

Отчёт по лабораторной работе №9

Понятие подпрограммы. Отладчик GDB.

Карпухин Клим

Содержание

1 Цель работы	6
2 Выполнение лабораторной работы	7
2.1 Реализация подпрограмм в NASM	7
2.2 Отладка программ с помощью GDB	11
2.3 Обработка аргументов командной строки в GDB	20
3 Задание для самостоятельной работы	23
3.1 Задание 1. Преобразование программы ЛР8 с использованием подпрограммы	23
3.2 Задание 2. Поиск и исправление ошибки в программе вычисления выражения	25
4 Выводы	30

Список иллюстраций

2.1	Создание каталога для программ лабораторной работы №9, переход в него и создание файла lab09-1.asm	7
2.2	Текст программы lab09-1.asm из листинга 9.1	8
2.3	Трансляция, компоновка и запуск программы lab09-1.asm	8
2.4	Изменённый текст программы lab09-1.asm с подпрограммами _calcul и _subcalcul	10
2.5	Запуск модифицированной программы и проверка корректности вычисления $f(g(x))$	10
2.6	Текст программы lab09-2.asm – вывод сообщения Hello, world!	11
2.7	Трансляция и компоновка программы lab09-2 с ключом -g	11
2.8	Запуск программы lab09-2 внутри GDB	12
2.9	Установка точки останова на метку _start и запуск программы до точки останова	12
2.10	Дизассемблирование программы lab09-2 в режимах ATT и Intel	13
2.11	Режим псевдографики GDB (layout asm и layout regs)	14
2.12	Список точек останова и установка брейкпоинта по адресу	15
2.13	Пошаговое выполнение программы и анализ изменений регистров	16
2.14	Использование команды x для просмотра содержимого переменных msg1 и msg2	17
2.15	Изменение содержимого строковых переменных с помощью команды set	18
2.16	Примеры использования команды print и изменения регистра ebx	19
2.17	Завершение выполнения программы и выход из GDB	20
2.18	Копирование файла программы вывода аргументов командной строки из лабораторной работы №8, создание объектного и исполняемого файлов	21
2.19	Загрузка программы lab09-3 в GDB с аргументами командной строки	21
2.20	Установка точки останова и запуск программы	21
2.21	Просмотр расположения аргументов командной строки в стеке	22
3.1	Текст программы task1.asm для вычисления суммы значений функции $f(x)$ как подпрограммы	24
3.2	Запуск программы task1 с набором аргументов 1 2 3 4	25
3.3	Текст программы из листинга 9.3	26
3.4	Запуск программы из листинга 9.3	26
3.5	Загрузка программы в GDB и установка точки останова	27

3.6	Пошаговое выполнение программы	28
3.7	Исправленный текст программы вычисления выражения $(3+2)^*4+5$.	29
3.8	Запуск исправленной программы и вывод корректного результата .	29

Список таблиц

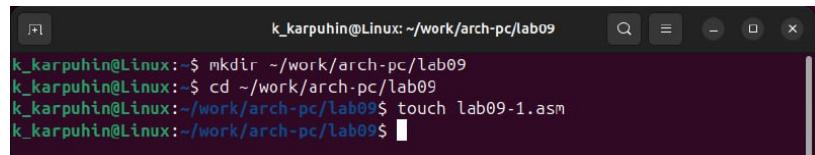
1 Цель работы

Цель данной лабораторной работы – приобрести навыки написания программ с использованием подпрограмм на языке ассемблера NASM, а также познакомиться с методами отладки программ с помощью отладчика GDB и его основными возможностями.

2 Выполнение лабораторной работы

2.1 Реализация подпрограмм в NASM

Создал каталог для выполнения лабораторной работы №9, перешёл в него и создал файл `lab09-1.asm` (рис. 2.1):



```
k_karpuhin@Linux:~/work/arch-pc/lab09
k_karpuhin@Linux:~/work/arch-pc/lab09$ mkdir ~/work/arch-pc/lab09
k_karpuhin@Linux:~/work/arch-pc/lab09$ cd ~/work/arch-pc/lab09
k_karpuhin@Linux:~/work/arch-pc/lab09$ touch lab09-1.asm
k_karpuhin@Linux:~/work/arch-pc/lab09$
```

Рисунок 2.1: Создание каталога для программ лабораторной работы №9, переход в него и создание файла `lab09-1.asm`

2.1.1 Программа вычисления выражения $f(x) = 2x + 7$ с использованием подпрограммы

Ввёл в файл `lab09-1.asm` текст программы из листинга 9.1, реализующей вычисление выражения $f(x) = 2x + 7$ с помощью подпрограммы `_calcul` (рис. 2.2).

```
k_karpuhin@Linux:~/work/arch-pc/lab09
/home/k_~09-1.asm [---] 11 L:[ 1+15 16/ 36] *(312 / 708b) 0010 0x00A [*][X]
%include 'in_out.asm'

SECTION .data
msg: DB 'Введите x: ',0
result: DB '2x+7=',0
SECTION .bss
x: RESB 80
res: RESB 80
SECTION .text
GLOBAL _start
_start:
;-----
; Основная программа
;-----
mov eax, msg
call sprint
mov ecx, x
mov edx, 80
call sread
mov eax,x
call atoi
call _calcul ; Вызов подпрограммы _calcul
mov eax,result
call sprint
mov eax,[res]
call iprintLF
call quit
;-----
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
mov ebx,2
mul ebx
add eax,7
mov [res],eax
ret ; выход из подпрограммы
1Help 2Save 3Mark 4Replace 5Copy 6Move 7Search 8Delete 9PullDown 10Quit
```

Рисунок 2.2: Текст программы lab09-1.asm из листинга 9.1

Создал объектный и исполняемый файлы и проверил работу программы (рис. 2.3).

```
k_karpuhin@Linux:~/work/arch-pc/lab09$ nasm -f elf lab09-1.asm -o lab09-1.o
k_karpuhin@Linux:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
k_karpuhin@Linux:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 2
2x+7=11
k_karpuhin@Linux:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 6
2x+7=19
k_karpuhin@Linux:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 21
2x+7=49
k_karpuhin@Linux:~/work/arch-pc/lab09$ ./lab09-1
Введите x: -67
2x+7=-125
k_karpuhin@Linux:~/work/arch-pc/lab09$
```

Рисунок 2.3: Трансляция, компоновка и запуск программы lab09-1.asm

При вводе различных **неотрицательных** значений x (например, x = 2, x =

6 и $x = 21$) программа корректно вычисляла значение выражения $2x + 7$ и выводила результат на экран. При вводе отрицательных значений (например, $x = -67$) результат некорректен из-за того, что стандартная процедура `atoi` из файла `in_out.asm` не обрабатывает знак минус и предназначена только для преобразования неотрицательных целых чисел.

