

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. 0362

Мирошников Н.Ю.

Руководитель

Халиуллин Р.А.

Санкт-Петербург

2022

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студент Мирошников Н.Ю.

Группа 0362

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

Задание на практику: 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: 23.

Сроки прохождения практики: 29.06.2022 – 12.07.2022

Дата сдачи отчета: 12.07.2022

Дата защиты отчета: 12.07.2022

Студент

Мирошников Н.Ю.

Руководитель

Халиуллин Р.А.

АННОТАЦИЯ

Учебная практика посвящена изучению функциональности приложения, полученного в рамках задания по учебной практике, с помощью инструментальных средств реверс-инжиниринга и реализации функциональных аналогов на ассемблере и на языке программирования высокого уровня C или C++. Для выполнения работы были использованы: «Ghidra», для создания функциональных аналогов использовались среды разработки «Fresh» и «CodeBlocks», а в качестве отладчика – x64debugger.

SUMMARY

The educational practice is devoted to the study of the functionality of an application obtained as part of a training practice assignment using reverse engineering tools and the implementation of functional analogues in assembler and in a high-level programming language C or C++. To perform the work, the following were used: "Ghidra", the development environments "Fresh" and "CodeBlocks" were used to create functional analogues, and x64debugger was used as a debugger.

СОДЕРЖАНИЕ

	Введение	5
1.	Анализ функциональности приложения	6
2.	Реализация функциональных аналогов	11
2.1.	Анализ функциональности изученного приложения	11
2.2.	Описание инструкций процессора архитектуры IA-32 (x86) используемых в функциональном аналоге	11
3.	Результаты тестирования реализованных функциональных аналогов	13
3.1.	Прогон тестовых значений в исходном исполняемом файле	13
3.2.	Тестирование функционального аналога на Ассемблере	14
3.3.	Тестирование функционального аналога на языке С	15
	Заключение	16
	Список использованных источников	17
	Приложение 1. Исходный код функционального аналога на ассемблере	18
	Приложение 2. Исходный код функционального аналога на языке программирования С	22

ВВЕДЕНИЕ

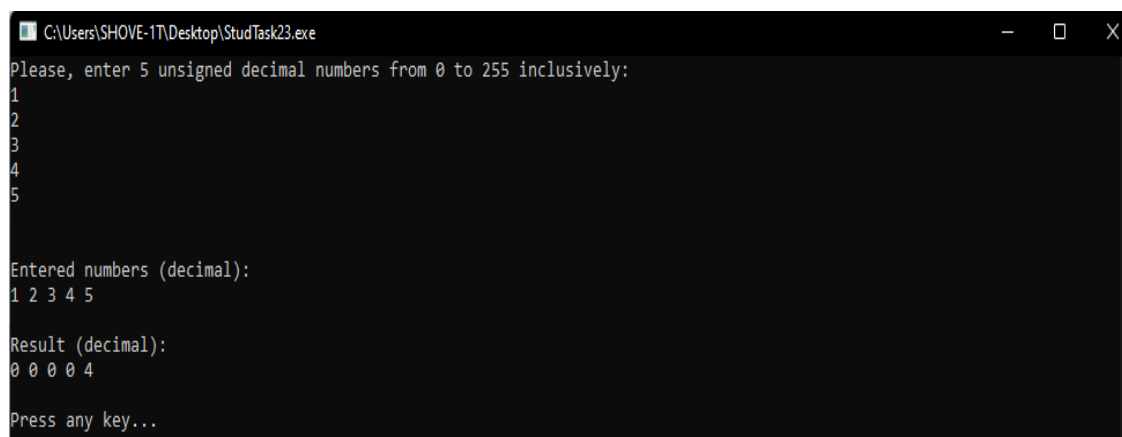
Ассемблер – важнейшая часть индустрии разработки компьютерных программ. Несмотря на то, что программы на Ассемблере получаются более объёмные, чем программы, написанные на языках более высокого уровня, Ассемблер даёт программисту доступ к написанию очень эффективных программ за счёт своей природы языка программирования низкого уровня.

Реверс-инжиниринг (reverse-engineering) – процесс создания функционального аналога объекта по уже существующему образцу, обладающей такими же физическими характеристиками. Реверс-инжиниринг применяется в целях импортозамещения, конкуренции или в случае восстановления процесса производства.

Целью выполнения предоставленного задания является приобретение навыков исследования функциональности приложений и разработки функциональных аналогов. Для проведения анализа предоставленного приложения будет использован дизассемблер Ghidra. После проведения анализа исходного приложения будут разработаны два функциональных аналога: один – на Ассемблере, другой – с использованием языка программирования C. Для работы с Ассемблером будут использованы: компилятор FASM (Flat Assembler), среда разработки Fresh IDE и внешний отладчик x64dbg. После реализации функциональных аналогов будет проведено тестирование работы программ, а его результат внесён в отчёт. На основе полученных данных будет сделан вывод.

