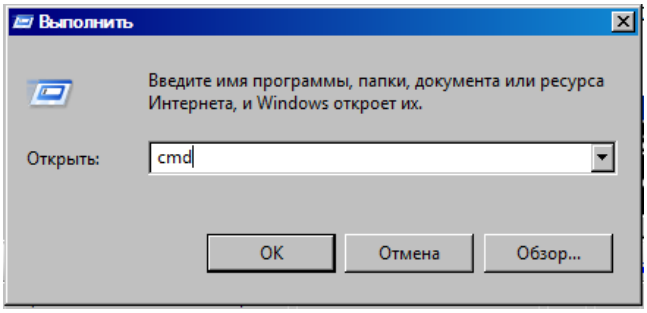

Рис. 2. Запуск командной строки.

После запуска командной строки перейдём к созданной папке (рис. 3). Переход осуществляется по команде cd c:\asm. После перехода к папке можно просмотреть содержимое папки командой dir.

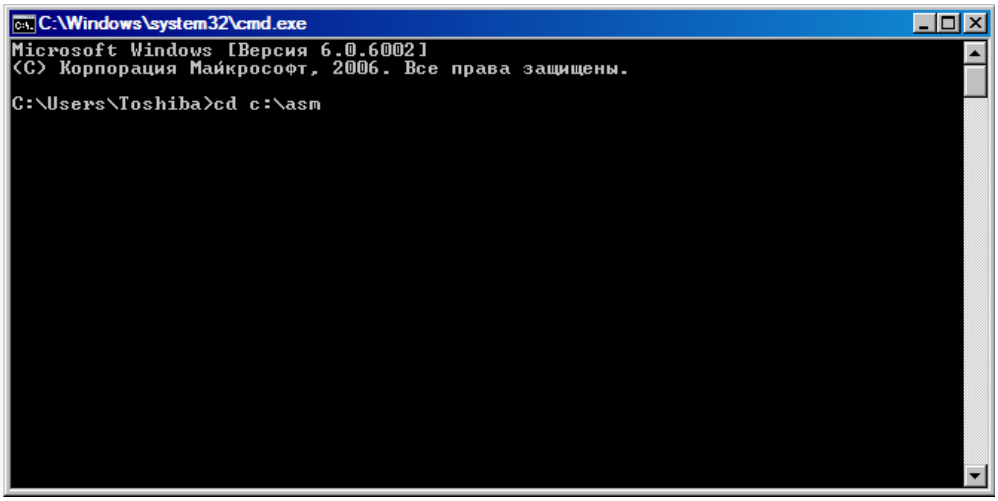

Рис. 3. Переход к папке asm.

Для компиляции исходного текста программы на ассемблере нужно запустить программу TASM.EXE. В дальнейшем предполагается отладка программы в программе

TD.EXE, поэтому при компиляции нужно добавить отладочную информацию в выполняемый модуль *.exe; для этого компилятору TASM нужно задать ключ /zi. Поэтому в командной строке следует набрать текст:

tasm.exe /zi source, object

Вместо source нужно подставить название файла с текстом программы, а вместо object нужно подставить название получаемого объектного файла (рис. 4). Для просмотра ключей управления программой нужно набрать tasm.exe

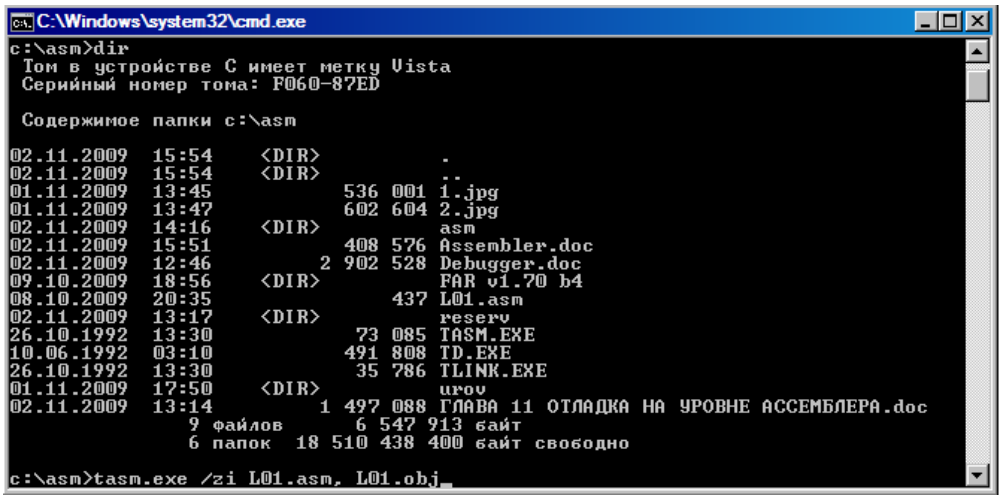

Рис. 4. Компиляция текста программы L01.asm

Если компилятор не выдаст сообщение об ошибке, то компиляция прошла успешно (рис. 5). В папке asm появится объектный файл L01.OBJ, и этот файл нужно обработать программой линковщиком для получения исполняемого файла.

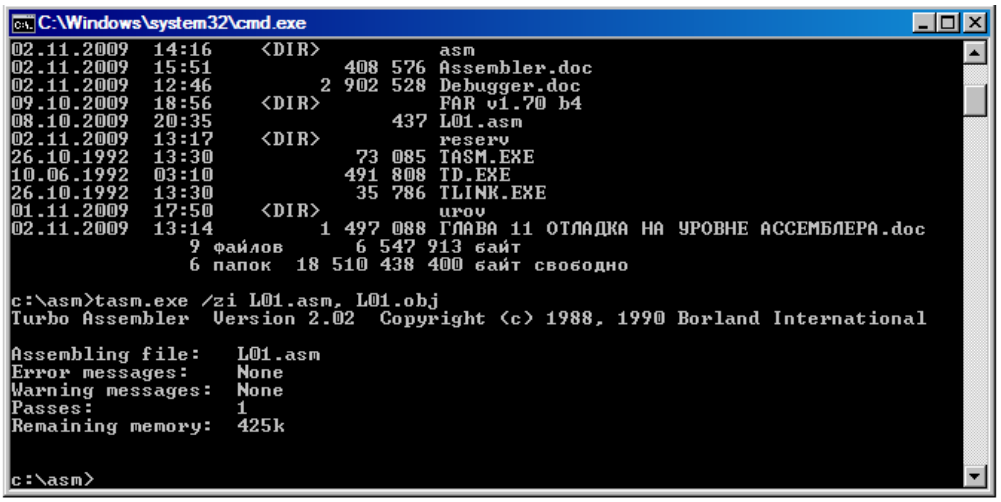

Рис. 5. Компиляция без ошибок.

Для получения исполняемого файла из объектного файла нужно запустить программу TLINK.EXE. В дальнейшем предполагается отладка программы в программе TD.EXE, поэтому при компиляции нужно добавить отладочную информацию в выполняемый модуль *.exe; для этого линковщику нужно задать ключ /v. Поэтому в командной строке следует набрать такой текст:

tlink.exe /v objfiles, exefile

Вместо objfiles нужно подставить название объектного файла, а вместо exefile название получаемого файла (рис. 6). Для просмотра ключей управления программой нужно набрать tlink.exe

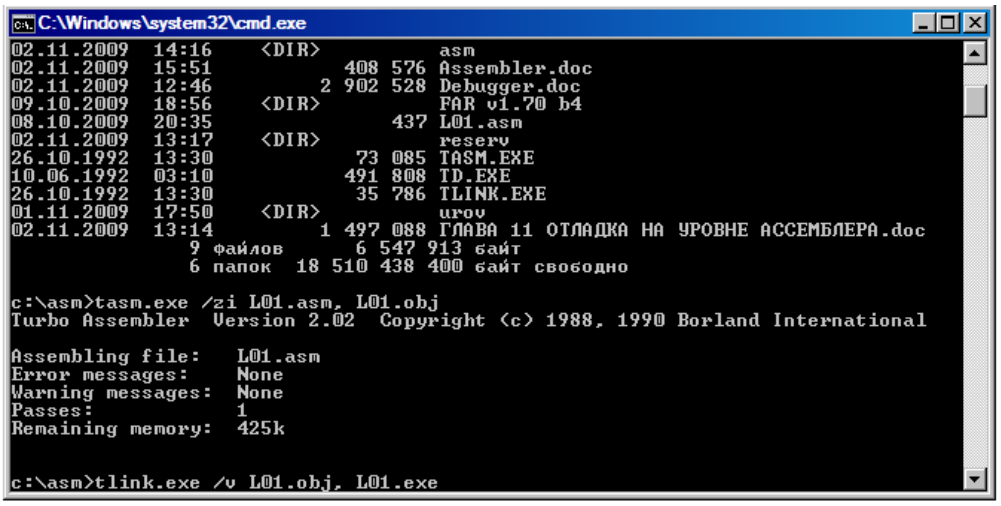

Рис. 6. Создание исполняемого файла.

Если линковка (работа программы TLINK.EXE) прошла успешно (рис 7), тогда в папке asm появится исполняемый файл. В нашем случае это L01.exe

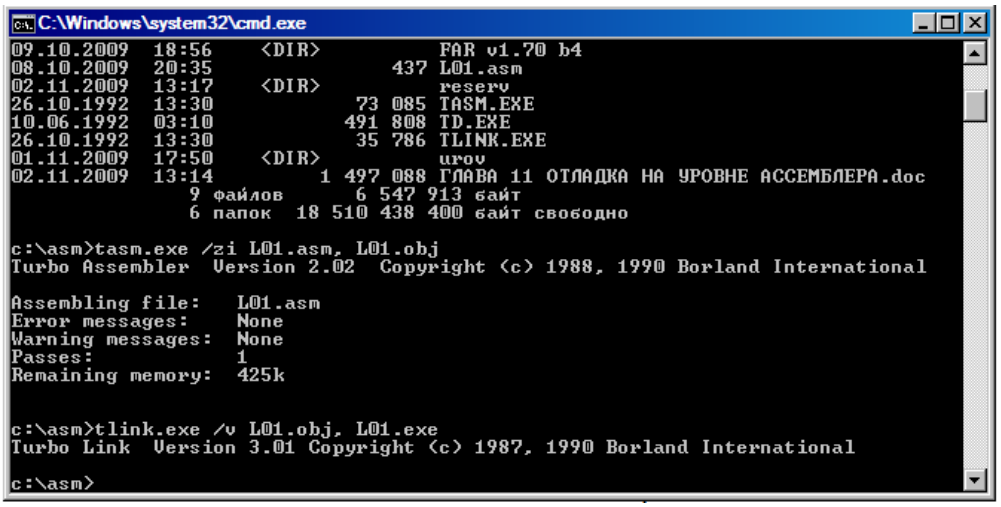

Рис. 7. Окно после успешной линковки.

Теперь, когда у нас всё готово, можно запустить отладчик. Если мы находимся в папке asm, куда мы заранее помести все программы, то запуск отладчика осуществляется командой td.exe (рис. 8).

После запуска отладчика нужно мышкой нажать кнопку OK (рис. 9).

