

Лабораторная работа №10. Понятие подпрограммы. Отладчик GDB.

Дисциплина: Архитектура ЭВМ

Плескачева Елизавета Андреевна

Содержание

1	Цель работы	5
1.1	Создание подпрограммы	7
1.2	Отладка с помощью GDB	7
1.2.1	Просмотр дизасемблированного кода	8
1.2.2	Режим псевдографики	9
1.2.3	Информация о точках остановки	10
1.2.4	Работа с данными программы	10
1.2.5	Изменение данных в памяти	11
1.2.6	Изменение содержимого регистра	11
1.3	Обработка аргументов командной строки	12
2	Задания для самостоятельной работы	15
2.1	Написание подпрограммы	15
2.2	Поиск ошибки в программе через GDB	17
3	Выводы	22

Список иллюстраций

1.1	создавание каталога и файла	5
1.2	Листинг 10.1 в gedit	6
1.3	Запуск программы lab10-1	6
1.4	Добавление подпрограммы	7
1.5	Запуск измененной программы lab10-1	7
1.6	Вывод lab10-2.asm	8
1.7	Создание листинга и добавление отладочной информации, открытие через GDB	8
1.8	Запуск программы внутри GDB	8
1.9	Точка останова на start	8
1.10	Просмотр дизассемблированного кода	9
1.11	Изменение отображения на синтаксис intel	9
1.12	Включение псевдографики	10
1.13	Информация об установленных точках	10
1.14	Вывод точек, после добавления новой	10
1.15	Вывод msg1	11
1.16	Вывод содержимого msg2 по адресу	11
1.17	замена символа в msg1	11
1.18	Замена W на K в msg2	11
1.19	Изменение содержимого eax	12
1.20	Завершение программы	12
1.21	Копирование и открытие программы с введенными аргументами	12
1.22	Запуск lab10-3 с точкой останова	13
1.23	Значение esp	13
1.24	Просмотр содержимого стека	13
1.25	Завершение программы	14
2.1	Подпрограмма	16
2.2	Измененная lab9-4.asm	16
2.3	Наблюдение за изменением eax	17
2.4	Изменение неправильного кода	18
2.5	Просмотр eax измененной программы	18
2.6	Дальнейший просмотр программы	19
2.7	Изменение программы	19
2.8	Просмотр eax	20
2.9	Завершение программы	21

Список таблиц

1 Цель работы

Приобретение навыков написания программ с использованием подпрограмм.
Знакомство с методами отладки при помощи GDB и его основными возможностями # Выполнение лабораторной работы

Создадим каталог и файл для выполнения лабораторной работы

```
[eapleskacheva@localhost ~]$ mkdir ~/work/arch-pc/lab10  
cd ~/work/arch-pc/lab10  
touch lab10-1.asm  
[eapleskacheva@localhost lab10]$ gedit lab10-1.asm
```

Рис. 1.1: создание каталога и файла

Введем листинг 10.1 в lab10-1.asm

```

1 %include 'in_out.asm'
2
3 SECTION .data
4 msg:
5     DB 'Введите x: ',0
6 result:
7     DB '2x+7=',0
8 SECTION .bss
9 x: RESB 80
10 rezs: RESB 80
11
12 SECTION .text
13 GLOBAL _start
14 _start:
15
16 ;-----
17 ; Основная программа
18 ;-----
19     mov eax, msg
20     call sprint
21     mov ecx, x
22     mov edx, 80
23     call sread
24     mov eax, x
25     call atoi
26     call _calcul ; Вызов подпрограммы _calcul
27
28     mov eax, result
29     call sprint
30     mov eax, [result]
31     call iprintLF
32     call quit
33
34 ;-----
35 ; Подпрограмма вычисления
36 ; выражения "2x+7"
37
38 _calcul:
39     mov ebx, 2
40     mul ebx
41     add eax, 7
42     mov [result], eax
43     ret
44
45 ; выход из подпрограммы
46

