

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра информационной безопасности

ОТЧЕТ

по учебной практике

**Тема: Основы исследования функциональности программного
обеспечения и разработки функциональных аналогов**

Студент гр.

Преподаватель

Санкт-Петербург

2022

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студент

Группа

Тема отчета: Основы исследования функциональности программного обеспечения и разработки функциональных аналогов.

Задание: изучить функциональность приложения, полученного в рамках задания по учебной практике, с помощью инструментальных средств реверс-инжиниринга и реализовать функциональные аналоги на ассемблере и на языке программирования высокого уровня С или С++. Вариант задания для учебной практики: 6.

Содержание пояснительной записки: введение, анализ функциональности приложения, реализация функциональных аналогов, результаты тестирования реализованных функциональных аналогов, заключение, список использованных источников, приложение 1 – исходный код функционального аналога на ассемблере, приложение 2 – исходный код функционального аналога на языке программирования С.

Предполагаемый объем пояснительной записки:

Не менее 20 страниц.

Сроки прохождения практики: 29.06.2022 – 12.07.2022

Дата сдачи отчета: 12.07.2021

Дата защиты отчета: 12.07.2021

Студент

Преподаватель

АННОТАЦИЯ

Учебная практика посвящена изучению основных инструкций процессора архитектуры IA-32 (x86) и реализации функциональных аналогов выданного .exe файла на ассемблере и на C/C++. Для создания использовались среда разработки для программирования на ассемблере «FASM editor», дизассемблер «Ghidra» и среда разработки для C «Clion».

Аналоги способны выдавать значения, идентичные оригиналу в пределах допустимых значений.

SUMMARY

The training practice is devoted to the study of the basic instructions of the processor architecture IA-32 (x86) and the implementation of functional analogues issued.exe file in assembly language and in C/C++. The development environment for assembly programming «FASM editor», disassembler «Ghidra» and development environment for C «Clion» were used for creation.

Analogs are able to output values identical to the original within acceptable values.

СОДЕРЖАНИЕ

	Введение	5
1.	Анализ функциональности приложения	6
2.	Реализация функциональных аналогов	14
3.	Результаты тестирования реализованных функциональных аналогов	16
	Заключение	19
	Список использованных источников	20
	Приложение 1. Исходный код функционального аналога на ассемблере	21
	Приложение 2. Исходный код функционального аналога на языке программирования С	24

ВВЕДЕНИЕ

Тема данной учебной практики очень актуальна. Существует целая сфера, в которой занимаются исследованием некоторого готового устройства или программы, а также документации на него с целью понять принцип его работы. Это называют обратной разработкой. Деятельность в данной области очень тесно связана с пониманием машинного кода. Самым распространенным языком низкого уровня является ассемблер.

В курсовой работе нам была поставлена задача реализовать функциональные аналоги на ассемблере и на C/C++ для предложенной программы в виде файла формата EXE.

Разрешить данную задачу получилось посредством дизассемблирования исходного файла и написания аналогичных инструкций на ассемблере и C.

В курсовой работе удалось достигнуть успешного написания рабочего кода, который способен выдавать значения, идентичные оригиналу.

Подробнее о процессе дизассемблирования файла можно прочитать в последующем изложении теоретического материала.

1. АНАЛИЗ ФУНКЦИОНАЛЬНОСТИ ПРИЛОЖЕНИЯ

Чтобы найти ключевой алгоритм .exe файла, воспользуемся дизассемблером «Ghidra».

В начале необходимо создать новый проект. Для этого выберем вкладку «File», затем «New project», как показано на рисунке 1.

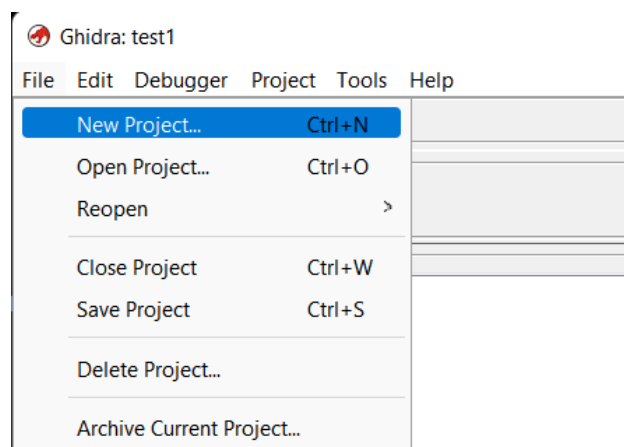


Рисунок 1 – Создание нового проекта в Ghidra

Далее выбираем «Non-Shared Project» и кликаем «Next», как на рисунке 2.