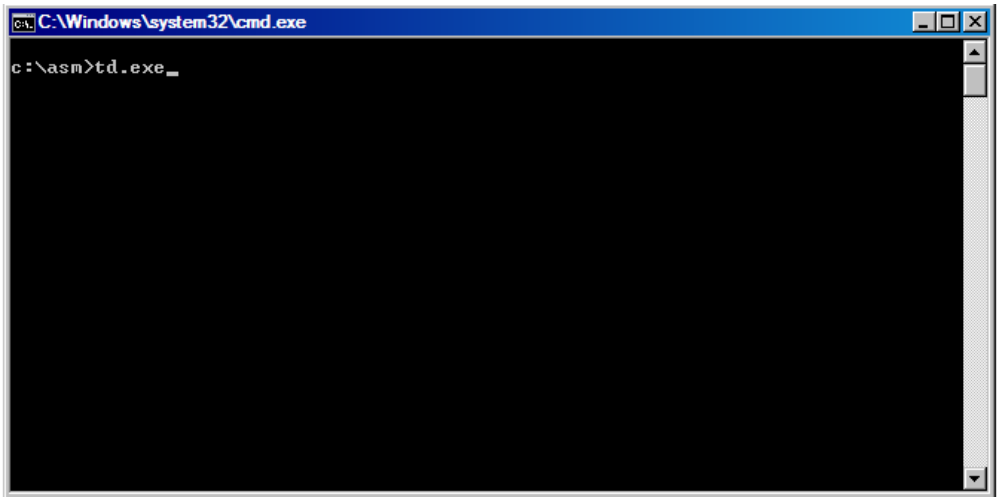

Рис. 8. Запуск отладчика td.exe.

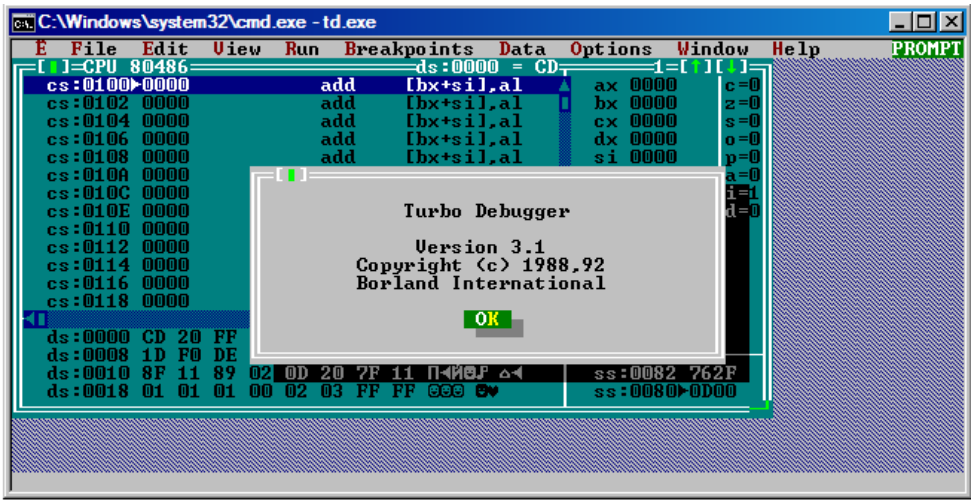

Рис. 9. Запущенный отладчик td.exe

Настройка отладчика TURBO DEBUGGER

Для начала настроим параметры отображения отладчика в окне. Для этого в меню Options выберем Display options (рис. 10).

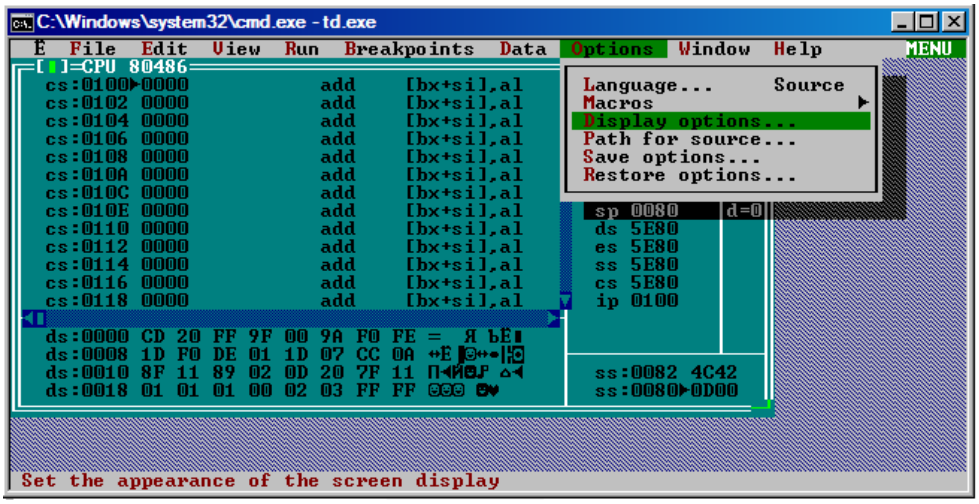

Рис. 10. Выбор параметров отображения отладчика в окне.

В появившемся окне выберем Screen lines 43/50. Для выбора нужно щёлкнуть мышкой – появится точка (Рис. 11).

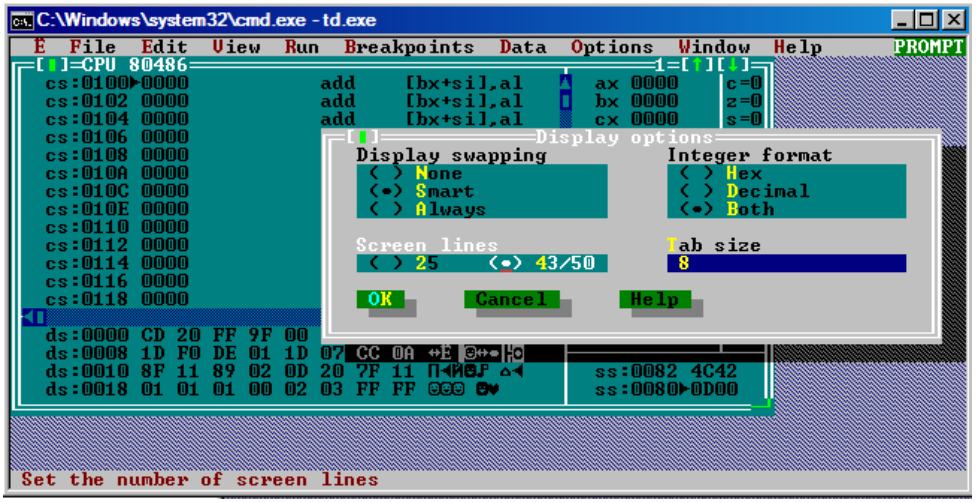

Рис. 11. Настройка окна.

Затем нужно нажать OK. После этого окно примет вид как показано на рис. 12. Такой вид позволит отображать больше информации в доступной форме. Затем откроем файл, который мы собираемся отладить. Пусть это будет l01.exe. Для этого в меню File выбираем Open (рис. 13). В открывшемся диалоге достаточно два раза щёлкнуть левой кнопкой мыши на отлаживаемом файле (в данном случае l01.exe на рис. 14).

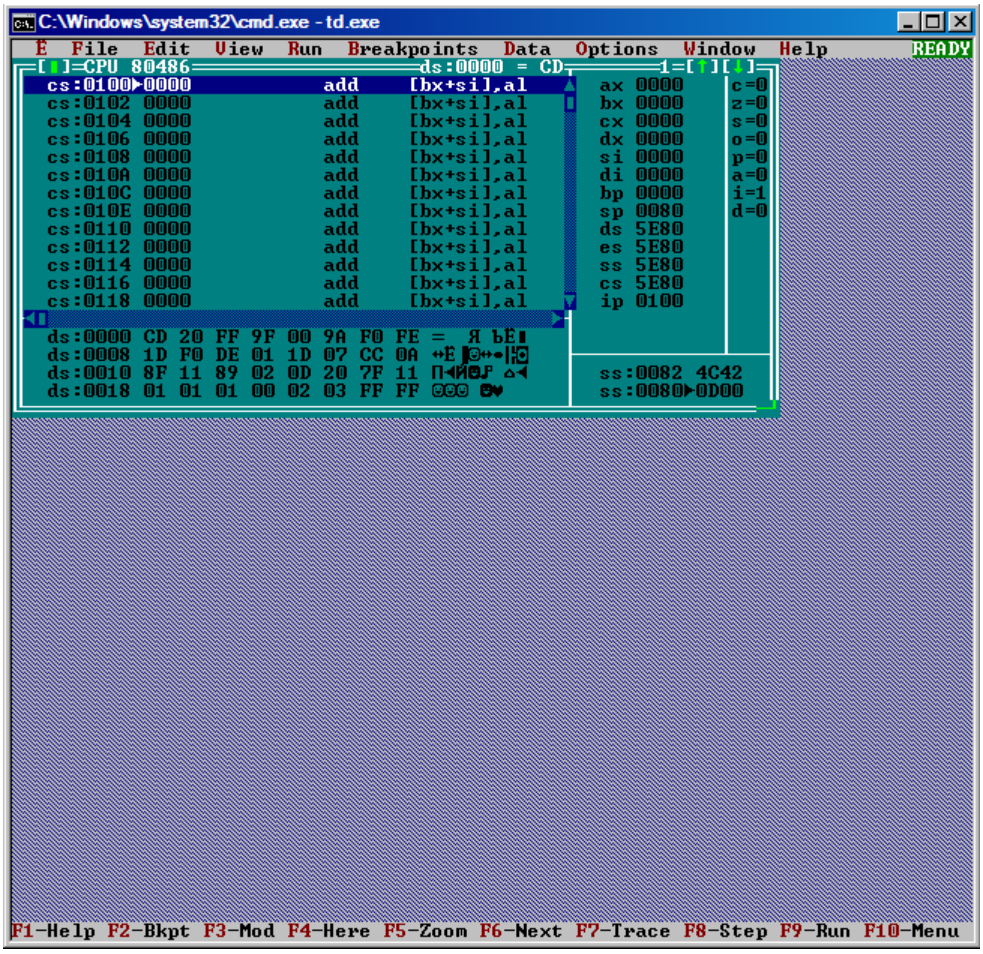

Рис. 12. Вид после настройки.

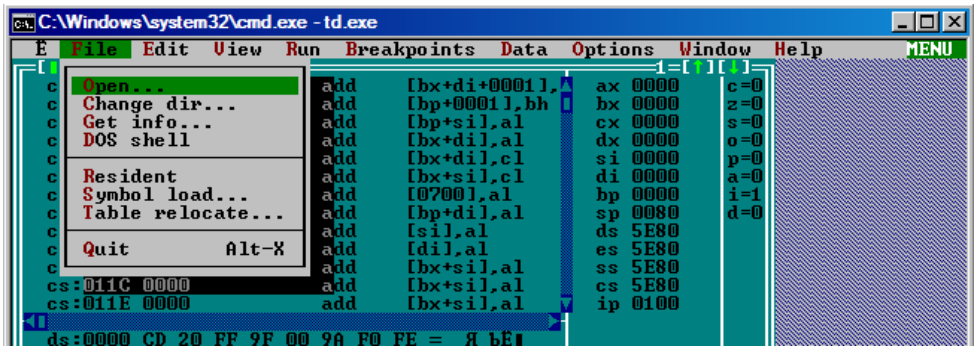

Рис. 13. Выбор файла.

