МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Чувашский государственный университет имени И.Н. Ульянова»

А.А. Андреева

ОПРЕДЕЛЕНИЕ И НАСТРОЙКА ПАРАМЕТРОВ КОМПЬЮТЕРА

Практикум

Чебоксары
2020

Введение

Современные микропроцессоры семейства x86-64 фирмы Intel – 64-разрядные однокристальные центральные процессоры с фиксированным набором команд [1, 6]. На основе процессоров Intel x86-64 (а также аналогичных процессоров AMD) выпускаются различные типы вычислительных систем, включая широко распространенные IBM-совместимые настольные компьютеры, которые имеют развитое программное обеспечение, в том числе трансляторы с языков высокого уровня и среды визуального программирования. В данной статье рассматривается программирование компьютера на языке низкого уровня – языке ассемблера, несмотря на трудоемкость и меньшее удобство его использования. В каких случаях необходимо использование языка ассемблера?

Во-первых, для создания программ с максимальной производительностью, например, в сложных игровых программах, машинной графике и т.п. Во-вторых, для написания интерфейсных программ, обеспечивающих взаимодействие между языками высокого уровня и некоторыми служебными процедурами операционной системы. Наконец, для самостоятельной доработки готовых программ. Таким образом, изучение программирования компьютера на языке ассемблера является необходимым элементом подготовки будущего специалиста по вычислительной технике.

Для программирования на ассемблере 16-/32-битных процессоров семейства x86 в операционных системах MS DOS и Windows широко использовалась система TASM/TLINK/TD фирмы Borland [2, 3, 4]. Однако поддержка этой системы была прекращена, поэтому возникла задача выбора компилятора языка ассемблера для 64-битных процессоров x86 под Windows.

Свободно распространяемые ассемблеры FASM [8] и NASM [9] имеют свои сильные и слабые стороны [7]. Так FASM выбран для компиляции всех примеров известной книги Р. Аблязова [5], но в исходном тексте программ нужно «вручную» создавать секции импортируемых функций, недостатком также является отсутствие встроенного отладчика. Ассемблер NASM требует использования дополнительных программ для создания (GoLink) и

отладки (X64dbg) exe-файла, его рекомендуют применять в операционной системе Linux [7].

Результатом нашего выбора является ассемблерный компилятор MASM фирмы Microsoft, известный еще с времен 16-битных процессоров x86. Кроме поддержки всех машинных команд компилятор MASM имеет следующие достоинства:

похожесть синтаксиса операторов ассемблера с компилятором TASM (вернее, в TASM мы использовали режим MASM);

нахождение компилятора в составе мощной визуальной среды разработки Visual Studio Community [10], что позволяет получать различные варианты исполняемых файлов (консольное приложение, динамическая библиотека), проводить отладку программ с просмотром регистров процессора, дампов памяти, стека.

РАБОТА 1. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ В Win64

Набор регистров процессоров Intel x86-64

Процессоры Intel x86-64 могут работать в следующих режимах (рис. 1): в режиме совместимости с x86 (Legacy Mode) и длинном режиме (Long Mode), который делится на два подрежима:

- режим совместимости с x86 (compatibility mode);
- 64-битный режим (64-bit mode).

Режим совместимости предусмотрен только для того, чтобы 64-разрядная операционная система могла выполнять старые 32-битные приложения.

При переключении процессора в 64-разрядный режим программам доступны следующие регистры (рис. 2):

регистры общего назначения (рис. 3): 64-разрядные RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP и R8, R9,..., R15; 32-разрядные EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, R8D - R15D (являются младшими частями 64-разрядных регистров); 16-разрядные AX, BX, CX, DX, SI, DI,SP, BP, R8W - R15W (являются младшими частями 32-разрядных регистров); 8-битные регистры AH, BH, CH, DH и AL, BL, CL, DL, SIL, DIL, SPL, BPL, R8L -

R15L (старшие и младшие части 16-битных регистров соответственно);