```

Рис. 1.2: Листинг 10.1 в gedit

Скомпилируем и запустим программу.

```

[eapleskacheva@localhost lab10]$ nasm -f elf ./lab10-1.asm
ld -m elf_i386 -o ./lab10-1 ./lab10-1.o
./lab10-1
Введите x: 3
13
[eapleskacheva@localhost lab10]$ ./lab10-1
Введите x: 5
17
[eapleskacheva@localhost lab10]$ 

```

Рис. 1.3: Запуск программы lab10-1

Программа выводит 13 и 7 для 3 и 5, потому что функция $2x + 7$

1.1 Создание подпрограммы

Создадим подпрограмму, которая будет вычислять $g(x) = 3x - 1$

Добавим ее вниз нашего кода и в вычисления

```
38 _calcul:
39     call _subcalcul
40     mov ebx, 2
41     mul ebx
42     add eax, 7
43     mov [result], eax
44     ret
45 ; выход из подпрограммы
46
47 ;; --- SUBCALCUAL
48 ;; g(x) = 3x - 1
49 _subcalcul:
50     mov ebx, 3          ; ebx = 3
51     mul ebx             ; eax = eax*3
52     dec eax             ; eax = eax - 1
53
54     ret
55
56
```

Рис. 1.4: Добавление подпрограммы

Снова запустим и посмотрим, как изменился результат

```
[eapleskacheva@localhost lab10]$ nasm -f elf ./lab10-1.asm
ld -m elf_i386 -o ./lab10-1 ./lab10-1.o
./lab10-1
Введите x: 3
23
[eapleskacheva@localhost lab10]$ ./lab10-1
Введите x: 2
17
[eapleskacheva@localhost lab10]$
```

Рис. 1.5: Запуск измененной программы lab10-1

$$f(g(x)) = 2(3x - 1) + 7 = 6x + 5 \quad f(g(3)) = 18 + 5 = 23 \quad f(g(2)) = 12 + 5 = 17$$

Результат верный

1.2 Отладка с помощью GDB

Скопируем листинг 10.2 в lab10-2.asm, скомпилируем и запустим программу.

```
[eapleskacheva@localhost lab10]$ nasm -f elf ./lab10-2.asm
ld -m elf_i386 -o ./lab10-2 ./lab10-2.o
./lab10-2
Hello, world!
[eapleskacheva@localhost lab10]$
```

Рис. 1.6: Вывод lab10-2.asm

Программа выводит на экран сообщение

Теперь создадим листинг для lab10-2 и откроем исполняемый файл через GDB

```
[eapleskacheva@localhost lab10]$ nasm -f elf -g -l lab10-2.lst lab10-2.asm
ld -m elf_i386 -o lab10-2 lab10-2.o
[eapleskacheva@localhost lab10]$ gdb lab10-2
```

Рис. 1.7: Создание листинга и добавление отладочной информации, открытие через GDB

Открыв программу в gdb запустим ее, написав run

```
(gdb) run
Starting program: /home/eapleskacheva/work/arch-pc/lab10/lab10-2

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.archlinux.org
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.
Hello, world!
[Inferior 1 (process 778059) exited normally]
(gdb)
```

Рис. 1.8: Запуск программы внутри GDB

Поставим точку остановки на _start и запустим программу

```
[Inferior 1 (process 778059) exited normally]
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab10-2.asm, line 13.
(gdb) run
Starting program: /home/eapleskacheva/work/arch-pc/lab10/lab10-2

Breakpoint 1, _start () at lab10-2.asm:13
13      mov eax, 4
(gdb)
```

Рис. 1.9: Точка останова на start

Теперь программа запустилась, но остановилась на _start

1.2.1 Просмотр дизасемблированного кода

Напишем disassemble _start что бы посмотреть дизасемблированный код


```

(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.
(gdb) 

```

Рис. 1.10: Просмотр дизассемблированного кода

Изменим отображение на intel

```

End of assembler dump.
(gdb) 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.
(gdb) 

```

Рис. 1.11: Изменение отображения на синтаксис intel

Теперь программа выглядит как на NASM

1.2.2 Режим псевдографики

Напишем `layout asm layout regs` и `run` что бы включить режим псевдографики и запустить программу