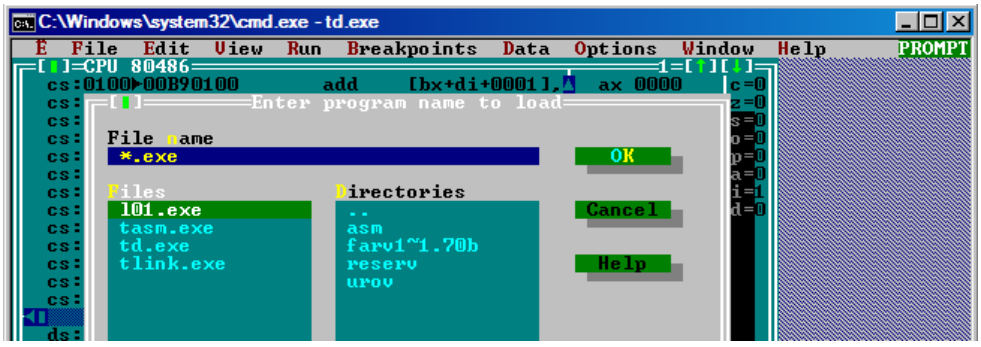

Рис. 14. Выбор отлаживаемого файла.

Если все ключи при трансляции и компоновке (линковке) были заданы верно, тогда вся отладочная информация, включая текст программы была помещена в l01.exe файл, и тогда в окне отладчика появится текст программы (рис. 15).

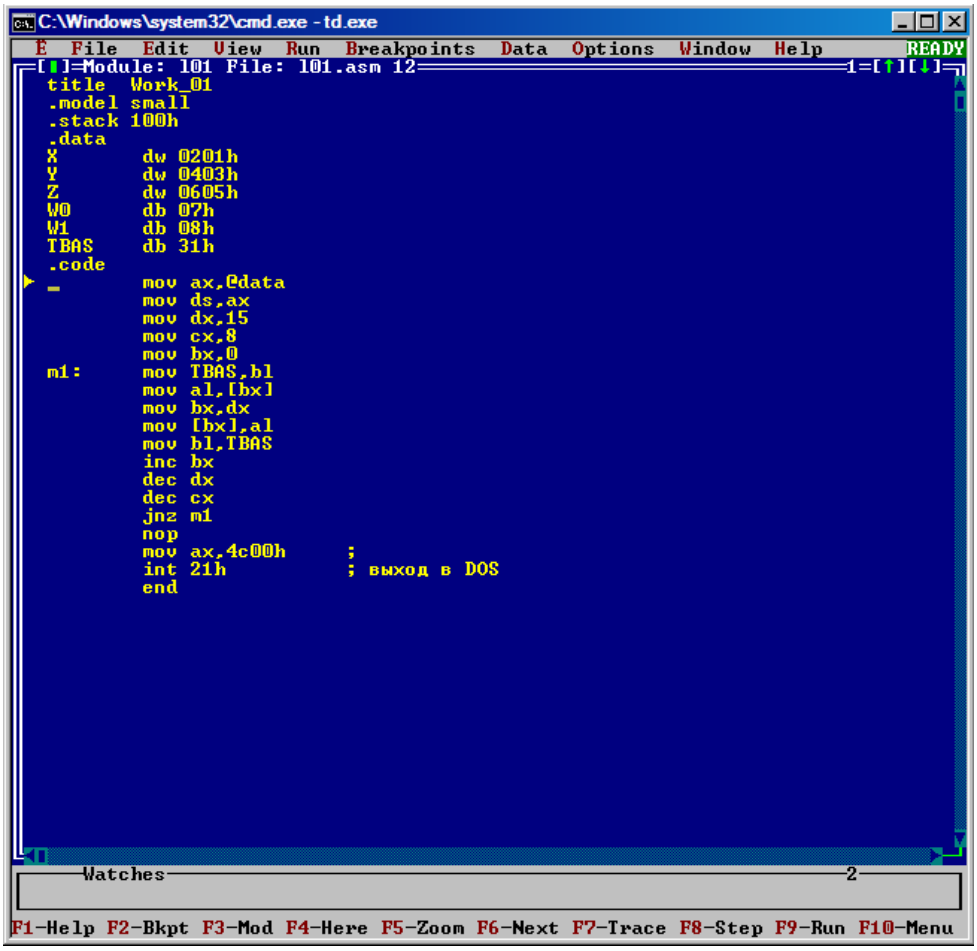

Рис. 15. Исходный текст программы в окне отладчика.

Поскольку нас интересуют данные, которые мы обрабатываем, то с помощью отладчика нужно посмотреть область памяти, содержащую наш сегмент данных. Откроем окно Dump. В меню View выбираем Dump (рис. 16). После этого нужно настроить его на адрес начала сегмента данных. Этот адрес должен содержаться в сегментном регистре ds. Но перед началом выполнения программы адрес в ds не соответствует началу сегмента данных. Нужно перед первым обращением к любому символическому имени произвести загрузку действительного физического адреса сегмента данных. Обычно это действие производят первыми двумя командами в сегменте кода. Действительный физический адрес сегмента данных извлекают как значение предопределённой переменой @data. В нашей программе эти действия выполняют команды

mov ax, @data

mov ds, ax

Для того чтобы посмотреть содержимое сегмента данных нужно остановить выполнение программы после этих двух команд. Это можно сделать, если перевести отладчик в пошаговый режим с помощью клавиш F7 или F8. Нажмите два раза F8. В окне Dump вызовите контекстное меню, щёлкнув правой кнопкой мыши. В появившемся контекстном меню выберите команду Goto (рис. 17). Появится диалоговое окно, в котором нужно ввести начальный адрес памяти, начиная с которого будет выводиться информация в окне Dump (рис. 18). Синтаксис задания этого адреса должен соответствовать

синтаксису задания адресного операнда в программе на ассемблере. Мы хотим увидеть содержимое памяти нашего сегмента данных, начиная с начала, поэтому введём ds:0000.

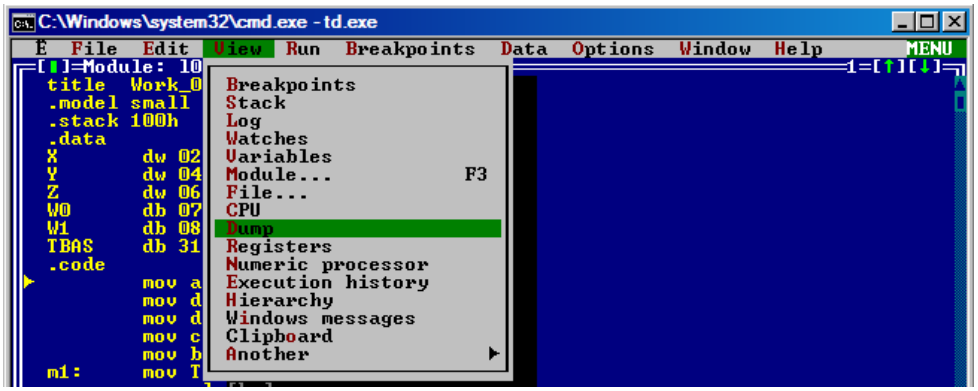

Рис. 16. Выбор Dump в меню View.

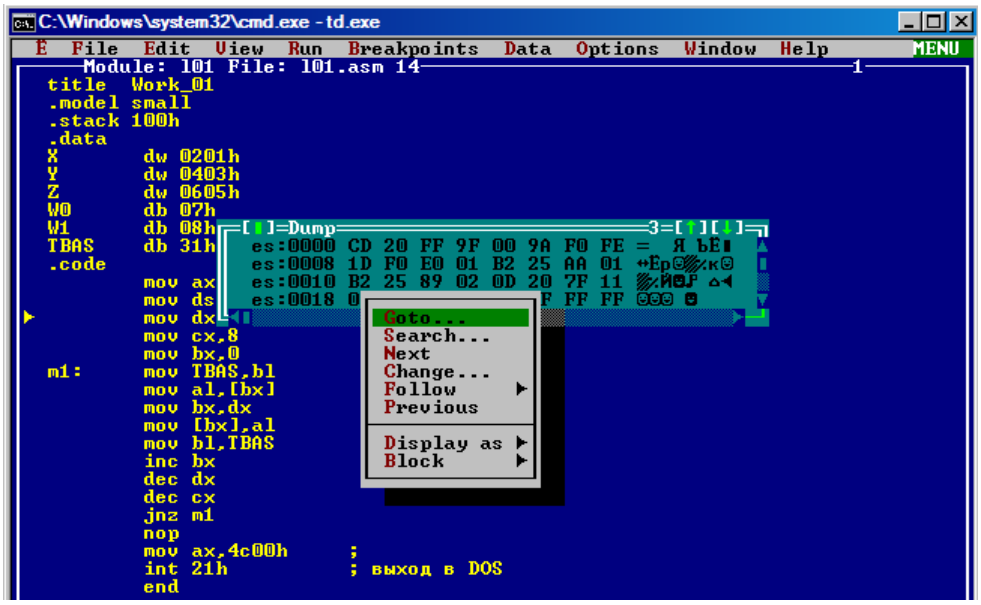

Рис. 17. Настройка окна Dump.

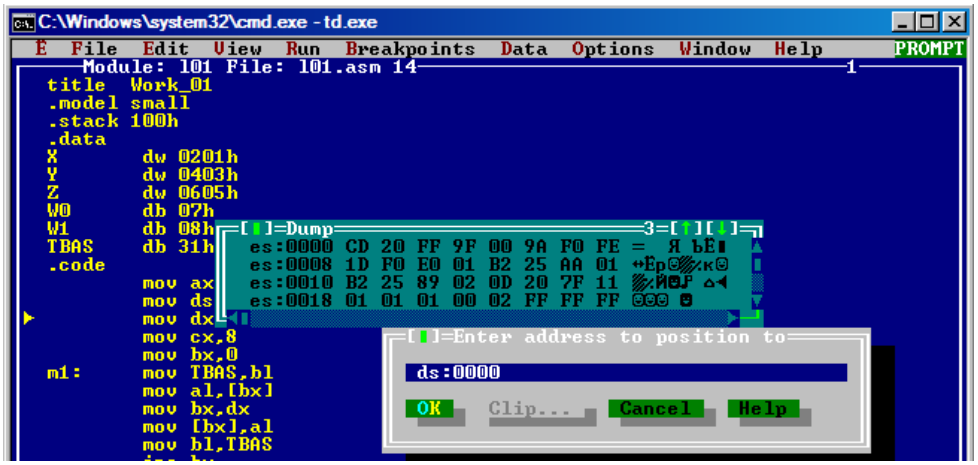

Рис. 18. Диалоговое окно, в котором нужно ввести начальный адрес памяти.

После того как окно Dump настроено на верный адрес в памяти нужно нажать кнопку OK.

В окне Dump отобразится содержимое сегмента памяти, в котором находятся данные программы l01.exe. Обратите внимание на порядок следования байтов в памяти – младший байт находится по младшему адресу. Поэтому данные, которые определены как слова и имеют значение 0201, 0403, 0605 (переменные X, Y, Z) отображаются как 01 02 03 04. Переменные W0, W1 и TBAS определены как байты, поэтому в памяти они следуют как 07 08 и 31 (рис. 19).