1. АНАЛИЗ ФУНКЦИОНАЛЬНОСТИ ПРИЛОЖЕНИЯ

Для начала ознакомимся с работой приложения «StudTask23.exe», дабы получить базовое представление о сути программы. На рисунке 1 представлено запущенное приложение и результаты работы с ним.



```
C:\Users\SHOVE-17\Desktop\StudTask23.exe
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
1
2
3
4
5

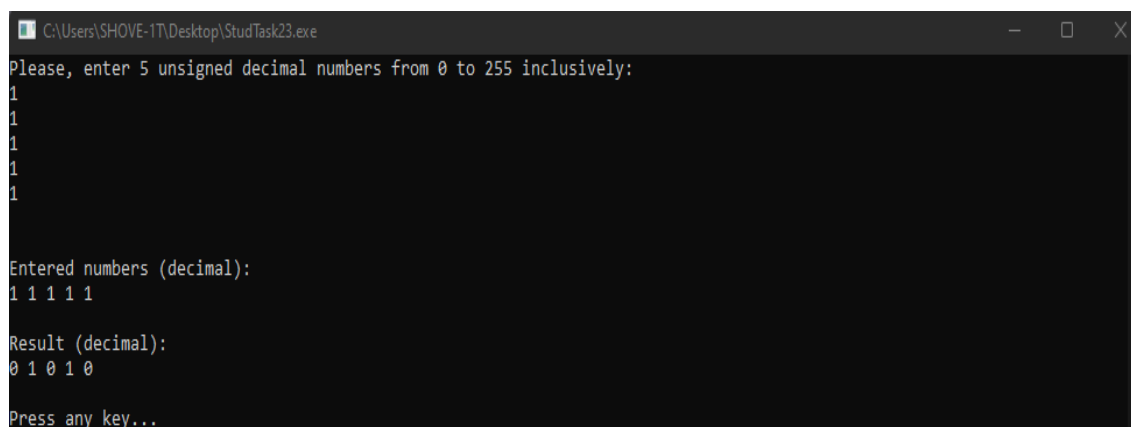
Entered numbers (decimal):
1 2 3 4 5

Result (decimal):
0 0 0 0 4

Press any key...
```

Рисунок 1 – Запуск программы StudTask30.exe

Итак, при запуске программа просит ввести 5 беззнаковых целых чисел в промежутке от 0 до 255. После ввода последовательности от 1 до 5 программа вывела введённую последовательность и ещё некую последовательность из 5 чисел, являющуюся результатом работы. Запустим программу ещё раз и введём другую последовательность.



```
C:\Users\SHOVE-17\Desktop\StudTask23.exe
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
1
1
1
1
1

Entered numbers (decimal):
1 1 1 1 1

Result (decimal):
0 1 0 1 0

Press any key...
```

Рисунок 2 – Пример работы с программой StudTask30.exe

Результат работы программы отличается от предыдущего, из чего можно сделать вывод, что результат работы программы является следствием каких-то вычислений, производимых программой с последовательностью, поданной на ВХОД.

Теперь исследуем устройство приложения с помощью дизассемблера Ghidra. Для этого создадим новый проект, дав ему название Algorithym, после чего перенесём в него исходное приложение «StudTask30.exe». Откроем его в Ghidra CodeBrowser двойным нажатием ЛКМ по перенесённому файлу.

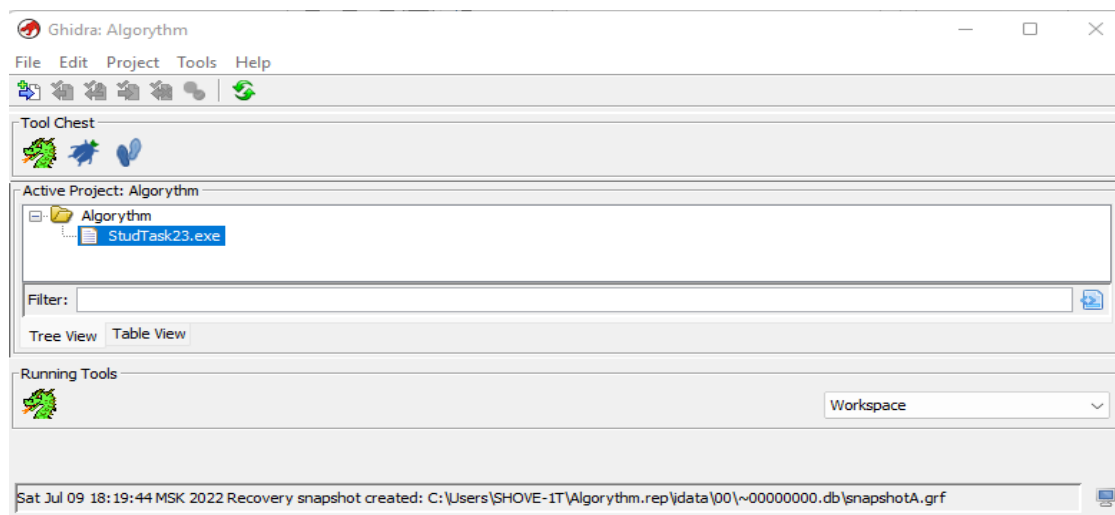


Рисунок 3 – Открытие приложения в Ghidra CodeBrowser

Проведём анализ приложения с использованием стандартных настроек, как показано на рисунке 4.