2.1.2 Модификация программы: вычисление $f(g(x))$

По заданию изменил текст программы, добавив подпрограмму `_subcalcul` в подпрограмму `_calcul` для вычисления выражения

$$f(g(x)), \quad f(x) = 2x + 7, \quad g(x) = 3x - 1.$$

Логика работы: 1. Пользователь вводит x 2. x передаётся в `_calcul` 3. `_calcul` вызывает `_subcalcul` с x 4. `_subcalcul` вычисляет $3x - 1$ и возвращает результат 5. `_calcul` вычисляет $2 \cdot (3x - 1) + 7$ 6. Результат выводится на экран

Модифицированный фрагмент программы имеет вид (рис. 2.4):

```
k_karpuhin@Linux: ~/work/arch-pc/lab09
/home/k_karpuhin@Linux:~/work/arch-pc/lab09 [ -M-- ] 7 L:[ 1+39 40/ 40 ] *(550 / 550b) <EOF> [*][X]
%include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
result: DB 'f(g(x)) = ',0
SECTION .bss
x: RESB 80
res: RESB 80
SECTION .text
GLOBAL _start

_start:
    mov eax, msg
    call sprint
    mov ecx, x
    mov edx, 80
    call sread
    mov eax, x
    call atoi
    ...
    call _calcul
    ...
    mov eax, result
    call sprint
    mov eax, [res]
    call iprintLF
    call quit

_calcul:
    call _subcalcul
    mov ebx, 2
    mul ebx
    add eax, 7
    mov [res], eax
    ret

_subcalcul:
    mov ebx, 3
    mul ebx
    sub eax, 1
    ret
1Help 2Save 3Mark 4Replace 5Copy 6Move 7Search 8Delete 9PullDown 10Quit
```

Рисунок 2.4: Изменённый текст программы lab09-1.asm с подпрограммами `_calcul` и `_subcalcul`

После пересборки программы и запуска с несколькими тестовыми значениями x убедился, что программа корректно вычисляет $f(g(x))$ (рис. 2.5).

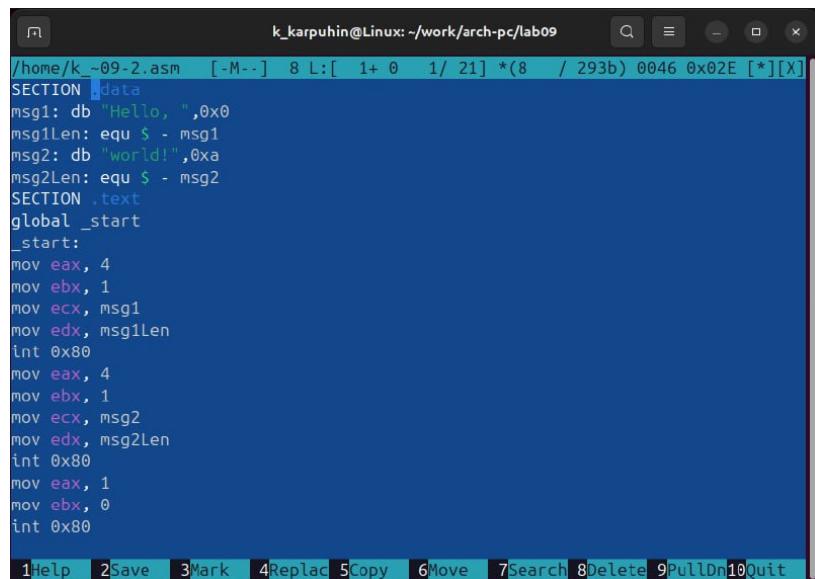
```
k_karpuhin@Linux:~/work/arch-pc/lab09$ nasm -f elf lab09-1.asm -o lab09-1.o
k_karpuhin@Linux:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
k_karpuhin@Linux:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 6
f(g(x)) = 41
k_karpuhin@Linux:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 7
f(g(x)) = 47
k_karpuhin@Linux:~/work/arch-pc/lab09$
```

Рисунок 2.5: Запуск модифицированной программы и проверка корректности вычисления $f(g(x))$

2.2 Отладка программ с помощью GDB

2.2.1 Программа вывода сообщения «Hello, world!»

Создал файл lab09-2.asm с текстом программы из листинга 9.2, выводящей сообщение «Hello, world!» (рис. 2.6).

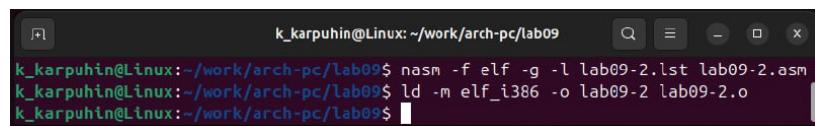


```
k_karpuhin@Linux: ~/work/arch-pc/lab09
/home/k_-09-2.asm [-M--] 8 L:[ 1+ 0 1/ 21 ] *(8 / 293b) 0046 0x02E [*][X]
SECTION .data
msg1: db "Hello, ",0x0
msg1Len: equ $ - msg1
msg2: db "world!",0xa
msg2Len: equ $ - msg2
SECTION .text
global _start
_start:
    mov eax, 4
    mov ebx, 1
    mov ecx, msg1
    mov edx, msg1Len
    int 0x80
    mov eax, 4
    mov ebx, 1
    mov ecx, msg2
    mov edx, msg2Len
    int 0x80
    mov eax, 1
    mov ebx, 0
    int 0x80
1Help 2Save 3Mark 4Replace 5Copy 6Move 7Search 8Delete 9PullDown 10Quit
```

Рисунок 2.6: Текст программы lab09-2.asm — вывод сообщения Hello, world!