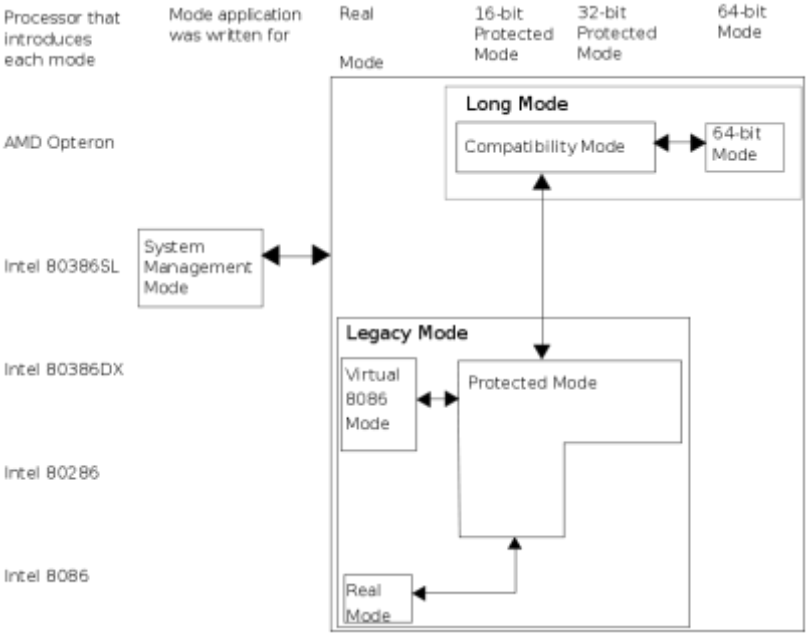

Рис.1. Режимы работы процессоров Intel x86-64

64-разрядный RIP - указатель инструкции;

16-разрядные сегментные регистры: CS, DS, SS, ES, FS, GS;

64-разрядный регистр флагов RFLAGS;

80-битные регистры математического сопроцессора ST0 - ST7;

64-битные MMX-регистры (MM0 - MM7);

128-разрядные XMM-регистры XMM0 - XMM15 и 32-битный MXCSR;

64-разрядные регистры управления CR0 - CR4 и CR8; регистры-указатели системных таблиц GDTR, LDTR, IDTR и регистр задачи TR;

64-разрядные регистры отладки DR0 - DR3, DR6, DR7;

MSR-регистры.

Register or Stack	Legacy and Compatibility Modes			64-Bit Mode		
	Name	Number	Size (bits)	Name	Number	Size (bits)
General-Purpose Registers (GPRs)	EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP	8	32	RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8–R15	16	64
128-Bit XMM Registers	XMM0–XMM7	8	128	XMM0–XMM15	16	128
64-Bit MMX Registers	MMX0–MMX7	8	64	MMX0–MMX7	8	64
x87 Registers	FPR0–FPR7	8	80	FPR0–FPR7	8	80
Instruction Pointer	EIP	1	32	RIP	1	64
Flags	EFLAGS	1	32	RFLAGS	1	64
Stack	–		16 or 32	–		64

Рис.2. Сравнение 32- и 64-битных регистров

64-битные	32-битные	16-битные	8-битные
RAX	EAX	AX	AL/AH
RBX	EBX	BX	BL/BH
RCX	ECX	CX	CL/CH
RDX	EDX	DX	DL/DH
RSI	ESI	SI	
RDI	EDI	DI	
RSP	ESP	SP	
RBP	EBP	BP	
R8	R8D	R8W	R8B

R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

Рис. 3. Регистры общего назначения

Теперь немного пояснений. Регистры сегментов напрямую участвуют в формировании адресов, каждый сегментный регистр указывает на свой сегмент памяти, а именно: CS - сегмент кода, DS - сегмент данных, SS - сегмент стека; остальные три регистра дополнительные и могут не использоваться программой. Свободная работа с ними не всегда возможна; например, в защищённом и 64-разрядном режимах загружать в них можно лишь определённые значения. В защищённом и 64-разрядном режимах доступность регистров зависит уровня привилегий, на котором выполняется программа.

Регистр общего назначения ESP (RSP) всегда указывает на вершину стека, но при этом нам ничто не мешает использовать его в других целях, хотя тогда будет потеряна возможность нормальной работы со стеком. Вообще все регистры общего назначения можно свободно использовать в своих целях, но следует помнить, что некоторые регистры используются некоторыми, командами: например, EBP(RBP) обычно указывает на начало фрейма в стеке, где хранятся локальные данные подпрограмм.

Указатель инструкции EIP (RIP) напрямую использовать нельзя - данный регистр используется самим процессором.

Регистры STn, MMп, XMMп используются математическим сопроцессором при работе с числами с плавающей точкой.

Регистры CRn, DRn, регистры-указатели системных таблиц, MSR-регистры являются системными и управляются ключевыми механизмами работы процессора.

Память

В Win64 отличий от Win32 не так много. По-прежнему каждый процесс находится в своём собственном виртуальном адресном пространстве. По-прежнему память разделена на две части, но уже не пополам. Память третьего кольца находится в диапазоне 0h-80000000000h - в нём недоступны по 64 Кбайт «сверху» и «снизу», и в итоге у нас имеется 8 Тбайт минус 128 Кбайт. Память ядра находится в диапазоне 0000080000000000b - FFFFFFFFFFFFFFFFh. На первый взгляд кажется, что это очень большой диапазон памяти, но если учесть, что адрес в long mode может быть только в канонической форме (фактически используются только 48 бит), то это равносильно диапазону FFFF080000000000h - FFFFFFFFFFFFFFFFh. Старшие 16 бит адреса могут быть как нулями, так и единицами, просто исторически сложилось так, что адреса памяти ядра принято обозначать единицами в старших битах.

