

**РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ**

**Факультет физико-математических и естественных наук**

**Кафедра прикладной информатики и теории вероятностей**

**ОТЧЕТ**

**ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9**

дисциплина: *Архитектура компьютера*

Студент: Ибрагимов Гаджимурад Шамильевич

Группа: НКАбд-02-25

**МОСКВА**

2025 г.

## 1. СОДЕРЖАНИЕ

Цель работы - - - - -	3
Теоретическое введение - - - - -	4
Выполнение лабораторной работы - - - - -	6
Выполнение самостоятельной работы - - - - -	16
Вывод - - - - -	20
Источники - - - - -	21

# 1. ЦЕЛЬ РАБОТЫ

Приобретение навыков написания программ с использованием подпрограмм.

Знакомство с методами отладки при помощи GDB и его основными возможностями

## 2. ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ

### 2.1. Отладка программ

Отладка — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа:

- обнаружение ошибки;
- поиск её местонахождения;
- определение причины ошибки;
- исправление ошибки.

Можно выделить следующие типы ошибок:

- синтаксические ошибки — обнаруживаются во время трансляции исходного кода и вызваны нарушением ожидаемой формы или структуры языка;
- семантические ошибки — являются логическими и приводят к тому, что программа запускается, отрабатывает, но не даёт желаемого результата;
- ошибки в процессе выполнения — не обнаруживаются при трансляции и вызывают прерывание выполнения программы (например, это ошибки, связанные с переполнением или делением на ноль)

### 2.2. Основные возможности отладчика GDB

GDB (GNU Debugger — отладчик проекта GNU) [1] работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования. GDB предлагает обширные средства для слежения и контроля за выполнением компьютерных программ. Отладчик не содержит собственного графического пользовательского интерфейса и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторонних графических надстроек, а кроме того, некоторые интегрированные среды разработки используют его в качестве базовой подсистемы отладки.

Отладчик GDB (как и любой другой отладчик) позволяет увидеть, что происходит «внутри» программы в момент её выполнения или что делает программа в момент сбоя. GDB может выполнять следующие действия:

- начать выполнение программы, задав всё, что может повлиять на её поведение;
- остановить программу при указанных условиях;
- исследовать, что случилось, когда программа остановилась
- изменить программу так, чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

### 2.3. Понятие подпрограммы

Подпрограмма — это, как правило, функционально законченный участок кода, который можно многократно вызывать из разных мест программы. В отличие от простых переходов из подпрограмм существует возврат на команду, следующую за вызовом. Если в программе встречается одинаковый участок кода, его можно оформить в виде подпрограммы, а во всех нужных местах поставить её вызов. При этом подпрограмма будет содержаться в коде в одном экземпляре, что позволит уменьшить размер кода всей программы

### 3. ВЫПОЛНЕНИЕ ЛАБОРАТОРНОЙ РАБОТЫ

#### 3.1. Реализация подпрограмм в NASM

Создаём каталог для выполнения лабораторной работы № 9, перейдите в него и создайте файл lab09-1.asm:

```
gsibragimov@dk3n55 ~ $ mkdir ~/work/arch-pc/lab09
gsibragimov@dk3n55 ~ $ cd ~/work/arch-pc/lab09
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ touch lab09-1.asm
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ ls
lab09-1.asm
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $
```

В качестве примера рассмотрим программу вычисления арифметического выражения  $f(x) = 2x + 7$  с помощью подпрограммы `_calcul`.

```
Открыть ▾ + lab09-1.asm
~/work/arch-pc/lab09 Сохранить ⌫ - □

1 %include 'in_out.asm'
2 SECTION .data
3 msg: DB 'Введите x: ',0
4 result: DB '2x+7=',0
5 SECTION .bss
6 x: RESB 80
7 res: RESB 80
8 SECTION .text
9 GLOBAL _start
10 _start:
11 ;-----
12 ; Основная программа
13 ;-----
14 mov eax, msg
15 call sprint
16 mov ecx, x
17 mov edx, 80
18 call sread
19 mov eax,x
20 call atoi
21 call _calcul ; Вызов подпрограммы _calcul
22 mov eax,result
23 call sprint
24 mov eax,[res]
25 call iprintLF
26 call quit
27 ;-----
28 ; Подпрограмма вычисления
29 ; выражения "2x+7"
30 _calcul:
31 mov ebx,2
32 mul ebx
33 add eax,7
34 mov [res],eax
35 ret ; выход из подпрограммы
```