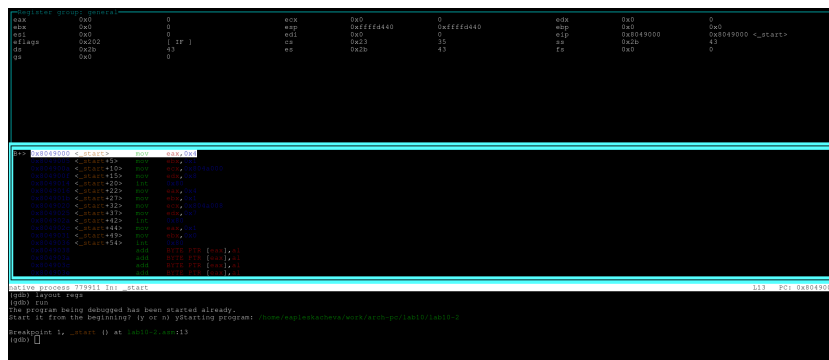


Рис. 1.12: Включение псевдографики

1.2.3 Информация о точках останковки

Посмотрим информацию о поставленных точках останковки, написав `i b`

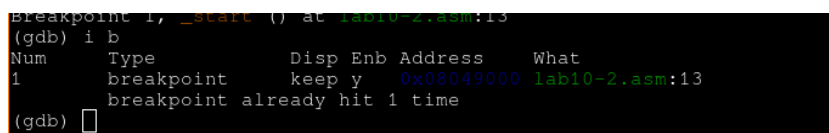


Рис. 1.13: Информация об установленных точках

Поставим точку останковки на `0x8049031`

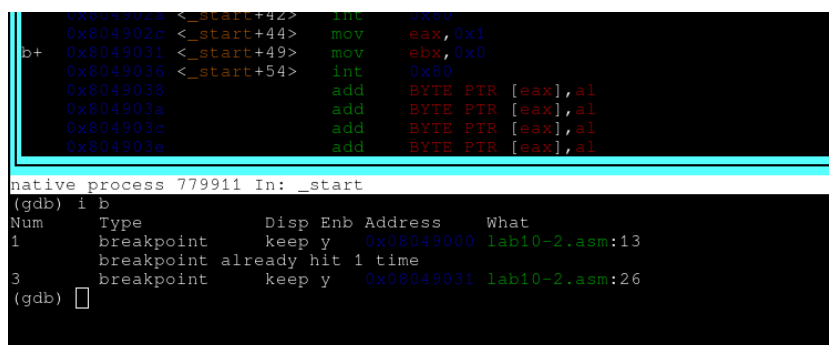


Рис. 1.14: Вывод точек, после добавления новой

Посмотрим информацию о точках останковки и увидим там новую точку

1.2.4 Работа с данными программы

Выведем содержимое переменной `msg1`

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

Рис. 1.15: Вывод msg1

Выведем содержимое msg2 по адресу

```
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n\034"
(gdb) □
```

Рис. 1.16: Вывод содержимого msg2 по адресу

1.2.5 Изменение данных в памяти

Заменяем “H” на “h” в msg1

```
msg1 has unknown type, cast it to its def
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hello, "
(gdb) □
```

Рис. 1.17: замена символа в msg1

Изменим любой символ в msg2

```
(gdb) set {char}&msg2='K'
(gdb) x/1sb &msg2
0x804a008 <msg2>:      "Korld!\n\034"
(gdb) □
```

Рис. 1.18: Замена W на K в msg2

1.2.6 Изменение содержимого регистра

Изменим значение ebx на ‘2’ и на 2

```

Register group: general
eax 0x0 0 ecx 0x0 0
edx 0x0 0 ebx 0x2 2
esp 0xffffd440 0xffffd440 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

0x804903c add BYTE PTR [eax],al
0x804903e add BYTE PTR [eax],al
0x8049040 add BYTE PTR [eax],al
0x8049042 add BYTE PTR [eax],al
0x8049044 add BYTE PTR [eax],al
0x8049046 add BYTE PTR [eax],al
0x8049048 add BYTE PTR [eax],al

native_process 779911 In: _start L13 P
(gdb) set (char*)&msg2='K'
(gdb) x/1sb &msg2
0x804a008 <msg2>: "Korld!\n\034"
(gdb) set $ebx='2'
(gdb) p/s $ebx
$1 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$2 = 2
(gdb)