Таким образом, 8 Тбайт виртуальной памяти отведено для приложений третьего кольца и 248 Тбайт виртуальной памяти - для ядра системы; этого с избытком хватит даже самым требовательным к памяти приложениям и драйверам.

Ограничения в системе команд

Отсутствуют команды:

для работы с двоично-десятичными числами (`aaa, aad, aam, aas, daa и das`);

`bound`;

`pusha/popa`;

`inc и dec` с однобайтовой кодировкой.

Ограничения:

В архитектуре x86 присутствует ограничение на длину команды в 15 байт. Из этого вытекает невозможность записи команд вида

```
mov [mem64], imm64
```

Так как это займёт больше 15 байт (8 байт адреса + 8 байт непосредственного операнда + код операции). Поэтому в x64 можно делать только так

```
mov [mem64], imm32
```

При этом imm32 будет преобразован в qword с учётом знака (`movsx`).

Команда не может ссылаться одновременно на `ah, bh, dh, ch` и младший байт новых регистров. То есть "`mov ah,dl`" - допустимо, а "`mov ah,r8b`" - не допустимо.

Операции с 32 битными операндами обнуляют старшие 4 байта результата. То есть команда "`add eax, ebx`" обнулит старшую часть `RAX`. К 8 и 16 битным операндам это не относится.

Исчезла прямая адресация. Если нам надо изменить содержимое ячейки памяти по конкретному адресу, на x86 мы поступаем приблизительно так:

```
dec byte ptr ds:[666h]; уменьшить содержимое байта по адресу 666h на единицу
```

Под x86-64 транслятор выдает ошибку ассемблирования, вынуждая нас прибегать к фиктивному базированию:

```
xor r9, r9; обнулить регистр r9
dec byte ptr [r9+666h]; уменьшить содержимое байта по адресу 666h на единицу
```

Программирование в Win64

Изменений в типах данных очень много; будут приведены только общие сведения о типах данных.

Очевидно то, что все указатели теперь имеют размер 8 байт. Это значит, что все типы, названия которых начинаются с буквы p (по терминологии Microsoft), теперь имеют размер 8 байт.

Во-вторых, все handle теперь имеют размер 8 байт, т. е. все типы, названия которых начинаются с буквы h, а именно: handle, hbrush, hbitm ap, hcolorspace, hcursor, HDC, HFONT, HICON, HINSTANCE, HKEY, HMENU, HMODULE, HPEN, HPALETTE, HWND и т.д.

В-третьих, параметры оконных сообщений wparam и lparam тоже теперь имеют размер 8 байт.

Типы данных, названия которых говорят сами за себя, и те типы, размеры которых сложились исторически, не изменились, например: char - по-прежнему 1 байт, uint, UINT32, ulong, ULONG32 - по-прежнему 4-байтовые числа без знака, UINT64, ULONG64, ulonglong - по-прежнему 8-байтовые числа без знака т. д.

Изменения касаются и ядра системы. Поскольку в ядре системы в основном используются указатели, то большинство параметров, принимаемых функциями, теперь имеют размер 8 байт, за исключением тех, о которых было сказано в предыдущем абзаце.

Вызов функций Win64

В Win64 используется модель вызова fastcall. Основные соглашения:

- Параметр, размер которого 8, 4, 2 или 1 байт передается по значению.
- Если размер параметра не совпадает с выше перечисленными, то параметр передается по ссылке.
- Первые 4 целочисленных аргумента передаются через регистры RCX, RDX, R8, R9 слева направо.
- Параметры, имеющие тип float или double передаются через регистры XMM0, XMM1, XMM2, XMM3.
- Если число параметров больше 4, то параметры, начиная с 5-го передаются через стек в обратном порядке (как в stdcall).
- Перед вызовом любой функцией с соглашением fastcall в стеке должно быть зарезервировано место под 4 параметра + параметры, которые не передаются через регистры.
- Результат функции передается через регистр RAX или через XMM0, если он является числом с плавающей точкой.
- Если результат не помещается в RAX, то в RAX содержится ссылка на результат функции.
- Выравнивание стека на границу 16.

Резервирование места в стеке под первые четыре параметра (даже если количество параметров меньше четырёх) производится для того, чтобы потом можно было переместить значения из регистров в стек, а сами регистры использовать для других целей. Место в стеке для первых четырёх параметров, которые

передаются через регистры, называется теневой частью. Резервирование места достигается тем, что из регистра RSP вычитается значение 32 (стек растёт сверху вниз). За очистку стека от параметров и теневой части отвечает вызывающая процедура. Следует помнить, что стек всегда должен быть выровненным на границу 16 байт, т. е. указатель в RSP должен быть кратен 16. Это требование неочевидное, но тем не менее так нам говорит документация. В большинстве случаев отсутствие выравнивания на 16 байт не повлечет за собой никаких последствий, однако некоторые функции вызовут исключение именно по этой причине. В общем случае отсутствие выравнивания уменьшает производительность приложения.

