

# **Отчёт по лабораторной работе №10**

**Подпрограммы. Отладчик GDB**

Мулин Иван Владимирович

# Содержание

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Цель работы</b>                                      | <b>4</b>  |
| <b>2</b> | <b>Ход работы</b>                                       | <b>5</b>  |
| 2.1      | Выполнение лабораторной работы . . . . .                | 5         |
| 2.1.1    | Работа с подпрограммами . . . . .                       | 5         |
| 2.1.2    | Отладчик GDB . . . . .                                  | 5         |
| 2.2      | Выполнение заданий для самостоятельной работы . . . . . | 11        |
| <b>3</b> | <b>Листинги написанных программ</b>                     | <b>12</b> |
| <b>4</b> | <b>Заключение</b>                                       | <b>19</b> |

## Список иллюстраций

|      |   |    |
|------|---|----|
| 2.1  | Значение выражения $f(x)=2x+7$ . . . . .              | 5  |
| 2.2  | Значение выражения $f(g(x))$ . . . . .                | 5  |
| 2.3  | Запуск программы в отладчике . . . . .                | 6  |
| 2.4  | Установка первой точки останова . . . . .             | 6  |
| 2.5  | Дизассемблирование программы в режиме АТТ . . . . .   | 6  |
| 2.6  | Дизассемблирование программы в режиме Intel . . . . . | 7  |
| 2.7  | layout regs . . . . .                                 | 7  |
| 2.8  | layout asm . . . . .                                  | 8  |
| 2.9  | Просмотр точек останова . . . . .                     | 8  |
| 2.10 | Обзор новых точек останова . . . . .                  | 8  |
| 2.11 | Обзор значений регистров . . . . .                    | 9  |
| 2.12 | Просмотр значения строки msg1 . . . . .               | 9  |
| 2.13 | Обращение к строке msg2 по её адресу . . . . .        | 9  |
| 2.14 | Замена буквы в msg1 . . . . .                         | 10 |
| 2.15 | Замена буквы в msg2 . . . . .                         | 10 |
| 2.16 | Значение регистра edx в разных форматах . . . . .     | 10 |
| 2.17 | Изменение регистра edx . . . . .                      | 10 |
| 2.18 | Обзор значений регистров . . . . .                    | 11 |
| 2.19 | Работающая программа lab10-4 . . . . .                | 11 |

# 1 Цель работы

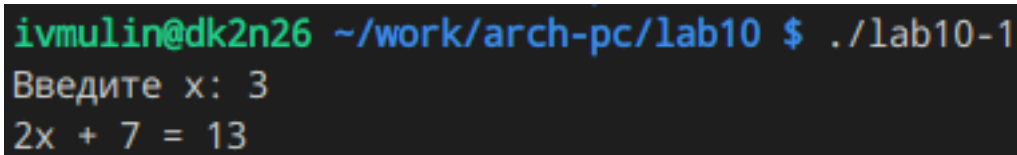
Цель работы - изучить написание программ, использующих подпрограммы, а также ознакомиться с основными возможностями отладчика GDB. Репозиторий автора расположен по адресу [https://github.com/ivmulin/study\\_2022-2023\\_arch-ps](https://github.com/ivmulin/study_2022-2023_arch-ps).

## 2 Ход работы

### 2.1 Выполнение лабораторной работы

#### 2.1.1 Работа с подпрограммами

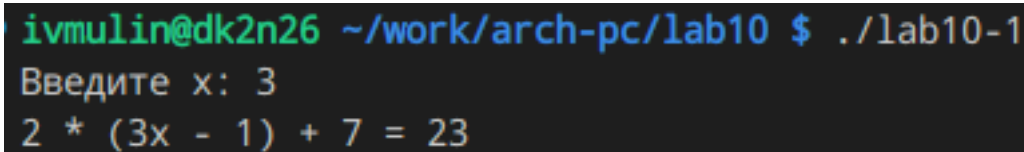
Напишем программу `lab10-1.asm`, которая использует подпрограмму для вычисления значения функции  $f(x) = 2x + 7$  в зависимости от введённого значения аргумента:



```
ivmulin@dk2n26 ~/work/arch-pc/lab10 $ ./lab10-1
Введите x: 3
2x + 7 = 13
```

Рис. 2.1: Значение выражения  $f(x)=2x+7$

Перепишем эту программу так, чтобы она при помощи подпрограмм выводила значение выражения  $f(g(x))$ , где  $g(x) = 3x - 1$ :



```
ivmulin@dk2n26 ~/work/arch-pc/lab10 $ ./lab10-1
Введите x: 3
2 * (3x - 1) + 7 = 23
```

