Лабораторная работа №1

Разработка программ на ассемблере.
Работа с отладчиком x32dbg, пакетом masm32

Цель работы: получить навыки создания простейших ассемблерных программ с использованием пакета masm32 и научиться пользоваться отладчиком x32dbg.

Теоретические сведения

Язык «ассемблер» представляет собой низкоуровневый язык программирования, оперирующий отдельными командами процессора. Существует большое количество разновидностей данного языка, каждый из которых соответствует определённой архитектуре процессора. К примеру, обычный компьютер пользователя, построенный на базе процессора Intel или AMD, имеет архитектуру x86 (ранняя) или x64 (поздняя); телефон или смартфон имеет ARM-архитектуру и может быть построен, к примеру, на базе процессора Qualcomm Snapdragon. Каждая архитектура программируется своим ассемблером, соответствующим только её системе команд. Большое количество микроконтроллеров различных производителей также имеют свои разновидности ассемблера. Можно сказать проще: язык ассемблера – это такой язык, который понимает процессор.

В данном курсе будет изучен ассемблер архитектуры x86 и её расширение x64. Программный код на любом языке высокого уровне в конечном итоге компилируется в последовательность инструкций процессора или ассемблерных команд. Основное преимущество ассемблера перед другими языками программирования – скорость выполнения программ. Конечно, данное преимущество обеспечивается в основном квалификацией программиста. Второе преимущество – малый объём итогового программного кода. Программы, написанные на ассемблере, обычно имеют меньший размер по сравнению с программами на Java, C# и другими языками.

Существуют следующие основные сферы программирования, в которых используется язык ассемблера:

1. Оптимизация программного кода. Если требуется ускорить выполнение некоторых критически важных участков кода или таких, которые выполняются наибольший процент времени, можно написать их с помощью ассемблера. Например, можно оптимизировать математические вычисления или работу с оперативной памятью. Также могут возникать ситуации, когда компилятор не поддерживает какие-либо инструкции процессора. Тогда программе можно «объяснить», что она должна делать только ассемблерными инструкциями или вовсе последовательностью машинного кода (т.е. последовательностью байтов определённой длины).
2. Анализ машинного кода. Если интересно знать, как работает какая-либо программа, но нет её исходного кода, то единственный способ её исследовать – использовать отладчик, который покажет из каких ассемблерных инструкций (или дизассемблирует) она состоит.

3. Разработка компиляторов. Задача любого компилятора – транслировать текст программы на языке высокого уровня в набор ассемблерных инструкций. Процессоры всё время усложняются, в них появляются новые инструкции. Практически ежегодно выходят новые версии стандартов и языков программирования. Всё это требует от программистов разработки новых компиляторов и хорошее знание ассемблера, чтобы соответствующий компилятор создать.
4. Информационная безопасность операционных систем и антивирусная защита. Разработка программного обеспечения для обнаружения вирусов основана на анализе машинного кода исполняемых программ. Большое количество уязвимостей операционных систем эксплуатируется программным кодом, написанным на ассемблере.
5. Разработка драйверов устройств и обмен информацией с периферийными устройствами.

Ассемблер является довольно специфическим языком программирования и применяется в тех случаях, когда это действительно обосновано. Например, нет смысла писать оконное приложение или компьютерную игру полностью на ассемблере. Потому что и объём программного кода, и затраченное время будет очень большим. В основном, его используют, когда нужно написать небольшой объём очень эффективного кода.

Masm32 (Microsoft Assembler) – низкоуровневая среда разработки 32-разрядных приложений на языке ассемблер для операционной системы Windows. Masm32 включает в себя большое количество библиотек, в том числе для работы с консолью и оконными приложениями.

Программист, использующий ассемблер имеет в своём распоряжении следующие аппаратные ресурсы:

1. Восемь 32-разрядных регистров центрального процессора для 32-разрядной архитектуры и шестнадцать 64-разрядных регистров для 64-разрядной;
2. Регистры сопроцессора для выполнения операций с вещественными числами;
3. 2 Гб оперативной памяти (предназначена для размещения переменных, стека, динамического выделения памяти) для 32-разрядной архитектуры и весь доступный объём для 64-разрядный (теоретическое ограничение – порядка 2^{64} байт).

Регистры процессора являются наиболее «быстрой» памятью, т.к. расположены непосредственно на самом кристалле процессора. Но их количество очень мало и поэтому регистры используется непосредственно при выполнении операций с данными, а пока они (данные) не нужны, они хранятся в оперативной памяти.

В 32-разрядной архитектуре доступны для использования следующие регистры: EAX, EBX, ECX, CDX, ESI, EDI, EBP, ESP. В ранних процессорах, которые работали в ОС MS-DOS, архитектура была 16-разрядной и регистры общего назначения получили следующие названия:

AX – аккумулятор (Accumulator),
BX – базовый регистр (Base register),
CX – счётчик (Counter)
DX – регистр данных (Data register),
BP – указатель на участок памяти (Base pointer),
SI – индексный регистр источника (Source index),
DI – индексный регистр приёмника (Destination index),
SP – указатель на вершину стека (Stack pointer).

Позднее, в 32-разрядных процессорах эти регистры были расширены до 4 байт (32 бит) и получили приставку «E» (от англ. Extended, т.е. расширенные). В современных процессорах регистры имеют ещё больший размер – 64 бита (8 байт). Название 64-разрядного регистра начинается с буквы «R». У всех регистров, представленных ниже, похожая структура. Отличие заключается в том, что у некоторых из них недоступен младший байт. Зелёным цветом показаны регистры, доступные только в 64-разрядных архитектурах.

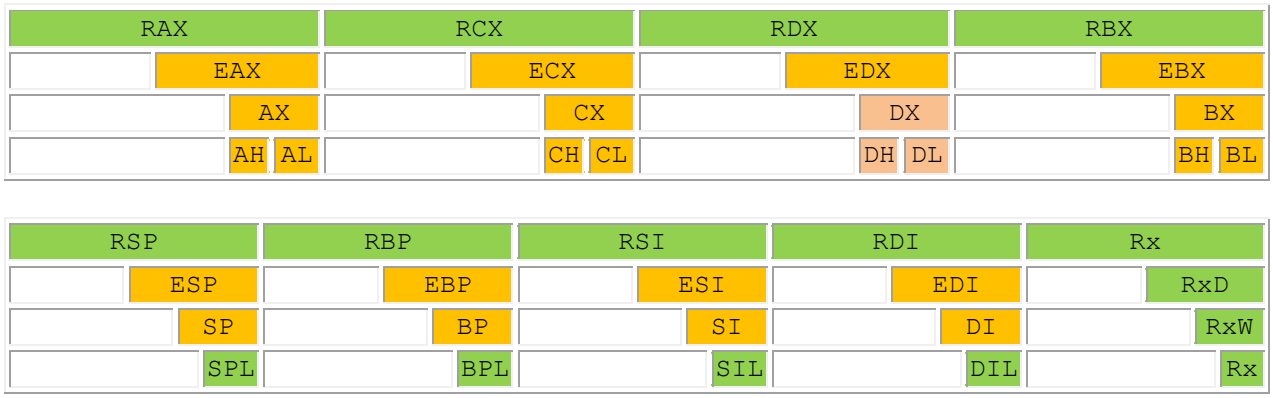
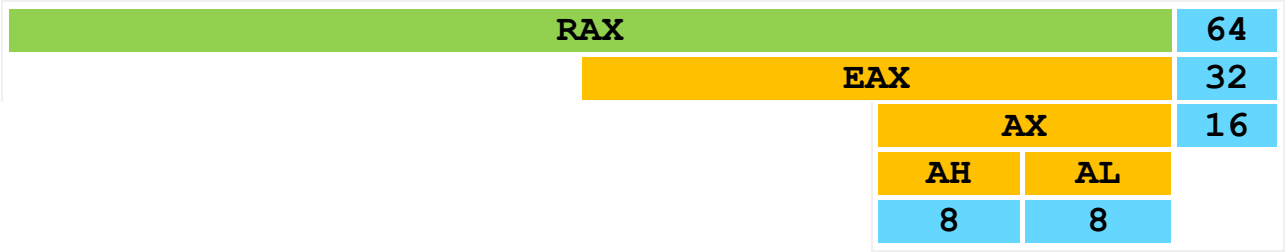


Таблица 1. Описание регистров центрального процессора

Ниже показаны размеры составных частей регистров на примере регистра RAX:



Буквы «L» и «H» обозначают соответственно младшую (Low, регистры AL, BL, CL, DL) и старшую (High, регистры AH, BH, CH, DH) части 16-разрядного регистра. Регистры с окончанием «L» и «H» имеют размер 8 бит (1 байт).