Если размер параметра меньше 64 бит, то он расширяется до 64 бит нулями или единицами (если он отрицателен) и только потом помещается в регистр или стек.

Обычно резервирование места происходит в самом начале, а параметры помещаются в стек не командами PUSH, а командой MOV.

При правильном написании программ в Win64 можно минимизировать количество модификаций регистра RSP и количество обращений к стеку; следовательно, увеличивается и скорость вызова процедур и функций.

Следующие регистры могут измениться после вызова fastcall функции: RAX, RCX, RDX, R8-R11, XMM0-XMM5. Если значения этих регистров нужны вызывающему, он должен их сохранить. Остальные регистры (RBX, RSI, RDI, RSP, RBP, R12-R15, XMM6-XMM15) не должны измениться после вызова fastcall-функции, и если вызываемая функция их использует, то она должна их сохранить.

Для примера рассмотрим вызов ExitProcess.

На входе у него один параметр – код выхода. Его мы сохраним в регистре RCX командой:

```
xor rcx, rcx
```

Резервируем место в стеке для 4 (минимум) переменных. Это обязательно!

```
sub rsp, 32
```

Вызываем функцию

```
call ExitProcess
```

Если нам было необходимо передать параметры через стек, то мы должны были сохранить командой

```
mov qword[rsp+n], val
```

Так вызов функции с пятью аргументами API_func (1,2,3,4,5) выглядит так:

```
sub rsp, 40
mov qword ptr [rsp+32],5; в стек пятый слева аргумент
mov r9d, 4; передаем четвертый слева аргумент
mov r8d, 3; передаем третий слева аргумент
mov edx, 2; передаем второй слева аргумент
mov ecx, 1; передаем первый слева аргумент
call API_func
add rsp, 40
```

В качестве третьего примера передачи параметров и резервирования места в стеке рассмотрим простую программу на C++, использующую функцию WinAPI:

```
#include "Windows.h"
const char* fileName = "file.txt";
int main()
{
    CreateFileA(
        fileName,
        GENERIC_READ,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );
    return 0;
}
```

Дизассемблированный код (Release-сборка) имеет вид:

```
#include "Windows.h"
const char* fileName = "file.txt";
```

```
int main()
{
000000013FA81000 sub rsp,48h
    CreateFileA(fileName,   GENERIC_READ,   0,
NULL,   CREATE_ALWAYS,   FILE_ATTRIBUTE_NORMAL,
NULL);
000000013FA81004 mov qword ptr [rsp+48],0
000000013FA8100D  lea  rcx,[string  "file.txt"
(013FA82200h)]
000000013FA81014 mov dword ptr [rsp+40],80h
000000013FA8101C xor r9d,r9d
000000013FA8101F xor r8d,r8d
000000013FA81022 mov dword ptr [rsp+32],2
000000013FA8102A mov edx,80000000h
000000013FA8102F call qword ptr [__imp_Create-
FileA (013FA82000h)]
    return 0;
000000013FA81035 xor eax,eax
}
000000013FA81037 add rsp,48h
000000013FA8103B ret
```

Как видно, в начале программы выделяется место в стеке под параметры вызываемых из main функций. Данный компилятор C++ выделяет сразу 48h = 72 байта под 7 параметров с учетом выравнивания стека на границу, кратную 16 (учитывается дальнейшее сохранение в стеке 8 байт адреса возврата при вызове функции, т.е. 72 + 8 = 80 – кратно 16).

Отладка программ x64 с помощью отладчика x64dbg

Для отладки и проверки работоспособности программы можно использовать отладчик x64dbg. Выделим следующие его преимущества:

- бесплатность;
- удобность интерфейса, схожесть его с Turbo Debugger (вплоть до горячих клавиш);
- возможность представления данных в нескольких форматах (шестнадцатеричный, текстовый, целочисленный, вещественный);
- возможность просмотра секций, создаваемых программой во время выполнения;
- отображение состояний регистров, присутствующих в нынешних процессорах;

Программа поставляется в двух вариантах:

x32dbg – отладка программ, транслированных для 32-х битных систем (папка release\x32);

x64dbg – отладка программ, транслированных для 64-х битных систем (папка release\x64).

В данной работе используется версия для x64 систем (файл x64dbg.exe). Окно программы состоит из следующих частей, обозначенных цифрами на рис. 5.