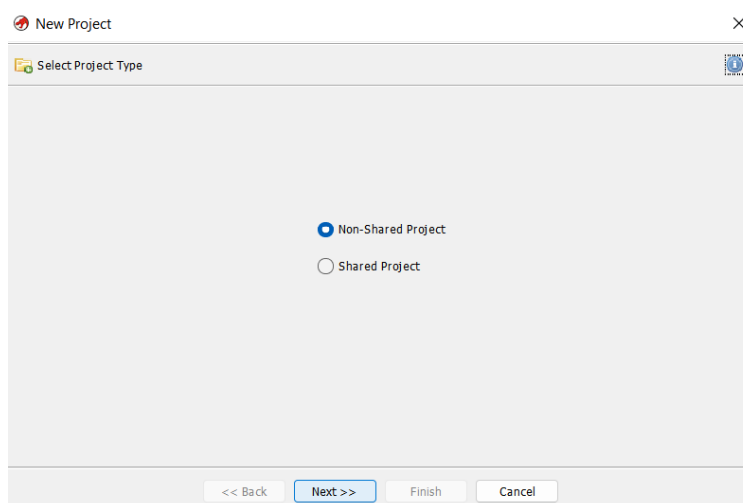


Рисунок 2 – Выбор типа проекта

После выбора типа проекта, надо определить его расположение и задать название. Я назову проект «test_project». Для завершения создания проекта надо нажать кнопку «Finish». Пример ввода необходимой информации изображен на рисунке 3.

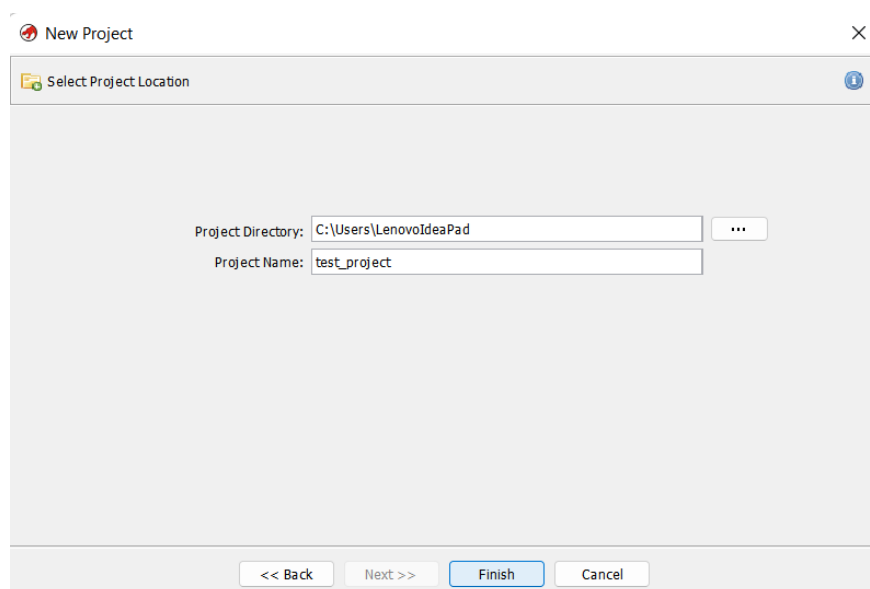


Рисунок 3 – Выбор расположения и названия проекта

Проект создан и отображается в окне, как показано на рисунке 4.