Рис. 2.2: Значение выражения  $f(g(x))$

#### 2.1.2 Отладчик GDB

Запустим программу `lab10-2.asm` в отладчике GDB:

```
(gdb) r
Starting program: /afs/.dk.sci.pfu.edu.ru/home/i/v/ivmulin/work/arch-pc/lab10/lab10-2
Hello, vadim!
[Inferior 1 (process 5270) exited normally]
```

Рис. 2.3: Запуск программы в отладчике

Далее установим точку останова на метке `_start`:

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab10-2.asm, line 10.
(gdb) r
Starting program: /afs/.dk.sci.pfu.edu.ru/home/i/v/ivmulin/work/arch-pc/lab10/lab10-2

Breakpoint 1, _start () at lab10-2.asm:10
10      mov eax, 4
```

Рис. 2.4: Установка первой точки останова

Дизассемблируем программу, начиная с метки `_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
--Type <RET> for more, q to quit, c to continue without paging--c
    0x0804902c <+44>:     mov     $0x1,%eax
    0x08049031 <+49>:     mov     $0x0,%ebx
    0x08049036 <+54>:     int     $0x80
End of assembler dump.
```

Рис. 2.5: Дизассемблирование программы в режиме АТТ

Изначально дизассемблированный код отображается в стиле АТТ. Переключим его на Intel:

```

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.

```

Рис. 2.6: Дизассемблирование программы в режиме Intel

Как видно, отображение в стиле АТТ и в стиле Intel отличаются: к примеру, дизассемблированный код АТТ устанавливает символ \$ перед ячейками памяти и числами и % перед названием регистра, чего не делает отображение Intel. Более того, порядок операндов в инструкциях с двумя операндами (таких, как, например, mov или add) различен в разных видах отображений.

Отобразим окно регистров при помощи команды `layout regs` и `layout asm`:

```

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

exec No process In:                                L??  PC: ??
(gdb) layout regs

```

Рис. 2.7: `layout regs`

```
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

exec No process In:                                     L??  PC: ??
(gdb) layout regs
```

Рис. 2.8: layout asm

Выведем информацию обо всех добавленных точках останова:

```
(gdb) i b
Num      Type      Disp Enb Address      What
1        breakpoint keep y   0x08049000 lab10-2.asm:10
```

Рис. 2.9: Просмотр точек останова

Установим точку останова по адресу предпоследней инструкции в программе:

```
(gdb) i b
Num      Type      Disp Enb Address      What
1        breakpoint keep y   0x08049000 lab10-2.asm:10
2        breakpoint keep y   0x08049031 lab10-2.asm:21
```

Рис. 2.10: Обзор новых точек останова

При помощи GDB можно просматривать информацию о регистрах:



|        |            |                       |
|--------|------------|-----------------------|
| eax    | 0x8        | 8                     |
| ecx    | 0x804a000  | 134520832             |
| edx    | 0x8        | 8                     |
| ebx    | 0x1        | 1                     |
| esp    | 0xffffc3e0 | 0xffffc3e0            |
| 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                     |

Рис. 2.11: Обзор значений регистров

Кроме того, можно получать значение памяти по нужному адресу:

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

Рис. 2.12: Просмотр значения строки msg1

Для печати обращение к памяти можно выполнять с использованием адресов:

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

Рис. 2.13: Обращение к строке msg2 по её адресу

Используя команду `set`, заменим первую букву в строке `msg1`. Попутно изменим вторую букву в `msg2`.

```
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hello, "
```

Рис. 2.14: Замена буквы в msg1

```
(gdb) set {char}0x804a009='o'
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "vodim!\n\034"
```

Рис. 2.15: Замена буквы в msg2

Выведем в шестнадцатичном, двоичном и символьном форматах значение регистра edx:

```
(gdb) p/x $edx
$4 = 0x8
(gdb) p/t $edx
$5 = 1000
(gdb) p/s $edx
$6 = 8
```

Рис. 2.16: Значение регистра edx в разных форматах

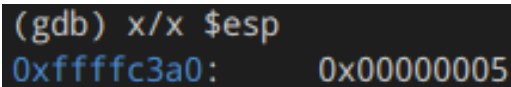
Изменим значение регистра edx сначала на '2', затем на 2. Значения '2' и 2 отличаются, и отладчик выводит их коды в таблице ASCII.

```
(gdb) set $ebx='2'
(gdb) p/s $ebx
$7 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$8 = 2
```

Рис. 2.17: Изменение регистра edx

Напишем программу lab10-3.asm и запустим её в отладчике при помощи команды

```
gdb --args lab10-3 аргумент1 аргумент 2 'аргумент 3'
```