1. Окно дизассемблера – окно, отображающее текущий указатель на команду (RIP – instruction pointer), точку останова, адреса расположения команд и сами инструкции. Состояние точки останова указывается кликом на кружок около адреса: красный цвет – точка останова установлена, зелёный цвет – точка останова выключена, серый цвет – точка останова удалена.

2. Окно регистров – окно, показывающий состояние регистров в данный момент времени.

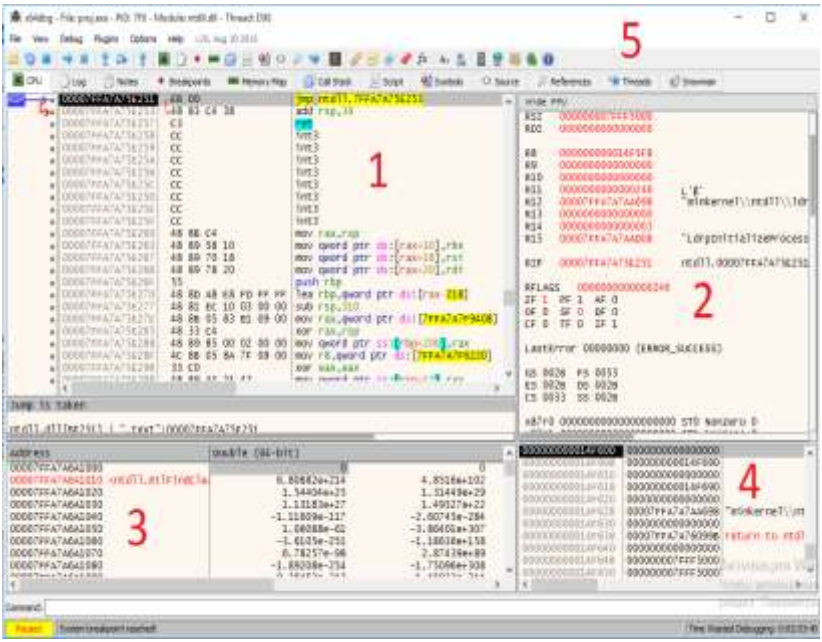

Рис. 5. Окно отладчика x64dbg

Для наблюдения доступны:

регистры общего назначения: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8-R15;

указатель на команду: RIP;

регистр флагов RFLAGS и часто используемые флаги ZF, PF, AF, OF, SF, DF, CF, RF, IF;

регистр, содержащий код последней ошибки LastError;

регистры сопроцессора: x87r0 - x87r7;

регистры тегов, управления, состояния сопроцессора x87TagWord, x87StatusWord, x87ControlWord и состояния каждого бита из этих флагов

регистр управления/статуса SIMD (MXCSR) и состояния используемых флагов;

регистры расширения MMX0-MMX7, регистры блока XMM(0-15).

3. Дамп (снимок) памяти. В этом окне мы можем наблюдать за значениями, хранящимися в памяти. Возможны различные виды представления данных: символьное, шестнадцатеричное, целочисленное, вещественное, адресное.

4. Окно стека – информация о состоянии стека: адрес и значение (в некоторых случаях – предназначение значения).

5. Панель управления – на ней находятся различные кнопки быстрого доступа и вкладки.

Пример отладки программы

Для примера отладим программу test.exe, предварительно скомпилированную и слинкованную. Для загрузки программы необходимо выполнить следующие действия.

1. Указать путь к ней. Делается это с помощью меню File (File -> Open), либо клавиши F3. Если программа была недавно открыта, то можно просто выбрать необходимый файл. (File -> Recent Files)

Рис. 6. Вкладка File

2. Установить адрес смещения, начиная с которого хранятся данные. Переходим во вкладку Memory Map. Находим нашу программу, смотрим, где находятся наши данные (ищем строку .data после test.exe). С помощью мыши сохраняем значение в буфере обмена. (ПКМ –> Copy -> Address)

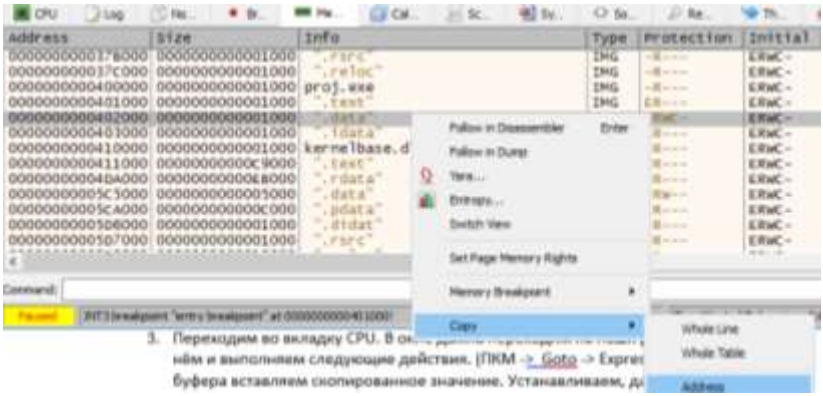

Рис. 7. Вкладка Memory Map

3. Переходим во вкладку CPU. В окне дампа переходим на наши данные и выполняем следующие действия. (ПКМ -> Goto -> Expression (Ctrl-G)). Из буфера вставляем скопированное значение. Устанавливаем, данные какого типа хранятся в памяти (ПКМ и выбираем, в каком формате мы хотим представить данные). В нашем случае выбираем Float -> Double (64 – bit).