EAX, EBX, ECX, EDX – регистры общего назначения. ESP (Stack Pointer) – регистр указатель на вершину стека. EBP (Base Pointer) можно использовать по собственному усмотрению, но обычно он используется для работы со стеком. ESI, EDI – индексные регистры, которые используются при работе с массивами. EIP (Instruction Pointer) – указатель на текущую выполняемую инструкцию.

В 64-разрядной архитектуре доступны ещё 8 регистров общего назначения R0-R7.

Значения в регистрах обычно рассматривают в шестнадцатеричной системе счисления. Удобство шестнадцатеричной системы счисления состоит в том, что в неё очень легко можно переводить двоичные числа (и в обратную сторону тоже). Четыре разряда двоичного числа (тетрада) представляются одним разрядом шестнадцатеричного. Для перевода достаточно разбить число на группы по 4 бита и заменить каждую тетраду соответствующей шестнадцатеричной цифрой.

Двоичная тетрада	Шестнадцатеричная цифра
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Программа на ассемблере состоит из следующих сегментов (областей памяти): сегмента данных, сегмента кода, сегмента стека. Ниже приведён пример простой программы на ассемблере, которая выводит на консоль сумму двух чисел типа unsigned char:

```
 .686                    ; Тип процессора
 .model flat, stdcall    ; Модель памяти и стиль вызова подпрограмм
 option casemap: none    ; Чувствительность к регистру

  ; --- Подключение файлов с кодом, макросами, константами, прототипами
функций и т.д.
 include windows.inc
 include kernel32.inc
 include msvcrt.inc
 includelib kernel32.lib
 includelib msvcrt.lib

  ; --- Сегмент данных ---
 .data
      a db 127     ; Первое число
      b db 128     ; Второе число
      sum db 8 dup(?)   ; Буфер для хранения строки с суммой чисел a и b
      message db "It is assembler sum of unsigned char numbers", 0

  ; --- Сегмент кода ---
 .code
 start:
      MOV AX, 0           ; AX = 0
      MOV AL, a           ; AL = a
      MOV BL, b           ; BL = b
      ADD AL, BL          ; AL = AL + BL
```

```
        MOV ECX, 10         ; ECX = 10

        DIV CL              ; AL = AX / 10, AH = AX % 10
        MOV sum[2], AH      ; sum[2] = a % 10

        MOV AH, 0
        DIV CL              ; AL = AX / 10, AH = AX % 10
        MOV sum[1], AH      ; sum[1] = a / 10 % 10

        MOV sum[0], AL      ; sum[0] = a / 100

        ; Нужно добавить код цифры '0', чтобы преобразовать цифры от 0 до 9 в
символы от '0' до '9'
        ADD sum[0], '0'     ; sum[0] += '0'
        ADD sum[1], '0'     ; sum[1] += '0'
        ADD sum[2], '0'     ; sum[2] += '0'
        MOV sum[3], 0       ; sum[3] = 0        ; Признак конца строки - нулевой
символ

        push offset sum
        call crt_puts       ; puts(sum)
        ADD ESP, 4  ; Очистка стека от аргумента

        push offset message
        call crt_puts       ; puts(message)
        ADD ESP, 4  ; Очистка стека от аргумента

        call crt__getch ; Задержка ввода, getch()
        ; Вызов функции ExitProcess(0)
        push 0      ; Поместить аргумент функции в стек
        call ExitProcess ; Выход из программы
 end start
```

`.686` – директива задания системы команд, которая указывает, что в программе будут использоваться команды процессора Pentium 6 (Pentium Pro, Pentium II).

`.model flat` – директива, указывающая модель памяти. При разработке программ для Windows используется сплошная (flat) модель памяти.

`stdcall` – это стиль вызова подпрограмм. Аргументы для данного стиля передаются справа налево, а вызываемая функция сама освобождает стек. В программах на ассемблере подпрограммы вызываются с помощью команды **call**. Аргументы при этом передаются в стек с помощью команды **push**. Выполнение двух последних команд «`push NULL`» и «`call ExitProcess`» равнозначно вызову ExitProcess(0), например, на языке C++. Для вывода строки на экран используется функция puts из библиотеки языка Си. В следующей таблице показано, как выполнить один и тот же вызов в разных языках программирования:

Ассемблер	Си

`push offset str`
`call crt_puts`
`ADD ESP, 4`
	puts(str);

`push NULL` `call ExitProcess`	ExitProcess(0);

Чувствительность к регистру в данном случае присутствует, т.е. «FIRE» и «fire» будут восприниматься компилятором по-разному.

Для задания размеров переменным в сегменте данных используются следующие директивы:

Директива	Размер	
db	1	Байт (BYTE)
dw	2	Слово (WORD)
dd	4	Двойное слово (DOUBLE WORD)
dq	8	Учетверённое слово (QUAD WORD)
df	6	6 байт
dt	10	10 байт (TEN BYTE)

В приведённом выше примере сегмент данных содержит только строки и два однобайтовых числа *a* и *b*. Создадим произвольный сегмент данных, который включает строки, массивы, целые и вещественные числа:

```
.data
a       dd 500000
x       dq -2500000
b       dd 3.0
m1      db 1, 2, 3, 4, 5, 6, 7, 8, 9, 0Ah, 0Bh, 0Ch
        dw -3, -2, -1, 0, 1, 2, 3
d1      dd 10100101001101b, 0AB10005h
m2      df 5 dup (11)
strs    db "some string", 13, 10, 0
m3      dq 1.0, 300h, -500
d2      df ?
```

В ассемблере названия переменных ассоциируются с адресами ячеек памяти, в которых хранятся данные.

Для понимания, как хранятся целые числа в памяти, нужно учитывать два обстоятельства:

1. Числа хранятся в памяти в дополнительном коде.
2. Младший байт числа хранится по младшему адресу.

Отпечаток памяти (hex dump), который соответствует содержимому секции «.data», имеет вид:

Адрес	Шестнадцатеричное												ASCII
00403000	20 A1 07 00	60 DA D9 FF	FF FF FF FF	00 00 40 40	¡.. `ÚÙÿÿÿÿ..@@
00403010	01 02 03 04	05 06 07 08	09 0A 0B 0C	FD FF FE FF	............ýÿþÿ
00403020	FF FF 00 00	01 00 02 00	03 00 4D 29	00 00 05 00	ÿÿ........M)....
00403030	B1 0A 0B 00	00 00 00 00	0B 00 00 00	00 00 0B 00	±..............
00403040	00 00 00 00	0B 00 00 00	00 00 0B 00	00 00 00 00	....
00403050	73 6F 6D 65	20 73 74 72	69 6E 67 0D	0A 00 00 00	some string.....
00403060	00 00 00 00	F0 3F 00 03	00 00 00 00	00 00 0C FE	....ð?.........þ
00403070	FF FF FF FF	FF FF 00 00	00 00 00 00	00 00 00 00	ÿÿÿÿÿÿ..........
00403080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
00403090	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
004030A0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....