```

Рис. 1.19: Изменение содержимого еах

В начале программа выводит код символа 2, а потом число 2

Продолжим исполнение программы и завершим ее

```

(gdb) p/s $ebx
$2 = 2
(gdb) c
Continuing.
hello, Korld!

Breakpoint 3, _start () at lab10-2.asm:26
(gdb) quit

```

Рис. 1.20: Завершение программы

Заметим, что выводимое сообщение изменилось

1.3 Обработка аргументов командной строки

Скопируем lab9-2.asm в lab10-3.asm создадим исполняемый файл с отладочной информацией и откроем программу через gdb добавив аргументы.

```

[teapleskacheva@localhost lab10]$ cp ~/work/arch-pc/lab09/lab9-2.asm ~/work/arch-pc/lab10/lab10-3.asm
[teapleskacheva@localhost lab10]$ nasm -f elf -g -l lab10-3.lst lab10-3.asm
ld -m elf_i386 -o lab10-3 lab10-3.o
[teapleskacheva@localhost lab10]$ gdb --args lab10-3 аргумент1 аргумент 2 'аргумент 3'

```

Рис. 1.21: Копирование и открытие программы с введенными аргументами

Поставим точку останова на _start и запустим программу

```

Reading symbols from lab10-3...
(gdb) b _start
Breakpoint 1 at 0x80490e5: file lab10-3.asm, line 7.
(gdb) run
Starting program: /home/eapleskacheva/work/arch-pc/lab10/lab10-3 аргумент1 аргумент 2 аргумент\ 3

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.archlinux.org
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.

Breakpoint 1, _start () at lab10-3.asm:7
7       pop    ecx ; Направляем из стека в 'ecx' количество

```

Рис. 1.22: Запуск lab10-3 с точкой останова

Выведем значение esp

```

End of assembler dump.
(gdb) x/x $esp
0xffffd3f0:      0x00000005
(gdb) 

```

Рис. 1.23: Значение esp

В esp лежит 5, потому что аргументов 5

Посмотрим остальное содержимое стека

```

(gdb) x/s *(void**)($esp + 4)
0xffffd569:      "/home/eapleskacheva/work/arch-pc/lab10/lab10-3"
(gdb) x/s *(void**)($esp + 8)
0xffffd598:      "аргумент1"
(gdb) x/s *(void**)($esp + 12)
0xffffd5aa:      "аргумент"
(gdb) x/s *(void**)($esp + 16)
0xffffd5bb:      "2"
(gdb) x/s *(void**)($esp + 20)
0xffffd5bd:      "аргумент 3"
(gdb) x/s *(void**)($esp + 24)
0x0:      <error: Cannot access memory at address 0x0>
(gdb) 

```

Рис. 1.24: Просмотр содержимого стека

Мы увеличиваем, значение адреса на 4 потому что столько занимает размер указателя на аргумент из стека.

Продолжим программу и завершим ее

```
(gdb) c
Continuing.
аргумент1
аргумент
2
аргумент 3
[Inferior 1 (process 812913) exited normally]
(gdb) q
[eapleskacheva@localhost lab10]$
```

Рис. 1.25: Завершение программы

2 Задания для самостоятельной работы

2.1 Написание подпрограммы

Изменим программу из самостоятельно лабораторной 9, представив функцию как подпрограмму.