Собрал программу с добавлением отладочной информации (-g) и получил исполняемый файл (рис. 2.7).

```
nasmd -f elf -g -l lab09-2.lst lab09-2.asm
ld -m elf_i386 -o lab09-2 lab09-2.o
```

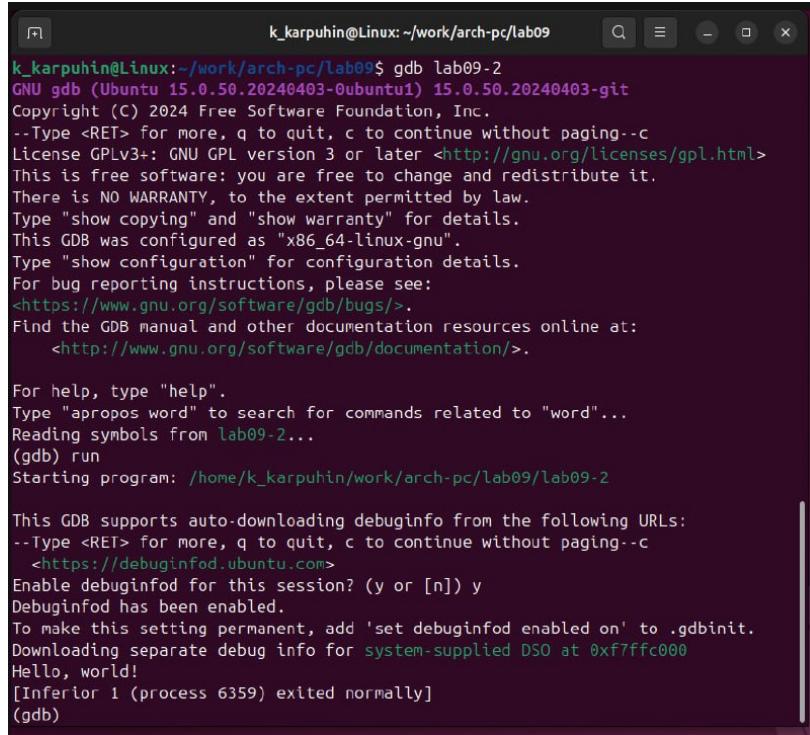


```
k_karpuhin@Linux:~/work/arch-pc/lab09$ nasmd -f elf -g -l lab09-2.lst lab09-2.asm
k_karpuhin@Linux:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-2 lab09-2.o
k_karpuhin@Linux:~/work/arch-pc/lab09$
```

Рисунок 2.7: Трансляция и компоновка программы lab09-2 с ключом -g

2.2.2 Запуск программы в GDB и установка точки останова

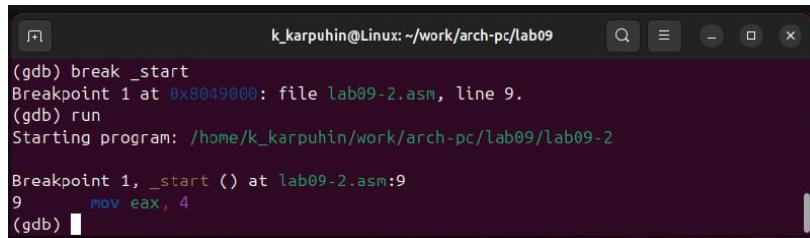
Загрузил исполняемый файл в отладчик GDB. Внутри GDB запустил программу командой `run` и убедился, что она корректно выводит строку (рис. 2.8).



A screenshot of a terminal window titled "k_karpuhin@Linux:~/work/arch-pc/lab09". The window shows the output of the command "gdb lab09-2". The terminal displays the GDB startup message, which includes copyright information, license details, and help instructions. It then shows the command "(gdb) run" being entered, followed by "Starting program: /home/k_karpuhin/work/arch-pc/lab09/lab09-2". The program outputs "Hello, world!" and exits. The terminal ends with "[Inferior 1 (process 6359) exited normally]".

Рисунок 2.8: Запуск программы lab09-2 внутри GDB

Далее установил точку останова на метку `_start` и снова запустил программу (рис. 2.9).



A screenshot of a terminal window titled "k_karpuhin@Linux:~/work/arch-pc/lab09". The window shows the command "(gdb) break _start" being entered, followed by "Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.". Then, the command "(gdb) run" is entered, followed by "Starting program: /home/k_karpuhin/work/arch-pc/lab09/lab09-2". The terminal then shows the assembly code at address 9: "9 mov eax, 4". The prompt "(gdb)" is visible at the bottom.

Рисунок 2.9: Установка точки останова на метку `_start` и запуск программы до точки останова

2.2.3 Дизассемблирование и режим Intel-синтаксиса

Посмотрел дизассемблированный код программы, начиная с метки `_start`, затем переключил отображение команд на Intel-синтаксис.

В режиме ATT и Intel различается порядок operandов и общее оформление инструкций. В режиме Intel синтаксис совпадает с используемым в NASM, что упрощает сопоставление дизассемблированного кода с исходником (рис. 2.10).

The screenshot shows a terminal window titled "k_karpuhin@Linux: ~/work/arch-pc/lab09". The terminal displays assembly code for the function `_start`. It starts with the command `(gdb) disassemble _start`. The first section shows the code in ATT syntax, with instructions like `mov $0x4,%eax`. The second section, after a `set disassembly-flavor intel` command, shows the same code in Intel syntax, with instructions like `mov eax,0x4`. The terminal also includes prompts for continuation (`c`), quitting (`q`), and more output (`<RET>`).

Рисунок 2.10: Дизассемблирование программы lab09-2 в режимах ATT и Intel

2.2.4 Режим псевдографики GDB

Для более удобного анализа программы включил псевдографический режим (рис. 2.11).