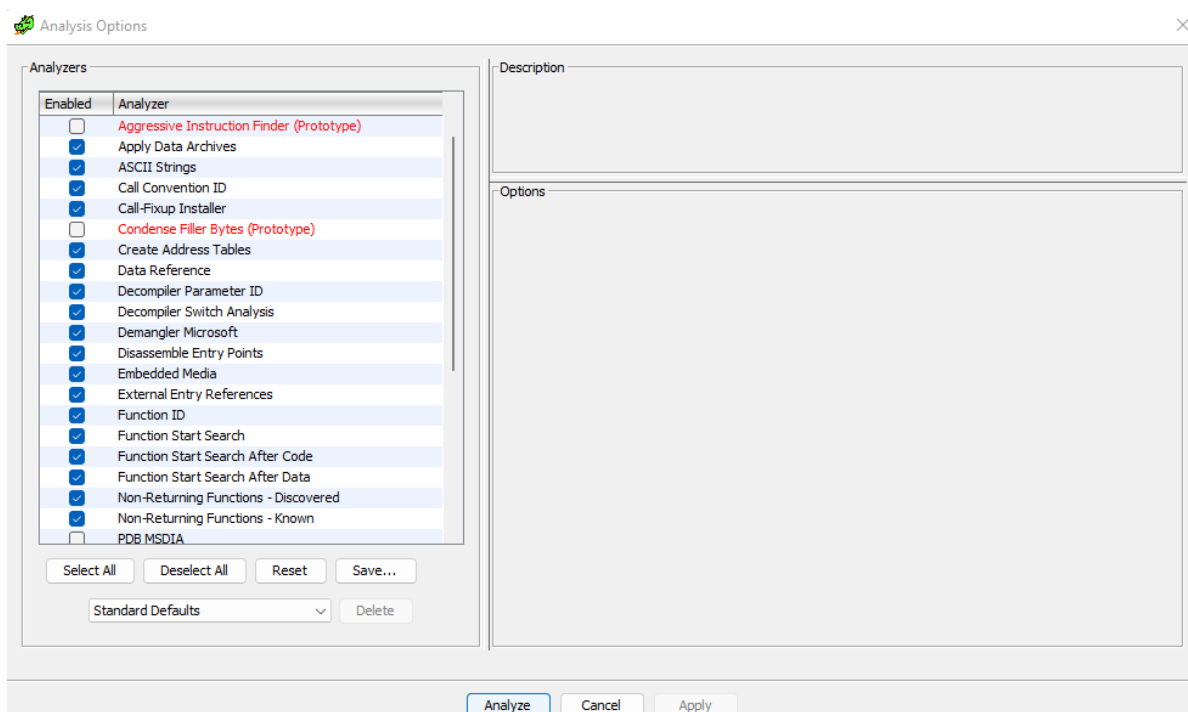


Рисунок 4 – Окно настроек проводимого анализа

По окончании проведения анализа приложения перейдём к секции, содержащей исходный код двойным нажатием ЛКМ по иконке «.text», как показано на рисунке 5.

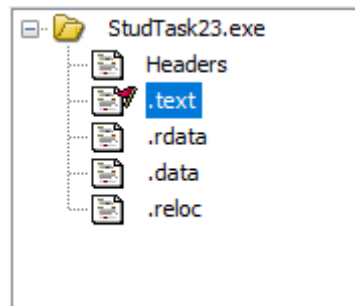


Рисунок 5 – Переход к секции с исходным кодом

На рисунке 6 представлен исходный код на языке C, восстановленный дизассемблером.

```
Decompile: FUN_00401000 - (StudTask23.exe)
2 void FUN_00401000(void)
3
4 {
5     byte bVar1;
6     byte bVar2;
7     uint uVar3;
8     HMODULE pHVar4;
9     undefined4 extraout_EDX;
10    uint uVar5;
11    undefined uVar6;
12
13    uVar3 = DAT_0040d008 ^ (uint)&stack0xffffffff;
14    uVar6 = 0;
15    FID_conflict:_wprintf("Please, enter %u unsigned decimal numbers from 0 to 255 inclusively:\n",5);
16    uVar5 = 0;
17    do {
18        FID_conflict:_wscanf(L"请输入",&stack0xffffffff + uVar5);
19        uVar5 = uVar5 + 1;
20    } while (uVar5 < 5);
21    FID_conflict:_wprintf("\n\nEntered numbers (decimal):\n");
22    uVar5 = 0;
23    do {
24        FID_conflict:_wprintf("%hhu ", (uint)(byte)(&stack0xffffffff)[uVar5]);
25        uVar5 = uVar5 + 1;
26    } while (uVar5 < 5);
27    pHVar4 = GetModuleHandleW((LPCWSTR)0x0);
28    if (pHVar4 != (HMODULE)0x0) {
29        bVar1 = *(byte *)spHVar4->unused;
30        bVar2 = *(byte *)((int)spHVar4->unused + 1);
31        uVar5 = 0;
32        do {
33            (&stack0xffffffff)[uVar5] = ((&stack0xffffffff)[uVar5] ^ bVar2) & bVar1 & (byte)uVar5;
34            uVar5 = uVar5 + 1;
35        } while (uVar5 < 5);
36    }
37    FID_conflict:_wprintf("\n\nResult (decimal):\n");
38    uVar5 = 0;
39    do {
40        FID_conflict:_wprintf("%hhu ", (uint)(byte)(&stack0xffffffff)[uVar5]);
41        uVar5 = uVar5 + 1;
42    } while (uVar5 < 5);
43    FID_conflict:_wprintf("\n\nPress any key...\n");
44    __getch();
45    FUN_004010f0(uVar3 ^ (uint)&stack0xffffffff,extraout_EDX,uVar6);
46    return;
47 }
```