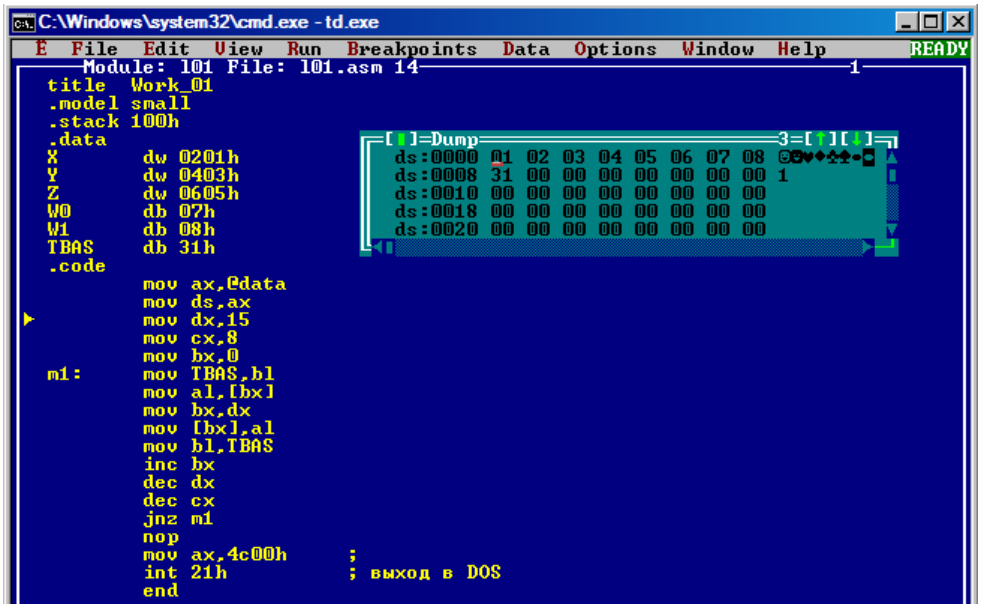

Рис. 19. Вид окна Dump с содержимым сегмента данных программы l01.exe.

По адресу ds:0000 – ds:0001 находится переменная X = 0201 (байты со значением 01 и 02).

По адресу ds:0002 – ds:0003 находится переменная Y = 0403 (байты со значением 03 и 04).

По адресу ds:0004 – ds:0005 находится переменная Z = 0605 (байты со значением 05 и 06).

По адресу ds:0006 находится переменная W0 = 07 (байт со значением 07).

По адресу ds:0007 находится переменная W1 = 08 (байт со значением 08).

По адресу ds:0008 находится переменная TBAS = 31 (байт со значением 31).

Программа, которая формирует массив из 8 чисел.

Перед тем, как приступить к работе с программами рассмотрим простой приём, который облегчит процесс компиляции и линковки (компоновки). Создадим файл, например Lab_assemble.txt, в файле напишем текст tasm.exe /z/zi lab1.asm. Затем переименуем расширение файла из txt в bat. Теперь для компиляции файла достаточно запустить bat файл. Для линковки создадим файл Lab_link.bat с текстом tlink.exe /v lab1.obj, lab1.exe.

Разберём подробно текст программы lab1.asm. В строке <1> память, выделяемая операционной системой DOS программе, распределяется в соответствии с моделью small. Модель памяти определяет число и расположение сегментов памяти для программы. В модели памяти Small сегмент кода (выполняемые команды) отделён от сегмента данных и стека. Данные и стек объединены в одну группу и поэтому соответствующие сегментные регистры DS и SS имеют одно и то же значение. Это наиболее распространённая модель памяти при разработке отдельных программ на языке ассемблера.

В строке <2> использована директива ассемблера, которая выделяет для стека 256 байт.

В строке <3> директива ассемблера определяет сегмент данных.

<4> в сегменте данных выделяется 8 байт, которые заполнены перечисленными значениями (0, 1, 2, 3, 4, 5, 6, 7)

<5> директива ассемблера определяет сегмент кода

<6> метка main:

<7> в регистр ax помещается адрес сегмента данных. Этот адрес назначается операционной системой.

<8> из регистра ax этот адрес помещается в регистр ds. Регистр ds предназначается для указания на сегмент данных. Прямой операции пересылки адреса в ds не существует, поэтому используется регистр ax, который здесь используется как промежуточный регистр для пересылки в ds.

<9> метка exit:

<10> в регистр ax помещается константа 4C00h

<11> вызывается прерывание с номером 21h (00 в al передаёт операционной системе 0, что трактуется как нормальное завершение программы.

<12> директива ассемблера end – конец компиляции

Текст программы lab1.asm приведён ниже:

```
<1>.model small ; память распределяется в соответствии с моделью small
<2>.stack 100h  ; стек – выделено 256 адресов
<3>.data        ; сегмент данных
<4>massive1 db 0h, 01h, 02h, 03h, 04h, 05h, 06h, 07h;
<5>.code        ; сегмент кода
<6>main:
<7>             mov ax,@data        ; точка входа в программу
<8>             mov ds,ax           ; в регистр ds смещение сегмента данных.
<9>exit:
<10>            mov ax,4c00h        ; стандартный выход
<11>            int 21h             ; для DOS
<12>            end                 ; директива ассемблера «конец компиляции»
```

Первая часть задания выполнена. Остаётся только правильно выполнить компиляцию, линковку (рис. 1 – 7), только вместо L01 везде нужно вписать Lab1. Для просмотра программы в отладчике нужно запустить программу td.exe (рис. 8) и посмотреть расположение данных massive1 в Turbo Debugger. Окно Dump после двух шагов (два нажатия на клавишу F8) также нужно настроить на начальный адрес сегмента данных (DS:0000).

Программа перестанови данных в обратном порядке

Эта программа является модифицированным вариантом Lab1.asm. В сегменте данных в строке 4 определён массив источник, который содержит цифры 0, 1, …7. В строке 5 определён массив приёмник, который заполнен ASCII кодом знака пробел; в этот массив будут помещены элементы массива source в обратном порядке. Переменная number равна длине массива source. Переменная second_number равна адресу последнего элемента двух массивов.

В сегменте кода строки 10 и 11 выполняют ту же функцию, что и строки 7, 8 в предыдущей программе. В 12 строке команда xor (логическое «исключающее или») очищает регистр ax (после выполнения команды ax = 00h). В строке 13 в регистр cx загружается число 8 – это количество элементов массива source. Далее, строка 14, в регистр si помещается адрес последнего элемента обоих массивов. В 15 строке в регистр bx помещается адрес начала массива source. 16 строка – метка. 17 стока – в регистр al помещаются данные, которые расположены по адресу, который находится в bx (mov al,[bx]), но в bx находится адрес нулевого элемента массива source (см. 15 строка). В 18 строке содержимое al располагается по адресу, который в si, а в si адрес последнего

элемента массива dest. В 20 строке адрес, который находится в bx, увеличивается на единицу, адресуя тем самым следующий элемент из таблицы source. В 21 строке si уменьшается на единицу, адресуя уже предпоследний элемент массива dest. В 22 строке уменьшается на единицу регистр cx, который служит счётчиком элементов таблицы. В строке 23 анализируется на ноль содержимое cx. Если cx не ноль (конец массива не достигнут) программа перенаправляется на метку source_to_aL. Если cx = 0, то перестановка массива закончена и программа завершается. Аналогичным образом (строки 24, 25 и 26) завершалась предыдущая программа.

Текст программы Lab2.asm приведён ниже:

```
<1>.model small
<2>.stack 100h
<3>.data
<4>source       db 0h, 01h, 02h, 03h, 04h, 05h, 06h, 07h
<5>dest         db 8 dup (" ")
<6>number       equ 8
<7>second_number equ 15
<8>.code
<9>main:
<10>            mov ax,@data
<11>            mov ds,ax
<12>            xor ax,ax
<13>            mov cx,number
<14>            mov si,second_number
<15>            lea bx,source
<16>source_to_aL:
<17>            mov al,[bx]             ; source -> al
<18>            mov [si],al
<19>next:
<20>            inc bx
<21>            dec si                  ; si--
<22>            dec cx                  ; cx--
<23>            jnz                     source_to_aL
<24>exit:
<25>            mov ax,4c00h
<26>            int 21h
<27>            end
```

Далее нужно компилировать исходный текст и линковать (компоновать) полученный объектный модуль (рис. 1 – 7), только вместо L01 везде нужно вписать Lab2. Для просмотра программы в отладчике нужно запустить программу td.exe (рис. 8). Окно Dump настраивается на DS:0000. На рисунке 21 показан вид отладчика, на момент завершения программы. В окне Dump при отладке (если Вы настроили окно после двух нажатий на F8) отображаются адреса, начиная с DS:0000. После остановки адреса приобретают вид 5F15:0000 – вместо DS подставляется 5F15 (у Вас может быть другое значение – этот адрес назначает операционная система). По адресам <DS:0000 – DS:007> расположены символы 0, 1, 2, 3, 4, 5, 6, 7 (справа рисунки, соответствующие ASCII кодам 0 – 7). В следующем диапазоне адресов <DS:0008 – DS:009> цифры 7, 6, 5, 4, 3, 2, 1, 0 (справа рисунки, соответствующие ASCII кодам 7 – 0). Это и есть результат работы программы Lab2.exe (рис. 20).

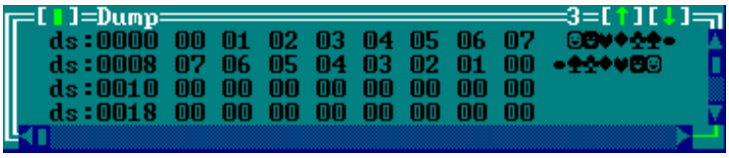

Рис. 20. Вид окна Dump с результатом работы программы.

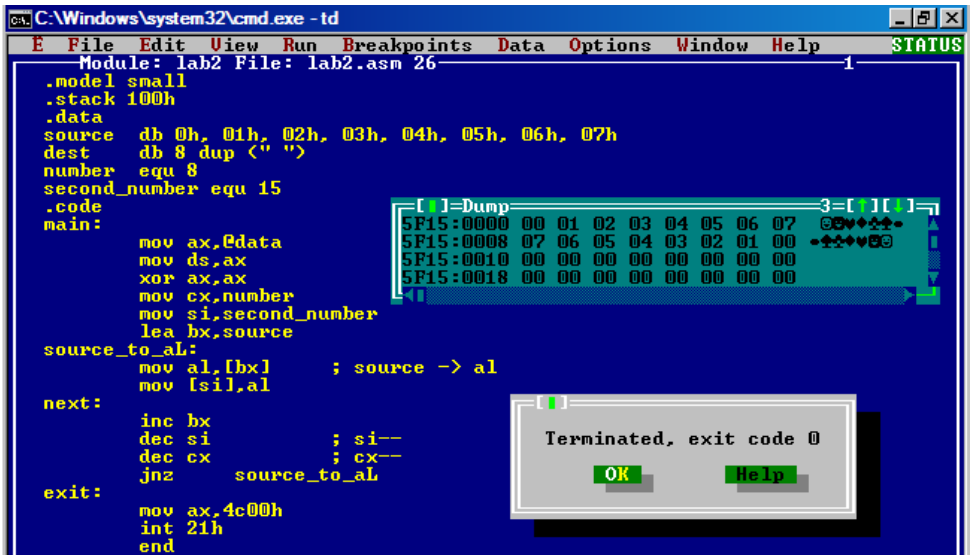

Рис. 21. Результат работы программы Lab2.exe

Программа умножения на 5 элементов массива dest.