The screenshot shows a terminal window titled "k_karpuhin@Linux: ~/work/arch-pc/lab09". The window has two main sections: "Registers" at the top and "Assembly" at the bottom. The Registers section displays the message "[Register Values Unavailable]". The Assembly section contains the following assembly code:

```
0x0049000 <_start>    mov    $0x4,%eax
0x0049005 <_start+5>   mov    $0x1,%ebx
0x004900a <_start+10>  mov    $0x804a000,%ecx
0x004900f <_start+15>  mov    $0x8,%edx
0x0049014 <_start+20>  int    $0x80
0x0049016 <_start+22>  mov    $0x4,%eax
0x004901b <_start+27>  mov    $0x1,%ebx
0x0049020 <_start+32>  mov    $0x804a008,%ecx
0x0049025 <_start+37>  mov    $0x7,%edx
0x004902a <_start+42>  int    $0x80
0x004902c <_start+44>  mov    $0x1,%eax
0x0049031 <_start+49>  mov    $0x0,%ebx
0x0049036 <_start+54>  int    $0x80
```

At the bottom of the terminal, the command history shows:

```
exec No process (asm) In:
(gdb) layout regs
(gdb)
```

Рисунок 2.11: Режим псевдографики GDB (layout asm и layout regs)

2.2.5 Работа с точками останова

Проверил список установленных точек останова, затем по заданию установил ещё одну точку останова по адресу предпоследней инструкции. После этого ещё раз вывел информацию о всех точках останова (рис. 2.12).

The screenshot shows the GDB debugger interface. At the top, there's a terminal window with the command:

```
k_karpuhin@Linux: ~/work/arch-pc/lab09
```

Below it is a register dump window titled "Register group: general". The registers show initial values (eax=0x0, ebx=0x0, etc.) and some modified values (esp=0xfffffcfc0, eip=0x8049000). The stack pointer (esp) has been explicitly set to 0xfffffcfc0.

The main part of the interface is a disassembly window. It displays assembly code starting at address 0x8049000, which is the entry point (_start). The assembly code consists of several mov and int instructions. A specific instruction at address 0x8049031 is highlighted with a blue border, indicating it is the current instruction being stepped or examined.

At the bottom, there's a terminal window showing GDB commands:

```
native process 6954 (asm) In: _start
(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
(gdb) break *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
breakpoint already hit 1 time
2 breakpoint keep y 0x08049031 lab09-2.asm:20
(gdb)
```

Рисунок 2.12: Список точек останова и установка брейкпоинта по адресу

2.2.6 Пошаговое выполнение и анализ регистров

С помощью команды `stepi` последовательно выполнил первые пять инструкций функции `_start` (`mov eax, 4, mov ebx, 1, mov ecx, msg1, mov edx, msg1Len, int 0x80`) и по окну регистров (`layout regs`) проследил, что изменяются значения регистров `eax`, `ebx`, `ecx`, `edx`, а также счётчика команд

eip.

The screenshot shows a terminal window with the following content:

```
k_karpuhin@Linux: ~/work/arch-pc/lab09
Register group: general
eax      0x8          8
ecx      0x804a000    134520832
edx      0x8          8
ebx      0x1          1
esp      0xfffffcfc0  0xfffffcfc0
ebp      0x0          0x0
esi      0x0          0
edi      0x0          0
eip      0x8049016    0x8049016 <_start+22>
eflags   0x202       [ IF ]
cs       0x23        35
ss       0xb          43
ds       0xb          43
es       0xb          43
fs       0x0          0
gs       0x0          0

B+ 0x8049000 <_start>    mov    $0x4,%eax
0x8049005 <_start+5>    mov    $0x1,%ebx
0x804900a <_start+10>   mov    $0x804a000,%ecx
0x804900f <_start+15>   mov    $0x8,%edx
0x8049014 <_start+20>   int    $0x80
>0x8049016 <_start+22>  mov    $0x4,%eax
0x804901b <_start+27>  mov    $0x1,%ebx
0x8049020 <_start+32>  mov    $0x804a008,%ecx
0x8049025 <_start+37>  mov    $0x7,%edx
0x804902a <_start+42>  int    $0x80
0x804902c <_start+44>  mov    $0x1,%eax
b+ 0x8049031 <_start+49> mov    $0x0,%ebx
0x8049036 <_start+54>  int    $0x80
0x804903b add    %al,(%eax)
0x804903a add    %al,(%eax)
0x804903c add    %al,(%eax)
0x804903e add    %al,(%eax)
0x8049040 add    %al,(%eax)

native process 6954 (asm) In: _start
(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
(gdb) break *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
      breakpoint already hit 1 time
2    breakpoint keep y  0x08049031 lab09-2.asm:20
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) ■
```

Рисунок 2.13: Пошаговое выполнение программы и анализ изменений регистров

2.2.7 Работа с данными программы в GDB

Для просмотра содержимого памяти использовал команду `x` в различных форматах. По имени переменной `msg1` просмотрел её содержимое. В ответ отладчик показал строку "Hello, ".

Аналогично, по адресу, записанному в регистре `ecx` после инструкции `mov ecx, msg2`, просмотрел содержимое переменной `msg2` (рис. 2.14).

The screenshot shows the GDB debugger interface. The assembly code area displays the following assembly instructions:

```
B+ 0x0049000 <_start>    mov    $0x4,%eax
0x0049005 <_start+5>    mov    $0x1,%ebx
0x004900a <_start+10>   mov    $0x804a000,%ecx
0x004900f <_start+15>   mov    $0x8,%edx
0x0049014 <_start+20>   int    $0x80
>0x0049016 <_start+22>   mov    $0x4,%eax
0x004901b <_start+27>   mov    $0x1,%ebx
0x0049020 <_start+32>   mov    $0x804a008,%ecx
0x0049025 <_start+37>   mov    $0x7,%edx
0x004902a <_start+42>   int    $0x80
0x004902c <_start+44>   mov    $0x1,%eax
b+ 0x0049031 <_start+49>   mov    $0x0,%ebx
0x0049036 <_start+54>   int    $0x80
0x0049038 add    %al,(%eax)
0x004903a add    %al,(%eax)
0x004903c add    %al,(%eax)
0x004903e add    %al,(%eax)
0x0049040 add    %al,(%eax)
```

The command line shows the assembly code and the current state of the registers and memory:

```
native process 6954 (asm) In: _start
Num  Type      Disp Enb Address  What
1    breakpoint  keep y  0x0049000 lab09-2.asm:9
breakpoint already hit 1 time
(gdb) break *0x0049031
Breakpoint 2 at 0x0049031: file lab09-2.asm, line 20.
(gdb) i b
Num  Type      Disp Enb Address  What
1    breakpoint  keep y  0x0049000 lab09-2.asm:9
breakpoint already hit 1 time
2    breakpoint  keep y  0x0049031 lab09-2.asm:20
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb) x/1sb &msg2
0x804a008 <msg2>:      "world!\n\034"
(gdb) █
```

Рисунок 2.14: Использование команды `x` для просмотра содержимого переменных `msg1` и `msg2`

Изменил первый символ переменной `msg1` с помощью команды `set`.