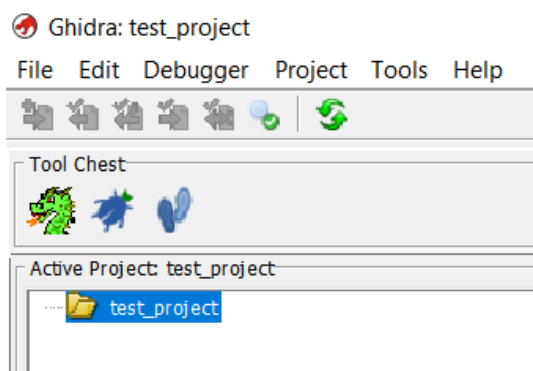


Рисунок 4 – Отображение проекта

Теперь, чтобы загрузить .exe файл, необходимо нажать «File», затем «Import file» и выбрать нужный файл, как на рисунке 5. Успешно импортированный файл показан на рисунке 6.

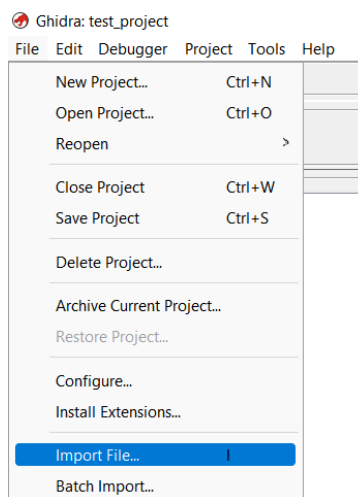


Рисунок 5 – Импорт файла

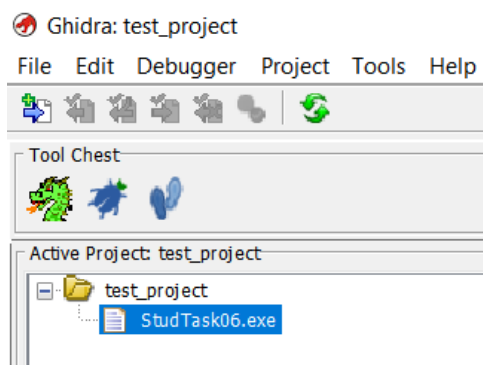


Рисунок 6 – Успешно импортированный файл

Теперь, чтобы открыть загруженный файл в дизассемблере, необходимо кликнуть по файлу два раза ПКМ. Открывшиеся окно показано на рисунке 7.

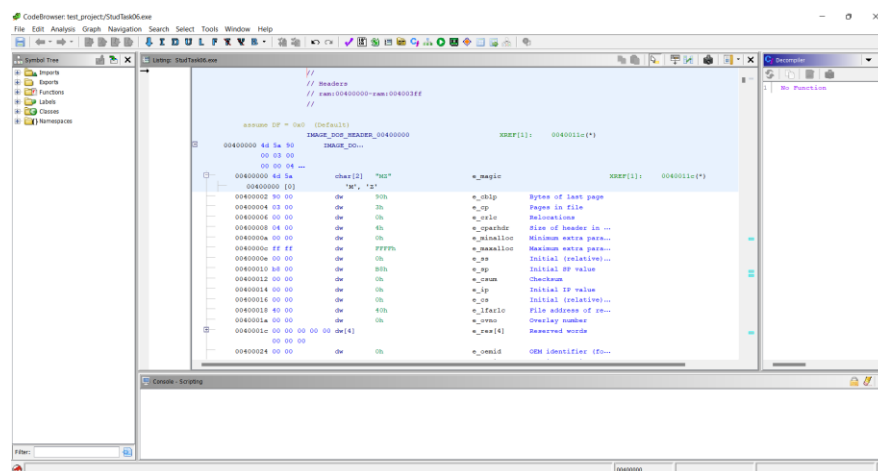


Рисунок 7 – Открытие файла в дизассемблере

Чтобы быстро найти точку входа, необходимо проанализировать файл. Для этого выберем сверху «Analysis», затем «Auto Analyze...», как показано на рисунке 8.