Программа во многом похожа на предыдущие программы. Но в этой программе создаётся три массива source, temporary и dest. От метки source_to_aL до строки jnz source_to_aL, как и в предыдущей программе, производится перезапись данных из одного массива в другой в обратном порядке. Затем производится загрузка числа элементов массива в регистр cx. Затем в регистр si заносится адрес последнего элемента массива dest (dest_addr equ 23) который был посчитан заранее. В регистр bx заносится адрес нулевого элемента массива source. Теперь можно производить умножение элементов массива temporary на 5 и записывать их в массив dest. Следующий фрагмент программы аналогичен предыдущему, за исключением команды умножения mul multiplier. Здесь используется один из вариантов команды умножения целых чисел без знака. Формат команды mul:

mul <операнд>,

Если в качестве операнда используется восьмиразрядный регистр или байт в памяти, то в качестве первого операнда всегда используется al, а результат помещается в AX. В нашей программе именно так и происходит. Множитель определён как 5 (multiplier db 5). В следующей строке произведение помещается в место, адрес которого задан в регистре si. А в регистре si задаётся адрес последнего элемента массива dest. Потом происходит уменьшение адреса в si на единицу, и уменьшение на единицу числа обработанных элементов в регистре cx. Результат работы программы изображён на рисунке 22. В окне Dump числа отображаются в шестнадцатеричном формате.

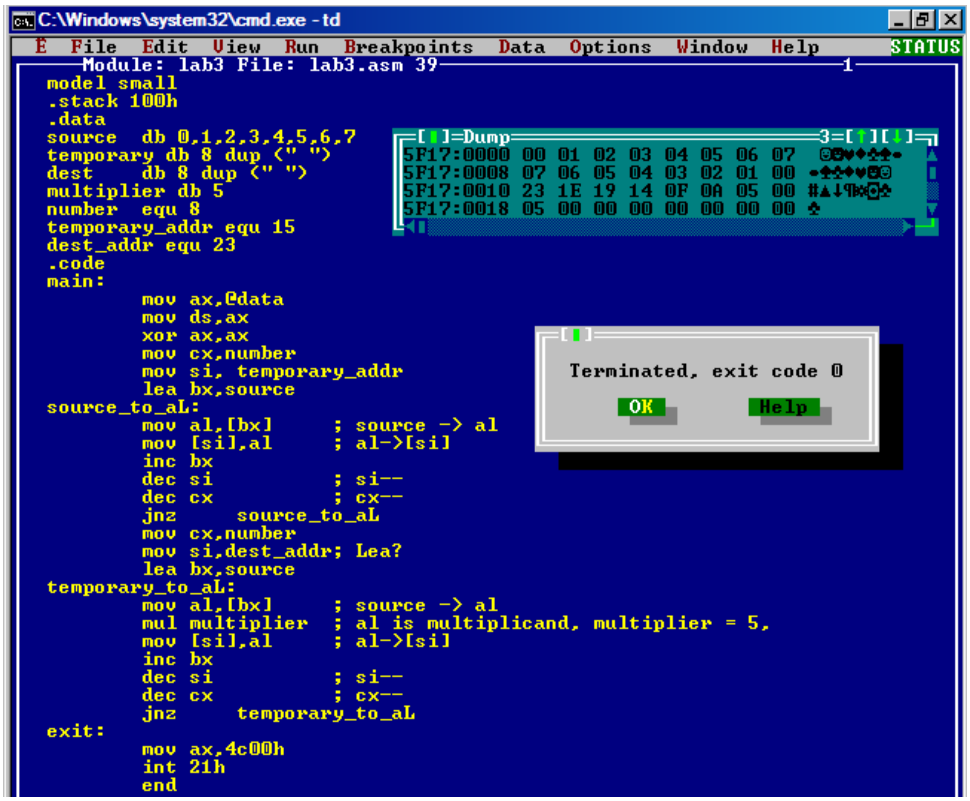

Рис. 22. Результат работы программы Lab3.

Проверить результат можно в калькуляторе Windows. Для этого нажмём кнопку пуск и выберем выполнить (рис. 1). В открывшемся окне наберём calc (рис. 23). Настроим калькулятор как инженерный (рис. 24).

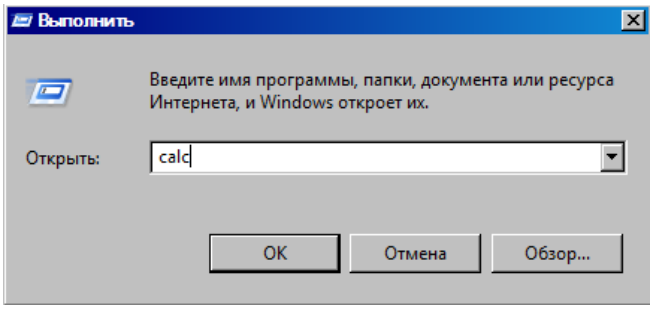

Рис. 23. Запуск калькулятора Windows.

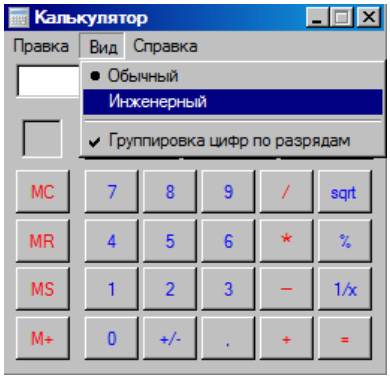

Рис. 24. Настройка калькулятора.

Для проверки перемножим число, например 7, на 5 и переведём это число в шестнадцатеричный вид (рис. 25). Для этого достаточно поставить точку напротив Hex. Десятичное число 35 становится шестнадцатеричным 23.

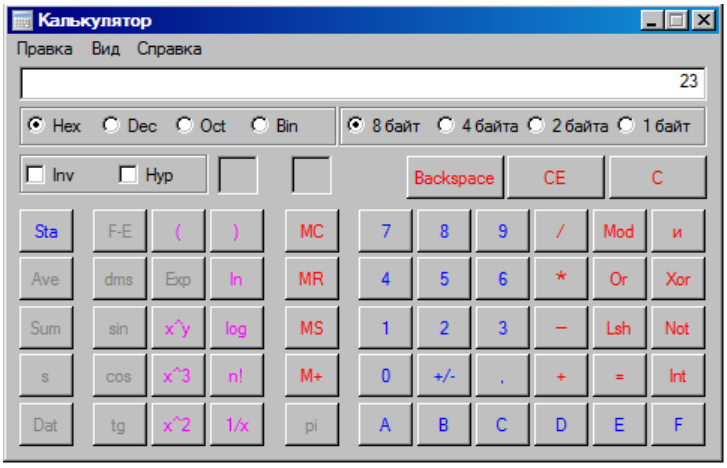

Рис. 25. Перевод из десятичной в шестнадцатеричную систему счисления.

Составим таблицу, в которой переведём десятичные числа (произведения первой строки на пять) в шестнадцатеричный вид.

Табл. 1. Перевод произведений в шестнадцатеричный вид.

Число	Произведение на пять – десятичное число	Шестнадцатеричное число
0	0	0h
1	5	5h
2	10	Ah
3	15	Fh
4	20	14h
5	25	19h
6	30	1E
7	35	23

Текст программы приведён ниже:

```
model small
.stack 100h
.data
source          db 0,1,2,3,4,5,6,7
temporary       db 8 dup (" ")
dest            db 8 dup (" ")
multiplier      db 5
number          equ 8
temporary_addr  equ 15
dest_addr       equ 23
.code
main:
                mov ax,@data
                mov ds,ax
                xor ax,ax
                mov cx,number
                mov si, temporary_addr
                lea bx,source
```

```
source_to_aL:
                mov al,[bx]             ; source -> al
                mov [si],al             ; al->[si]
                inc bx
                dec si                      ; si--
                dec cx                      ; cx--
                jnz                     source_to_aL
                mov cx,number
                mov si,dest_addr
                lea bx,source
temporary_to_aL:
                mov al,[bx]             ; source -> al
                mul multiplier          ; al is multiplicand, multiplier = 5,
                mov [si],al             ; al->[si]
                inc bx
                dec si                      ; si--
                dec cx                      ; cx--
                jnz                     temporary_to_aL
exit:
                mov ax,4c00h
                int 21h
                end
```

Программа подсчёта нулей в 5-м элементе модифицированного массива

Текст программы нужно откомпилировать и скомпоновать (линковать). Для этого из командной строки нужно сначала перейти в каталог asm, для чего набрать

cd c:\asm

Запустить компилятор:

tasm.exe /z/zi Lab4.asm

Затем запустить линковщик:

tlink.exe /v Lab4.obj, Lab4.exe

После того, как получен исполняемый модуль (программа exe), запустим td и откроем файл Lab4.exe. Настроим td на отображение в режиме 43/50 линий на экране (рис. 10 и рис. 11). Изменим размеры окна Module. Для изменения размеров окна нужно навести курсор мыши в нижний правый угол окна, зажать левую кнопку мыши на одиночной линии и, не отпуская левой кнопки, поднять вверх нижнюю часть окна (рис. 26).

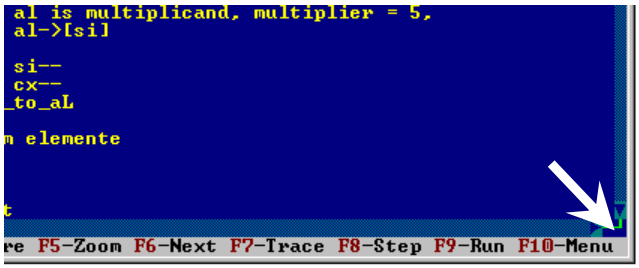

Рис. 26. Изменение размеров окна.

Теперь откроем окно Dump (рис. 16). И разместим его ниже окна Module (рис. 27).

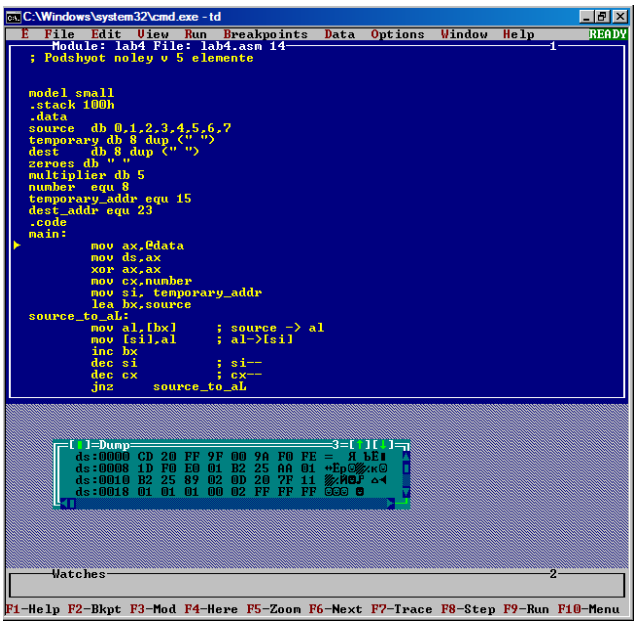

Рис. 27. Размещение окна Dump.