Аналогично изменил первый символов в переменной `msg2` (рис. 2.15).

```
>0x8049016 <_start+22>    mov    $0x4,%eax  
0x804901b <_start+27>    mov    $0x1,%ebx  
0x8049020 <_start+32>    mov    $0x804a008,%ecx  
0x8049025 <_start+37>    mov    $0x7,%edx  
0x804902a <_start+42>    int    $0x80  
0x804902c <_start+44>    mov    $0x1,%eax  
b+ 0x8049031 <_start+49>    mov    $0x0,%ebx  
0x8049036 <_start+54>    int    $0x80  
0x804903b add    %al,(%eax)  
0x804903a add    %al,(%eax)  
0x804903c add    %al,(%eax)  
0x804903e add    %al,(%eax)  
0x8049040 add    %al,(%eax)  
0x8049042 add    %al,(%eax)  
0x8049044 add    %al,(%eax)  
0x8049046 add    %al,(%eax)  
  
native process 6954 (status) In: _start L14 PC: 0x8049016  
(gdb) set {char} &msg1 = 'h'  
(gdb) x/1sb &msg1  
0x804a008 <msg1>:      "hello, "  
(gdb) set {char} &msg2 = 'W'  
(gdb) x/1sb &msg2  
0x804a008 <msg2>:      "World!\n\034"  
(gdb) █
```

Рисунок 2.15: Изменение содержимого строковых переменных с помощью команды `set`

Вывел в различных форматах значение регистра `edx`, затем изменил значение регистра `ebx` (рис. 2.16).

```

Register group: general
eax          0x8          8
ecx          0x804a000    134520832
edx          0x8          8
ebx          0x2          2
esp          0xfffffcf0    0xfffffcf0
ebp          0x0          0x0
esi          0x0          0
edi          0x0          0
eip          0x8049016    0x8049016 <_start+22>
eflags        0x202      [ IF ]
cs           0x23       35
ss           0x2b       43
ds           0x2b       43
es           0x2b       43
fs           0x0          0
gs           0x0          0

>0x8049016 <_start+22>  mov   $0x4,%eax
0x804901b <_start+27>  mov   $0x1,%ebx
0x8049020 <_start+32>  mov   $0x804a008,%ecx
0x8049025 <_start+37>  mov   $0x7,%edx
0x804902a <_start+42>  int   $0x80
0x804902c <_start+44>  mov   $0x1,%eax
b+ 0x8049031 <_start+49>  mov   $0x0,%ebx
0x8049036 <_start+54>  int   $0x80
0x8049038 add    %al,(%eax)
0x804903a add    %al,(%eax)
0x804903c add    %al,(%eax)
0x804903e add    %al,(%eax)
0x8049040 add    %al,(%eax)
0x8049042 add    %al,(%eax)
0x8049044 add    %al,(%eax)
0x8049046 add    %al,(%eax)
0x8049048 add    %al,(%eax)
0x804904a add    %al,(%eax)

native process 6954 (status) In: _start
(gdb) p/x $edx
$9 = 0x8
(gdb) p/t $edx
$10 = 1000
(gdb) p/c $edx
$11 = 8 'b'
(gdb) set $ebx = '2'
(gdb) p/s $ebx
$12 = 50
(gdb) set $ebx =
(gdb) p/s $ebx
$13 = 2
(gdb) █

```

Рисунок 2.16: Примеры использования команды print и изменения регистра ebx

Разница в выводе `p/s $ebx` связана с тем, что в первом случае в регистр `ebx` записан символьный код '`2`', и отладчик интерпретирует его как адрес строки, а во втором случае — числовое значение `2`.

Завершил выполнение программы командой `continue (c)`, а затем вышел из GDB (рис. 2.17).

The screenshot shows a terminal window with GDB running. At the top, there's a register dump titled "Register group: general". The "eip" register is highlighted, showing its value as 0x8049031 and its location as <_start+49>. Below the registers is a block of assembly code. The assembly code starts at address 0x8049016 and includes instructions like mov \$0x4,%eax, mov \$0x1,%ebx, and int \$0x80. The assembly code ends at address 0x804904a. The bottom part of the terminal shows the interaction with GDB, including setting breakpoints, changing registers, and continuing execution. The command "Breakpoint 2, _start () at lab09-2.asm:20" is visible, indicating a breakpoint was set at the start of the program.

```

Register group: general
eax          0x1          1
ecx          0x804a008      134520840
edx          0x7          7
ebx          0x1          1
esp          0xfffffcfc0    0xfffffcfc0
ebp          0x0          0x0
esi          0x0          0
edi          0x0          0
eip          0x8049031      0x8049031 <_start+49>
eflags        0x202      [ IF ]
cs           0x23         35
ss           0x2b         43
ds           0x2b         43
es           0x2b         43
fs           0x0          0
gs           0x0          0

0x8049016 <_start+22>  mov   $0x4,%eax
0x804901b <_start+27>  mov   $0x1,%ebx
0x8049020 <_start+32>  mov   $0x804a008,%ecx
0x8049025 <_start+37>  mov   $0x7,%edx
0x804902a <_start+42>  int   $0x80
0x804902c <_start+44>  mov   $0x1,%eax
B+>0x8049031 <_start+49> mov   $0x0,%ebx
0x8049036 <_start+54>  int   $0x80
0x8049038 add    %al,(%eax)
0x804903a add    %al,(%eax)
0x804903c add    %al,(%eax)
0x804903e add    %al,(%eax)
0x8049040 add    %al,(%eax)
0x8049042 add    %al,(%eax)
0x8049044 add    %al,(%eax)
0x8049046 add    %al,(%eax)
0x8049048 add    %al,(%eax)
0x804904a add    %al,(%eax)

native process 6954 (status) In: _start
(gdb) p/t $dx
$10 = 1000
(gdb) p/c $dx
$11 = 8 '\b'
(gdb) set $bx = '2'
(gdb) p/s $bx
$12 = 50
(gdb) set $bx =
(gdb) p/s $bx
$13 = 2
(gdb) continue
Continuing.
World!
Breakpoint 2, _start () at lab09-2.asm:20
(gdb) quit
A debugging session is active.

Inferior 1 [process 6954] will be killed.

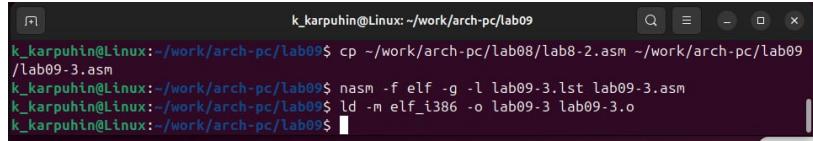
Quit anyway? (y/n) n