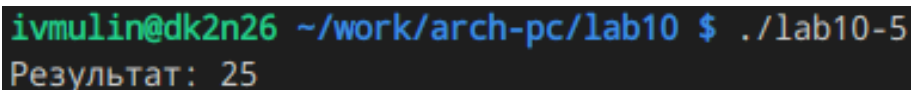
```
(gdb) x/x $esp
0xfffffc3a0: 0x00000005
```

Рис. 2.18: Обзор значений регистров

В регистре `esp` хранится число, равное количеству переданных аргументов. Остальные значения стека можно просмотреть при помощи четырёхкратного инкремента (к примеру, `[$esp+4]`, `[$esp+8]`). Инкремент равен четырём из-за того, что аргументы хранятся как двойное слово, то есть занимают объём в 4 байта.

## 2.2 Выполнение заданий для самостоятельной работы

Скорректируем код программы из задания 1 самостоятельной работы к лабораторной работе № 9 так, чтобы значение функции вычислялось в отдельной подпрограмме:



```
ivmulin@dk2n26 ~/work/arch-pc/lab10 $ ./lab10-5
Результат: 25
```

Рис. 2.19: Работающая программа `lab10-4`

Скопируем текст программы, вычисляющей значение выражения  $(3+2)*4+5$ . В ней, очевидно, допущена ошибка. Текст исправленной программы представлен ниже в качестве программы `lab10-5.asm`

## 3 Листинги написанных программ

1. lab10-1.asm

```
%include 'in_out.asm'
```

```
section .data
```

```
msg: db 'Введите x: ', 0
```

```
result: db '2 * (3x - 1) + 7 = ', 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:
    ;  $f(x) = 2x + 7$ 
    call _subcalcul
    mov ebx, 2
    mul ebx
    add eax, 7
    mov [res], eax
    ret

```

```

_subcalcul:
    ;  $g(x) = 3x - 1$ 
    mov ebx, 3
    mul ebx
    sub eax, 1
    ret

```

2. lab10-2.asm

```

section .data
    msg1: db "Hello, ", 0x0
    msg1Len: equ $-msg1
    msg2: db "vadim!", 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

```

### 3. lab10-3.asm

```

#include 'in_out.asm'

```

```

section .text
global _start

```

```

_start:

```

```
pop ecx
pop edx
sub ecx, 1
```

next:

```
cmp ecx, 0
jz _end

pop eax
call sprintf
loop next
```

\_end:

```
call quit
```

4. lab10-4.asm

```
%include 'in_out.asm'
```

```
; f(x) = 15x + 2
```

```
section .data
```

```
msg db "Результат: ", 0
fun db "f(x) = 15 * x + 2", 10
```

```
section .bss
```

```
result resb 10
```

```
section .text
```

```
global _start
```

```

_start:
    mov eax, fun
    call sprint

    pop ecx ; Извлекаем из стека в `ecx` количество аргументов
    pop edx ; Извлекаем из стека в `edx` имя программы
    sub ecx, 1 ; Уменьшаем `ecx` на 1 (количество аргументов без названия программы)
    mov esi, 0 ; используем 'esi' для хранения промежуточных сумм

    mov eax, 0
    mov [result], eax

extractArguments:
    cmp ecx, 0 ; проверяем, есть ли еще аргументы
    jz _end ; если аргументов нет, выходим из цикла

    pop eax
    call atoi

    call funkcja
    add [result], eax

    loop extractArguments

_end:
    mov eax, msg
    call sprint

    mov eax, [result]

```



```
call iprintLF
```

```
call quit
```

```
funkcia:
```

```
mov ebx, 15
```

```
mul ebx
```

```
add eax, 2
```

5. lab10-5.asm

```
%include 'in_out.asm'
```

```
section .data
```

```
div: db 'Результат: ',0
```

```
SECTION .text
```

```
global _start
```

```
_start:
```

```
; ---- Вычисление выражения (3+2)*4+5
```

```
mov ebx, 3
```

```
mov eax, 2
```

```
add eax, ebx
```

```
mov ecx, 4
```

```
mul ecx
```

```
add eax, 5
```

```
mov edi, eax
```

```
; ---- Вывод результата на экран
```

```
mov eax, div
```

```
call sprint
```

```
mov eax,edi  
call iprintLF  
  
call quit
```

## 4 Заключение

Поставленная в начале данного отчёта цель была, очевидно, достигнута, ведь был освоен процесс использования подпрограмм в языке ассемблера NASM и отладчика GDB.