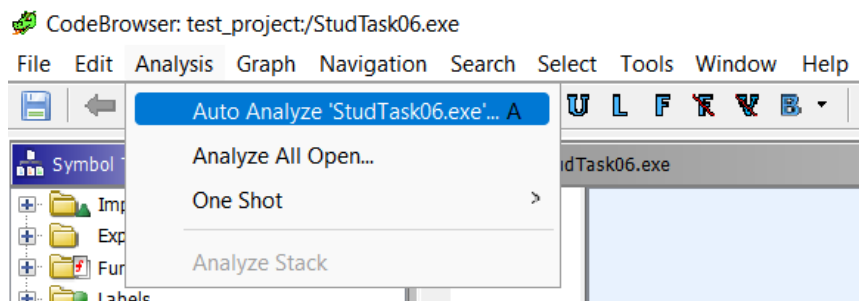


Рисунок 8 – Анализ файла

В открывшемся окне, показанном на рисунке 9, выбираем «Analyze».

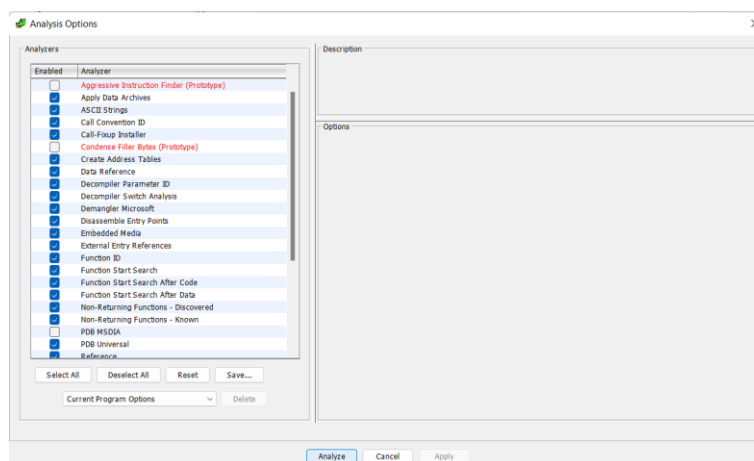


Рисунок 9 – Окно настройки анализа файла

Теперь, после проведенного анализа, мы можем воспользоваться поиском по функциям. В окне слева раскроем вкладку «Functions» и нажмем два раза на «entry» — это точка входа в программу. Действие отображено на рисунке 10.

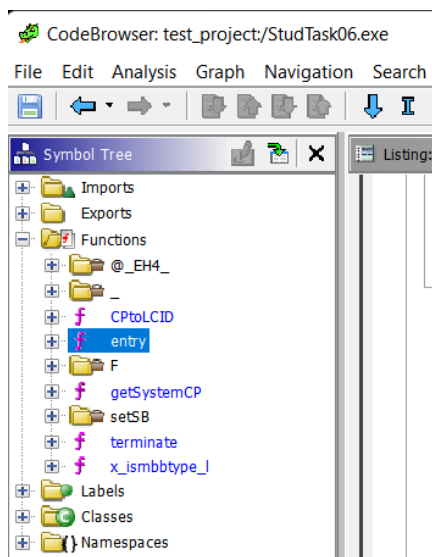


Рисунок 10 – Переход в метку «entry»

После перехода справа, в окне дизассемблирования, появится более понятный код, в котором можно увидеть, что в конце переменной iVar1 присваивается значение функции FUN_00401000. Отрывок дизассемблированного кода показан на рисунке 11.

```

43  _DAT_0040e36c = DAT_0040e368;
44  iVar1 = FUN_00401000();
45  _exit(iVar1);
46  __cexit();
47  return iVar1;
48  }

```

Рисунок 11 – Конец функции «entry» в дизассемблированном виде

Перейдем в функцию FUN_00401000, щелкнув по ее названию два раза. Теперь в дизассемблере справа покажется ее содержимое, а в основной секции посередине – код на ассемблере. Несложно догадаться, что основной алгоритм программы находится между выводом введенных чисел и выводом результата, поэтому перейдем к коду между этими сообщениями, который показан на рисунке 12.