Рисунок 6 – Исходный код на языке C

Проведём анализ полученного исходного кода. Весь исходный код можно разделить на 4 смысловые части: объявление переменных, используемых в коде программы (строки 5-14), запрос чисел у пользователя, запись считанных чисел и их последующий вывод (строки 15-26), преобразование введенных пользователем чисел (строки 27-36) и вывод результата преобразований пользователю, задержка закрытия программы до ввода пользователя, закрытие программы (строки 37-46). Интересующая нас часть кода – третья, она приведена на рисунке 7.

```

27 | pHVar4 = GetModuleHandleW((LPCWSTR)0x0);
28 | if (pHVar4 != (HMODULE)0x0) {
29 |     bVar1 = *(byte *)&pHVar4->unused;
30 |     bVar2 = *(byte *) ((int)&pHVar4->unused + 1);
31 |     uVar5 = 0;
32 |     do {
33 |         (&stack0xffffffff)[uVar5] = ((&stack0xffffffff)[uVar5] ^ bVar2) & bVar1 & (byte)uVar5;
34 |         uVar5 = uVar5 + 1;
35 |     } while (uVar5 < 5);
36 | }

```

Рисунок 7 – Код, преобразующий введенные данные

Как видно из рисунка 7, переменной pHVar4 присваивается значение функции GetModuleHandleW(NULL). При передаче в функцию GetModuleHandleW значения NULL данная функция вернёт дескриптор запущенного приложения (т.е. приложения StudTask23.exe). Поскольку дескриптор запущенного приложения соответствует значению адреса этого приложения, то код строк 29-30 означает, переменной bVar1 присваивается значение первого байта приложения «StudTask23.exe», а переменной bVar2 – значение второго байта приложения «StudTask23.exe». Так как файл StudTask23.exe имеет формат PE (portable executable), значение первых двух байт будет «М» и «Z» соответственно. Как видно на рисунке 6 (строка 18), значения введенных пользователем чисел записывались массив по адресу &stack0xffffffff0. Тогда очевидно, что в цикле, описанном строками 32-35, происходят некие математические действия с каждым введенным пользователем значением. Так каждое из чисел, введенных пользователем, заменяется на другое по формуле:

$$\text{array}[i] = (\text{array}[i] \wedge \text{bVar2}) \& \text{bVar1} \& i,$$

где $\text{array}[i]$ – значение элемента массива, содержащего значения введенных пользователем данных; bVar1 – значение «М»; bVar2 – значение «Z»; i – счётчик цикла от 0 до 4 включительно. Символ « \wedge » соответствует побитовой логической операции «исключающее ИЛИ».

2. РЕАЛИЗАЦИЯ ФУНКЦИОНАЛЬНЫХ АНАЛОГОВ

2.1. Анализ функциональности изученного приложения

В ходе анализа кода программы было выяснено, что программа запрашивает на вход у пользователя 5 целых неотрицательных чисел от 0 до 255 включительно, помещает значения введенных пользователем чисел в массив, выводит их на экран, а затем изменяет их по формуле $array[i] = (array[i] \wedge bVar2) \& bVar1 \& i$. Далее программа выводит изменённые числа.

2.2. Описание инструкций процессора архитектуры IA-32 (x86) используемых в функциональном аналоге

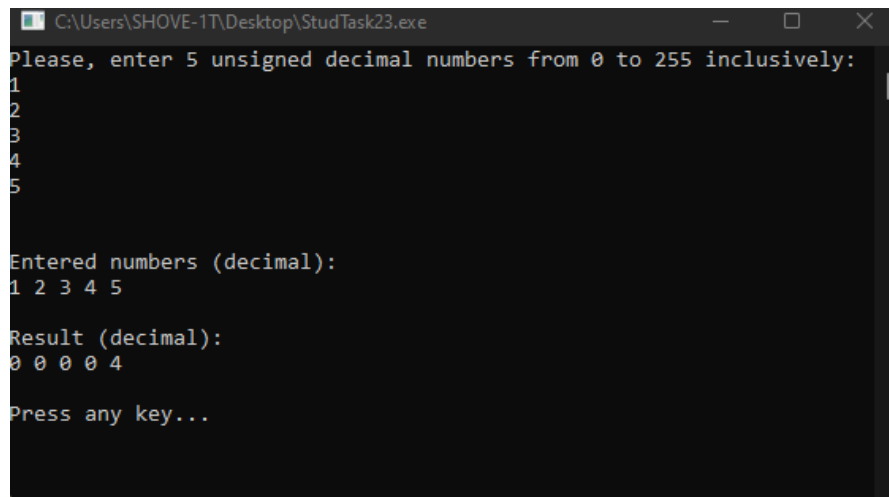
В реализованном функциональном аналоге на Ассемблере используются следующие инструкции процессора:

- `cinvoke/invoke` – сами по себе макроинструкции `invoke` (`cinvoke`) не являются как таковыми инструкциями процессора, а являются именно макроинструкциями, описанными в «%fasm%/win32wxp.inc», использующимися для вызова сторонних импортированных функций Windows API (библиотеки C) с помощью стека.
- `mov` – инструкция с двумя операндами, помещает значение из второго операнда в первый операнд. При этом нельзя использовать память в качестве обоих операндов сразу;
- `cmp` – инструкция с двумя операндами, выполняет вычитание операндов и по результатам сравнения устанавливает флаги;
- `je` – команда условного перехода с одним операндом, если первый операнд инструкции `cmp` равен второму (флаг $ZF = 1$), операнды могут быть любыми, то происходит переход по метке – операнду;
- `inc` – инструкция с одним операндом, увеличивает значение операнда на 1;