Создаем исполняемый файл и запускаем программу:

```
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ nasm -f elf lab09-1.asm
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o main lab09-1.o
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ ./main
Введите x: 1
2x+7=9
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $
```

Функционал `_calcul` напоминает функционал самописной функции в языках более высокого уровня. Инструкция `ret` является последней в подпрограмме и ее исполнение приводит к возвращению в основную программу к инструкции, следующей за инструкцией `call`, которая вызвала данную подпрограмму.

### 3.2. Отладка программ с помощью GDB

Создаём файл `lab09-2.asm` в нашей директории с текстом (Программа печати сообщения Hello world!):

```
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ touch lab09-2.asm
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ ls
in_out.asm  lab09-1.asm  lab09-1.o  lab09-2.asm  main
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $
```

lab09-2.asm

```
1 SECTION .data
2 msg1: db "Hello, ",0x0
3 msg1Len: equ $ - msg1
4 msg2: db "world!",0xa
5 msg2Len: equ $ - msg2
6 SECTION .text
7 global _start
8 _start:
9 mov eax, 4
10 mov ebx, 1
11 mov ecx, msg1
12 mov edx, msg1Len
13 int 0x80
14 mov eax, 4
15 mov ebx, 1
16 mov ecx, msg2
17 mov edx, msg2Len
18 int 0x80
19 mov eax, 1
20 mov ebx, 0
21 int 0x80
```

Получим исполняемый файл. Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию программ необходимо проводить с ключом ‘-g’.



```
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ nasm -f elf -g -l lab09-2.lst lab09-2.asm
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-2 lab09-2.o
```

Загружаем исполняемый файл в отладчик gdb:

```
gsibragimov@dk3n55 ~/work/arch-pc/lab09 $ gdb lab09-2
GNU gdb (Gentoo 15.2 vanilla) 15.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab09-2...
```

Проверяем работу программы, запустив ее в оболочке GDB с помощью команды run

```
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/g/s/gsibragimov/work/arch-pc/lab09/lab09-2
Hello, world!
[Inferior 1 (process 4905) exited normally]
```

Для более подробного анализа программы установим брейкпоинт на метку \_start, с которой начинается выполнение любой ассемблерной программы, и запускаем её

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/g/s/gsibragimov/work/arch-pc/lab09/lab09-2

Breakpoint 1, _start () at lab09-2.asm:9
9      _ mov eax, 4
```

Посмотрим дисассимилированный код программы с помощью команды disassemble начиная с метки \_start

```
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:    mov    $0x4,%eax
    0x08049005 <+5>:    mov    $0x1,%ebx
    0x0804900a <+10>:   mov    $0x804a000,%ecx
    0x0804900f <+15>:   mov    $0x8,%edx
    0x08049014 <+20>:   int    $0x80
    0x08049016 <+22>:   mov    $0x4,%eax
    0x0804901b <+27>:   mov    $0x1,%ebx
    0x08049020 <+32>:   mov    $0x804a008,%ecx
    0x08049025 <+37>:   mov    $0x7,%edx
    0x0804902a <+42>:   int    $0x80
    0x0804902c <+44>:   mov    $0x1,%eax
    0x08049031 <+49>:   mov    $0x0,%ebx
    0x08049036 <+54>:   int    $0x80
End of assembler dump.
```

Переключимся на отображение команд с Intel'овским синтаксисом, введя команду set disassembly-flavor intel