```

Рисунок 2.17: Завершение выполнения программы и выход из GDB

2.3 Обработка аргументов командной строки в GDB

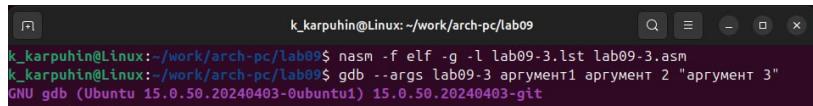
Скопировал файл программы вывода аргументов командной строки из лабораторной работы №8, создал объектный и исполняемый файлы с отладочной информацией (рис. 2.18).



```
k_karpuhin@Linux:~/work/arch-pc/lab09$ cp ~/work/arch-pc/lab08/lab08-2.asm ~/work/arch-pc/lab09  
/lab09-3.asm  
k_karpuhin@Linux:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-3.lst lab09-3.asm  
k_karpuhin@Linux:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-3 lab09-3.o  
k_karpuhin@Linux:~/work/arch-pc/lab09$
```

Рисунок 2.18: Копирование файла программы вывода аргументов командной строки из лабораторной работы №8, создание объектного и исполняемого файлов

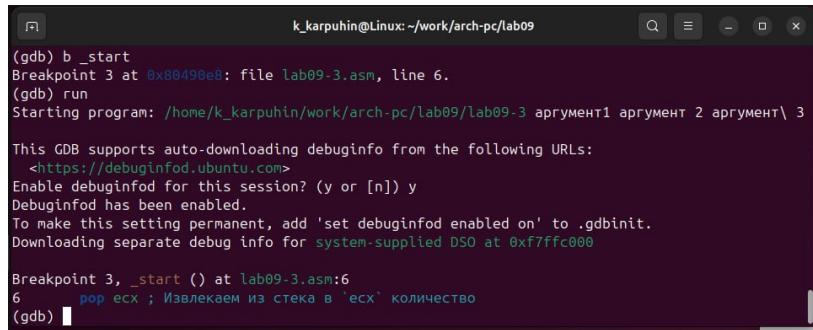
Загрузил программу в GDB с использованием ключа `--args`, указав аргументы (рис. 2.19).



```
k_karpuhin@Linux:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-3.lst lab09-3.asm  
k_karpuhin@Linux:~/work/arch-pc/lab09$ gdb --args lab09-3 аргумент1 аргумент 2 "аргумент 3"  
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
```

Рисунок 2.19: Загрузка программы lab09-3 в GDB с аргументами командной строки

Установил точку останова на метку `_start` и запустил программу (рис. 2.20).



```
(gdb) b _start  
Breakpoint 3 at 0x80490e8: file lab09-3.asm, line 6.  
(gdb) run  
Starting program: /home/k_karpuhin/work/arch-pc/lab09/lab09-3 аргумент1 аргумент 2 аргумент\ 3  
  
This GDB supports auto-downloading debuginfo from the following URLs:  
<https://debuginfod.ubuntu.com>  
Enable debuginfod for this session? (y or [n]) y  
Debuginfod has been enabled.  
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.  
Downloading separate debug info for system-supplied DSO at 0xf7ffc000  
  
Breakpoint 3, _start () at lab09-3.asm:6  
6      pop ecx ; Извлекаем из стека в 'ecx' количество  
(gdb) ■
```

Рисунок 2.20: Установка точки останова и запуск программы

Значение вершины стека хранится в регистре `esp`. Просмотрел содержимое по адресу `esp`. На вершине стека хранилось число `0x5`, то есть общее количество аргументов: `lab09-3`, `аргумент1`, `аргумент 2`, `аргумент 3`.

Далее просмотрел остальные элементы стека, интерпретируя их как адреса строк (рис. 2.21).

```
k_karpuhin@Linux: ~/work/arch-pc/lab09
(gdb) x/x $esp
0xfffffcf80:      0x00000005
(gdb) x/s *(void**)( $esp + 4)
0xfffffd158:      "/home/k_karpuhin/work/arch-pc/lab09/lab09-3"
(gdb) x/x $esp
0xfffffcf80:      0x05
(gdb) x/s *(void**)( $esp + 4)
0xfffffd158:      "/home/k_karpuhin/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)( $esp + 8)
0xfffffd184:      "аргумент1"
(gdb) x/s *(void**)( $esp + 12)
0xfffffd196:      "аргумент"
(gdb) x/s *(void**)( $esp + 16)
0xfffffd1a7:      "2"
(gdb) x/s *(void**)( $esp + 20)
0xfffffd1a9:      "аргумент 3"
(gdb)
```

Рисунок 2.21: Просмотр расположения аргументов командной строки в стеке

Шаг изменения адреса равен 4 байтам ([esp+4], [esp+8], [esp+12] и т.д.), так как в стеке хранятся указатели (адреса строк), каждый из которых занимает 4 байта в 32-разрядной архитектуре.

3 Задание для самостоятельной работы

3.1 Задание 1. Преобразование программы ЛР8 с использованием подпрограммы

В задании требуется преобразовать программу из лабораторной работы №8 (задание №1 для самостоятельной работы), реализуя вычисление значения функции $f(x)$ как подпрограмму.