- `jmp` – команда безусловного перехода с одним операндом (аналог `GoTo`), по которому происходит переход, расстояние от команды `jmp` до адреса перехода не должно превышать -128 или $+127$ байт;
- `xor` – инструкция с двумя операндами, команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов. Результат записывается на место первого операнда.
- `cld` – инструкция без операндов, снимает флаг `DF` (`DF = 0`), обработка происходит от начала строки к концу;
- `lodsb` – инструкция без операндов, загружает байт из цепочки в регистр `AL`;
- `and` – инструкция с двумя операндами, операция логического умножения. Команда выполняет поразрядно логическую операцию И (конъюнкцию) над битами операндов. Результат записывается на место первого операнда;
- `or` – инструкция с двумя операндами, операция логического сложения. Команда выполняет поразрядно логическую операцию ИЛИ (дизъюнкцию) над битами операндов. Результат записывается на место первого операнда;
- `add` – Цепочечная инструкция `lodsb` используется для записи значения из массива-источника (находящегося по адресу, хранящемуся в регистре `esi`) в регистр `al`. После выполнения инструкции адрес массива-источника при значении в регистре `df` равном `0` – увеличивается на `1`; при значении в регистре `df` равном `1` – уменьшается на `1`;
- `stosb` – Цепочечная инструкция `stosb` используется для записи значения из регистра `al` в массив-приёмник (находящийся по адресу, хранящемуся в регистре `edi`). После выполнения инструкции адрес массива-приёмника при значении в регистре `df` равном `0` увеличивается на `1`; при значении в регистре `df` равном `1` – уменьшается на `1`;

3. РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ РЕАЛИЗОВАННЫХ ФУНКЦИОНАЛЬНЫХ АНАЛОГОВ

3.1. Прогон тестовых значений в исходном исполняемом файле



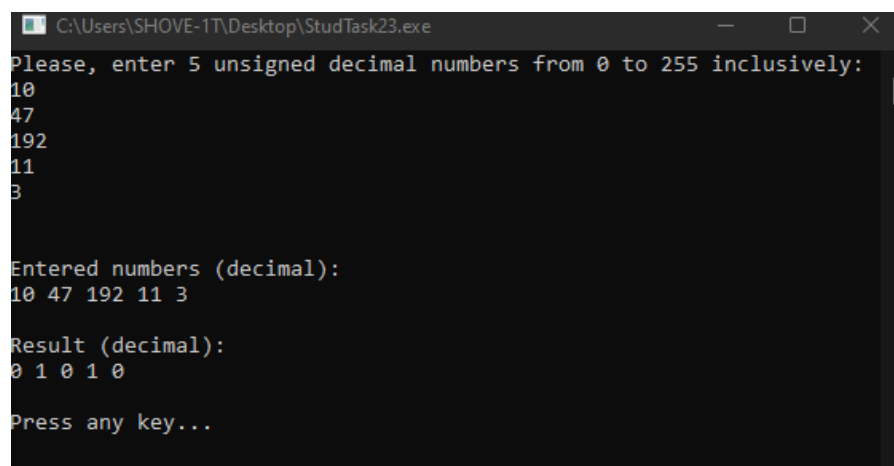
```
C:\Users\SHOVE-1T\Desktop\StudTask23.exe
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
1
2
3
4
5

Entered numbers (decimal):
1 2 3 4 5

Result (decimal):
0 0 0 0 4

Press any key...
```

Рисунок 10 – Результат теста №1 исходной программы



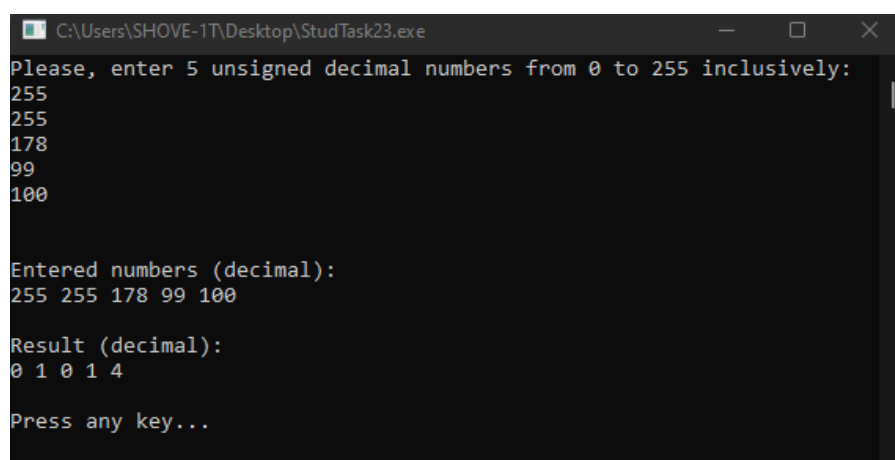
```
C:\Users\SHOVE-1T\Desktop\StudTask23.exe
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
10
47
192
11
3

Entered numbers (decimal):
10 47 192 11 3

Result (decimal):
0 1 0 1 0

Press any key...
```

