Програмирование на fasm под Win64

qwe8013

24 июля 2017 г.

Оглавление

0.1. Ассемблер, отладчик, IDE

0.1.1. Введение

Я начинаю цикл статей по ассемблеру fasm. Возможно у вас есть вопрос: "Зачем в 21 веке нужен ассемблер?". Я бы ответил так: Конечно, знание ассемблера не обязятельно, но оно способствует пониманию, во что превращается ваш код, как он работает, это позволяет почувствовать силу. Ну и в конце концов: писать на ассемблере просто приятно (ну по крайней мере небольшие приложения). Так что надеюсь, что мои статьи будут вам полезны.

0.1.2. Где взять fasm?

Собственно тут: http://flatassembler.net/download.php На этой странице Томаш Грыштар(создатель fasm-а) выкладывает последнюю версию ассемблера. Там есть версии для DOS, Linux, Unix и Windows, нам нужна для Windows. В скачанном архиве находятся следующие компоненты:

- fasm.exe – собственно сам ассемблер
- fasmw.exe – IDE (среда разработки)
- fasm.pdf – документация
- папка source – исходники fasm-а (написан на самом себе)
- папка include – папка с заголовками, импортами, и т.д.
- папка examples – примеры программ на fasm-е

Содержимое fasm.pdf дублирует 2 раздела документации "flat assembler 1.71 Programmer's Manual" и "Windows programming" отсюда: http://flatassembler.net/docs.php.

0.1.3. IDE (среда разработки)

Перед тем, как писать программы нужно определиться, в чём их писать. Для fasm-а существуют разные IDE, например: fasmw.exe(находится в архиве с fasm-ом), RadAsm, WinAsm Studio, Fresh, ... Выберайте, какая вам больше по вкусу. Сразу скажу, что IDE из поставки fasm-а обладает минимальным количеством фичей, так что я бы рекомендовал использовать альтернативную IDE. Я, например, использую RadAsm 3.0, его можно взять здесь: https://fbedit.svn.sourceforge.net/svnroot/fbedit/RadASM30/Release/RadASM.zip

(это хорошая IDE, но к сожалений она больше не обновляется). К статье приложен файл Fasm.ini (*attach\intro\Fasm.ini*), там выбрана чёрная тема, добавлены x64 регистры и добавлена подсветка для большего числа инструкций. Можете поставить его вместо Fasm.ini поумолчанию, только исправьте в нём пути к папке с fasm-ом в 6 и 7 строках. Там написано:

```
[Environment]
0=path,C:\Program Files (x86)\fasm;$A\..\Ollydbg
1=include,C:\Program Files (x86)\fasm\INCLUDE
```

Если вы распаковали fasm в папку <полный путь>, то надо заменить приведённый выше код на:

```
[Environment]
0=path,<полный путь>\fasm;$A\..\Ollydbg
1=include,<полный путь>\fasm\INCLUDE
```

0.1.4. Отладчик

Писать программы это — хорошо, но нужно находить и исправлять баги, для этого нужен отладчик. Существуют разные отладчики способные отлаживать 64-битный код например: WinDbg, fdbg, x64dbg. Наиболее удобный на мой взгляд - x64dbg, его можно скачать здесь. Это — всё, что я хотел рассказать в первой части.

Глава 1.

Основы

1.1. Системы счисления, память, регистры

Данная статья будет чисто теоретической, ничего программировать в ней мы не будем, но то, что здесь рассказывается знать необходимо.

1.1.1. Системы счисления

Начнём мы с систем счисления, они определяют способ записи чисел.

Вам хорошо знакома десятичная система счисления, в ней существует десять цифр от 0 до 9. Число в данной системе счисления представляется как сумма чисел, каждое из которых принимает значения от нуля до девяти, умноженных на 10 в соответствующей степени, например, рассмотрим число $32167 = 7 \cdot 10^0 + 6 \cdot 10^1 + 1 \cdot 10^2 + 2 \cdot 10^3 + 3 \cdot 10^4$. Т.о. произвольное целое число x записывается в виде:

$$x = \sum_{i=0}^{n} c_i \cdot 10^i$$

Где c_i - цифры от 0 до 9, а $n + 1$ – число разрядов. Далее будем говорить, что в i-ом разряде находится цифра c_i, i=0 – младший разряд, i=n – старший разряд.