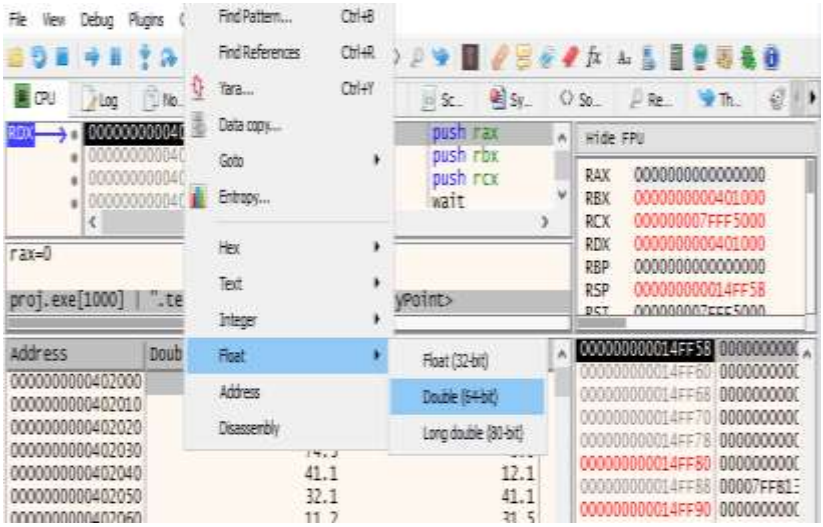

Рис.8. Установка формата представления данных

4. Нажимаем F9. Программа переходит на точку останова (начало нашей программы). Программа загружена. Теперь мы видим программу (рис. 9), загруженную в отладчик.

Управление процессом отладки осуществляется с помощью следующих клавиш:

F9 – запуск программы на выполнение, либо до ближайшей точки останова;

F7 – выполнение программы по шагам с заходом в функцию;

F8 – выполнение программы по шагам без захода в функцию;

Ctrl-F2 – перезапуск программы (после перезапуска необходимо нажать F9, чтобы перейти к началу программы)

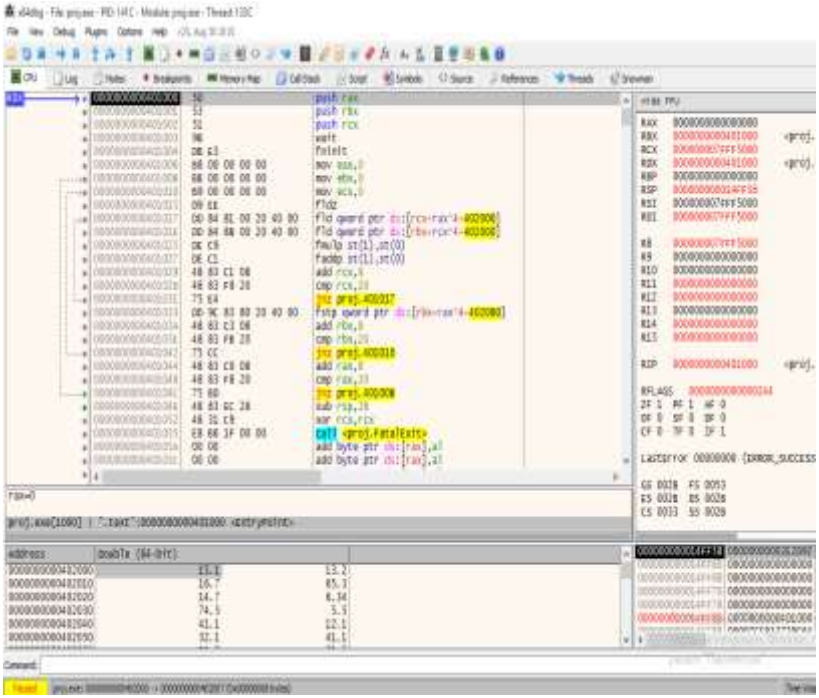

Рис. 9. Загруженная программа

В качестве упражнения установите точку останова по адресу 000000000040104E, с помощью клавиши F8 дойдите до адреса 0000000000401004, и нажмите F9. Проверьте результат выполнения с помощью перехода в окне дампа на адрес смещения данных.

