

## Лабораторная работа 3: Black Box

### Задача

Дан «чёрный ящик» в виде объектного файла, реализующего одну целую функцию от двух целых чисел. Написать на языке ассемблера программу, которая используя эту функцию, выясняет, что она вычисляет.

- Функция реализована на C и имеет следующий интерфейс:  
`int deadbeef(int a, int b);`
- Функция `deadbeef` реализует одну из следующих операций:

<code>a + b</code>	сложение
<code>a - b</code>	вычитание
<code>a * b</code>	умножение
<code>a / b</code>	деление
<code>a % b</code>	остаток
<code>a &amp; b</code>	побитовое <b>and</b>
<code>a   b</code>	побитовое <b>or</b>
<code>a ^ b</code>	побитовое <b>xor</b>

Соответственно, ваша программа должна вывести знак или название операции.

- Функция на самом деле называется не `deadbeef`.
- Для анализа объектных файлов использовать один из следующих инструментов:
  - `objdump` (из комплекта [MinGW](#) под Windows или набора [GNU binutils](#) в Linux).
  - `dumpbin` (из комплекта [MS Visual Studio Express](#) под Windows).

### Указания

1. Скопируйте [свой вариант](#) чёрного ящика (для Linux выбирайте из [другого списка](#)).
2. Чтобы перечислить все символы, экспортируемые объектным файлом, используйте `dumpbin` (или `objdump`):

```
1 dumpbin /symbols blackbox.obj
```

В полученном списке вы найдёте только один экспортируемый символ — это и будет название функции.

3. Чтобы использовать функцию `deadbeef`, нужно передать через стек два аргумента, а потом результат (который функция поместила в регистр `eax`) сравнить с ожидаемым.

```
1 push dword 456
2 push dword 123
3 call deadbeef
```

Такая запись эквивалентна вызову `deadbeef(123, 456)` в C.

4. Чтобы определить, какая операция выполняется в чёрном ящике, используйте конструкцию ветвления. Сравните результат работы чёрного ящика с ожидаемым. Например, если для пары чисел 123 и 456 чёрный ящик выдал 579, значит он считает сумму. В x86 ветвление реализуется с помощью флагов и инструкций перехода.

Простейшим переходом является инструкция безусловного перехода `jmp` (от англ. `jump`), которая имеет следующий синтаксис:

1 `jmp <адрес> ; где <адрес> - адрес ячейки памяти, куда передается управление`

Аналогом `jmp` в некоторых ЯВУ (Pascal, C/C++) является оператор `goto`.

Кроме безусловного перехода, в x86 существуют команды условного перехода. Команды условного перехода учитывают состояние битов служебного регистра `eflags` (их называют «флагами»).

Общий синтаксис инструкций условного перехода имеет следующий вид:

1 `j<сс> <адрес> ; где <сс> - код условия`

Список всех кодов условий можно посмотреть в [этом справочнике](#). Ниже приведён список тех, которые могут вам понадобиться в ходе этой лабораторной работы.

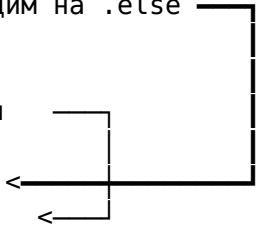
Команда	Перейти, если
<code>jz</code>	нуль (zero)
<code>jnz</code>	не нуль (not zero)
<code>je</code>	равно (equal)
<code>jne</code>	не равно (not equal)
<code>jg</code>	больше (greater)
<code>jge</code>	больше или равно (greater or equal)
<code>jl</code>	меньше (less)
<code>jle</code>	меньше или равно (less or equal)

Команда `cmp` (`compare`) выполняет операцию вычитания над операндами, однако никуда не сохраняет результат — только выставляет флаги в регистре `eflags`. Синтаксис команды `cmp` следующий:

1 `cmp <операнд1>, <операнд2>`

Ниже приведен пример, который демонстрирует реализацию конструкции ветвления на языке ассемблера. Если регистр `eax` хранит 9, то ему присваивается значение 1 (метка `.then`), иначе присваивается `-1` (метка `.else`).

```
1      cmp eax, 9 ; сравниваем значение регистра eax с девяткой
2      jne .else ; если не равно, то переходим на .else
3  .then:
4          mov eax, 1 ;
5          jmp endif ; переход в конец ветвления
6  .else:
7          mov eax, -1
8  .endif: ; конец ветвления
```



## **Справочники**

1. [Руководство по link.](#)
2. [Руководство по nasm.](#)
3. [Руководство по соглашениям о вызовах.](#)
4. [Руководство по командам x86.](#)
5. [Руководство по dumpbin](#)