Аналогично определяются и другие системы счисления, например, в восьмеричной системе счисления 8 цифр: от 0 до 7 и умножаются они на степени восьмёрки, например, число в восьмеричной системе счисления $32167_8 = 7 \cdot 8^0 + 6 \cdot 8^1 + 7 \cdot 8^2 + 2 \cdot 8^3 + 3 \cdot 8^4$. Вполне очевидным образом можно перевести число из восьмеричной системы в десятичную просто посчитав эту сумму. Ещё одна важная система счисления – двоичная, в ней, как вы уже наверно догадались, 2 цифры 0 и 1 и каждый разряд умножается на степень двойки: $100101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5$. Так же нам будет нужна шестнадцатеричная

система счисления, в ней 16 цифр: от 0 до 9 – вполне очевидно, а цифры, соответствующие числам от 10 до 15 – просто первые буквы латинского алфавита: A-F, умножение происходит на степень 16. Приведём пример: $20EF1_{16} = 1 \cdot 16^0 + 15 \cdot 16^1 + 14 \cdot 16^2 + 0 \cdot 16^3 + 2 \cdot 16^4$. На том, как переводить из одной системы счисления в другую я не буду останавливаться, это описано в википедии: https://ru.wikipedia.org/wiki/Позиционная_система_счисления, так же работать с перечисленными четырьмя системами счисления умеет калькулятор в windows.

1.1.2. Память

Для начала договоримся, о том, в чём память будем измерять. Бит (bit) – одна двоичная цифра, т.е. он может принимать значения 0 и 1. Байт (byte) – 8 бит, соответственно он может принимать значения от 0 до 255 (про числа со знаком пока умолчу). Слово (word) – 2 байта, значения от 0 до 65535. Двойной слово (double word) – 4 байта. Счетверённое слово (quad word) – 8 байт. Так же используют сокращения: килобайт – 1024 байт, мегабайт – 1024 килобайт, гигабайт – 1024 мегабайт.

Для работы с оперативной памятью существует адресное пространство, адресное пространство – множество адресов, адрес – некоторое 64-х разрядное число (64 двоичных разряда, т.е. 8 байт). Каждому адресу соответствует 1 байт (ячейка) в памяти вашей программы (в виртуальной памяти и соответственно адрес – виртуальный). Тут надо кое-что уточнить: дело в том, что виртуальная память не просто так называется виртуальной, чтобы работать с памятью по данному виртуальному адресу нужно чтобы данной ячейке виртуальной памяти соответствовала ячейка в физической памяти, иначе произойдёт ошибка.

Каждая программа (весь код и все данные) располагается в своей виртуальной памяти и доступа к чужой памяти у неё нет (на самом деле всё хитрее, и доступ к чужой памяти получить можно, но пока что я не буду об этом упоминать).

1.1.3. Регистры

Регистры – это память, расположенная прямо рядом с процессором, и соответственно доступ к ним гораздо быстрее, чем к оперативной памяти. У современных процессоров куча регистров у каждого из которых своё особое назначение, все эти регистры я здесь перечислять не буду, их полный список можно найти здесь: http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf. У процессора есть 16 регистров общего назначения, каждый из них содержит 64 бита (8 байт): rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15. Так же можно обращаться к их младшим 32-м битам, младшим 16-и битам

и младшим 8 битам. По каким именам обращаться к регистрам и их частям представлено на Рис. 1.1. Так же нам в дальнейшем понадобится регистр