```
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:    mov    eax,0x4
    0x08049005 <+5>:    mov    ebx,0x1
    0x0804900a <+10>:   mov    ecx,0x804a000
    0x0804900f <+15>:   mov    edx,0x8
    0x08049014 <+20>:   int    0x80
    0x08049016 <+22>:   mov    eax,0x4
    0x0804901b <+27>:   mov    ebx,0x1
    0x08049020 <+32>:   mov    ecx,0x804a008
    0x08049025 <+37>:   mov    edx,0x7
    0x0804902a <+42>:   int    0x80
    0x0804902c <+44>:   mov    eax,0x1
    0x08049031 <+49>:   mov    ebx,0x0
    0x08049036 <+54>:   int    0x80
End of assembler dump.
```

Основные различия между синтаксисом AT&T и Intel заключаются в порядке operandов (источник-приемник в AT&T против приемник-источник в Intel), использовании префикса % для имен регистров в AT&T, использовании \$ для непосредственных значений в AT&T, а также в разном обозначении непосредственных значений и смещений в режиме Intel.

Включим режим псевдографики для более удобного анализа программы

layout asm

```
B+>0x8049000 <_start>    mov    eax, 0x4
 0x8049005 <_start+5>    mov    ebx, 0x1
 0x804900a <_start+10>   mov    ecx, 0x804a000
 0x804900f <_start+15>   mov    edx, 0x8
 0x8049014 <_start+20>   int    0x80
 0x8049016 <_start+22>   mov    eax, 0x4
 0x804901b <_start+27>   mov    ebx, 0x1
 0x8049020 <_start+32>   mov    ecx, 0x804a008
 0x8049025 <_start+37>   mov    edx, 0x7
 0x804902a <_start+42>   int    0x80
 0x804902c <_start+44>   mov    eax, 0x1
 0x8049031 <_start+49>   mov    ebx, 0x0
 0x8049036 <_start+54>   int    0x80
 0x8049038           add    BYTE PTR [eax],al
 0x804903a           add    BYTE PTR [eax],al
```

после ввода layout regs

```
Register group: general
eax          0x0          0
ecx          0x0          0
edx          0x0          0
ebx          0x0          0
esp 0xfffffc3b0 0xfffffc3b0
ebp          0x0          0x0
esi          0x0          0
edi          0x0          0
eip 0x8049000 0x8049000 <_start>
eflags        0x202      [ IF ]
cs            0x23        35
ss            0x2b        43
ds            0x2b        43

0x8049130    add  BYTE PTR [eax],al
0x8049132    add  BYTE PTR [eax],al
0x8049134    add  BYTE PTR [eax],al
0x8049136    add  BYTE PTR [eax],al
0x8049138    add  BYTE PTR [eax],al
0x804913a    add  BYTE PTR [eax],al
0x804913c    add  BYTE PTR [eax],al
0x804913e    add  BYTE PTR [eax],al
0x8049140    add  BYTE PTR [eax],al
0x8049142    add  BYTE PTR [eax],al
0x8049144    add  BYTE PTR [eax],al
0x8049146    add  BYTE PTR [eax],al

native process 4971 (asm) In: _start                                L9      PC: 0x8049000
(gdb) layout regs
Undefined command: "layout". Try "help".
(gdb) layout regs
(gdb) layout regs
(gdb) 
```

В этом режиме есть три окна:

- В верхней части видны названия регистров и их текущие значения;
- В средней части виден результат дисассимилирования программы;
- Нижняя часть доступна для ввода команд.

### 3.3. Добавление точек останова

На предыдущих шагах была установлена точка останова по имени метки (\_start).

Проверим это с помощью команды info breakpoints (кратко i b):

```

gsibragimov@dk1n22 - lab09
Register group: general
eax          0x0          0
ecx          0x0          0
edx          0x0          0
ebx          0x0          0
esp 0xfffffc380 0xfffffc380
ebp          0x0          0x0

B+>0x8049000 <_start>    mov    eax,0x4
 0x8049005 <_start+5>    mov    ebx,0x1
 0x804900a <_start+10>   mov    ecx,0x804a000
 0x804900f <_start+15>   mov    edx,0x8
 0x8049014 <_start+20>   int    0x80
 0x8049016 <_start+22>   mov    eax,0x4

native process 3588 (asm) In: _start
L9      PC: 0x8049000

Breakpoint 1, _start () at lab09-2.asm:9
(gdb) layout regs
(gdb) i b
Num      Type            Disp Enb Address      What
1        breakpoint      keep y  0x08049000  lab09-2.asm:9
                                breakpoint already hit 1 time

```

Добавим еще одну точку остановы. Сделать это с помощью команды:

`break *<адрес>`