Рисунок 11 – Результат теста №2 исходной программы



```
C:\Users\SHOVE-1T\Desktop\StudTask23.exe
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
255
255
178
99
100

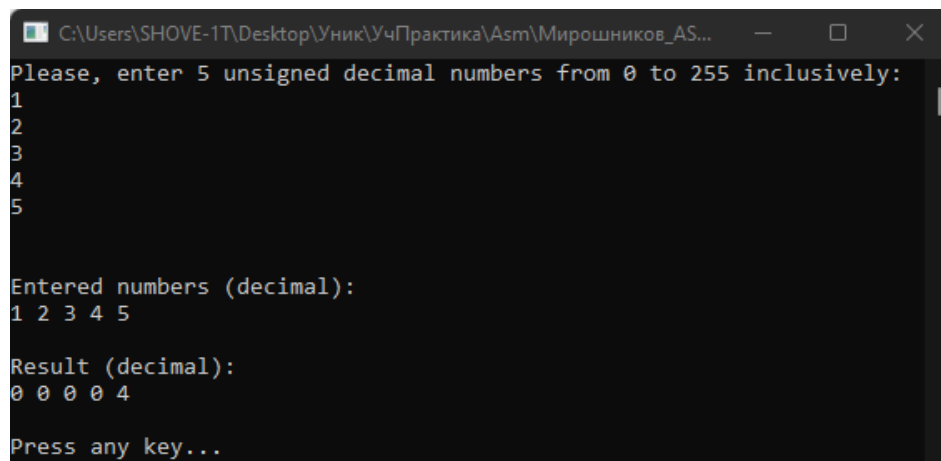
Entered numbers (decimal):
255 255 178 99 100

Result (decimal):
0 1 0 1 4

Press any key...
```

Рисунок 12 – Результат теста №3 исходной программы

3.2. Тестирование функционального аналога на Ассемблере



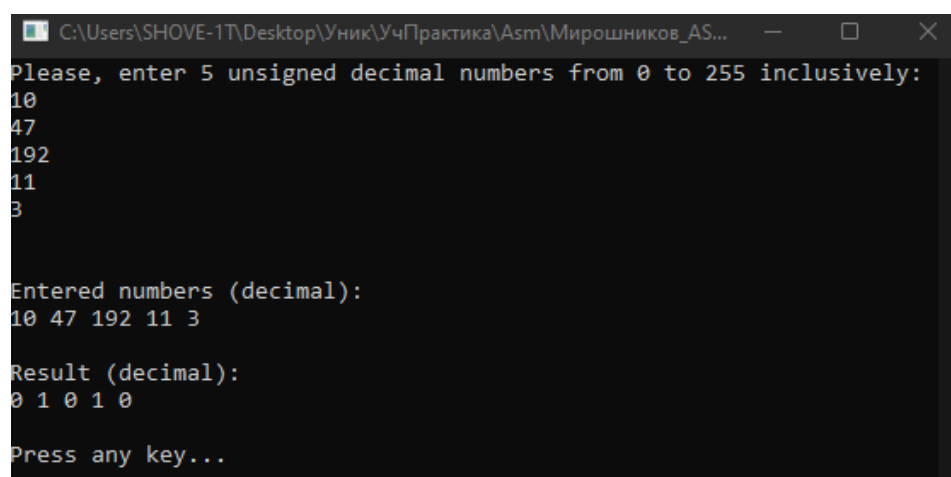
```
C:\Users\SHOVE-1T\Desktop\Уник\УчПрактика\Asm\Мирошников_AS...
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
1
2
3
4
5

Entered numbers (decimal):
1 2 3 4 5

Result (decimal):
0 0 0 0 4

Press any key...
```

Рисунок 15 – Результат теста №1 функционального аналога на ассемблере



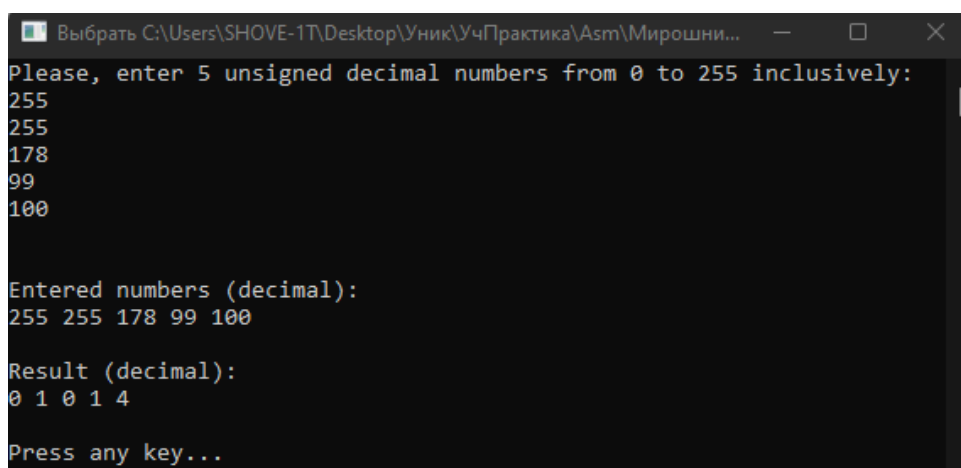
```
C:\Users\SHOVE-1T\Desktop\Уник\УчПрактика\Asm\Мирошников_AS...
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
10
47
192
11
3

Entered numbers (decimal):
10 47 192 11 3

Result (decimal):
0 1 0 1 0

Press any key...
```