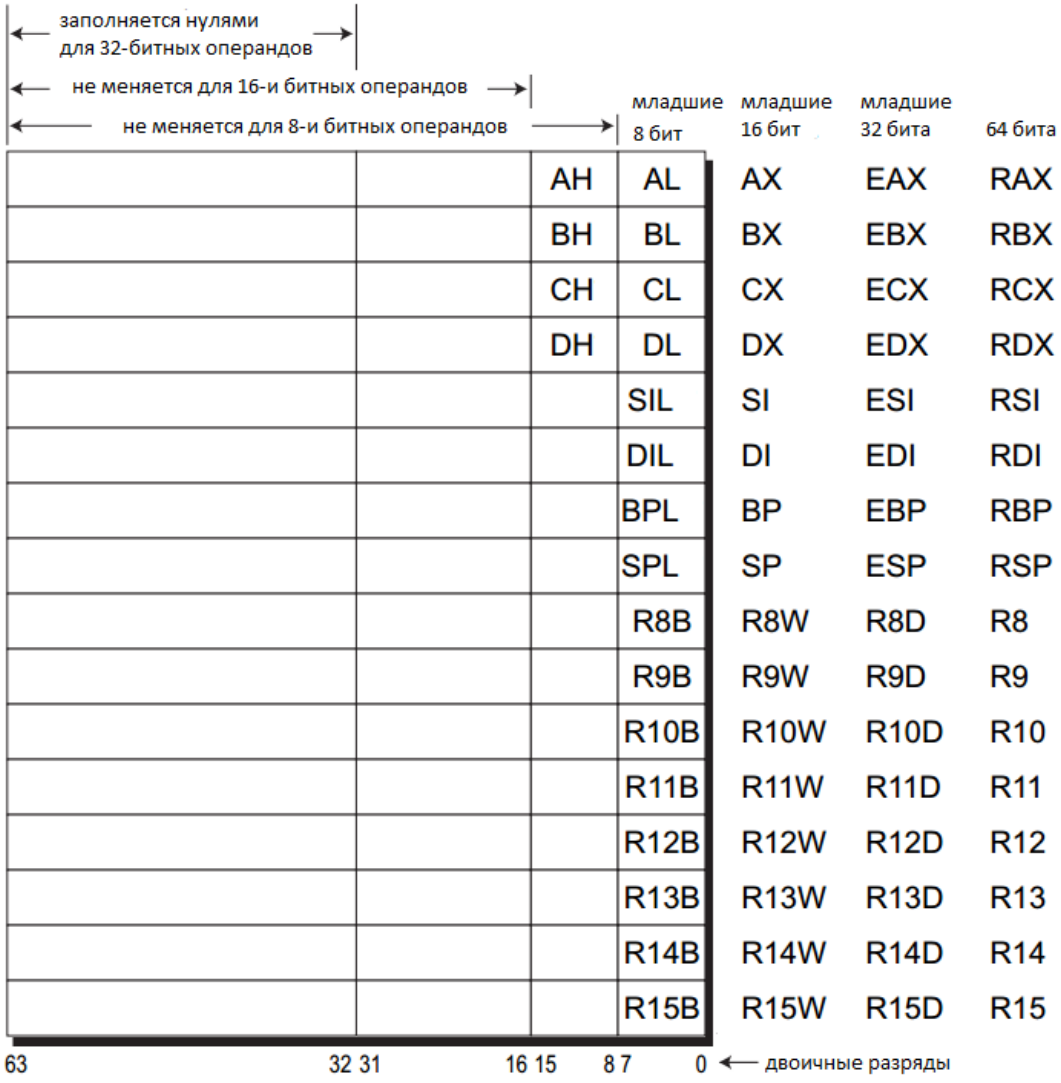

Рис. 1.1. Регистры общего назначения, их нижние 32, 16, 8 бит.

флагов, но о нём – в отдельной статье. Ну, а полномочия данной статьи на этом – всё.

1.2. Арифметика, отладчик, первые инструкции

В данной статье мы рассмотрим, как пользоваться отладчиком x64dbg, как выглядит простейшая программа на fasm-е и инструкцию «mov».

1.2.1. Отладчик

Запустите отладчик x64dbg и откройте любой exe-файл, который найдёте(Файл->открыть). После этого отладчик будет выглядеть примерно как на Рис. 1.2.