00401079	53	PUSH	EBX	
0040107a	ff 15 00	CALL	dword ptr [->KERNEL32.DLL::GetModuleHandleW]	= 0000c238
	a0 40 00			
00401080	3b c3	CMP	EAX,EBX	
00401082	74 20	JZ	LAB_004010a4	
00401084	8a 08	MOV	CL,byte ptr [EAX]	
00401086	8a 50 01	MOV	DL,byte ptr [EAX + 0x1]	
00401089	33 c0	XOR	EAX,EAX	
0040108b	eb 03	JMP	LAB_00401090	
0040108d	8d	??	8Dh	
0040108e	49	??	49h	I
0040108f	00	??	00h	
		LAB_00401090		XREF[2]: 0040108b(j), 004010a2(j)
00401090	8a 5c 05 f4	MOV	BL,byte ptr [EBP + EAX*0x1 + local_f]	
00401094	02 d9	ADD	BL,CL	
00401096	02 d8	ADD	BL,AL	
00401098	32 da	XOR	BL,DL	
0040109a	88 5c 05 f4	MOV	byte ptr [EBP + EAX*0x1 + local_f],BL	
0040109e	40	INC	EAX	
0040109f	83 f8 05	CMP	EAX,0x5	
004010a2	72 ec	JC	LAB_00401090	

Рисунок 12 – Код алгоритма на ассемблере

CALL dword ptr [->KERNEL32.DLL::GetModuleHandleW]

Выполнение алгоритма начинается с вызова функции GetModuleHandleW, которая берет аргумент из регистра EBX, равный 0, и возвращает значение в регистр EAX.

GetModuleHandleW извлекает дескриптор указанного модуля, если файл был отображен в адресном пространстве вызывающего процесса. Принимает в качестве аргумента указатель на символьную строку с нулем в конце в кодировке Unicode, которая содержит имя модуля. Если этот параметр – NULL или 0, как в нашем случае, GetModuleHandleW возвращает дескриптор файла, используемый чтобы создать вызывающий процесс (.exe файл).

То есть после отработки функции, в EAX содержится адрес начала массива файла формата .exe.

```
CMP EAX, EBX
JZ LAB_00401060
```

Указатель, находящийся в EAX сравнивается с 0, находящимся в EBX и, если условие истинно, происходит переход на метку с выводением результата.

```
MOV CL, byte ptr [EAX]
```

В регистр CL кладется значение размера 8 бит, находящееся по адресу EAX. По адресу EAX находится первый символ дескриптора файла .exe, а именно «M».

```
MOV DL, byte ptr [EAX + 0x1]
```

В регистр DL кладется значение размера 8 бит, находящееся по адресу EAX + 1. По адресу EAX + 1 находится второй символ дескриптора файла .exe, а именно «Z».

```
XOR EAX, EAX
```

Обнуление регистра EAX.

```
JMP LAB_00401090
```

Переход на метку LAB_00401090.

```
MOV BL, byte ptr [EBP + EAX * 0x1 + local_f]
```

В регистр BL кладется значение размера 8 бит, находящееся по адресу EBP + EAX * 0x1 + local_f.

```
ADD BL, CL
```

Операция складывает значения регистров BL и CL и записывает результат в BL.

```
ADD BL, AL
```

Операция складывает значения регистров BL и AL и записывает результат в BL.

```
XOR BL, DL
```

Операция выполняет побитовое исключающее или между BL и DL и записывает результат в BL.

```
MOV byte ptr [EBP + EAX * 0x1 + local_f], BL
```

Помещение значение BL обратно в массив по адресу EBP + EAX * 0x1 + local_f.

```
INC EAX
```

Операция инкремента для EAX

```
CMP EAX, 0x5
```

```
JC LAB_00401090
```

Сравнение значения регистра EAX с 5 и перемещение на метку вывода результата в случае, если значение EAX не меньше 5.

2. РЕАЛИЗАЦИЯ ФУНКЦИОНАЛЬНЫХ АНАЛОГОВ

Из изученного кода приложения можно сделать вывод, о том, что алгоритм проходит в цикле по всем введенным элементам и выполняет над каждым один и тот же набор арифметических операций, зависящий от самого элемента и его номера по порядку, и записывает результат на изначальное место. В конце выводится измененный таким образом массив чисел.

Если условно ввести термины «элемент» и его «индекс» («индекс» принимает целые значения от 0 до 5), то преобразование над конкретным элементом, выглядит так:

$$(\text{элемент} + M + \text{индекс} \wedge Z) \% 256,$$

где M – номер 'M' по таблице ASCII, Z – номер 'Z' по таблице ASCII, + - операция сложения, \wedge - операция побитового исключающего или, % - деление по модулю.