Напишем эту программу в низу кода

```

47 subfunc:
48 ; f(x) = 3x -1
49 ; eax = x
50 ; eax = res
51 push ebx
52 push ecx
53 push edx
54
55 mov ecx,3
56 mul ecx
57 dec eax
58
59 pop edx
60 pop ecx
61 pop ebx
62
63
64 ret
65

```

Рис. 2.1: Подпрограмма

Запустим и проверим правильность программы

```

[eaпleskacheva@localhost lab10]$ nasm -f elf ./lab9-4.asm
[eaпleskacheva@localhost lab10]$ ld -m elf_i386 -o ./lab9-4 ./lab9-4.o
[eaпleskacheva@localhost lab10]$ ./lab9-4 1 2 3 4
Функция: f(x) = 3x + -1:
Результат: 26
[eaпleskacheva@localhost lab10]$ 

```

Рис. 2.2: Измененная lab9-4.asm

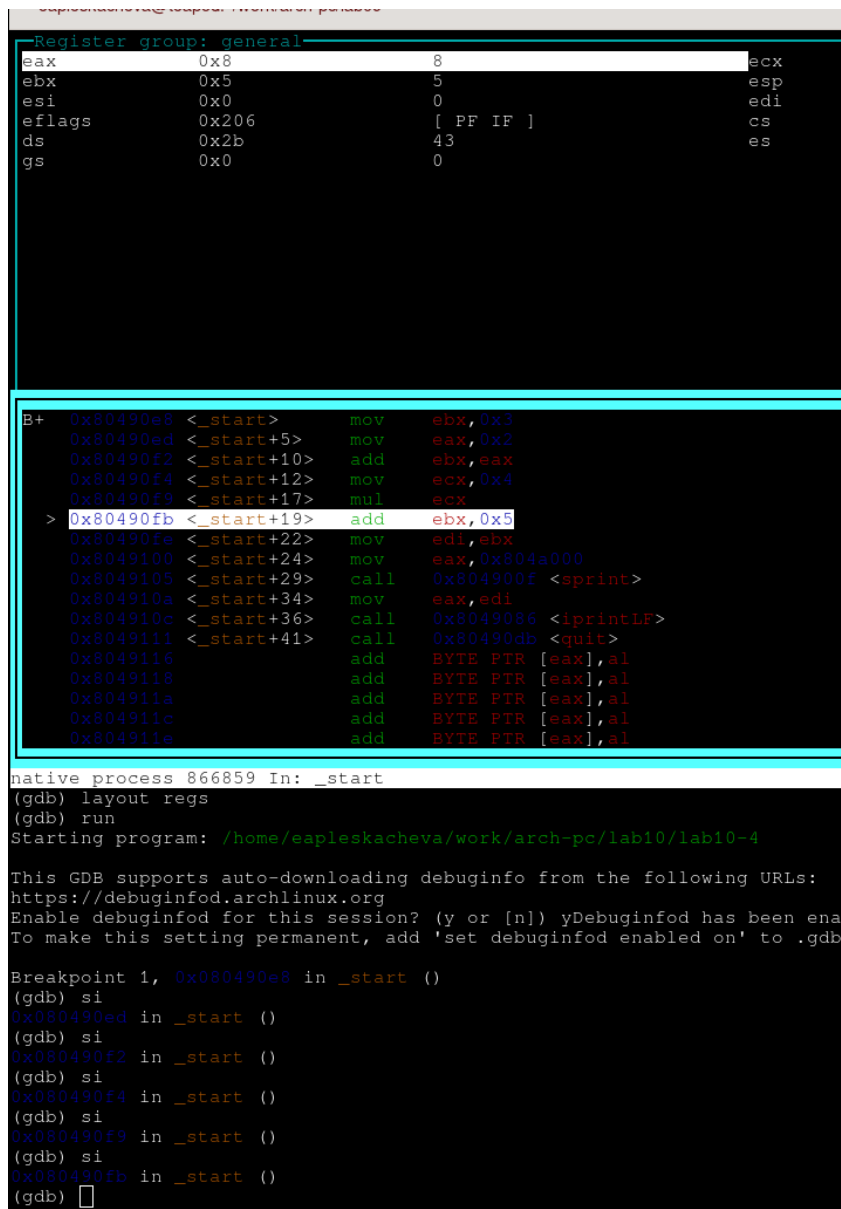
Результат верный

2.2 Поиск ошибки в программа через GDB

Введем в lab10-4.asm код из листинга 10.3, скомпилируем его и запустим через GDB

Так как происходит умножение, то после исполнения `mul` в `eax` должен оказаться результат вычисления. Там должно лежать $(3+2)*4 = 20$