Примеры программ

Пример 1. Установить файлу, имя которого задано в командной строке, атрибут «только для чтения».

Текст программы (файл lab5.asm):

```
   includelib kernel32.lib
   extrn GetCommandLineA:near
   extrn WriteConsoleA:near
   extrn SetFileAttributesA:near
   extrn CreateFileA:near
   extrn WriteFile:near
   GetCommandLine equ GetCommandLineA
   WriteConsole equ WriteConsoleA
   SetFileAttributes equ SetFileAttributesA
   CreateFile equ CreateFileA

   STD_OUTPUT_HANDLE equ -11
   FILE_ATTRIBUTE_READ_ONLY equ 01h
   GENERIC_READ equ 80000000h
   GENERIC_WRITE equ 40000000h
   OPEN_EXISTING equ 3

.data
   real_num      dq 0
   StrFirstMsg db 'Set Read-Only',10,13,0
   FileNamePtr dq ?
   StrError      db 'Something wrong',10,13,0
   StrSuccess    db 'Success!',10,13,0
   numw          dq ?
   hcons   dq ?
   nameout       db 'CONOUT$',0

.code
```

```
Start proc
   sub     rsp, 72;
; 7 параметров * 8 + выравнивание до величины,
; кратной 16, минус 8, если второй параметр 64,
; программа не работает, если 56, 88 – работает

; открыть консоль как файл
   mov     rcx, offset nameout
   mov     rdx,GENERIC_READ+GENERIC_WRITE
   mov     r8,0
   mov     r9,0
   mov     qword ptr [rsp+32], OPEN_EXISTING
   mov     qword ptr [rsp+40],0
   mov     qword ptr [rsp+48],0
   call    CreateFile
   cmp     rax,0
   e  Errorr
   mov     hcons,rax
   call    GetCommandLine
NextChar:  cmp  byte ptr[rax], 0
     je    StopSkip
   cmp     byte ptr[rax], ' '
   je NextWord
   inc     rax
   jmp     NextChar

NextWord:  cmp  byte ptr[rax], 0
   Je StopSkip
   cmp     byte ptr[rax], ' '
   jne     StopSkip
   inc     rax
   jmp     NextWord

StopSkip: mov rcx, rax
   call    SetAttribute
   jmp     Exit

Errorr:    mov  rcx, hcons
```

```
   mov     rdx, offset StrError
   mov     r8, 17
   mov     r9, offset real_num
   mov     qword ptr [rsp+32],0
   call    WriteFile
Exit: call ExitProcess

SetAttribute proc
   sub     rsp, 40; выделение места в стеке
   mov     FileNamePtr, rcx
   mov     rcx, FileNamePtr
   mov     rdx, FILE_ATTRIBUTE_READ_ONLY
   call    SetFileAttributes
   cmp     rax,0
   je Wrong
   mov     rcx,hcons
   mov     rdx,offset StrSuccess
   mov     r8, 10
   mov     r9, offset real_num
   mov     qword ptr [rsp+32],0
   call    WriteFile
   jmp     ExitProc

Wrong: mov rcx, hcons
   mov     rdx, offset StrError
   mov     r8, 17
   mov     r9, offset real_num
   mov     qword ptr [rsp+32],0
   call    WriteFile

ExitProc:  add   rsp, 40
   ret
SetAttribute endp
Start endp
   End
```

Пакетный файл для создания и запуска программы:

```
ml64 /Cp /c lab5.asm
pause
link /SUBSYSTEM:CONSOLE /ENTRY:start lab5.obj
pause
lab5.exe file.txt
pause
```

Ключи ml64:

/Cp - сохраняет регистр всех идентификаторов пользователя;

/c - только сборка, без линковки.

Ключи link:

/SUBSYSTEM:CONSOLE - консольное приложение

или

/SUBSYSTEM:WINDOWS - оконное приложение;

/ENTRY:<proc> - название процедуры, с которой начинается работа программы (не метки!);

/DLL - если создается dll-файл;

/DEF:<file.def> - указывает используемый DEF-файл.

В ml64 можно вместо ключа /c можно указать ключ /link, после чего идут ключи link, например:

```
ml64 /Cp /c prog.asm
link /SUBSYSTEM:CONSOLE /ENTRY:start prog.obj
```

можно заменить на

```
ml64 /Cp /link /SUBSYSTEM:CONSOLE /ENTRY:start
prog.asm
```

При создании dll-файла автоматически создается lib-файл.

Пример 2. Создать динамическую библиотеку, удаляющую файл, имя которого задано в командной строке. В основной программе использовать неявное связывание.