В ЛР8 была реализована программа, вычисляющая сумму

$$f(x_1) + f(x_2) + \cdots + f(x_n),$$

где значения x_i передаются через аргументы командной строки.

Для своего варианта (вариант 1) использую функцию:

$$f(x) = 2x + 15$$

Создал файл `lab09-task1.asm` и реализовал следующую структуру программы:

- основная программа:
 - извлекает из стека количество аргументов и имя программы;

- в цикле читает очередной аргумент x_i в виде строки;
 - преобразует строку в число (процедура `atoi`);
 - вызывает подпрограмму `_func` для вычисления $f(x_i)$;
 - накапливает сумму значений функции;
 - по завершении цикла выводит результат;
- подпрограмма `_func` реализует вычисление $f(x) = 7 + 2x$.

Текст программы приведён ниже (рис. 3.1):

```

k_karpuhin@Linux:~/work/arch-pc/lab09
/home/k_karpuhin-lab09-task1.asm [-M--] 7 L:[ 1+ 2 3/ 39 ]*(30 / 529b) 0032 0x020 [*][X]
#include "in_out.asm"

SECTION .data
    msg_func db "f(x)=2x+15", 0
    msg_res db "Результат: ", 0

SECTION .text
    global _start

_start:
    pop ecx
    pop edx
    sub ecx, 1
    mov eax, msg_func
    call sprintF
    mov esi, 0

next_arg:
    cmp ecx, 0
    jz _end
    pop eax
    call atoi
    call _func
    add esi, eax
    dec ecx
    jmp next_arg

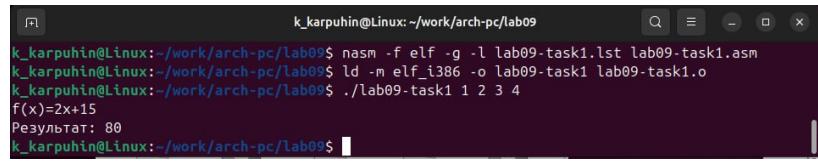
_end:
    mov eax, msg_res
    call sprint
    mov eax, esi
    call iprintLF
    call quit

_func:
    mov ebx, eax
    add eax, eax
    add eax, 15
    ret

```

Рисунок 3.1: Текст программы task1.asm для вычисления суммы значений функции $f(x)$ как подпрограммы

Собрал и запустил программу с набором аргументов 1 2 3 4 (рис. 3.2).



```
k_karpuhin@Linux:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-task1.lst lab09-task1.asm
k_karpuhin@Linux:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-task1 lab09-task1.o
k_karpuhin@Linux:~/work/arch-pc/lab09$ ./lab09-task1 1 2 3 4
f(x)=2x+15
Результат: 80
k_karpuhin@Linux:~/work/arch-pc/lab09$
```

Рисунок 3.2: Запуск программы task1 с набором аргументов 1 2 3 4

В этом случае:

- $f(1) = 15 + 2 \cdot 1 = 17$
- $f(2) = 15 + 2 \cdot 2 = 19$
- $f(3) = 15 + 2 \cdot 3 = 21$
- $f(4) = 15 + 2 \cdot 4 = 23$

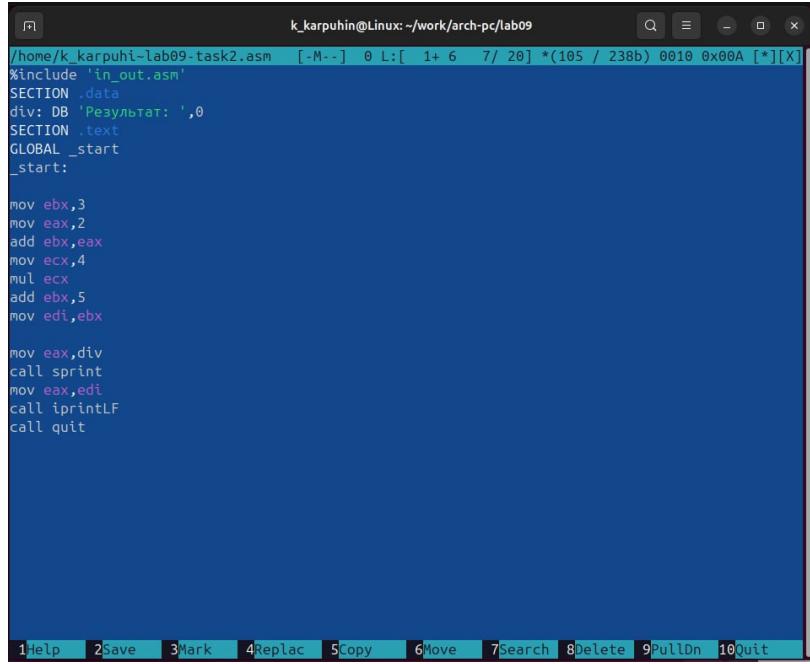
Сумма:

$$17 + 19 + 21 + 23 = 80.$$

Программа вывела результат 80, что подтверждает корректность реализации подпрограммы `_func` и всей программы в целом.

3.2 Задание 2. Поиск и исправление ошибки в программе вычисления выражения

Скопировал программу из листинга 9.3 (рис. 3.3).



```
k_karpuhin@Linux:~/work/arch-pc/lab09
/home/k_karpuhin-lab09-task2.asm [-M--] 0 L:[ 1+ 6 7/ 20] *(105 / 238b) 0010 0x00A [*][X]
%include 'in_out.asm'
SECTION .data
div: DB 'Результат: ',0
SECTION .text
GLOBAL _start
_start:

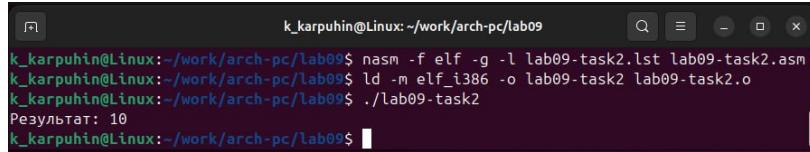
    mov ebx,3
    mov eax,2
    add ebx,eax
    mov ecx,4
    mul ecx
    add ebx,5
    mov edi,ebx

    mov eax,div
    call sprint
    mov eax,edi
    call iprintLF
    call quit

1Help 2Save 3Mark 4Replace 5Copy 6Move 7Search 8Delete 9PullDown 10Quit
```

Рисунок 3.3: Текст программы из листинга 9.3

Собral и запустил программу из листинга 9.3 (рис. 3.4).



```
k_karpuhin@Linux:~/work/arch-pc/lab09
k_karpuhin@Linux:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-task2.lst lab09-task2.asm
k_karpuhin@Linux:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-task2 lab09-task2.o
k_karpuhin@Linux:~/work/arch-pc/lab09$ ./lab09-task2
Результат: 10
k_karpuhin@Linux:~/work/arch-pc/lab09$
```

Рисунок 3.4: Запуск программы из листинга 9.3

Ожидаемое значение выражения:

$$(3 + 2) * 4 + 5 = 5 * 4 + 5 = 20 + 5 = 25.$$

Однако программа выводит значение 10.