Рисунок 16 – Результат теста №2 функционального аналога на ассемблере



```
Выбрать C:\Users\SHOVE-1T\Desktop\Уник\УчПрактика\Asm\Мирошни...
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
255
255
178
99
100

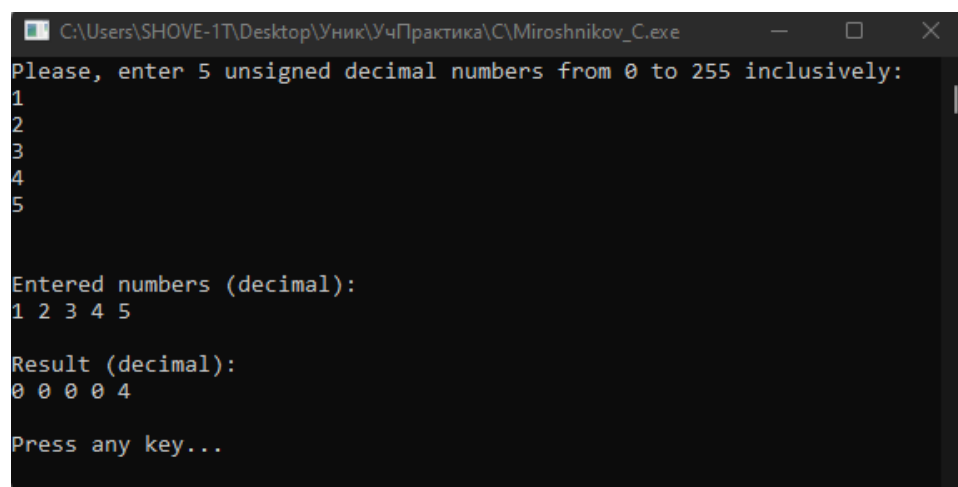
Entered numbers (decimal):
255 255 178 99 100

Result (decimal):
0 1 0 1 4

Press any key...
```

Рисунок 17 – Результат теста №3 функционального аналога на ассемблере

3.3. Тестирование функционального аналога на языке С



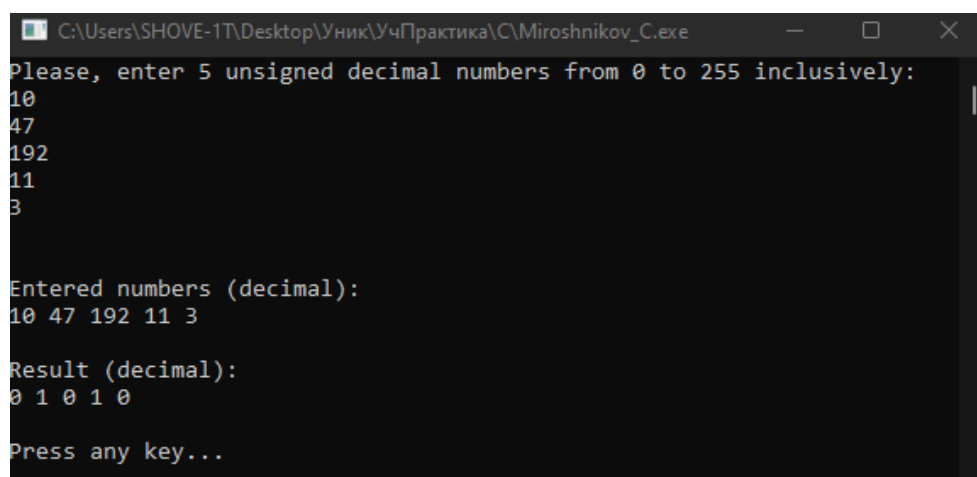
```
C:\Users\SHOVE-1T\Desktop\Уник\УчПрактика\C\Miroshnikov_C.exe
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
1
2
3
4
5

Entered numbers (decimal):
1 2 3 4 5

Result (decimal):
0 0 0 0 4

Press any key...
```

Рисунок 20 – Результат теста №1 функционального аналога на языке С



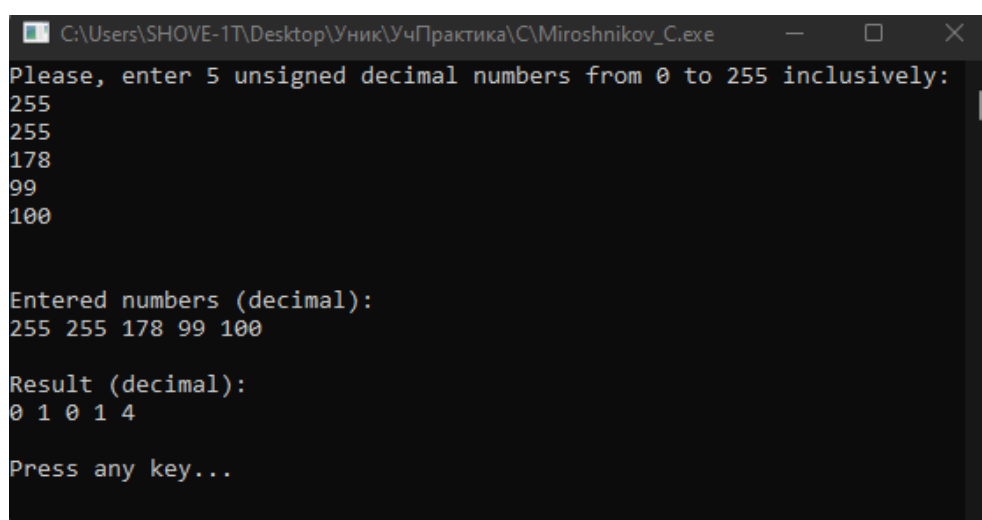
```
C:\Users\SHOVE-1T\Desktop\Уник\УчПрактика\C\Miroshnikov_C.exe
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
10
47
192
11
3

Entered numbers (decimal):
10 47 192 11 3

Result (decimal):
0 1 0 1 0

Press any key...
```