Рассмотрим подробнее, как интерпретировать содержимое ячеек памяти.

```
a dd 500000
```

4-байтовая переменная *a* располагается в начале сегмента по адресу 0x00403000. Она имеет значение $500000_{10} = 7A120_{16} = 00\ 07\ A1\ 20_{16}$. Поскольку байты числа хранятся в памяти в обратном порядке, младший байт – «20_{16}» располагается по младшему адресу 0x00403000, второй байт «$A1_{16}$» – по адресу 0x00403001, третий байт «07_{16}» – по адресу 0x00403002, старший байт «00_{16}» – по адресу 0x00403003. Можно считать, что переменная *a* имеет тип int или unsigned int.

```
x dq -2500000
```

Со следующего адреса 0x00403004 начинается 8-байтовая отрицательная переменная $x = -2500000$. Чтобы преобразовать её в дополнительный код, нужно число 2500000 расширить до 8 байт:

$$2500000_{10} = 2625A0_{16} = 00\ 00\ 00\ 00\ 00\ 26\ 25\ A0_{16},$$

затем инвертировать его биты в двоичном (или 16-ричном) представлении и добавить единицу:

$$\underbrace{00}_{1\text{ байт}}\ \underbrace{00}_{1\text{ байт}}\ \underbrace{00}_{1\text{ байт}}\ \underbrace{00}_{1\text{ байт}}\ \underbrace{00}_{1\text{ байт}}\ \underbrace{26}_{1\text{ байт}}\ \underbrace{25}_{1\text{ байт}}\ \underbrace{A0}_{1\text{ байт}}{}_{16} +1 =$$

$$= \text{FF FF FF FF FF D9 DA } 5\text{F}_{16} + 1 = \text{ FF FF FF FF FF D9 DA } 60_{16}.$$

В памяти байты данного числа располагаются в обратном порядке: «60 DA D9 FF FF FF FF FF», начиная с адреса 0x00403004. Значение переменной x соответствует типу long long.

```
b dd 3.0
```

Переменная b представляет собой 4-байтовое вещественное число (типа float) и располагается по адресу 0x0040300C.

```
m1 db 1, 2, 3, 4, 5, 6, 7, 8, 9, 0Ah, 0Bh, 0Ch
```

Массив $m1$, заполненный однобайтовыми числами от 1 до 12, имеет размер 12 байт и начинается с адреса 0x00403010. Последний элемент массива $m1$ располагается по адресу 0x0040301B, следовательно, следующее двухбайтовое число -3 расположено по адресу 0x0040301C. Тип элементов массива $m1$ – char или unsigned char.

```
dw -3, -2, -1, 0, 1, 2, 3
```

Область памяти 0x0040301C-0x00403029 никак не именована (не ассоциируется с какой-либо переменной) и содержит последовательно двухбайтовые числа от -3 до 3. Отрицательные числа из этого массива также кодируются в дополнительном коде. Поскольку в массиве есть отрицательные числа, базовый тип должен быть знаковым. Двухбайтовое знаковое – short.

В массиве $d1$ первое число записано в двоичной системе счисления (буква «b» в конце), второе – в 16-ричной (буква «h» в конце):

```
d1 dd 10100101001101b, 0AB10005h
```

4-байтовые числа $10100101001101_2 = 00\ 00\ 29\ 4\text{D}_{16}, 0\text{A B1 00 }05_{16}$ массива $d1$ начинаются с адреса 0x0040302A. Элементы массива d1 можно считать принадлежащими типу int или unsigned int . Далее в памяти следует массив $m2$ из пяти 6-байтовых целых чисел, заполненный числами 11 (для этого используется ключевое слово «dup»):

```
m2 df 5 dup (11)
```

Это объявление равнозначно следующему:

```
m2 df 11, 11, 11, 11, 11
```

Строки заключаются в кавычки. Они обязательно должны заканчиваться ноль-символом также, как и в языке Си. В строке strs присутствуют символы перевода строки (10) и возврата каретки (13):

```
strs    db "some string", 13, 10, 0
```

Область памяти 0x00403050-0x0040305A заполняется ASCII–кодами символов строки strs.

```
m3 dq 1.0, 300h, -500
```

В массиве $m3$, начиная с адреса 0x0040305E, размещаются три 8-байтовых числа разного типа. Вещественное число 1.0 типа double располагается по адресу 0x0040305E, целые числа 300_{16} и -500 начинаются с адреса 0x00403066.

6-байтовая переменная $d2$ не инициализирована:

```
d2 df ?
```

Для наглядности раскрасим в сегменте данных каждую переменную или массив отдельным цветом:

00403000	20 A1 07 00	60 DA D9 FF	FF FF FF FF	00 00 40 40	¡..`ÚÙÿÿÿÿ..@@
00403010	01 02 03 04	05 06 07 08	09 0A 0B 0C	FD FF FE FF	............ýÿþÿ
00403020	FF FF 00 00	01 00 02 00	03 00 4D 29	00 00 05 00	ÿÿ........M)....
00403030	B1 0A 0B 00	00 00 00 00	0B 00 00 00	00 00 0B 00	±..............
00403040	00 00 00 00	0B 00 00 00	00 00 0B 00	00 00 00 00	....
00403050	73 6F 6D 65	20 73 74 72	69 6E 67 0D	0A 00 00 00	some string....
00403060	00 00 00 00	F0 3F 00 03	00 00 00 00	00 00 0C FE	....ð?.........þ
00403070	FF FF FF FF	FF FF 00 00	00 00 00 00	00 00 00 00	ÿÿÿÿÿÿ..........

Осуществлять перевод вещественных чисел в двоичное представление вручную сложно, потому что формат их представления сложнее, чем у целых чисел (см. учебник В.И. Юрова «Assembler», стр. 458). Для такого перевода при необходимости лучше написать отдельную программу.

Компиляция программ, написанных на ассемблере, осуществляется с помощью командной строки в 2 этапа. Для этого необходимы следующие исполняемые файлы, которые находятся в каталоге masm32\bin:

ml.exe – транслятор. Он преобразует исходный текст в obj-файл (объектного формата COFF или OMF),

link.exe – компоновщик. Он создаёт исполняемый exe или dll-модуль.

Для автоматизации сборки и запуска ассемблерных программ с использованием masm32 лучше создать файл с расширением *.bat с примерным скриптом:

```
del lab1.exe
set masm32_path=d:\masm32
%masm32_path%\bin\ml /c /coff /I "%masm32_path%\include" lab1.asm
%masm32_path%\bin\link /SUBSYSTEM:CONSOLE /LIBPATH:%masm32_path%\lib
lab1.obj
pause
lab1.exe
pause
```