Пройдемся до исполнения этой части программы



The screenshot shows the GDB interface with the following content:

```
Register group: general
eax      0x8      8      ecx
ebx      0x5      5      esp
esi      0x0      0      edi
eflags   0x206    [ PF IF ] cs
ds       0x2b     43     es
gs       0x0      0

B+ 0x80490e8 <_start>    mov     ebx,0x3
0x80490ed <_start+5>    mov     eax,0x2
0x80490f2 <_start+10>   add     ebx,eax
0x80490f4 <_start+12>   mov     ecx,0x4
0x80490f9 <_start+17>   mul     ecx
> 0x80490fb <_start+19> add     ebx,0x5
0x80490fe <_start+22>   mov     edi,ebx
0x8049100 <_start+24>   mov     eax,0x804a000
0x8049105 <_start+29>   call   0x804900f <sprint>
0x804910a <_start+34>   mov     eax,edi
0x804910c <_start+36>   call   0x8049086 <iprintf>
0x8049111 <_start+41>   call   0x80490db <quit>
0x8049116             add     BYTE PTR [eax],al
0x8049118             add     BYTE PTR [eax],al
0x804911a             add     BYTE PTR [eax],al
0x804911c             add     BYTE PTR [eax],al
0x804911e             add     BYTE PTR [eax],al

native process 866859 In: _start
(gdb) layout regs
(gdb) run
Starting program: /home/eapleskacheva/work/arch-pc/lab10/lab10-4

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.archlinux.org
Enable debuginfod for this session? (y or [n]) yDebuginfod has been ena
To make this setting permanent, add 'set debuginfod enabled on' to .gdb

Breakpoint 1, 0x080490e8 in _start ()
(gdb) si
0x080490ed in _start ()
(gdb) si
0x080490f2 in _start ()
(gdb) si
0x080490f4 in _start ()
(gdb) si
0x080490f9 in _start ()
(gdb) si
0x080490fb in _start ()
(gdb) □
```

Рис. 2.3: Наблюдение за изменением `eax`

После исполнения умножения в еах лежит 8. Это потому что мы умножаем еах на есх, а сложение производилось в ебх

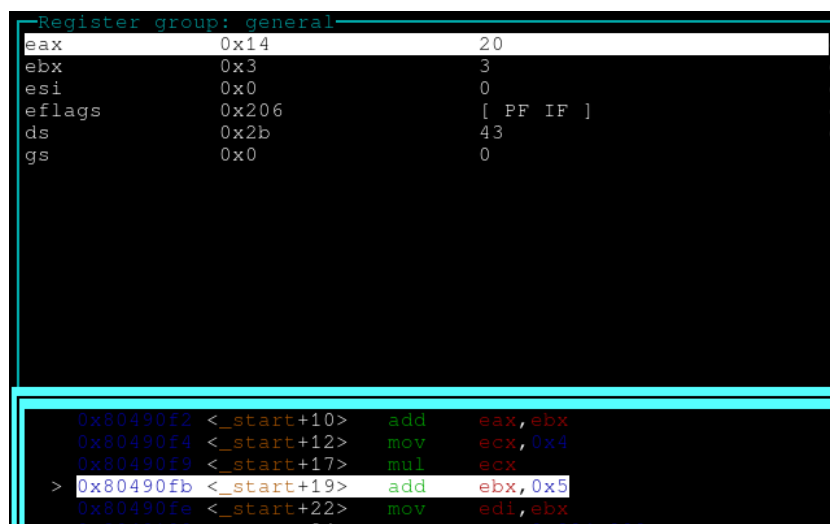
Изменим код так, что бы программа работала верно

```
14      mov     eax,2
15      add     eax, ebx
16      mov     ecx,4
```

Рис. 2.4: Изменение неправильного кода

Теперь суммирование будет в еах

Снова запустим GDB и дойдем до того же момента.