Следует пояснить, для чего в преобразовании нужно деление по модулю 256. Дело в том, что в дизассемблере показано, что в действиях участвуют регистры размером 8 бит, и, когда происходит переполнение, то бишь значение в регистре превышает 256, число записывается полностью в соответствующий 16-битный регистр, а в 8-битном остается лишь его часть. Такое переполнение можно симитировать делением числа по модулю 256.

Перейдем к описанию инструкций процессора архитектуры IA-32 (x86), использованных при реализации функционального аналога на ассемблере.

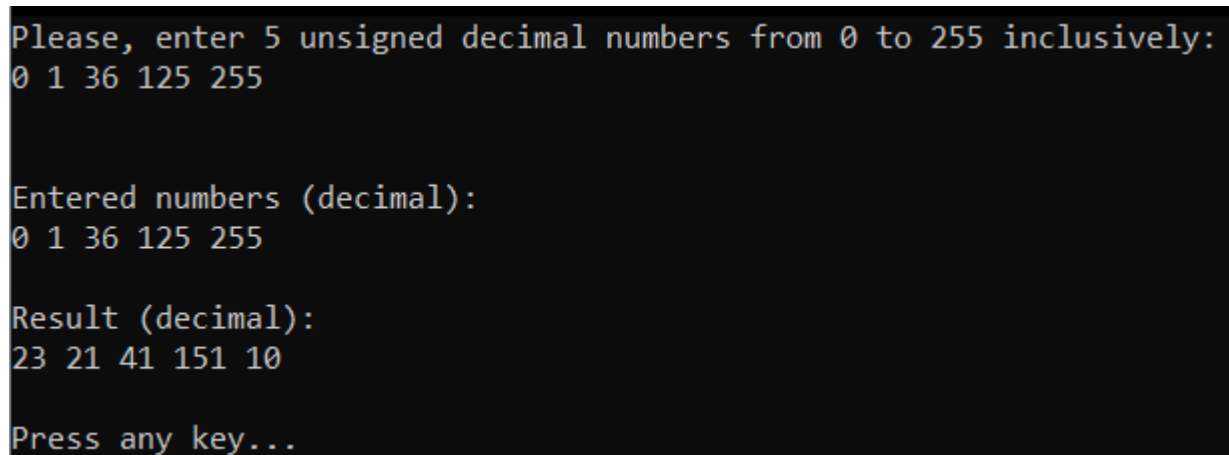
- `cinvoke` – макрос из «win32a.inc», вызывающий функции из стандартной библиотеки языка C;
- `invoke` - макрос из «win32a.inc», вызывающий импортированные функции;
- `push` - инструкция с одним операндом, размещающая операнд в стеке. Команда уменьшает значение регистра-указателя стека

SP/ESP на 2(4) и затем записывает значение источника в вершину стека;

- `pop` – инструкция с одним операндом, извлекающая значение из стека. Команда восстанавливает содержимое вершины стека в регистр, ячейку памяти или сегментный регистр, после чего содержимое ESP/SP увеличивается на 4 байта для `use32` и на 2 байта для `use16`. Недопустимо восстановление значения в сегментный регистр;
- `xor` – инструкция с двумя операндами, помещающая значение операции исключающего или между операндами в первый операнд;
- `mov` - инструкция с двумя операндами, помещающая значение второго операнда в первый операнд. Но невозможно помещать значения из памяти в память;
- `add` - инструкция с двумя операндами, увеличивающая значение первого операнда на значение второго операнда. Но невозможно увеличивать значение из памяти значением из памяти;
- `inc` - инструкция с одним операндом, увеличивающая значение операнда на 1;
- `cmp` – инструкция с двумя операндами, сравнивающая их методом вычитания. По результату сравнения устанавливаются соответствующие флаги;
- `jge` – инструкция условного перехода на метку. Переход выполняется, если $ZF = 0$, $SF = OF$;
- `jmp` – инструкция безусловного перехода на метку.

3. РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ РЕАЛИЗОВАННЫХ ФУНКЦИОНАЛЬНЫХ АНАЛОГОВ

Для первого примера возьмем пять чисел: 0, 1, 36, 125, 255 и посмотрим, какой результат выдаст функциональный аналог, реализованный на С. Результат представлен на рисунке 13.



```
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
0 1 36 125 255

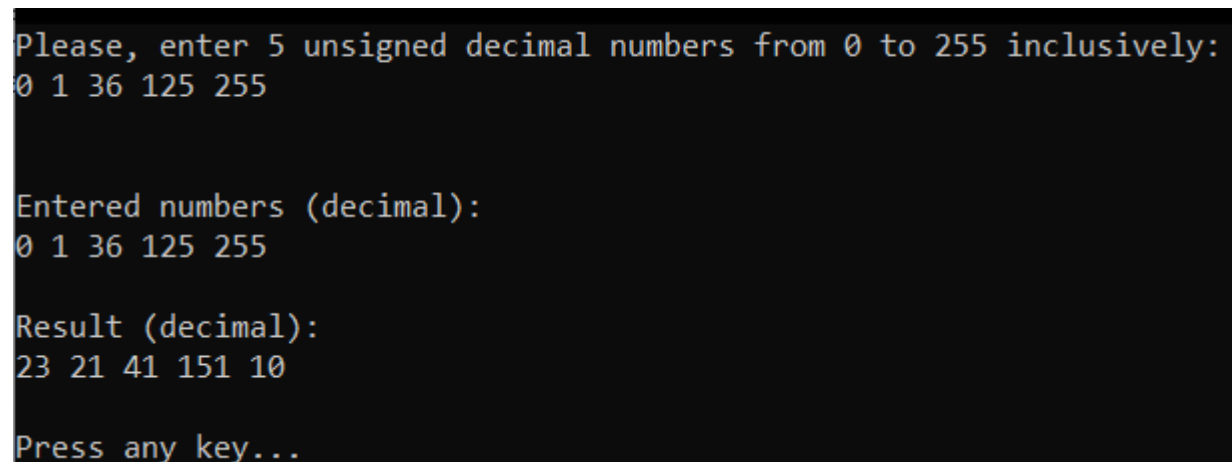
Entered numbers (decimal):
0 1 36 125 255

Result (decimal):
23 21 41 151 10

Press any key...
```

Рисунок 13 – Результат работы аналога на С

Теперь посмотрим, какой результат выведет функциональный аналог, реализованный на ассемблере. Результат представлен на рисунке 14.



```
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
0 1 36 125 255

Entered numbers (decimal):
0 1 36 125 255

Result (decimal):
23 21 41 151 10

Press any key...
```

Рисунок 14 - Результат работы аналога на ассемблере

Теперь посмотрим результат работы на тех же числах исходного .exe файла. Результат представлен на рисунке 15.

```
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
0 1 36 125 255

Entered numbers (decimal):
0 1 36 125 255

Result (decimal):
23 21 41 151 10

Press any key...
```

Рисунок 15 - Результат работы исходного .exe файла

Как видно из рисунков 13 – 15, результаты обоих аналогов совпадают с оригиналом.

Для второго примера возьмем пять чисел: 6, 66, 166, 116, 16 и посмотрим, какой результат выдаст функциональный аналог, реализованный на С. Результат представлен на рисунке 16.

```
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
6 66 166 116 16

Entered numbers (decimal):
6 66 166 116 16

Result (decimal):
9 202 175 158 59

Press any key...
```

Рисунок 26 – Результат работы аналога на С

Теперь посмотрим, какой результат выведет функциональный аналог, реализованный на ассемблере. Результат представлен на рисунке 17.

```
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
6 66 166 116 16

Entered numbers (decimal):
6 66 166 116 16

Result (decimal):
9 202 175 158 59

Press any key...
```

Рисунок 17 - Результат работы аналога на ассемблере

Теперь посмотрим результат работы на тех же числах исходного .exe файла. Результат представлен на рисунке 18.

```
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
6 66 166 116 16

Entered numbers (decimal):
6 66 166 116 16

Result (decimal):
9 202 175 158 59

Press any key...
```

Рисунок 18 - Результат работы исходного .exe файла

Как видно из рисунков 16 – 18, результаты обоих аналогов совпадают с оригиналом. Таким образом, мы убедились в корректной работе реализованных функциональных аналогов.