Ещё в этой работе понадобится наблюдать содержимое регистров процессора. Откроем окно Registers (рис. 28). И расставим эти окна ниже окна Module так, чтобы все три окна не перекрывались и были постоянно видны (рис. 29).

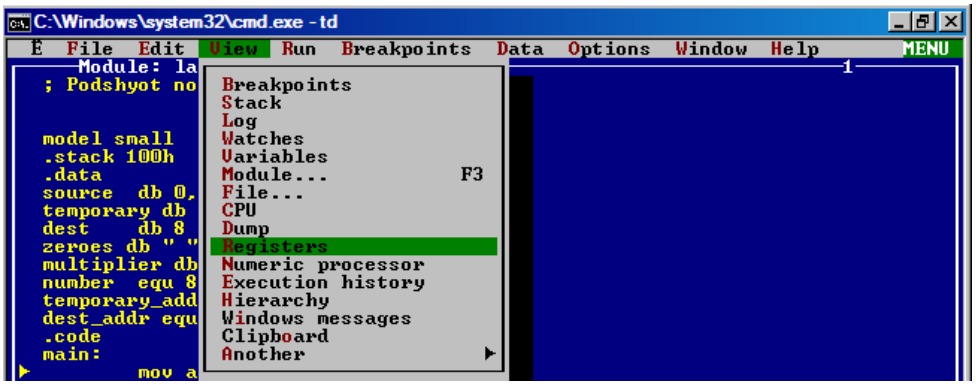

Рис. 28. Открытие окна Registers.

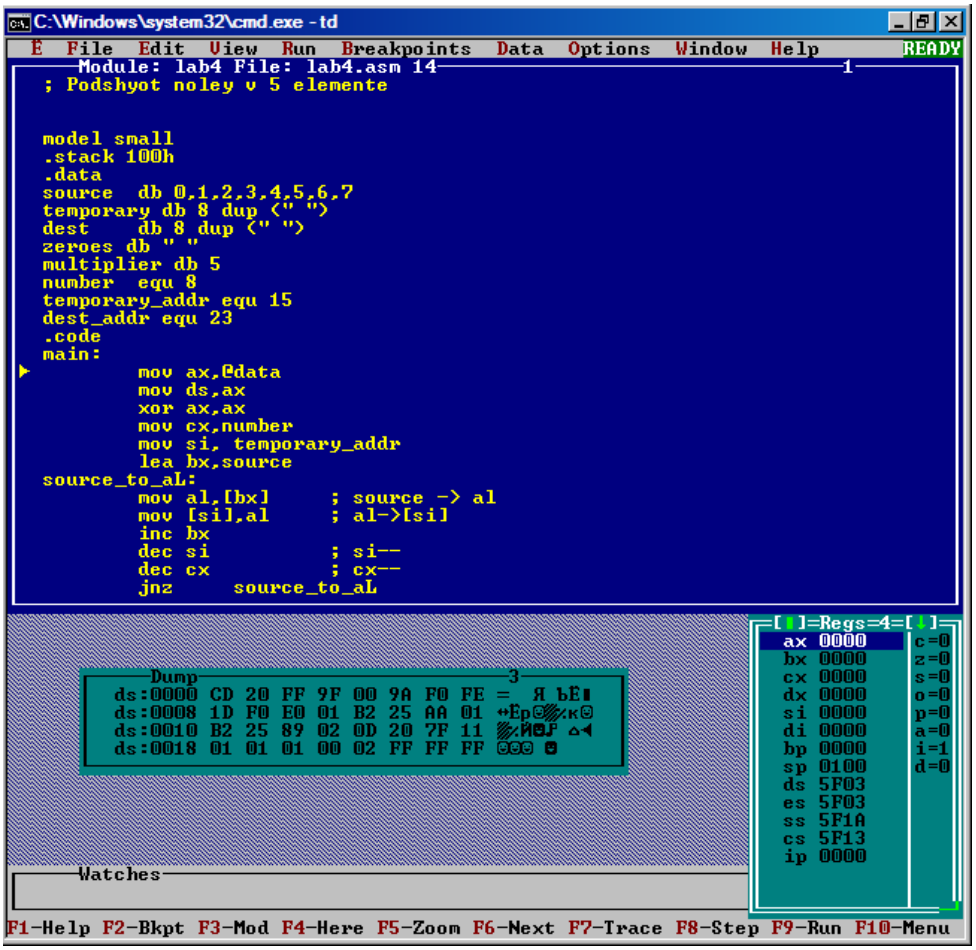

Рис. 29. Вид настроенного отладчика.

Теперь два раза нажмём F8 и настроим окно Dump для наблюдения содержимого сегмента данных (рис. 17, рис. 18). Для отображения сегмента данных, как обычно, в строке goto нужно указать ds:0000. Как и в предыдущих двух работах расположение данных по адресам следующее:

ds:0000 – массив source – источник, массив, который содержит начальные данные.

ds:0008 – массив temporary – массив, который хранит данные из source, расположенные в обратном порядке.

ds:0010 – массив dest – массив, в котором находятся данные из temporary, причём каждый элемент массива умножен на пять.

ds:0018 – байт zeroes, который после окончания программы будет содержать данные о количестве нулей в пятом элементе массива.

Перед началом выполнения программы в Байт zeroes занесена буква z, для того, чтобы было легче отыскать его в сегменте данных.

После метки podschyot_noley в регистр cx помещается число, которое определено как 8 (numberequ 8) в самом начале программы и соответствует числу бит в байте. Регистр cx используется совместно с командой loop в качестве счётчика. Команда loop имеет следующий формат:

loop, <offset>

при выполнении этой команды сначала осуществляется уменьшение содержимого регистра cx на 1, затем содержимое этого регистра сравнивается с нулём. Если оно не равно нулю, то осуществляется близкий переход к началу цикла, если же оно равно нулю, цикл заканчивается и выполняется следующая команда.

Затем регистр bx обнуляется (в нём мы будем считать количество нулей – сначала количество нулей ноль). Затем в режим si помещается адрес начала массива dest. Затем из адреса [si+4], что соответствует пятому элементу, в регистр al помещается содержимое ячейки памяти. То есть в al помещается содержимое ячейки памяти по адресу offset dest + 4 – там расположен пятый элемент.

Содержимое этих регистров процессора нужно просматривать в окне Registers турбо дебаггера. После метки sdvig происходит сдвиг регистра al вправо на одно место. Если последним была единица, то устанавливается флаг C. Флаги также просматривают в окне Registers. По команде jc осуществляется условный переход. Логика перехода такова: если флаг C установлен (крайнее правое число было единицей), то команда увеличения регистра bx на единицу пропускается. А в регистре bx как раз и ведётся подсчёт нулей. Затем выполняется команда loop до тех пор, пока cx не обнулится, поскольку регистр cx используется совместно с командой loop в качестве счётчика. Команда loop имеет следующий формат:

loop, <offset>

при выполнении этой команды сначала осуществляется уменьшение содержимого регистра cx на 1, затем содержимое этого регистра сравнивается с нулём. Если оно не равно нулю, то осуществляется близкий переход к началу цикла (к метке sdvig), если же оно равно нулю, то в ячейку памяти zeroes помещается содержимое регистра bl. Фрагмент текста программы, начиная с метки podschyot_noley, приведён ниже:

```
podschyot_noley:
                mov             cx,number
                mov             bx,0
                mov             si,offset dest
                mov             al,[si+4]
sdvig:
                shr             al,1
                jc              no_figure_one
                inc             bx
no_figure_one:
                loop            sdvig
                mov             zeroes,bl
```

Результат работы программы изображён на рисунке 30. По адресу 5F18:0018 расположено содержимое байта zeroes (у вас вместо 5F18 может быть другое число, так как этот адрес назначается операционной системой), который после окончания программы содержит данные о количестве нулей в пятом элементе массива. Количество нулей составило 4. И действительно, если перевести 0F в двоичную систему при помощи калькулятора Windows (рис. 23 – рис. 25, только выбрать надо сначала Hex, набрать данные, а потом выбрать Bin), мы увидим, что 0F это 1111. Но старшая тетрада содержит нули и их как раз четыре. Полный формат пятого элемента массива (байта) 0000 1111.

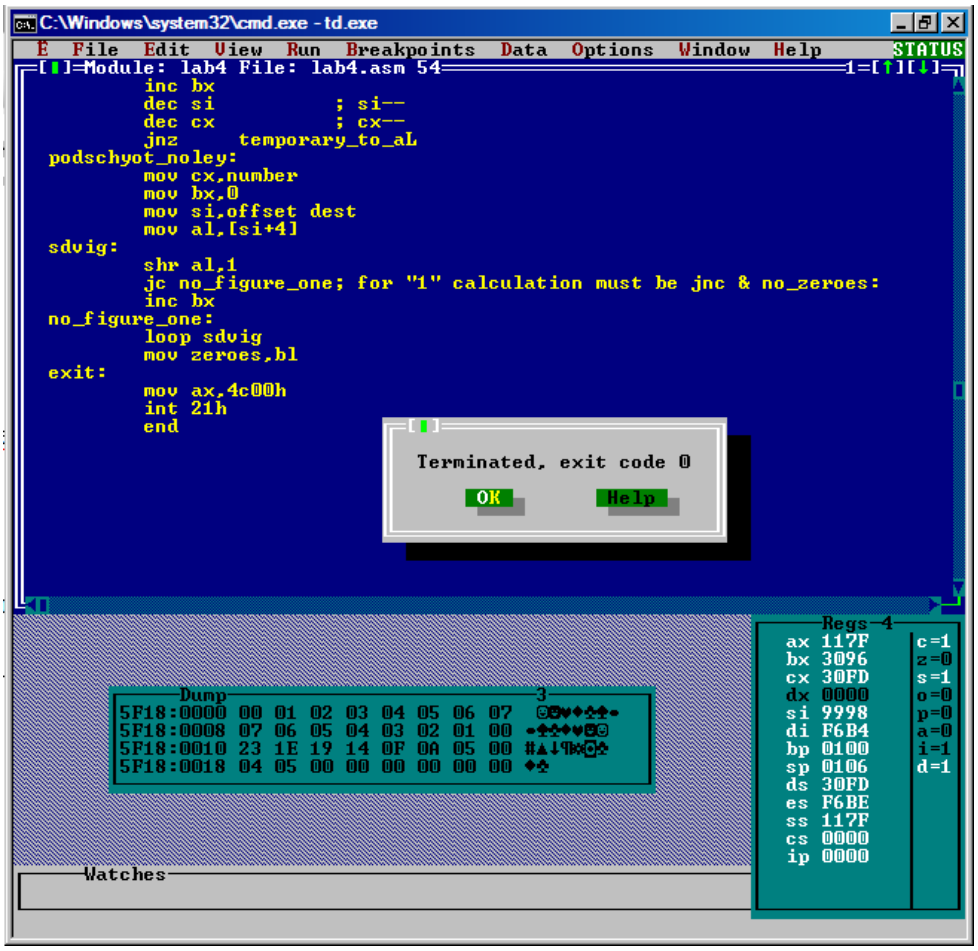

Рис. 30. Результат работы программы Lab4.exe.