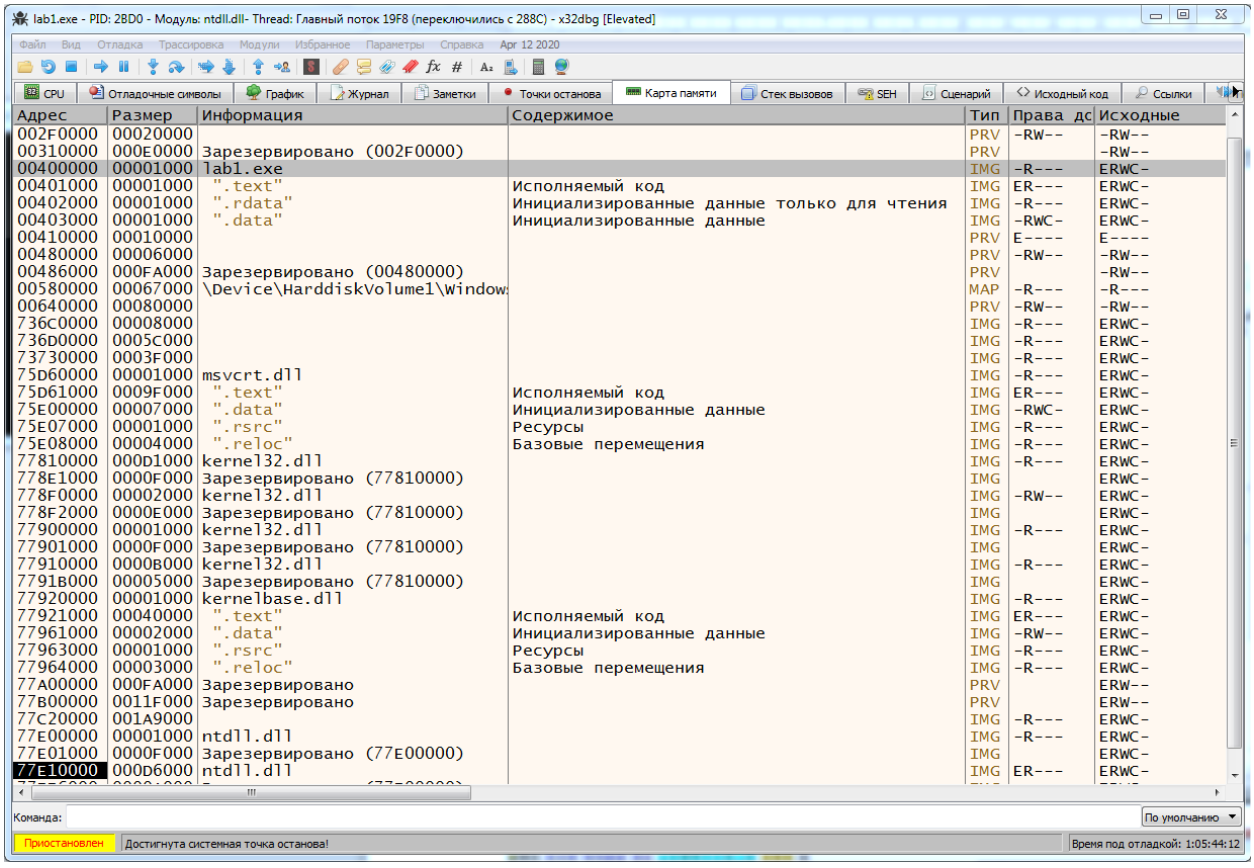
При этом следует в переменной masm32_path указать правильный путь к каталогу masm32 на конкретном компьютере. В данном скрипте строка %masm32_path% будет всюду разворачиваться в значение «d:\masm32». Это можно увидеть в консоли, запустив данный скрипт. Исходный код программы при этом необходимо предварительно сохранить в файле lab1.asm. Если всё сделано правильно, то после запуска данного скрипта в каталоге с asm-файлом должны появиться файлы с расширением *.obj и *.exe. Если файлы не были созданы, значит в программе имеются синтаксические ошибки или неправильно заданы команды для компиляции (скорее всего неверно задан каталог masm32_path). Сообщения о синтаксических ошибках (номер строки и код ошибки) следует смотреть в этой же консоли. К примеру, следующее сообщение транслятора говорит о том, что имеется ошибка в 21 строке:

```
lab1.asm(21) : error A2044: invalid character in file
```

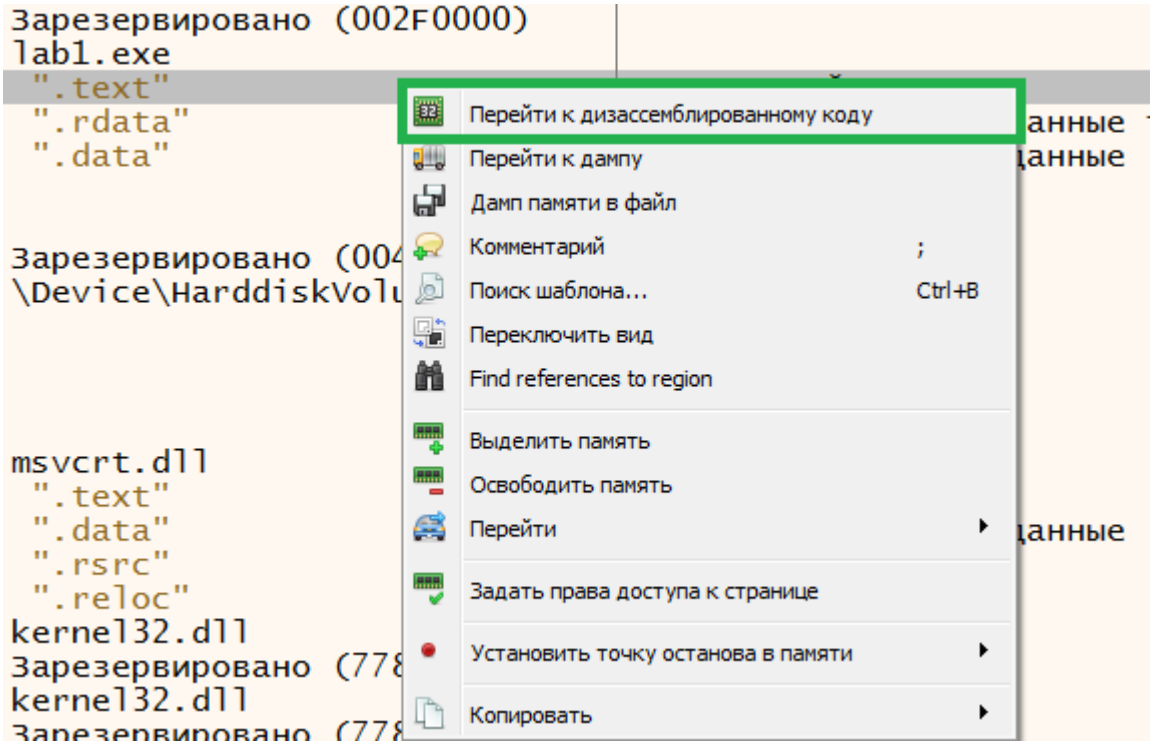
Для написания программ на ассемблере лучше использовать какой-нибудь блокнот с подсветкой синтаксиса, например, Notepad++.

Поскольку полученный исполняемый файл lab1.exe является 32-разрядным его необходимо открыть в отладчике x32dbg (клавиша **F3**). Для начала следует перейти на вкладку «Карта памяти». Карта памяти содержит информацию о загруженных в процесс dll-библиотеках и об основном исполняемом exe-файле. Видно, что в процесс загружены исполняемые модули msvcrt.dll и kernel32.dll, которые были подключены ранее в файле с исходным кодом. Компилятор masm32 размещает основной модуль lab1.exe по адресу 00400000_{16}. По данному адресу размещается информация о количестве секций исполняемого модуля и другая служебная информация. По данной таблице видно, что модуль lab1.exe состоит из трёх секций: «.text», «.rdata», «.data». Каждая секция имеет свой 32-разрядный адрес в 16-ричной системе счисления, который записан в первой колонке, и размер. Каждая секция представляет собой область памяти, содержащая определённую информацию, необходимую для выполнения программы. Секция «.text» соответствует сегменту кода и содержит последовательность ассемблерных инструкций, которая в исходном asm-файле следует после ключевых слов «.code». Сегмент данных содержит глобальные переменные программы. В нашем случае, это переменные *a*, *b*;