Рисунок 21 – Результат теста №2 функционального аналога на языке С



```
C:\Users\SHOVE-1T\Desktop\Уник\УчПрактика\C\Miroshnikov_C.exe
Please, enter 5 unsigned decimal numbers from 0 to 255 inclusively:
255
255
178
99
100

Entered numbers (decimal):
255 255 178 99 100

Result (decimal):
0 1 0 1 4

Press any key...
```

Рисунок 22 – Результат теста №3 функционального аналога на языке С

ЗАКЛЮЧЕНИЕ

В ходе проделанной работы был изучен исходный исполняемый файл, в котором был найден и проанализирован основной алгоритм преобразования чисел. Были написаны и протестированы функциональные аналоги предложенной программы на языке Ассемблер и языке программирования С. Тестирование показало, что функциональные аналоги соответствуют всем выдвинутым к ним требованиям.

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

```
format PE Console 4.0
entry start

include '%finc%\win32\win32a.inc'
include 'Console.inc'

section '.idata' import data readable writable
    library kernel32, 'KERNEL32.DLL', \msvcrt, 'msvcrt.dll'
    import kernel32, ExitProcess, 'ExitProcess',
GetModuleHandleW, 'GetModuleHandleW'
    import msvcrt, printf, 'printf', scanf, 'scanf', getch,
'_getch'

section '.data' data readable writable
    specHHU_in db '%hhu', 0
    specHHU_out db '%hhu ', 0
    array dd 5 dup(0)
    specS_out db '%s', 0
    entry_line db "Please, enter 5 unsigned decimal numbers
from 0 to 255 inclusively: ", 0x0A, 0          ;0x0A = /n
    entered_numbers_line db "Entered numbers (decimal): ",
0x0A, 0
    result_line db "Result (decimal): ", 0x0A, 0
    press_any_key_line db "Press any key...", 0
    clear_line db " ", 0x0A, 0
    entered_number dd 0
    index dd 0

section '.code' code readable executable

start:
    cinvoke printf, specS_out, entry_line
```

```

xor ebx, ebx
xor edx, edx
for:
    cmp ebx, 0x05
    je after_input
    invoke scanf, specHHU_in, entered_number
    mov edx, [entered_number]
    mov [array + ebx], edx
    inc ebx
    jmp for

after_input:
    cinvoke printf, specS_out, clear_line
    cinvoke printf, specS_out, clear_line
    xor ebx, ebx
    xor edx, edx
    mov esi, array
    cinvoke printf, specS_out, entered_numbers_line
    cld
    print_entered_numbers:
        cmp ebx, 5
        je algorithm
        lodsb
        mov [entered_number], eax
        cinvoke printf, specHHU_out, [entered_number]
        inc ebx
    jmp print_entered_numbers

algorithm:
    cinvoke printf, specS_out, clear_line
    invoke GetModuleHandleW, 0
    cmp eax, 0x00
    je final
    xor edx, edx
    xor ebx, ebx

```

```

    xor ecx, ecx
    mov cl, [eax]
    mov dl, [eax +1]
    xor eax, eax
    mov esi, array
    mov edi, array
    mov [index], 0
    cld
    jmp trans

trans:
    cmp [index], 5
    je final
    xor eax, eax
    lodsb
    xor al, dl
    and al, cl
    and al, byte[index]
    stosb
    inc [index]
    jmp trans

final:
    cinvoke printf, specS_out, clear_line
    xor ebx, ebx
    xor edx, edx
    mov esi, array
    cinvoke printf, specS_out, result_line
    cld
    print_result_numbers:
        cmp ebx, 5
        je exit
        lodsb
        mov [entered_number], eax
        cinvoke printf, specHHU_out, [entered_number]

```

```
        inc ebx
        jmp print_result_numbers

exit:
        cinvoke printf, specS_out, clear_line
        cinvoke printf, specS_out, clear_line
        invoke printf, specS_out, press_any_key_line
        invoke getch
        invoke ExitProcess, 0
```

ПРИЛОЖЕНИЕ 2

ИСХОДНЫЙ КОД ФУНКЦИОНАЛЬНОГО АНАЛОГА НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C

```
#include <stdio.h>
#include <conio.h>
#include <Windows.h>

int main(){
    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("%d ", array[i]);
    HMODULE hModule = GetModuleHandleW(NULL);
    if(hModule != NULL){
        for(int i = 0; i < 5; i++){
            array[i] = (array[i] ^ 0x5A) & 0x4D & (byte)i;
        }
    }

    printf("\n\nResult (decimal):\n");
    for(size_t i = 0; i < 5; i++){
        printf("%d ", array[i]);
    }
    printf("\n\nPress any key...\n");
    getch();
    return 0;
}
```