ЗАКЛЮЧЕНИЕ

В ходе выполнения задания по учебной практике мною были написаны функциональные аналоги на языке программирования С и на ассемблере. Созданные приложения выдают числовые значения, идентичные оригиналу при корректных входных данных.

Задание по учебной практике, а также информация, почерпанная из дополнительных источников, расширила мой кругозор в области реверс-инжиниринга и программирования на ассемблере.

В ходе написания кода я закрепила знания об основных инструкциях языка «Ассемблер», улучшила навык построения логически структурированной программы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Юров В.И. Assembler: Специальный справочник. 2-е изд. СПб.: Питер, 2005. 412 с.

ПРИЛОЖЕНИЕ 1. ИСХОДНЫЙ КОД ФУНКЦИОНАЛЬНОГО АНАЛОГА НА АССЕМБЛЕРЕ

Файл main.asm:

```
format PE Console
```

```
entry start
```

```
include 'win32a.inc'
```

```
section '.data' data readable writeable
```

```
a dd ?
```

```
b dd ?
```

```
c dd ?
```

```
d dd ?
```

```
e dd ?
```

```
SpecD db '%d %d %d %d %d', 0
```

```
MsgEntry db 'Please, enter 5 unsigned decimal numbers  
from 0 to 255 inclusively:', 0x0D, 0x0A, 0
```

```
MsgEntered db 0x0D, 0x0A, 0x0D, 0x0A, 'Entered numbers  
(decimal):', 0x0D, 0x0A, 0
```

```
MsgRes db 0x0D, 0x0A, 0x0D, 0x0A, 'Result (decimal):',  
0x0D, 0x0A, 0
```

```
MsgKey db 0x0D, 0x0A, 0x0D, 0x0A, 'Press any key...',  
0x0D, 0x0A, 0
```

```
section '.text' code readable executable
```

```
start:
```

```
cinvoke printf, MsgEntry
cinvoke scanf, SpecD, a, b, c, d, e
cinvoke printf, MsgEntered
cinvoke printf, SpecD, [a], [b], [c], [d], [e]
```

```
push [e]
push [d]
push [c]
push [b]
push [a]
```

```
xor ecx, ecx
```

operation:

```
mov al, [esp + ecx * 4]
add al, 'M'
add al, cl
xor al, 'Z'
mov [esp + ecx * 4], al
```

```
inc ecx
cmp ecx, 5
jge result
```

```
jmp operation
```

result:

```
pop [a]
pop [b]
pop [c]
```

```
pop [d]
```

```
pop [e]
```

```
cinvoke printf, MsgRes
```

```
cinvoke printf, SpecD, [a], [b], [c], [d], [e]
```

```
cinvoke printf, MsgKey
```

```
cinvoke getch
```

```
invoke ExitProcess, 0
```

```
section '.idata' data import readable
```

```
    library kernel, 'kernel32.dll', msvcrt,  
'msvcrt.dll'
```

```
import kernel, ExitProcess, 'ExitProcess'
```

```
import msvcrt, puts, 'puts', printf, 'printf', scanf,  
'scanf', getch, '_getch'
```

ПРИЛОЖЕНИЕ 2. ИСХОДНЫЙ КОД ФУНКЦИОНАЛЬНОГО АНАЛОГА НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C

Файл main.c:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    unsigned int array[5];

    printf("Please, enter 5 unsigned decimal numbers
    from 0 to 255 inclusively:\n");
    for (int i = 0; i < 5; i++) scanf("%hhu",
    &array[i]);

    printf("\n\nEntered numbers (decimal):\n");
    for (int i = 0; i < 5; i++) printf("%hhu ",
    array[i]);
    printf("\n");

    for (int i = 0; i < 5; i++) array[i] = (array[i] +
    'M' + i ^ 'Z') % 256;

    printf("\nResult (decimal):\n");
    for (int i = 0; i < 5; i++) printf("%hhu ",
    array[i]);
    printf("\n\nPress any key...\n");
    getch();

    return 0;
}
```