```

gsibragimov@dk1n22 - lab09
b+ 0x8049000 <_start>    mov    eax,0x4
 0x8049005 <_start+5>    mov    ebx,0x1
 0x804900a <_start+10>   mov    ecx,0x804a000
 0x804900f <_start+15>   mov    edx,0x8
 0x8049014 <_start+20>   int    0x80
 0x8049016 <_start+22>   mov    eax,0x4
 0x804901b <_start+27>   mov    ebx,0x1
 0x8049020 <_start+32>   mov    ecx,0x804a008
 0x8049025 <_start+37>   mov    edx,0x7
 0x804902a <_start+42>   int    0x80
 0x804902c <_start+44>   mov    eax,0x1
b+ 0x8049031 <_start+49>   mov    ebx,0x0
 0x8049036 <_start+54>   int    0x80

exec No process (asm) In:
L??      PC: ??

Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) b *0x8049031
Breakpoint 2 at 0x8049031: file lab09-2.asm, line 20.
(gdb) i b
Num      Type            Disp Enb Address      What
1        breakpoint      keep y  0x08049000  lab09-2.asm:9
2        breakpoint      keep y  0x08049031  lab09-2.asm:20

```

### 3.4. Работа с данными программы в GDB

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных.

Посмотрим значения msg1:

```
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb) █
```

Посмотрите значение переменной msg2 по адресу. Адрес переменной можно определить по дизассемблированной инструкции. Посмотрим инструкцию mov ecx,msg2 которая записывает в регистр ecx адрес переменной msg2

```
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n"
(gdb) █
```

Изменить значение для регистра или ячейки памяти можно с помощью команды set, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс \$, а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си).

Изменим первый символ переменной msg1

Для этого пропишем команду: `(gdb) set {char}msg1 = 'h'`

Далее проверим значение:

```
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hello, "
(gdb) █
```

Заменим любой символ во второй переменной msg2.

Прописываем команду:

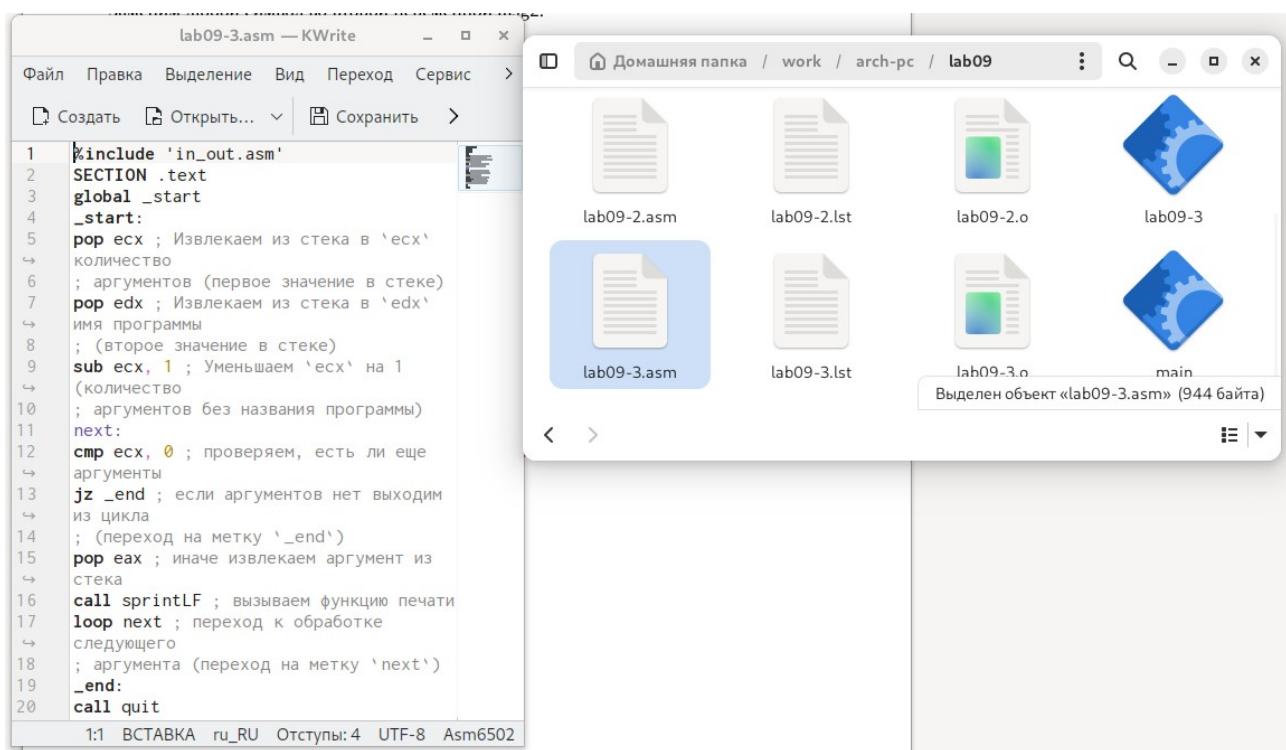
```
(gdb) set {char}msg2 = 'h'
```

Результат:

```
(gdb) x/1sb &msg2
0x804a008 <msg2>:      "horld!\n"
(gdb) █
```

### 3.5. Обработка аргументов командной строки в GDB

Скопируем файл lab8-2.asm, созданный при выполнении лабораторной работы №8, с программой выводящей на экран аргументы командной строки в файл с именем lab09-3.asm:



Для загрузки в gdb программы с аргументами необходимо использовать ключ --args.

Загрузим исполняемый файл в отладчик, указав аргументы

```
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ gdb --args lab09-3 аргумент1 аргумент 2 'аргумент 3'
```

Для начала установим точку останова перед первой инструкцией в программе и запустим ее:

```
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 5.
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/g/s/gsibragimov/work/arch-pc/lab09/lab09-3 аргумент1 аргумент 2 аргумент\ 3
Breakpoint 1, _start () at lab09-3.asm:5
5      pop ecx ; Извлекаем из стека в `есх` количество
```

Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы):

```
(gdb) x/x $esp
0xfffffc340:    0x00000005
```

Посмотрим остальные позиции стека – по адресу [esp+4] с шагом в 4, так как число аргументов является 4.

```
(gdb) x/s *(void**)( $esp + 4)
0xfffffc5ab:    "/afs/.dk.sci.pfu.edu.ru/home/g/s/gsibragimov/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)( $esp + 12)
0xfffffc605:    "аргумент"
(gdb) x/s *(void**)( $esp + 16)
0xfffffc616:    "2"
(gdb) x/s *(void**)( $esp + 20)
0xfffffc618:    "аргумент 3"
(gdb) x/s *(void**)( $esp + 8)
0xfffffc5f3:    "аргумент1"
```

## 4. ВЫПОЛНЕНИЕ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

4.1. Преобразуйте программу из лабораторной работы №8 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции  $f(x)$  как подпрограмму.

Изменим текст программы , выделив под вычисление в цикле отдельный программный модуль \_calcul :

```
1  %include 'in_out.asm'
2  SECTION .data
3  msg db "Результат: ",0h
4  section .bss
5  x resb 10
6  SECTION .text
7  global _start
8  _start:
9  pop ecx ; Извлекаем из стека в 'ecx' количество
10 ; аргументов (первое значение в стеке)
11 pop edx ; Извлекаем из стека в 'edx' имя программы
12 ; (второе значение в стеке)
13 sub ecx,1 ; Уменьшаем 'ecx' на 1 (количество
14 ; аргументов без названия программы)
15 mov esi,0 ; Используем 'esi' для хранения
16 ; промежуточных значений
17
18 next:
19 cmp ecx, 0h ; Если аргументов нет, то переход к завершению программы
20 jz _end
21
22 pop eax ; извлекаем аргумент из стека
23 call atoi ; преобразуем в число
24 call _calcul
25 loop next
26
27 _end:|
28 mov eax, msg ; вывод сообщения "Результат: "
29 call sprint
30 mov eax, esi ; записываем сумму в регистр 'eax'
31 call iprintLF ; печать результата
32 call quit ; завершение программы
33
34 _calcul:
35 mov [x], eax
36 mov ebx, 12
37 imul ebx
38 sub eax, 7
39 add esi, eax
40 ret
```

Создадим исполняемый файл и запустим программу, указав произвольные аргументы:

```
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ nasm -f elf -g -l 1.lst 1.asm
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o mn 1.o
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ ./mn 1 2 3 4 5
Результат: 145
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ ./mn 1 2 3
Результат: 51
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ ./mn 1
Результат: 5
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ ./mn 2 2
Результат: 34
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $
```

4.2. В листинге приведена программа вычисления выражения  $(3 + 2) * 4 + 5$ . При запуске данная программа дает неверный результат. Проверьте это. С помощью отладчика GDB, анализируя изменения значений регистров, определите ошибку и исправьте ее.

В ходе разбора текста программы мною было замечено

eax	group: general	2
eax	0x8	8
ecx	0x4	4
edx	0x0	0
ebx	0x5ffffc390	5fffffc390
esp	0xfffffc390	0xfffffc390
ebp	0x0	0x0
esi	0x0	0
edi	0x00490f9	0x80490f9 <_start+17>
eipags	0x80490fb	0x80490fb <_start+19>
eflags	0x202	mov e[ IF ]
cs	0x23	35

```
0x80490f4 <_start+12>    mov    ecx,0x4
0x80490f9 <_start+17>    mul    ecx
B+>0x80490fb <_start+19> add    ebx,0x5
    0x804910e <_start+22>    mov    edi,ebx
    0x8049100 <_start+24>    mov    eax,0x804a000rint>
    0x8049105 <_start+29>    call   0x804900f <sprint>
    0x804910a <_start+34>    mov    eax,edi
    0x804910c <_start+36>    call   0x8049086 <iprintLF>
    0x8049111 <_start+41>    call   0x80490db <quit>
    0x8049116                 add    BYTE PTR [eax],al
    0x8049118                 add    BYTE PTR [eax],al
```

Незапланированное целями задачи заполнение регистров

```
(gdb) p/s $ebx
$7 = 5
(gdb) p/s $ebx
$8 = 5
(gdb) p/s$ eax
A syntax error in expression, near 'eax'.
(gdb) p/s $eax
$9 = 8
(gdb) p/s $ecx
$10 = 4
```

Происходит перемножение регистров ecx и eax; eax = eax\*ecx = 4 \* 2 = 8

Изменим текст программы:

```
1 %include 'in_out.asm'
2 SECTION .data
3 div: DB 'Результат: ',0
4 SECTION .text
5 GLOBAL _start
6 _start:
7 ; ---- Вычисление выражения (3+2)*4+5
8 mov eax,3
9 mov ebx,2
10 add eax, ebx
11 mov ecx,4
12 mul ecx
13 add eax,5
14 mov edi,eax
15 ; ---- Вывод результата на экран
16 mov eax,div
17 call sprint
18 mov eax,edi
19 call iprintLF
20 call quit
```

Создаем исполняемый файл и запускаем

```
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ nasm -f elf -g -l 2.lst 2.asm
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o m 2.o
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ ./m
Результат: 25
gsibragimov@dk1n22 ~/work/arch-pc/lab09 $ █
```

Новый алгоритм:

eax = 3

ebx = 2

eax = 3 + 2

ecx = 4

eax = 5 \* 4

eax = 20 + 5

edi = eax (edi = 25)

## 5. ВЫВОД

В ходе выполнения данной лабораторной работы я познакомился с отладчиком GDB и также научился реализовывать подпрограммы в NASM

## 6. ИСТОЧНИКИ

- 1Столяров А. Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011. — URL: [http://www.stolyarov.info/books/asm\\_unix](http://www.stolyarov.info/books/asm_unix).
2. Таненбаум Э. Архитектура компьютера. — 6-е изд. — СПб. : Питер, 2013. — 874 с. — (Классика Computer Science).
3. Расширенный ассемблер: NASM. — 2021. — URL: <https://www.opennet.ru/docs/RUS/nasm/>