The screenshot shows the GDB interface. The top panel displays the 'Register group: general' with the following values: eax (0x14, 20), ebx (0x3, 3), esi (0x0, 0), eflags (0x206, [PF IF]), ds (0x2b, 43), and gs (0x0, 0). The bottom panel shows assembly code with the current instruction highlighted: > 0x80490fb <_start+19> add ebx,0x5. Other visible instructions include 0x80490f2 <_start+10> add eax,ebx, 0x80490f4 <_start+12> mov ecx,0x4, 0x80490f9 <_start+17> mul ecx, and 0x80490fe <_start+22> mov edi,ebx.

Рис. 2.5: Просмотр еах измененной программы

Теперь в еах лежит 20

Будем просматривать прогармум дальше

```

Register group: general
eax      0x804a000      134520832
ebx      0x8           8
esi      0x0           0
eflags   0x202         [ IF ]
ds       0x2b          43
gs       0x0           0

0x80490f2 <_start+10>  add    eax,ebx
0x80490f4 <_start+12>  mov    ecx,0x4
0x80490f9 <_start+17>  mul    ecx
0x80490fb <_start+19>  add    ebx,0x5
0x80490fe <_start+22>  mov    edi,ebx
0x8049100 <_start+24>  mov    eax,0x804a000
> 0x8049105 <_start+29>  call   0x804900f <sprint>

```

Рис. 2.6: Дальнейший просмотр программы

Заметим, что в `eax` записывается сообщение для вывода, а само значение никуда не сохраняется, а так же 5 прибавляется к `ebx`, в котором до этого лежало 3, поэтому теперь там 8

Изменим программу так что бы она работала корректно

```

17      mul    ecx
18      add    eax,5
19      mov    edi,eax
20

```

Рис. 2.7: Изменение программы

Запустим программу теперь и понаблюдаем за ее исполнением

```

eax      0x0      0
eax      0x19     25
ebx      0x3      3
ds       0x2b     43
eflags   0x202    [ IF ]

0x804900f <sprint>    push    edx
B+> 0x80490e8 <_start>    mov     ebx, 0x3
B+  0x80490e8 <_start>    mov     ebx, 0x3
    0x80490ed <_start+5>    mov     eax, 0x2
    0x80490f2 <_start+10>   add     eax, ebx00 <slen>
    0x80490f4 <_start+12>   mov     ecx, 0x4
    0x80490f9 <_start+17>   mul     ecx, 0x5
    0x80490fb <_start+19>   add     eax, 0x5
    0x80490fe <_start+22>   mov     edi, eax04a000
> 0x8049100 <_start+24>   mov     eax, 0x804a000 <rint>
    0x8049105 <_start+29>   call    0x804900f <sprint>
    0x804910a <_start+34>   mov     eax, edi86 <iprintLF>
    0x804910c <_start+36>   call    0x8049086 <iprintLF>
    0x8049111 <_start+41>   call    0x80490db <quit>1
    0x8049116               add     BYTE PTR [eax], al
    0x8049118               add     BYTE PTR [eax], al>
    0x804911a               add     BYTE PTR [eax], al
    0x804911c               add     BYTE PTR [eax], al
nativ0x804911e 877524 In: spradd     BYTE PTR [eax], al
(gdb) sprocess 877917 In: _start
0x08049process 878107 In: _start
Pezynbrat: 25
[Inferior 1 (process 877917) exited normally]
(gdb) run
Starting program: /home/eapleskacheva/work/arch-pc/lab10.

Breakpoint 1, 0x080490e8 in _start ()
(gdb) si
0x080490ed in _start ()
(gdb) si
0x080490f2 in _start ()
(gdb) si
0x080490f4 in _start ()
(gdb) si
0x080490f9 in _start ()
(gdb) si
0x080490fb in _start ()
(gdb) si
0x080490fe in _start ()
(gdb) si
0x08049100 in _start ()

```

Рис. 2.8: Просмотр eax

```
(gdb) c
Continuing.
Результат: 25
[Inferior 1 (process 878107) exited normally]
(gdb) □
```

Рис. 2.9: Завершение программы

Теперь программа вычисляет значение правильно, на этом ее можно завершить

3 Выводы

Мы приобрели навыки написания программ с использованием подпрограмм и ознакомились с процессом отладки через программу GDB и научились пользоваться его основными возможностями.