Программа выделения тетрады во втором элементе массива

Текст программы Lab5.asm нужно откомпилировать и скомпоновать.

Для компиляции исходного текста программы на ассемблере нужно запустить программу TASM.EXE. В дальнейшем предполагается отладка программы в программе TD.EXE, поэтому при компиляции нужно добавить отладочную информацию в выполняемый модуль *.exe; для этого компилятору TASM нужно задать ключ /zi. Поэтому в командной строке следует набрать текст:

tasm.exe /zi Lab5.asm, Lab5.obj

Для получения исполняемого файла (*.exe) из объектного файла (*.obj) нужно запустить программу TLINK.EXE. В дальнейшем предполагается отладка программы в программе TD.EXE, поэтому при компиляции нужно добавить отладочную информацию в выполняемый файл *.exe; для этого линковщику нужно задать ключ /v. Поэтому в командной строке следует набрать такой текст:

tlink.exe /v Lab5.obj, Lab5.exe

Затем запустить Турбо Дебаггер и открыть исполняемый модуль Lab5.exe. После этого нужно настроить отладчик – открыть окно Dump (рис. 31) и окно Registers (рис. 32).

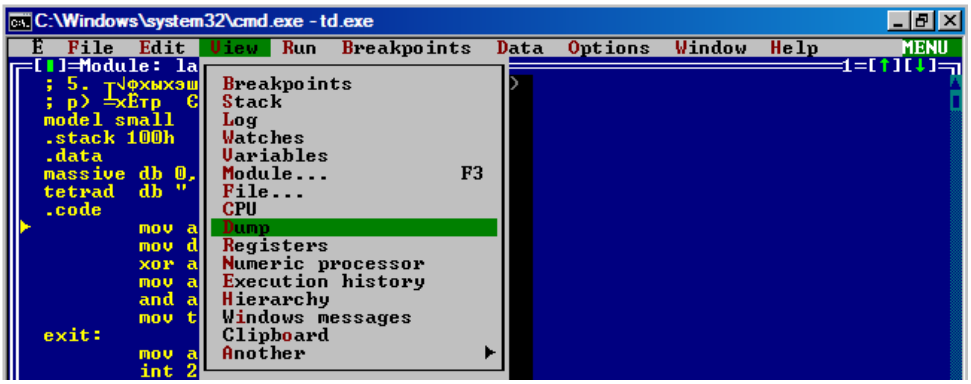

Рис. 31. Открывание окна Dump.

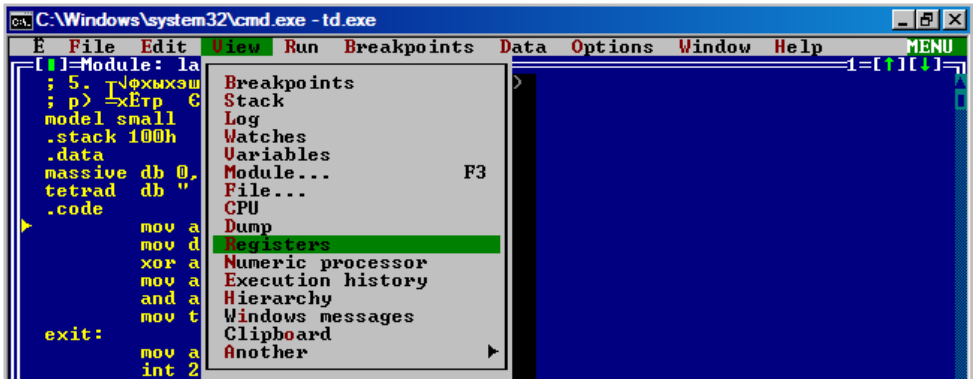

Рис. 32. Открывание окна Registers (внутренние регистры процессора).

Вид настроенного отладчика показан на рис. 33.

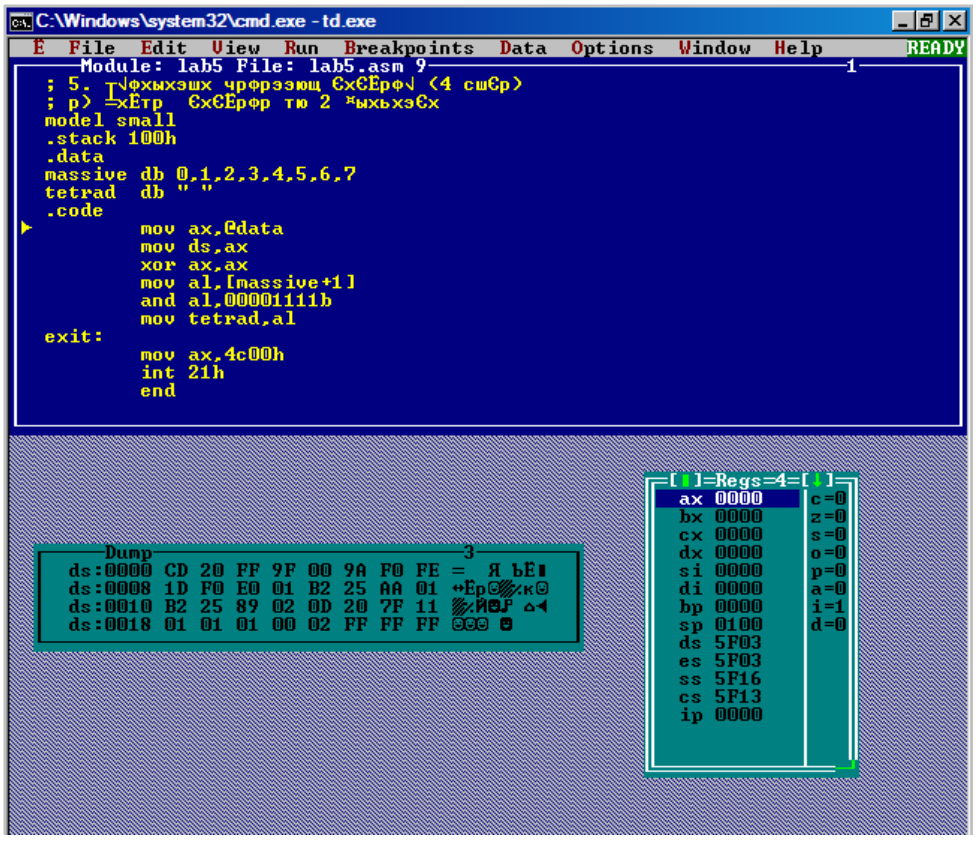

Рис. 33. Вид настроенного отладчика.

В программе сначала определяется модель памяти типа small. Затем выделяется пространство для стека размером в 256 байт. В сегменте данных определяется массив massive, состоящий из чисел 0,1,2…7. После этого определяется байт с названием tetrad. В этот байт будет помещёна тетрада из второго элемента. Затем следует сегмент кода. В первых двух строках сегмента кода настраивается регистр DS, он должен указывать на начало сегмента данных. В третьей строке производится очистка регистра AX. Затем в байт AL помещается копия содержимого памяти из адреса [massive+1]. Этот адрес соответствует второму элементу (адресация ведётся с нуля, поэтому второй элемент имеет адрес 1).

Теперь выделим тетраду из байта. Вспомним, чем байт отличается от тетрады. Байт – это восемь бит, а тетрада – это четыре бита. По определению получается, что байт состоит из двух тетрад. В задание не указано старшую или младшую тетраду следует выделить из байта. Выделим, например младшую тетраду. Для этого в строке

```
and al,00001111b
```

производим логическое «или» регистра AL и числа 00001111b (в регистре AL уже находится копия второго элемента массива). Так, старшие четыре байта обнулятся, и происходит выделение младшей тетрады копии второго элемента массива (копия находилась в регистре AL). В следующей строке содержимое AL перемещается в ячейку памяти с названием tetrad. То есть в ячейке tetrad находится младшая тетрада второго элемента массива. После метки exit следует стандартное завершение exe программы. Результат работы программы показан на рис. 34. Текст программы приведён ниже:

```
; Lab5. Выделение заданной тетрады (4 бита)
; a) Первая тетрада во 2 элементе
model small
.stack 100h
.data
massive         db 0,1,2,3,4,5,6,7
tetrad          db " "
.code
                mov ax,@data
                mov ds,ax
                xor ax,ax
                mov al,[massive+1]
                and al,00001111b
                mov tetrade, al
exit:
                mov ax,4c00h
                int 21h
                end
```

В результате работы программы появляется число 1 в ячейке tetrad, которая расположена по адресу 5F15:0008 (У Вас может быть число отличное от 5F15 – этот адрес назначает операционная система, именно он попадает в DS в первых двух строках сегмента кода).

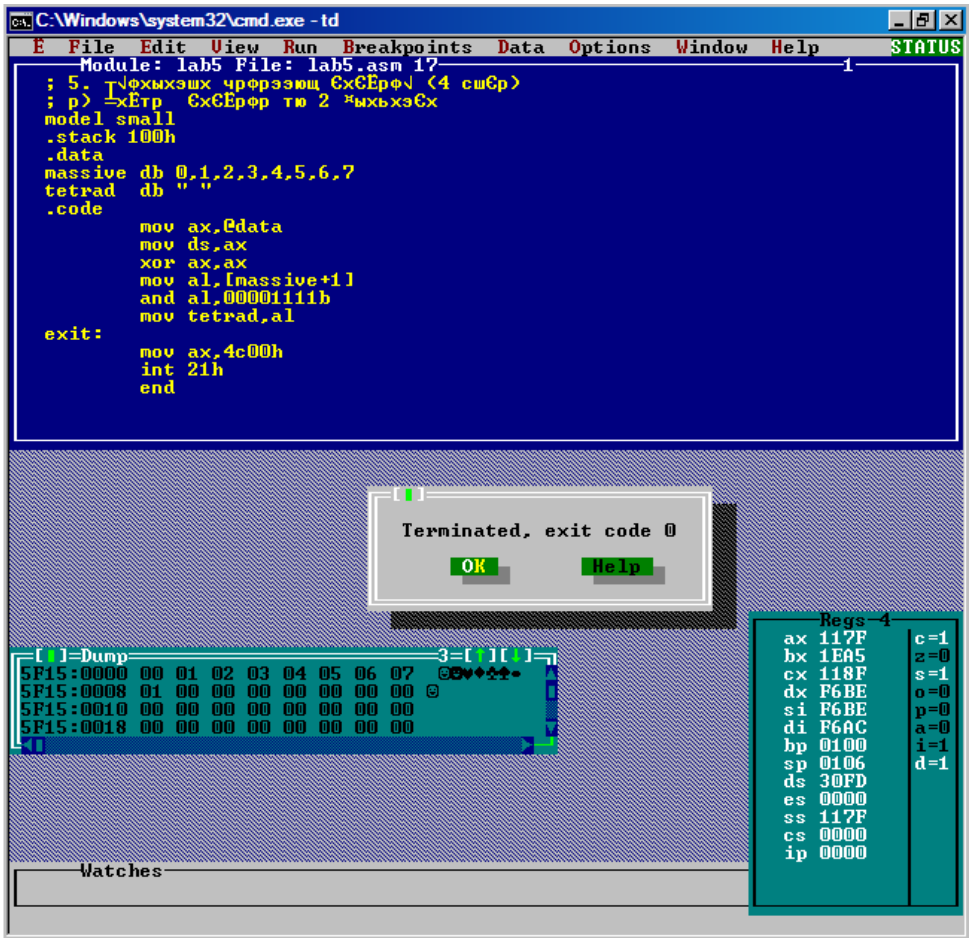

Рис. 34. Результат работы программы Lab5.exe

Более наглядный вариант программы выделения тетрады во втором элементе массива

Текст программы более подходящий для исследований, предложен в файле Lab5_1.asm. Рассмотрим этот текст:

```
; 5. Выделение заданной тетрады (4 бита)
; a) Первая тетрада во 2 элементе
model small
.stack 100h
.data
massive         db 37h,36h,35h,34h,33h,32h,31h,30h
tetrad          db " "
number          dw 3
maska equ 00001111b
.code
                mov ax,@data
                mov ds,ax
                xor ax,ax
                mov bx,number
                mov al,[bx]
                and al,maska
                mov tetrad,al
exit:
                mov ax,4c00h
```

int 21h
end

Основное отличие заключается в данных массива massive – это ASCII коды символов 7 – 0. Почему ASCII коды? Ответ прост – чтобы не утрачивалась наглядность. Ведь байт состоит из двух тетрад, а если старшая тетрада равна 0000, то невозможно увидеть, как она отбрасывается. Поэтому был выбран вариант, когда старшая тетрада равна 3, а младшая 4.