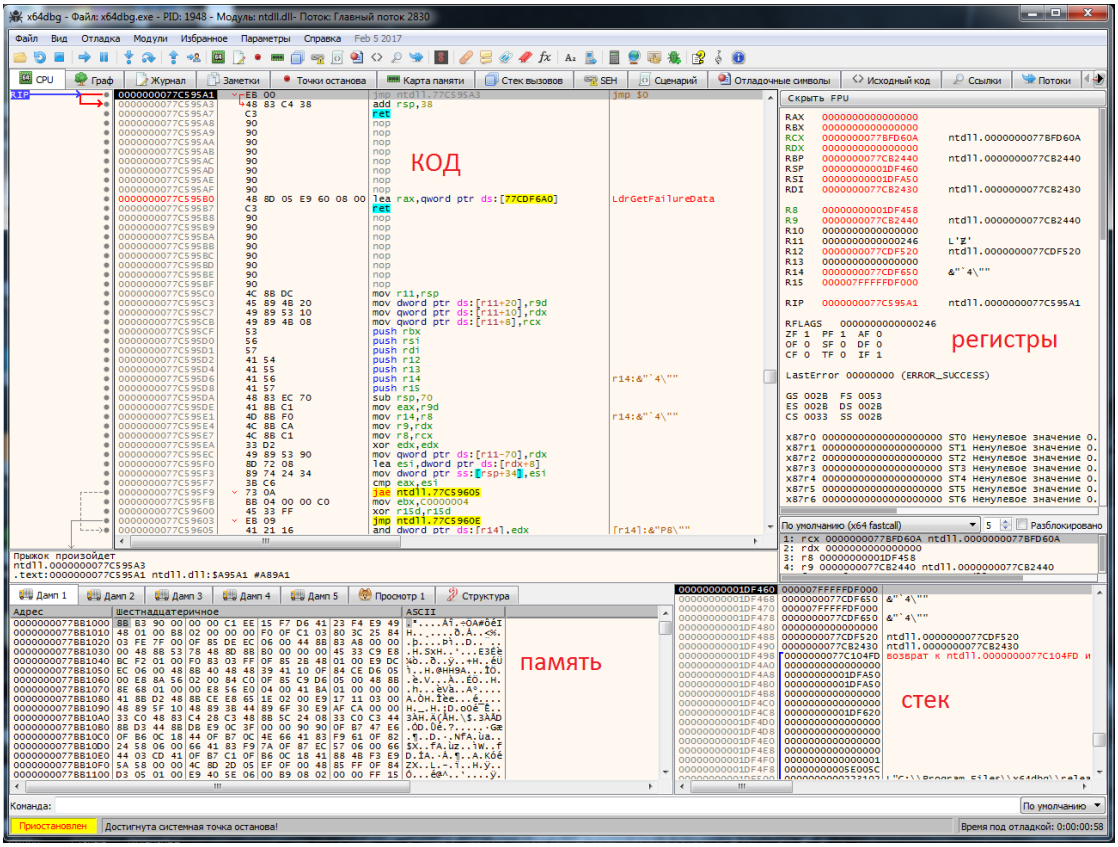

Рис. 1.2. Окно отладчика

В окне, где красным написано «КОД» находится список машинных инструкций: в левом столбце (не который почти пустой с точками, а следующий) находятся адреса начал машинных инструкций. В следующем столбце находится машинный код инструкции, находящейся по этому адресу. В следующем столбце находятся машинные инструкции на языке ассемблера. Ещё в данном окне есть строка, слева от которой находится синяя стрелка и белым по синему написано «RIP».

Не волнуйтесь, никто не умер, просто RIP – регистр, о котором я забыл рассказать в предыдущей статье, он содержит в себе адрес инструкции, следующей за той, которая исполняется в данный момент. Как видно значение этого регистра не меняется, не меняется оно по той причине, что программа приостановлена (это написано в левом нижнем углу).

Там, где я написал «регистры» написаны значения разных регистров, значения регистров общего назначения (те, которые я описывал в предыдущей статье) даны в шестнадцатеричной системе счисления.

Про стек пока ничего говорить не буду, потом к нему вернёмся.

Ну, и последнее окно – память (или дамп). Каждая строка в этом окне соответствует 16-и байтам. В столбце с названием «Адрес» находится адрес начала соответствующей последовательности из 16-и байтов. В столбце «Шестнадцатеричное» идут подряд (слева направо) значения 16-и байт в шестнадцатеричной системе счисления. В столбце «ASCII» - символы, соответствующие этим 16-и байтам (из таблицы ASCII). Формат представления дампа можно менять.

Щёлкнув 2 раза по регистру в окне регистров, можно менять их значения, так же щёлкнув дважды в дампе можно менять значения соответствующих байт.

Из горячих клавиш: F9 – запустить программу, F7 – сделать один шаг, F8 – сделать шаг, отличается от предыдущего на исполнении инструкции call, если при нажатии F7 отладчик остановится на той инструкции, на которую указывает call, то F8 приостановит отладчик сразу после возврата на инструкцию, стоящую после call.

Как видно на Рис. 1.2 программа под отладчиком находится в приостановленном состоянии, но код, на котором остановлена программа – это не код самой программы, это системный код. Чтобы дойти до начала кода программы нажмите F9, после нажатия программа опять остановится, но в правой колонке напротив инструкции, на которую указывает RIP, написано «EntryPoint», это – начало кода самой программы (см. Рис. 1.3).