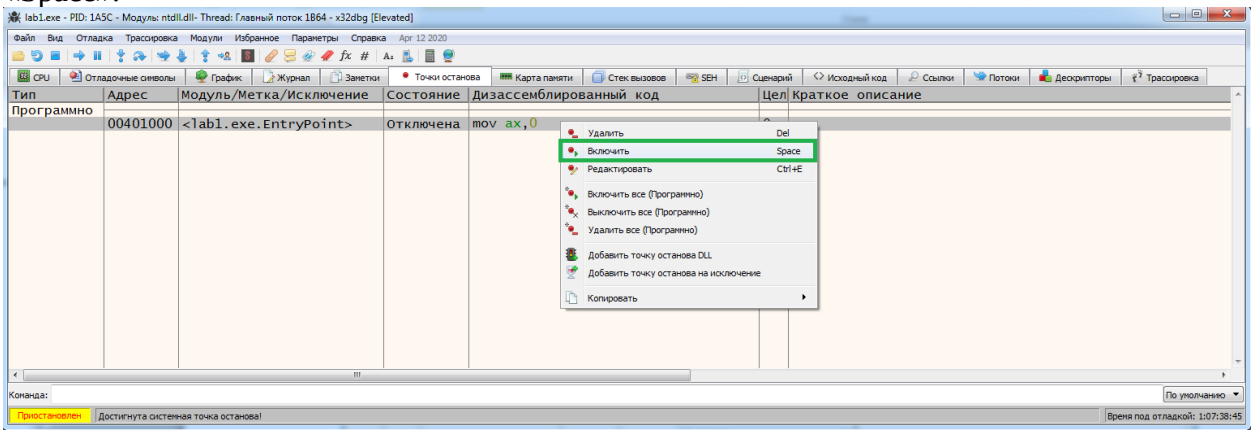
строки *sum* и *message*. Секция «.rdata» содержит данные, доступные только для чтения, которые нельзя изменить. В данной секции может располагаться как служебная информация, например, таблицы импорта и экспорта, так и статические данные пользователя. Каждая секция имеет размер, равный целому количеству страниц. Можно посчитать, что размер секции «.text» – 1000_{16} байт или 4 килобайта. Даже если в секции полезной информации не хватает до её полного заполнения, оставшаяся часть секции заполняется нулями. Также каждая секция имеет заранее заданные права доступа. Каждая буква характеризует определённое право доступа к странице: **E** – исполнение кода (execute), **R** – чтение данных (read), **W** – запись данных (write), **C** – копирование данных (copy). Если попытаться выполнить какое-либо действие, на которое недостаточно прав, то произойдёт аварийное завершение программы. К примеру, нельзя изменять сегмент кода или исполнять инструкции, которые заданы в сегменте данных.



Далее следует выделить строку, соответствующую секции «.text», и нажать клавишу Enter (можно выбрать пункт контекстного меню «Перейти к дизассемблированному коду» или выполнить двойной щелчок мыши). Аналогичным образом нужно выбрать секцию «.text», нажать клавишу Enter или выбрать пункт контекстного меню «Перейти к дампу» (можно также выполнить двойной щелчок мыши).

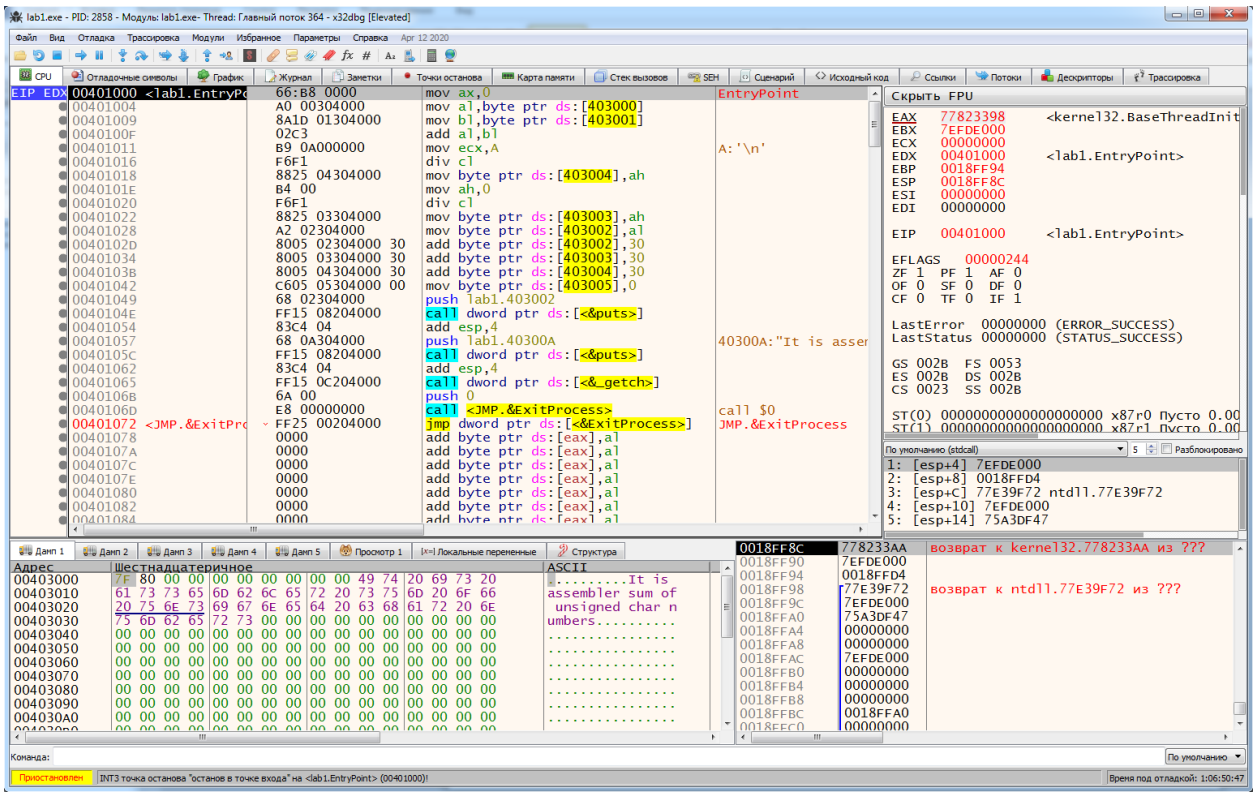


После этого вкладка «CPU» отображает основные секции модуля lab1.exe. Чтобы выполнить программу пошагово (трассировать), нужно сначала убедиться, что отладчик поставил точку останова в точке входа (Entry Point). Для этого следует переключиться на вкладку «Точки останова» и включить её с помощью контекстного меню или клавишей «Space».



Если данная таблица пустая, то нужно переключиться на вкладку «CPU», найти и выделить первую инструкцию своей программы, а затем нажать клавишу **F2**. После этого точка останова будет поставлена и появится во вкладке «Точки останова». Обычно во время отладки требуется создавать несколько точек останова. Все их можно удалять (Del) или временно включать/отключать (Space) во вкладке «Точки останова».

Нажав клавишу «**F9**», выполним программу до инструкции по адресу 0x00401000. Данный адрес называется точкой входа в программу (или EntryPoint), начальным адресом программы, по которому операционная система передаёт управление после того, как она загрузит исполняемый файл в оперативную память.



Окно отладчика «CPU» состоит из 4 основных частей. В левом верхнем углу отображаются команды. В трёх колонках содержится информация об адресах команд (`Address`), их коды (`Hex dump`) в шестнадцатеричном представлении, и мнемоническое описание (`Command`). В окне, содержащем команды, можно по значению регистра EIP найти текущую выполняемую команду.

Правая верхняя часть отображает содержимое регистров и флагов процессора.

Нижняя левая часть изначально содержит информацию о *глобальных данных*, а именно о глобальных переменных, массивах, строках и т.д. Но в общем, здесь можно увидеть содержимое участка памяти по любому интересующему адресу (комбинация клавиш Ctrl+G). Первая колонка (`Address`) – начальный адрес блока размером 16 байт. Следующие четыре колонки (`Hex dump`) содержат значения этой последовательности из 16 байтов. Вся информация выводится в шестнадцатеричном представлении. Последняя колонка (`ASCII`) содержит символьное представление последовательности байтов из предшествующих колонок.