3.2.1 Поиск ошибки с помощью GDB

Загрузил программу в GDB и установил точку останова на `_start` (рис. 3.5).

```
k_karpuhin@Linux:~/work/arch-pc/lab09$ gdb lab09-task2
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
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.
--Type <RET> for more, q to quit, c to continue without paging--c
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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
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-task2...
(gdb) break _start
Breakpoint 1 at 0x80499e8: file lab09-task2.asm, line 8.
(gdb) run
Starting program: /home/k_karpuhin/work/arch-pc/lab09/lab09-task2

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for system-supplied DSO at 0xf7ffc000

Breakpoint 1, _start () at lab09-task2.asm:8
8  mov ebx,3
(gdb)
```

Рисунок 3.5: Загрузка программы в GDB и установка точки останова

Пошагово выполнял программу командой `step i`, одновременно просматривая значения регистров (рис. 3.6).

The screenshot shows the GDB debugger interface. The top half displays the register dump:

Register group: general		
eax	0x8	8
ecx	0x4	4
edx	0x0	0
ebx	0xa	10
esp	0xfffffcf90	0xfffffcf90
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x80490fe	0x80490fe <_start+22>
eflags	0x10206	[PF IF RF]
cs	0x23	35
ss	0x2b	43
ds	0x2b	43
es	0x2b	43
fs	0x0	0
gs	0x0	0

The bottom half shows the assembly code:

```

0x80490e7 <quit+12>    ret
B+ 0x80490ed <_start>    mov   $0x1,%ebx
0x80490ed <_start+5>    mov   $0x2,%eax
0x80490f2 <_start+10>   add   %eax,%ebx
0x80490f4 <_start+12>   mov   $0x4,%ecx
0x80490f9 <_start+17>   mul   %ecx
0x80490fb <_start+19>   add   $0x5,%ebx
>0x80490fe <_start+22>  mov   %ebx,%edi
0x8049100 <_start+24>  mov   $0x804a000,%eax
0x8049105 <_start+29>  call  0x804900f <sprint>
0x804910a <_start+34>  mov   %edi,%eax
0x804910c <_start+36>  call  0x8049086 <iprintLF>
0x8049111 <_start+41>  call  0x80490db <quit>
0x8049116          add   %al,(%eax)
0x8049118          add   %al,(%eax)
0x804911a          add   %al,(%eax)
0x804911c          add   %al,(%eax)
0x804911e          add   %al,(%eax)

```

Below the assembly code, the command history and current state are shown:

```

native process 9016 (asm) In: _start
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb)
L14 PC: 0x80490fe

```

Рисунок 3.6: Пошаговое выполнение программы

После выполнения инструкций:

- `mov ebx, 3 → ebx = 3`
- `mov eax, 2 → eax = 2`
- `add ebx, eax → ebx = 5 (сумма 3 + 2)`
- `mov ecx, 4 → ecx = 4`

- `mul ecx → eax = 2 * 4 = 8, edx = 0`

Видно, что умножение выполняется над значением `eax = 2`, а не над суммой $3 + 2$, которая хранится в `ebx`. Далее выполняется `add ebx, 5`, и в итоге в `ebx` оказывается значение 10 , которое и печатается на экран.

Ошибка состоит в том, что перед умножением в `eax` не записана сумма $(3 + 2)$.

3.2.2 Исправление программы

Чтобы получить правильное выражение $(3 + 2) * 4 + 5$, нужно умножать именно сумму, а затем добавить 5 (рис. 3.7).

```

/home/k_karpuhin@Linux:~/work/arch-pc/lab09
k_karpuhin@Linux:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-task2.lst lab09-task2.asm
k_karpuhin@Linux:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-task2 lab09-task2.o
k_karpuhin@Linux:~/work/arch-pc/lab09$ ./lab09-task2
Результат: 25
k_karpuhin@Linux:~/work/arch-pc/lab09$ 
```

Рисунок 3.7: Исправленный текст программы вычисления выражения $(3+2)^*4+5$

Пересобрал и запустил исправленную программу (рис. 3.8).

Рисунок 3.8: Запуск исправленной программы и вывод корректного результата

Теперь результат 25 , что соответствует ожидаемому значению выражения.

4 Выводы

В ходе выполнения данной лабораторной работы я:

- закрепил понятие подпрограммы и на практике реализовал вызов и возврат из подпрограмм с помощью инструкций `call` и `ret`;
- модифицировал программу для вычисления выражения, добавив вложенную подпрограмму и реализовав вычисление композиции функций $f(g(x))$;
- освоил базовые приёмы отладки программ в отладчике GDB: запуск программы, установка и просмотр точек останова, пошаговое выполнение, дизассемблирование кода, переключение синтаксиса команд, использование режима псевдографики;
- научился просматривать и изменять содержимое регистров и ячеек памяти, а также анализировать расположение аргументов командной строки в стеке;
- выполнил задания для самостоятельной работы: преобразовал программу из ЛР8, выделив вычисление функции $f(x)$ в отдельную подпрограмму, и с помощью GDB нашёл и исправил ошибку в программе вычисления выражения $(3 + 2) * 4 + 5$.

В результате работы были укреплены навыки структурирования программ с использованием подпрограмм и навыки отладки на низком уровне, что важно для понимания процессов выполнения программ и эффективного поиска ошибок.