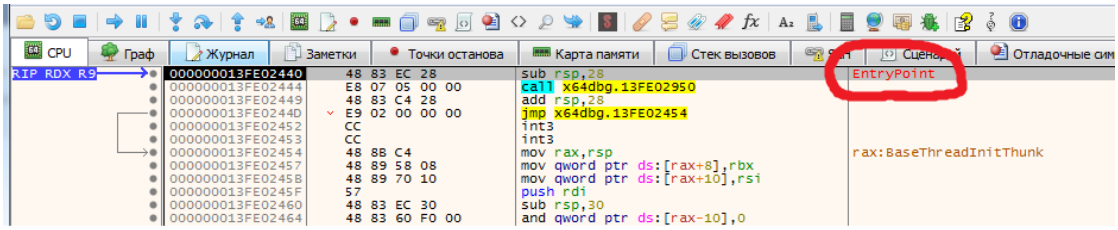

Рис. 1.3.

1.2.2. Целые числа со знаком

Перед тем, как приступать к прогрпммированию, нужно выяснить, как представлены в памяти целые числа.

Т.к. ячейка памяти - один байт, то число нужно разделить на группы по 8 бит (т.е. на байты). Сначала в памяти располагается младший байт, а в конце старший. Т.о. если дано 2-х байтовое число 0xFFEE (0x обозначает 16-ричную систему счисления, к этому мы ещё вернёмся), то в памяти сначала будет располагаться байт 0xEE, а за ним - 0xFF. Такой порядок ещё называется «little-endian».

Теперь перейдём к отрицательным числам. Отрицательные числа хранятся в т.н. дополнительном коде (вообще хранить их можно как угодно, но процессор работает именно с таким кодом), чтобы изменить знак числа в данном коде, нужно инвертировать все его биты (0 заменить на 1 и 1 - на 0) и прибавить к нему, как к положительному числу, единицу. На первый взгляд дополнительный код может показаться довольно сложным, но у него есть свои приимущества, например алгоритмы сложения и вычетания для чисел любого знака одинаковы.

1.2.3. Программа

Здесь я приведу пример простой программы.

Листинг 1.1. Простая программа

```
format PE64 console
; Заголовок, говорим, что нужен 64-х битный формат
; файла и у нашей программы будет консоль

entry start
; Точка входа, показывает, где начинается программа
include 'win64w.inc'; Стандартные макросы

section '.code' readable executable; секция кода

proc start; Начало нашего кода

        ; 3 инструкции "nop",
        ;данная инструкция ничего не делает
        nop
        nop
        nop

        invoke ExitProcess,0; завершение работы программы
        ret
endp

section '.idata' readable;секция импорта
data import
library kernel32,'kernel32.dll'
include 'api\kernel32.inc'
end data
```

Я пока не буду пояснять большую часть приведённого кода. Скажу лишь, что точка с запятой - начало комментария, всё содержимое строки, стоящее справа от неё ассемблер игнорирует. Скомпилируйте программу и откройте её в отладчике, затем нажмите F9, чтобы перейти на точку входа. На точке входа вы увидите примерно следующее:

```
1          push rbp
2          mov rbp,rsp
3          nop
4          nop
5          nop
```

первые две инструкции нам не важны, а следующие три - те самые три nop-а, которые мы написали в коде (Листинг 1.1). Можете пройтись по функции кнопкой F8. Вы увидите, что «nop» действительно ничего не делает.

Теперь рассмотрим первую инструкцию, которая делает что-то полезное. У каждой инструкции есть операнды, они могут быть как явными, так и неявными. Явные операнды - это операнды, которые вы указываете при написании машинной инструкции, неявные - которые вы не указываете, но они подразумеваются. Операнд может быть одного из трёх типов:

1) Регистр.
2) Память.
3) Непосредственное значение (константа).

Количество явных операндов «память» может быть не больше одного.

Первая инструкция, которую мы рассмотрим это - «mov». Её синтаксис следующий:

```
1          mov op1,op2
```

эта инструкция записывает значение операнда op2 в op1. op1 может быть либо памятью, либо регистром, op2 - либо памятью, либо регистром, либо константой. Операнд-константа может быть записан в 4-х разных системах счисления:

1) Десятичная: просто пишем число.
2) Двоичная: после числа из нулей и единиц без пробела пишется «b».
3) Восьмеричная: после числа без пробела пишется «o».
4) Шестнадцатеричная: либо после числа без пробела пишется «h», либо перед числом без пробела пишется «0x».

Напишите вместо nop-ов например: mov eax,0xFFEE, пройдитесь отладчиком, и вы увидите, что значение регистра rax изменилось.

Ещё инструкции и обращение с памятью - в следующей статье.