Исходный текст dll-файла:

```
includelib  kernel32.lib
extrn   GetStdHandle:proc
extrn   WriteConsoleA:proc
extrn   DeleteFileA:proc
WriteConsole  equ WriteConsoleA
DeleteFile    equ DeleteFileA
STD_OUTPUT_HANDLE  equ -11
```

```
.data
   OutHandle     dq ?
   real_num      dq 0
   StrFirstMsg db 'Delete the file',10,13,0
   StrFileName dq ?
   StrError      db 'Something wrong',10,13,0
   StrSuccess    db 'Success!',10,13,0

.code
DLLMain    proc
   mov      rax, 1
   ret
DLLMain    endp

Delete     proc
   sub rsp, 40; выделение места в стеке
   mov      StrFileName, rcx
   mov      rcx, STD_OUTPUT_HANDLE
   call     GetStdHandle
   mov      OutHandle, rax
   mov      rcx, OutHandle
   mov      rdx, offset StrFirstMsg
   mov      r8, 17
   mov      r9, offset real_num
   call     WriteConsole
   cmp      byte ptr[StrFileName], 0
   je Wrong
   mov      rcx, StrFileName
   call     DeleteFile
   cmp      rax, 0
   je Wrong
   mov      rcx, OutHandle
   mov      rdx, offset StrSuccess
   mov      r8, 10
   mov      r9, offset real_num
   call     WriteConsole
   jmp           ExitProc
Wrong: mov rcx, OutHandle
```

```
   mov     rdx, offset StrError
   mov     r8, 17
   mov     r9, offset real_num
   call    WriteConsole
   ExitProc: add rsp, 40
   ret
Delete     endp
   End
```

Текст программы, использующей динамическую библиотеку:

```
   includelib  kernel32.lib
   includelib   DeleteFile.lib
   extrn   GetCommandLineA:proc
   extrn   Delete:proc
   extrn   ExitProcess:proc
   GetCommandLine     equ GetCommandLineA

.code
Start proc
   sub     rsp, 40
   call    GetCommandLine
NextChar:  cmp  byte ptr[rax], 0
   je      StopSkip
   cmp     byte ptr[rax], ' '
   je      NextWord
   inc     rax
   jmp     NextChar
NextWord:  cmp  byte ptr[rax], 0
   je      StopSkip
   cmp     byte ptr[rax], ' '
   jne     StopSkip
   inc     rax
   jmp     NextWord
StopSkip:  mov rcx, rax
   call    Delete
   mov     rcx, 0
   call    ExitProcess
start      endp
   end
```

Литература

1.Intel® 64 and IA-32 Architectures Developer's Manual [Электронный ресурс]. – Электрон. текстовые дан. – Режим доступа: https://www.intel.ru/content/www/ru/ru/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html (дата обращения 20.11.2018)

Intel® 64 and IA-32 Architectures Software Developer Manuals https://software.intel.com/en-us/articles/intel-sdm

Как выглядит многоядерный язык ассемблера? http://qaru.site/questions/35323/what-does-multicore-assembly-language-look-like

2.Зубков, С. В. Assembler для DOS, Windows и UNIX / С.В. Зубков. – М.: ДМК Пресс, 2015. – 638 с.

3.Андреева, А. А. Основы программирования персонального компьютера на языке ассемблера: лабораторный практикум / А.А. Андреева. – Чебоксары: Изд-во Чуваш. ун-та, 2013. – 84 с.

4.Андреева, А. А. Программирование на языке ассемблера в операционной системе Windows: лабораторный практикум / А.А. Андреева, Д.Ф. Волков, А.Л. Иванов, И.А. Капустина. – Чебоксары: Изд-во Чуваш. ун-та, 2006. – 104 с.

5.Аблязов, Р. З. Программирование на ассемблере на платформе x86-64 / Р.З. Аблязов. – М.: ДМК Пресс, 2016. – 302 с.

6.Краснов, В. И. Исследование Long Mode процессоров архитектуры x86 / В.И. Краснов, А.А. Андреева // Информатика и вычислительная техника: сб. науч. трудов. – Чебоксары: Изд-во Чуваш. ун-та, 2016. – С. 113-119.

7.MASM, TASM, FASM, NASM под Windows и Linux [Электронный ресурс]. – Электрон. текстовые дан. – Режим доступа: https://habr.com/post/326078/ (дата обращения 20.11.2018).

8.Flat assembler. Assembly language resources [Электронный ресурс]. – Официальный сайт. – Режим доступа: http://flatassembler.net/ (дата обращения 20.11.2018).

9.Nasm [Электронный ресурс]. – Официальный сайт. – Режим доступа: https://www.nasm.us/ (дата обращения 20.11.2018).

10. Free Developer Software & Services – Visual Studio [Электронный ресурс]. – Официальный сайт. – Режим доступа:

https://visualstudio.microsoft.com/free-developer-offers/ (дата обращения 20.11.2018).