В правой нижней части окна изображён стек. Стек предназначен для передачи аргументов в процедуры и функции, хранения адресов возврата из подпрограмм и для хранения локальных переменных и массивов. При вызове подпрограмм её аргументы помещаются в стек. Содержимое стека отображается с адреса, находящегося в регистре ESP.

32-разрядный регистр флагов EFLAGS содержит несколько флагов, основные из которых:

CF – флаг переноса,
ZF – флаг нуля,
OF – флаг переполнения,
PF – флаг чётности,
SF – флаг знака,
TF – флаг трассировки,
AF – вспомогательный флаг переноса,
DF – флаг направления,
IF – флаг прерывания.

Каждый флаг представляет собой регистр, который может иметь состояние 0 или 1. Флаги устанавливаются командами в зависимости от результата их выполнения.

x32dbg содержит следующие основные команды для отладки программ:

F7 – выполнить одну команду с заходом в подпрограмму;

F8 – выполнить одну команду с обходом подпрограммы;

F2 – создать или удалить точку останова;

F4 – выполнить до выделенной строки;

F9 – запустить программу до ближайшей точки останова;

Ctrl+F2 – перезагрузка программы (возврат в начало).

Также полезны следующие комбинации клавиш:

F3 – открыть и запустить файл;

Alt+A – присоединиться к уже запущенному в ОС процессу;

Alt+F2 – закрыть текущий модуль;

Alt+X – закрыть отладчик.

Задания для выполнения к работе

1. Ознакомиться со средой x32dbg и компилятором masm32.
2. Создать и скомпилировать программу в соответствии с вариантом задания. В программу включить комментарии с описанием, что делает каждая инструкция. Подробное описание каждой команды можно найти в приложении учебника В.И. Юрова «Assembler», начиная со стр. 511. Комментарии следует выровнять по левому краю (как в примере).
3. С помощью отладчика определить местонахождение переменных, строк и массивов в сегменте данных, а также их размер. Составить таблицу и подробное описание ячеек сегмента данных (как в примере).
4. Выполнить пошаговую трассировку программы. Определить какие регистры, флаги и ячейки памяти изменяют свои значения в процессе выполнения команд. Обеспечить корректное завершение программы вызовом системной функции ExitProcess с кодом завершения 0. Если в сегменте данных есть строки, то вывести её в консоль. Трассировку требуется выполнить до команды «call ExitProcess» включительно. Составить для каждой инструкции таблицу трассировки (как в примере).
5. Сделать выводы о проделанной работе.

Пример выполнения работы

Задание варианта №1

Сегменты данных и кода имеют следующее содержание:

```
.DATA
      str1  DB "Hello, World!", 13, 10, 0
      x     DB 50, -60
      y     DB -10, 11
      b     DD 10.0
      r     DW ?
      d     DF -15000, 15000, 16ABC1234h, 10111101010111110100010010b
.CODE
START:
      XOR EBX, EBX
      MOV AL, x[0]
      IMUL y[0]
      MOV BX, AX
      MOV AL, x[1]
      IMUL y[1]
      ADD BX, AX
      MOV r, BX
END START
```

Требуется определить местонахождение переменных, строк и массивов в сегменте данных, а также выполнить пошаговую трассировку программы.

Выполнение работы

1. Создать файл lab1.asm со следующим содержанием:

```
.686
```

```
.model flat, stdcall
option casemap: none

include windows.inc
include kernel32.inc
include msvcrt.inc
includelib kernel32.lib
includelib msvcrt.lib

.data
     str1  DB "Hello, World!", 13, 10, 0
     x     DB 50, -60
     y     DB -10, 11
     b     DD 10.0
     r     DW ?
     d     DF -15000, 15000, 16ABC1234h, 10111101010111101000010010b

.code
start:
     XOR EBX, EBX       ; EBX = 0
     MOV AL, x[0]       ; AL = x0
     IMUL y[0]          ; AX = x0 * y0
     MOV BX, AX         ; BX = x0 * y0
     MOV AL, x[1]       ; AL = x1
     IMUL y[1]          ; AX = x1 * y1
     ADD BX, AX         ; BX = BX + x1 * y1
     MOV r, BX          ; r = BX = x0*y0 + x1*y1

     push offset str1
     call crt_puts      ; puts(str1)
     ADD ESP, 4         ; Очистка стека от аргумента

     call crt__getch    ; Задержка ввода, getch()
     ; Вызов функции ExitProcess(0)
     push 0             ; Поместить аргумент функции в стек
     call ExitProcess   ; Выход из программы
end start
```

2. Скомпилировать программу и получить исполняемый файл lab1.exe.
3. Открыть файл lab1.exe в отладчике.
4. Сегмент данных содержит одну строку *str1*, три массива *x*, *y*, *d* и две переменные *b*, *r*:

Адрес	Шестнадцатеричное												ASCII
00403000	48	65	6C	6C	6F	2C	20	57	6F	72	6C	64	21	0D	0A	00	Hello, World!...
00403010	32	C4	F6	0B	00	00	20	41	00	00	68	C5	FF	FF	FF	FF	2Äö... A..hÅÿÿÿÿ
00403020	98	3A	00	00	00	00	34	12	BC	6A	01	00	12	BD	7A	01	.:....4.¼j...½z.
00403030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00403040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00403050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00403060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Адрес	Шестнадцатеричное												ASCII
00403000	48	65	6C	6C	6F	2C	20	57	6F	72	6C	64	21	0D	0A	00	Hello, World!...
00403010	32	C4	F6	0B	00	00	20	41	00	00	68	C5	FF	FF	FF	FF	2Äö... A..hÅÿÿÿÿ
00403020	98	3A	00	00	00	00	34	12	BC	6A	01	00	12	BD	7A	01	.:....4.¼j...½z.
00403030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00403040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

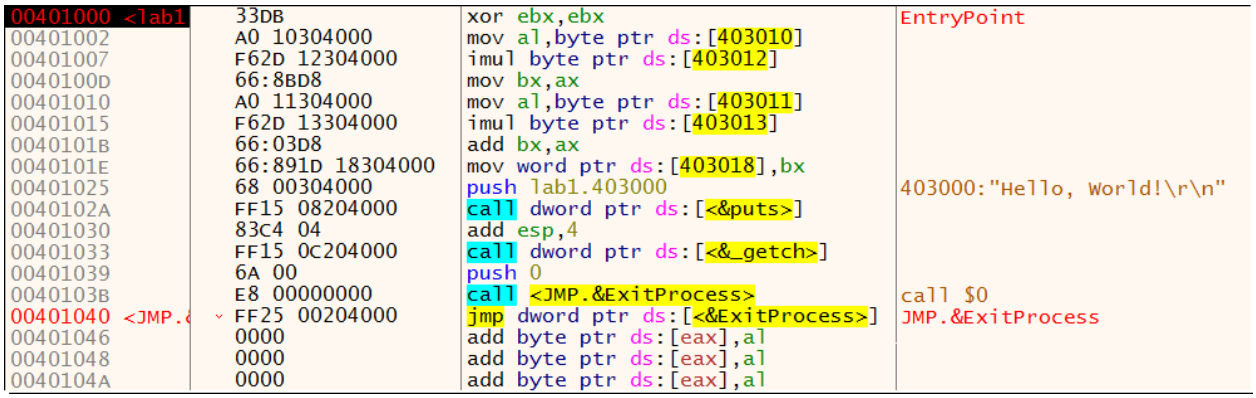
Название перемен-ной	Начальный адрес	Конечный адрес	Размер данных, байт	Описание
str1	00403000	0040300C	13	строка «Hello, World!»
–	0040300D	0040300F	3	символы возврата каретки (13), перевода строки (10), окончания строки (0)
x	00403010	00403011	2	два однобайтовых целых числа 50 и -60
y	00403012	00403013	2	два однобайтовых целых числа -10 и 11
b	00403014	00403017	4	4-байтовое вещественное 10.0
r	00403018	00403019	2	неинициализированная 2-байтовая переменная
d	0040301A	00403031	24	массив из четырёх 6-байтовых целых чисел -15000, 15000, $16ABC1234_{16}$, $101111010101111010001001 0_2$.
Общий размер сегмента данных:			**50**	

Ячейки памяти с адресами от 0x00403000 до 0x0040300C содержат ASCII-коды символов строки «Hello, World!». Далее следуют коды символов, необходимых для перевода строки и возврата каретки. Строка заканчивается ноль-символом, расположенным по адресу 0x0040300F. Массив *x* содержит два числа: $50=32_{16}$, -60 ($C4_{16}$ в дополнительном коде). Массив *y* начинается с адреса 0x00403012 и состоит из однобайтовых чисел -10 ($F6_{16}$ в дополнительном коде) и $11=0B_{16}$. По адресу 0x00403014 располагается 4-байтовое вещественное число 10.0 (типа float). Двухбайтовая переменная *r* по адресу 0x00403018 изначально не была инициализирована, но в сегменте данных данные ячейки заполнены нулями. В массиве *d* первые 6 байт представляют собой отрицательное число -15000. Чтобы перевести его в дополнительный код, нужно число $15000=3A98_{16}=000000003A98_{16}$ инвертировать в двоичном представлении и добавить единицу:

$$\overline{000000003A98}_{16} + 1 = \text{FF FF FF FF C5 67}_{16} + 1 = \text{FF FF FF FF C5 68}_{16}.$$

В памяти байты данного числа располагаются в обратном порядке: «68 C5 FF FF FF FF», начиная с адреса 0040301A. Следующее число $15000=3A98_{16}$ хранится в памяти как «98 3A 00 00 00 00», начиная с адреса 0x00403020. Байты следующих двух 6-байтовых чисел $16ABC1234_{16}$, $101111010101111010001001 0_2$ хранятся в памяти в обратном порядке как «34 12 BC 6A 01 00», «12 BD 7A 01 00 00». Адрес числа $16ABC1234_{16}$ равен 0x00403026, адрес числа $101111010101111010001001 0_2$ – 0x0040302C.

5. Пошаговая трассировка программы



Исходное состояние регистров:

EAX=	77823398	EBX=	7EFDE000	ECX=	00000000	EDX=	00401000
ESP=	0018FF8C	EBP=	0018FF94	ESI=	00000000	EDI=	00000000
EIP=	00401000						
ZF=	1	PF=	1	AF=	0		
OF=	0	SF=	0	DF=	0		
CF=	0	TF=	1	IF=	1		

xor ebx,ebx				КОП:	33DB		
EAX=	77823398	EBX=	00000000	ECX=	00000000	EDX=	00401000
ESP=	0018FF8C	EBP=	0018FF94	ESI=	00000000	EDI=	00000000
EIP=	00401002						
ZF=	1	PF=	1	AF=	0		
OF=	0	SF=	0	DF=	0		
CF=	0	TF=	0	IF=	1		
Выполняет побитовую операцию «исключающее или» над числами в регистре EBX: EBX = EBX xor EBX. Обнуляет регистр EBX. Увеличивает регистр EIP на 2 (размер кода 33DB).							

mov al,byte ptr ds:[403010]				КОП:	A0 10304000		
EAX=	77823332	EBX=	00000000	ECX=	00000000	EDX=	00401000
ESP=	0018FF8C	EBP=	0018FF94	ESI=	00000000	EDI=	00000000
EIP=	00401007						
ZF=	1	PF=	1	AF=	0		
OF=	0	SF=	0	DF=	0		
CF=	0	TF=	0	IF=	1		
Пересылает из ячейки памяти с адресом 0x403010 в регистр AL один байт. Увеличивает EIP на 5 (размер кода A010304000).							

imul byte ptr ds:[403012]						КОП:	F62D 12304000		
EAX=	7782FE0C		EBX=	00000000		ECX=	00000000	EDX=	00401000
ESP=	0018FF8C		EBP=	0018FF94		ESI=	00000000	EDI=	00000000
EIP=	0040100D								
ZF=	0	PF=	1	AF=	0				
OF=	1	SF=	0	DF=	0				
CF=	1	TF=	0	IF=	1				
Умножает значение регистра AL на однобайтовое число из ячейки памяти, расположенной по адресу 0x403012. Записывает результат в регистр AX, сбрасывает флаг ZF, устанавливает флаги OF, CF. Увеличивает EIP на 6.									

...

Варианты заданий

```
1. .DATA
        strm DB "MASM32", 0
        DB 250, 251, 252, 254
        a DD 50000
        b DQ 20000000
        cc DW 250h
        float1 DD 13.5
        float2 DD 26.5
        dmas DQ 5 DUP (5)
   .CODE
   START:
        MOV EAX, a
        MOV BX, 0A2h
        MOV CX, cc
        ADD BX, CX
        XOR EDX, EDX
        DIV BX
   END START
```

```
2. .DATA
        strt DB "Some string", 0
        DW 400
        a DF 900
        mas DD 4*8 DUP (3)
        s DQ 15.7
   .CODE
   START:
        MOV EAX, mas[0]
        MOV EBX, DWORD PTR [a]
        SUB EBX, EAX
        INC EBX
        MOV mas[4], EBX
   END START
```

```
3. .DATA
        stra DB 20 DUP ('e')
        DB 0
        n DB 20 DUP (8)
        a DW 500
        b DD 0AB120001h, 100000
        cc DQ 15.5, 15
        d DD 7.5
   .CODE
   START:
        MOV EAX, 03020100h
        MOV EBX, DWORD PTR stra
        ADD EBX, EAX
        DEC stra[6]
        MOV DWORD PTR stra, EBX
   END START
```

```
4. .DATA
        scanf_fmt DB "%d", 0
        printf_fmt DB "Result: %d", 13, 10, 0
        vec DD 4*4 DUP(5)
```

```
        x DW 2, 3, 4
        double DQ 2.5
    .CODE
    START:
        MOV EAX, 1
        XOR ESI, ESI
        INC ESI
        MOV EAX, vec[ESI]
        MUL x
        MOV vec[ESI], EAX
    END START
```

5.
```
    .DATA
        strl DB "Lfngth:  ", 0
        len DW 0
        mas DD 8 DUP(1)
        x DQ 1.0, 2.0
        ten DT 300000000
    .CODE

    START:
        XOR ESI, ESI
        ADD ESI, 8
        MOV strl[ESI], '9'
        DEC strl[1]
    END START
```

6.
```
    .DATA
        strs DB "some", 0, "string", 0
        a DD 2500000
        b DD 5000000
        vec DD 2000, 3000, 5000h, 0A00AAh
        r DQ 0
        ten DQ 18.5, 19.5, 100, 0B1200h
    .CODE

    START:
        MOV EAX, a
        ADD EAX, 100
        MUL b
        MOV dword ptr r[0], EAX
        MOV dword ptr r[4], EDX
        MOV strs[4], ' '
    END START
```

7.
```
    .DATA
        str1 DB "____age", 13, 10, 0
        p DB 5, 6, 7, 8
        k DW 16 dup(2)
        x DD 02C2A3A30h
        ten DQ 1.0, 2.0, -1.0, -2.0
    .CODE

    START:
        MOV EDI, dword ptr str1
        MOV EAX, x
        ADD EAX, dword ptr k
        XOR EAX, EDI
```

```
        MOV dword ptr str1, EAX
    END START
```

8.
```
    .DATA
        a DD 30201, 30201h
        b DB 43h, 0F3h, 0F3h, 0E5h
        DF 1500
        DD 1.5, 1.6, 1.9, -1.9
        t DQ 0E7D32A1h
        stra DB 16 dup(1)
    .CODE
    START:
        MOV ESI, 65737341h
        AND ESI, dword ptr b
        MOV dword ptr stra, ESI
        MOV ECX, dword ptr t
        IMUL ECX, 7
        ADD ECX, 6
        MOV dword ptr stra[4], ECX
        ADD stra[8], 'q'
        DEC stra[9]
    END START
```

9.
```
    .DATA
        strd DB 3 dup(5), 0, 4 dup (7)
        h DW -1, -2, -3
        w DQ 100000h, 100000
        s DD 1.0, -1.0
        DT 5.0, -5.0
    .CODE
    START:
        MOV EAX, 3
      cycle:
        MOV strd[EAX - 1], AL
        DEC EAX
        JNZ cycle
        ADD dword ptr strd, "000"
    END START
```

10.
```
    .DATA
        stre DB 8 dup(-3), "int", 0
        a DD 2840930783
        b DD 0A8794E3Ah
        x DW 16, 17, 2000, -2000
        t DD 4.0, 5.0, -1.0
        dop DF 5 DUP(65530)
    .CODE

    START:
        ; Обфускация строки
        MOV EAX, a
        MOV ECX, b
        MUL ECX
        SUB EAX, 'A'
        MOV dword ptr stre, EAX
        MOV dword ptr stre[4], EDX
    END START
```

11.
```
.DATA
    strs DB 10, 13, "unsigned long long sum", 0
    a DD 89000000h, 1000h
    b DD 1005000Fh, 2000h
    r DD 2 dup(?)
    mas DW 4 DUP(-5)
    lm DQ 1.5, 0, -7.0, -7
.CODE

START: ; Сложение 64-разрядных беззнаковых целых чисел a и b
    MOV EAX, a
    ADD EAX, b
    MOV r, EAX
    MOV EBX, a[4]
    ADC EBX, b[4]
    MOV r[4], EBX      ; r = a + b
END START
```

12.
```
.DATA
    strs DB "square", 0
    px DD -3, -2, -1, 0, 1, 2, 3
    a DW 4
    b DW 5
    r DW ?
    DF 15789, -10000000, -2, 2, 10000000
    lm DQ 1.0, -1.0, 1, -1
.CODE
START:      ; Арифметические операции
    XOR EAX, EAX
    MOV AX, a
    MUL EAX
    MOVZX EBX, b
    ADD EAX, EBX
    MOV r, AX    ; r = a*a + b
END START
```

13.
```
.DATA
    strd DB "Division", 0
    a DD 10500000h, 1200h
    b DD 10000
    m DD ?
    mas DW 8 DUP(1)
    ten DT 30000, -30000
    f DQ 1, 1.0, -1.0
    h DF -3, -2, -1, 0, 1, 2, 3, 4, 40000h, 40000

.CODE
START:      ; Арифметические операции
    MOV EDX, a[4]
    MOV EAX, a[0]
    DIV b
    MOV m, EDX         ; m = 0x120010500000 mod b
    ADD EAX, m
    IMUL EAX, 3
END START
```

14.
```
.DATA
    String1 DB 13, 10, "____", 0
```

```
        ds DW 5
        result DD ?
        p DQ 17.5
        ten DT 183333.5
    .CODE
    START:
        XOR ECX, ECX
        MOV CX, ds
        ADD CX, 5
        MOV EAX, ECX
        DIV 2
        MOV result, EAX
    END START
```

15.
```
    .DATA
        name DB 13, 10, "Andrey", 0
        a DW ?
        b DD ?
        c1 DB 16
        DF 15, 150
    .CODE
    START:
        MOV AL, c1
        DIV 4
        XOR EBX, EBX
        MOV BL, AL
        MOV a, BX
        MOV BH, AL
        MOV b, EBX
    END START
```

16.
```
    .DATA
        s DB 13, 10, "string", 0
        d DW 555
        subs DB "str", 0
        DD 15, 5
        a DQ ?
    .CODE
    START:
        XOR ECX, ECX
        MOV CX, d
        MOV EAX, ECX
        DIV 5
        MUL 3
        MOV d, AX
    END START
```

17.
```
    .DATA
        t1 DW 14, 15
        t2 DW 2 DUP(2)
        k DD ?
        b DD ?
        ddd DT 155000
    .CODE
    START:
        XOR EAX, EAX
        MOV AX, t1[0]
        MOV BX, t2[0]
```

```
    SUB AX, BX
    MOV k, EAX
    MOV b, EBX
END START
```

18.
```
.DATA
    String DB 13, 10, "RESULT: ", 0
    Mas DW 15, 16, 17, 18, 19, 20
    a DD ?
    b DD ?
    c1 DD ?
    DQ 1500.0
.CODE
START:
    XOR EAX, EAX
    MOV AX, Mas[0]
    MOV a, EAX
    MOV b, Mas[4]
    MOV AX, Mas[10]
    MOV c1, EAX
END START
```

19.
```
.DATA
    str1 DB "Some_S", 0
    DD 4*15 DUP (60)
    mas DD 20 DUP(20)
    dff DF ?
    DQ 15.5
.CODE
START:
    XOR ECX, ECX
    MOV ECX, mas[16]
    ADD CX, 5
    MOV AL, CL
    MOV mas[12], EAX
END START
```

20.
```
.DATA
    enter DB 13, 10, 0
    a DW 5
    b DW 5
    result DD ?
    p DF 17.5
    ten DT 15.5
.CODE
START:
    XOR ECX, ECX
    MOV CX, b
    XOR EAX, EAX
    MOV EAX, a
    MUL CX
    MOV result, EAX
END START
```

21.
```
.DATA
    hello DB "Hello", 0
```

```
        mas DW 232, 443, 567, 197
        bigMas DD 4 DUP(?)
        p DQ 156.43
        ten DT ?
    .CODE
    START:
        XOR EAX, EAX
        MOV ESI, 0
        MOV AX, mas[ESI]
        MOV bigMas[ESI], EAX
        ADD ESI, 2
        MOV AX, mas[ESI]
        ADD ESI, 2
        MOV bigMas[ESI], EAX
    END START
```

22.
```
    .DATA
        printf DB 13, 10, "Result: %d", 0
        a DW 5
        b DD 60
        r DD ?
        dqq DQ 171.233
    .CODE
    START:
        XOR EAX, EAX
        MOV AX, a
        MOV EBX, b
        ADD EAX, EBX
        MOV r, EAX
    END START
```

23.
```
    .DATA
        strt DB "Some String", 0
        v DD 4*4 DUP(5)
        DW 387
        r DQ ?
        ten DT 183333.423
    .CODE
    START:
        XOR EAX, EAX
        MOV EAX, v[0]
        MOV EBX, v[4]
        ADD EAX, EBX
    END START
```

24.
```
    .DATA
        scanf_s DB "%d", 0
        arr DW 4*5 DUP(?)
        con DD 5
        DD ?
        pi DF 3.14
    .CODE
    START:
        XOR EAX, EAX
        MOV ESI, 0
        MOV EBX, con
        ADD EAX, EBX
        MOV arr[ESI], AX
```

```
    ADD ESI, 2
END START
```

25.
```
.DATA
    scanf_s DB "%d", 0
    printf_s DB 13, 10, "Result: %d", 0
    arr1 DW 4*5 DUP(?)
    arr2 DW 5*4 DUP(2)
    DD ?
    e DT 2.75
.CODE
START:
    XOR EAX, EAX
    MOV AX, arr2[0]
    ADD AX, 15
    MOV arr1[0], AX
END START
```