Следующие отличия – в AL помещается число по адресу из BX. Этот адрес задаётся в переменной number. Получается, что переменная number определяет адрес элемента в массиве massive (в этой программе 4-й элемент). Далее, логическая операция «AND» производится над регистром AL и переменной maska. Таким образом, переменная maska определяет биты, выделяемые в байте. Для выделения старшей тетрады в переменную maska нужно занести «1111 0000». Далее выделяем в регистре AL тетраду и содержимое этого регистра помещаем в ячейку памяти tetrad.

Рассмотрим таблицу истинности для битовой логической операции «AND»:

Таблица истинности для операции логическое «AND»

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

Теперь рассмотрим таблицу для логической операции «AND» регистра AL и переменной maska:

Таблица истинности для операции «AND» регистра AL и переменной maska

Регистр AL двоичное значение	Регистр AL шестнад-цатеричное значение	Переменная maska двоичное значение	Переменная maska шестнад-цатеричное значение	Результат в регистре AL двоичное значение	Результат в регистре AL шестнад-цатеричное значение
0	3	0	0	0	0
0		0		0	
1		0		0	
1		0		0	
0	4	1	F	0	4
1		1		1	
0		1		0	
0		1		0	

В турбодебаггере эту операцию можно наблюдать в окне Registers при выполнении операции

and al,maska

При этом следует помнить, что курсор показывает на следующую команду. Содержимое окна Registers до выполнения команды показано на рис. 35, после выполнения команды на рис. 36. На рис. 37 показано содержимое переменной tetrad после помещения в неё содержимого регистра AX.

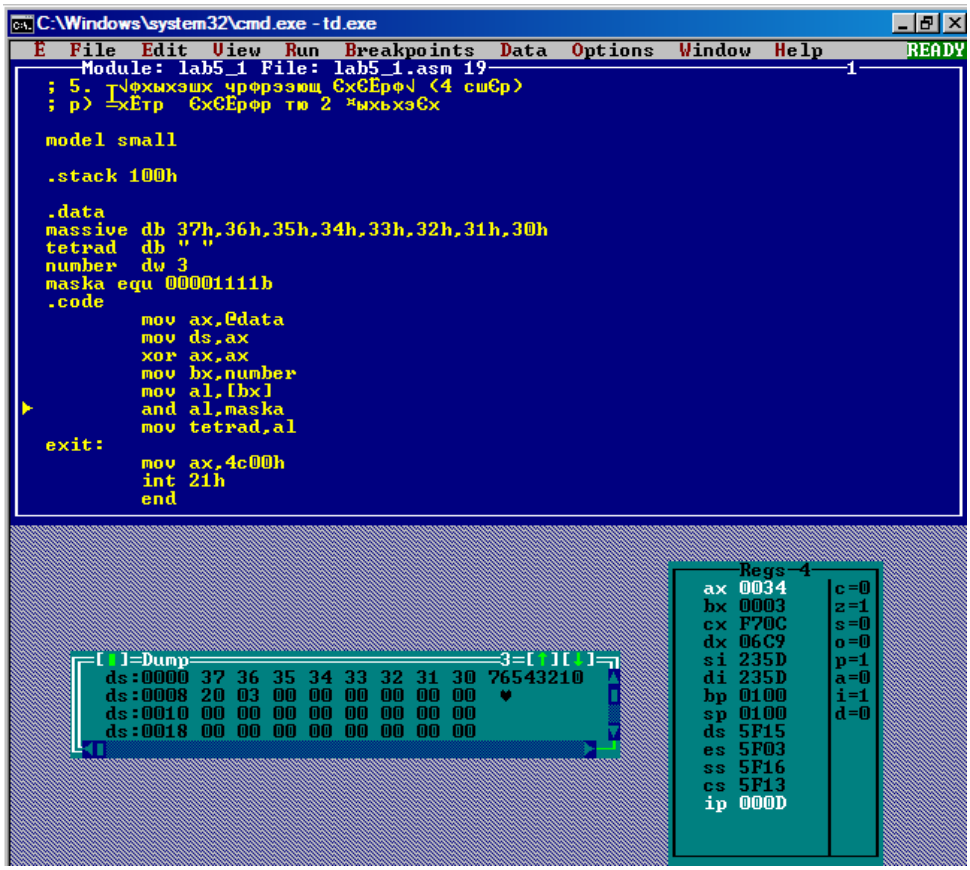

Рис. 35. Окно регистров процессора до выполнения команды (AL – младший байт или две правые цифры в регистре AX).

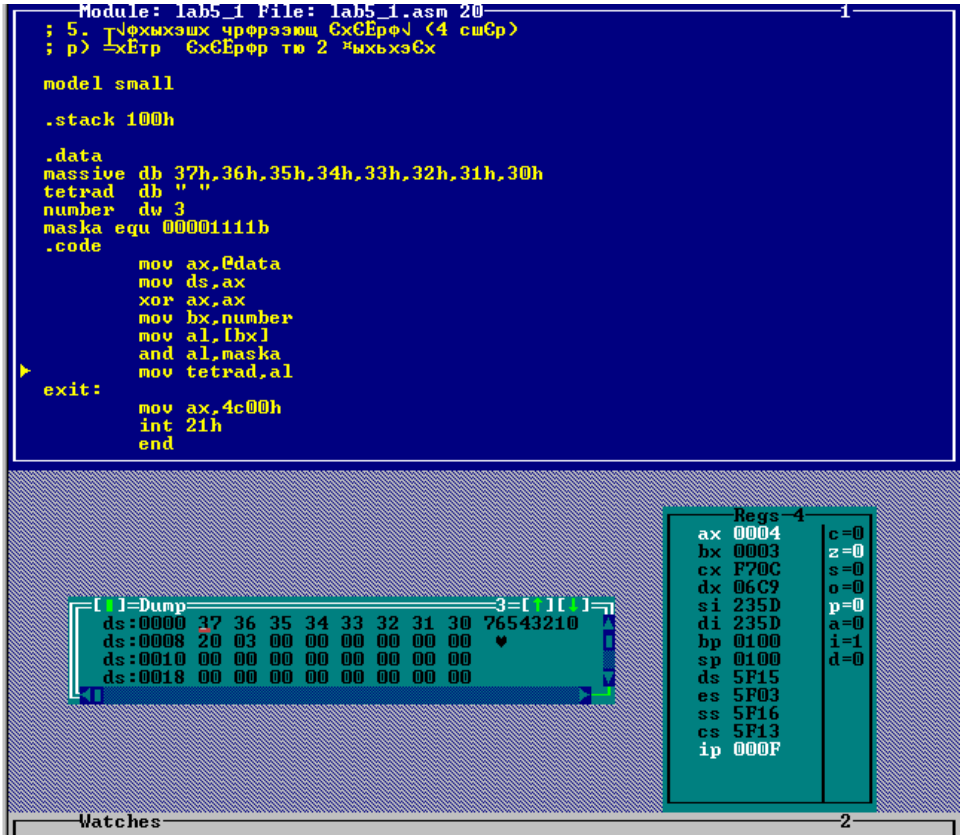

Рис. 36. Окно регистров процессора после выполнения команды (AL – младший байт или две правые цифры в регистре AX).

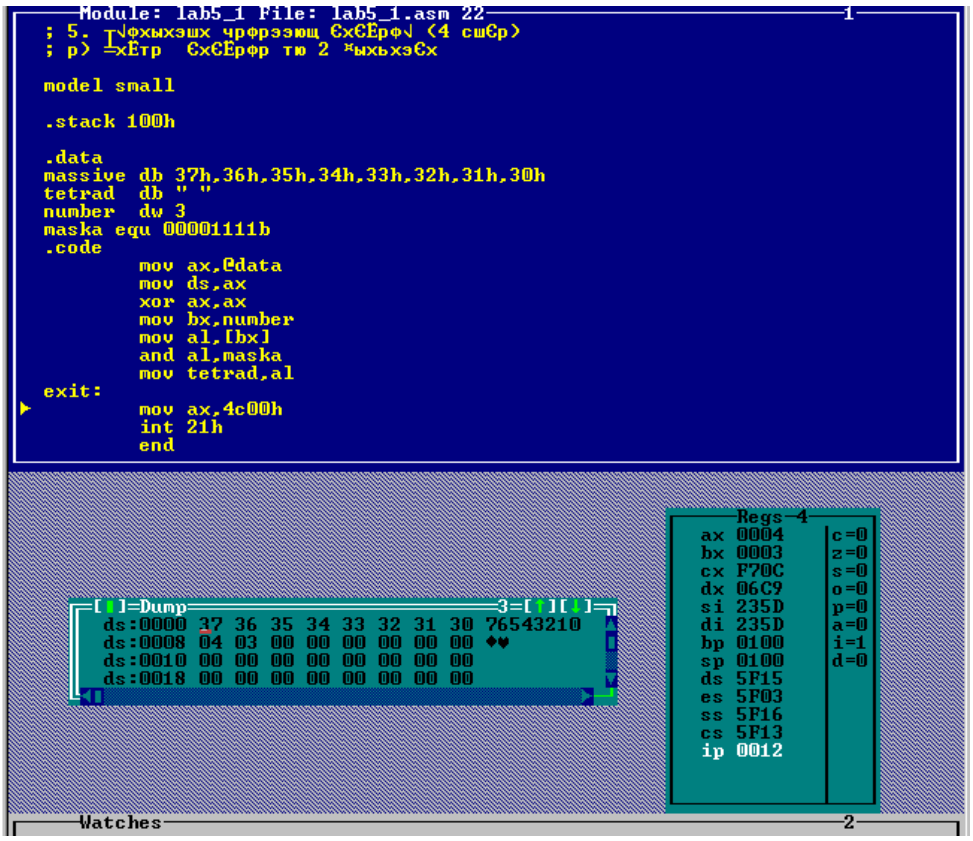

Рис. 37. Содержимое ячейки tetrad (ds:0008)
стало равно содержимому регистра AX.

Оглавление:

Рисунки: