

Лабораторная работа №10

Архитектура компьютера

Скандарова Полина Юрьевна

Содержание

1	Цель работы	5
2	Выполнение лабораторной работы	6
3	Задание для самостоятельной работы	20
4	Выводы	25

Список иллюстраций

2.1	Текст программы в файле	7
2.2	Проверка работы программы	7
2.3	Текст программы в файле	8
2.4	Проверка работы программы	8
2.5	Текст программы в файле	9
2.6	Загрузка программы в оболочку GDB и её запуск	10
2.7	Запуск программы с брейкпоинтом	10
2.8	Дисассимилированный код программы	11
2.9	Дисассимилированный код программы с Intel'овским синтаксисом	12
2.10	Режим псевдографики gdb	13
2.11	Информация о точке останова	14
2.12	Новая точка останова и информация о ней	14
2.13	Результат работы команды info registers	15
2.14	Содержимое переменной msg1	15
2.15	Содержимое переменной msg1	15
2.16	Изменение содержимого переменной msg1	16
2.17	Изменение содержимого переменной msg2	16
2.18	Значение регистра edx в разных форматах	16
2.19	Меняющиеся значения регистра ebx	17
2.20	Выход из GDB	17
2.21	Загрузка файла lab10-3 в GDB	18
2.22	Запуск файла lab10-3 в GDB	18
2.23	Аргументы программы по их адресам	19
3.1	Изменённая программа в файле	20
3.2	Проверка работы программы	20
3.3	Программа из листинга в файле	21
3.4	Значения регистров в процессе выполнения программы	22
3.5	В выводе участвует регистр ebx вместо регистра eax	23
3.6	Исправленная программа в файле	23
3.7	Результат выполнения исправленной программы	24

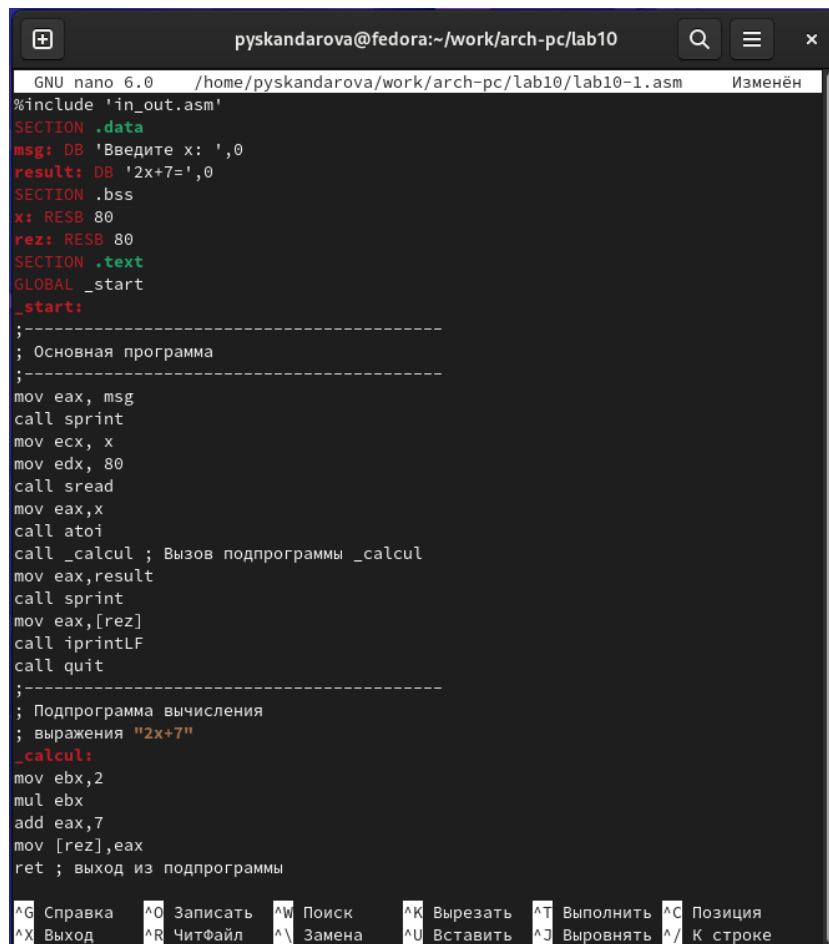
Список таблиц

1 Цель работы

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

2 Выполнение лабораторной работы

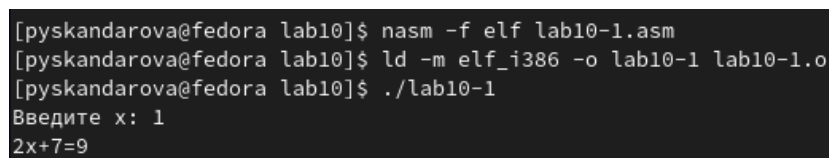
Для начала необходимо создать каталог для выполнения лабораторной работы №10, перейти в него и создать файл lab10-1.asm. В качестве примера рассматривается программа вычисления арифметического выражения $f(x) = 2x + 7$ с помощью подпрограммы _calcul. В данном примере x вводится с клавиатуры, а само выражение вычисляется в подпрограмме. Внимательно изучаю текст программы и ввожу её в файл. (рис. 2.1)



```
GNU nano 6.0 /home/pyskandarova/work/arch-pc/lab10/lab10-1.asm
%include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
result: DB '2x+7=',0
SECTION .bss
x: RESB 80
rez: 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, [rez]
call iprintLF
call quit
;-----
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
mov ebx, 2
mul ebx
add eax, 7
mov [rez], eax
ret ; выход из подпрограммы
```

Рис. 2.1: Текст программы в файле

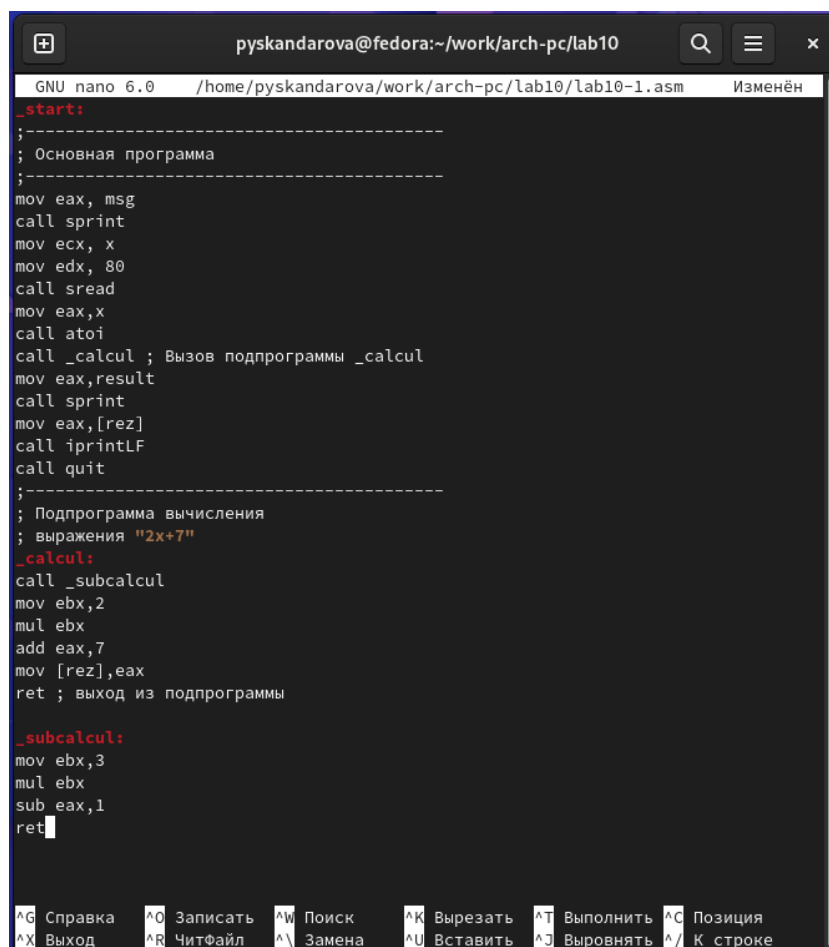
Создаю исполняемый файл и проверяю его работу. (рис. 2.2)



```
[pyskandarova@fedora lab10]$ nasm -f elf lab10-1.asm
[pyskandarova@fedora lab10]$ ld -m elf_i386 -o lab10-1 lab10-1.o
[pyskandarova@fedora lab10]$ ./lab10-1
Введите x: 1
2x+7=9
```

Рис. 2.2: Проверка работы программы

Изменяю текст программы, добавив подпрограмму `_subcalcul` в подпрограмму `_calcul`, для вычисления выражения $f(g(x))$, где x вводится с клавиатуры, $f(x) = 2x + 7$, $g(x) = 3x - 1$. (рис. 2.3)

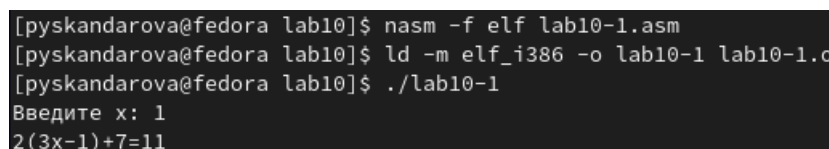


```
GNU nano 6.0 /home/pyskandarova/work/arch-pc/lab10/lab10-1.asm Изменён
_start:
;-----
; Основная программа
;-----
mov eax, msg
call sprint
mov ecx, x
mov edx, 80
call sread
mov eax, x
call atoi
call _calcul ; Вызов подпрограммы _calcul
mov eax, result
call sprint
mov eax, [rez]
call iprintLF
call quit
;-----
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
call _subcalcul
mov ebx, 2
mul ebx
add eax, 7
mov [rez], eax
ret ; выход из подпрограммы

_subcalcul:
mov ebx, 3
mul ebx
sub eax, 1
ret
```

Рис. 2.3: Текст программы в файле

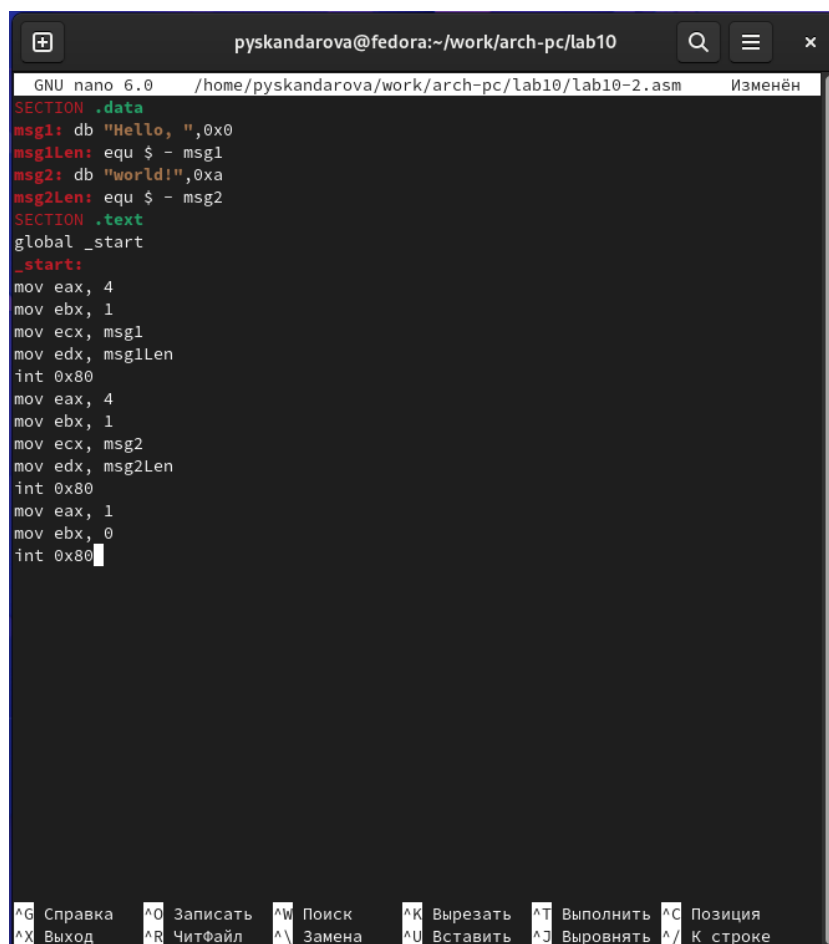
После чего её проверяю. (рис. 2.4)



```
[pyskandarova@fedora lab10]$ nasm -f elf lab10-1.asm
[pyskandarova@fedora lab10]$ ld -m elf_i386 -o lab10-1 lab10-1.o
[pyskandarova@fedora lab10]$ ./lab10-1
Введите x: 1
2(3x-1)+7=11
```

Рис. 2.4: Проверка работы программы

Дальше создаю файл lab10-2.asm с текстом программы из листинга. (Программа печатает сообщения Hello world!). (рис. 2.5)



```
GNU nano 6.0 /home/pyskandarova/work/arch-pc/lab10/lab10-2.asm
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
```

^G Справка ^O Записать ^W Поиск ^K Вырезать ^T Выполнить ^C Позиция
^X Выход ^R ЧитФайл ^\ Замена ^U Вставить ^J Выводить ^/_ К строке

Рис. 2.5: Текст программы в файле

Получаю исполняемый файл. Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию программ необходимо проводить с ключом '-g'. Потом загружаю исполняемый файл в отладчик gdb и проверяю работу программы, запустив ее в оболочке GDB с помощью команды run (сокращённо r).(рис. 2.6)

```
[pyskandarova@fedora lab10]$ nasm -f elf -g -l lab10-2.lst lab10-2.asm
[pyskandarova@fedora lab10]$ ld -m elf_i386 -o lab10-2 lab10-2.o
[pyskandarova@fedora lab10]$ gdb lab10-2
GNU gdb (GDB) Fedora 11.2-3.fc36
Copyright (C) 2022 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-redhat-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 lab10-2...
(gdb) run
Starting program: /home/pyskandarova/work/arch-pc/lab10/lab10-2

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.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.
Downloading separate debug info for /home/pyskandarova/work/arch-pc/lab10/system-
supplied DSO at 0xf7ffc000...
Hello, world!
[Inferior 1 (process 6026) exited normally]
```

Рис. 2.6: Загрузка программы в оболочку GDB и её запуск

Для более подробного анализа программы устанавливаю брейкпоинт на метку `_start`, с которой начинается выполнение любой ассемблерной программы, и запускаю её.(рис. 2.7)

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab10-2.asm, line 9.
(gdb) run
Starting program: /home/pyskandarova/work/arch-pc/lab10/lab10-2

Breakpoint 1, _start () at lab10-2.asm:9
9      mov eax, 4
```

Рис. 2.7: Запуск программы с брейкпоинтом

Теперь смотрю дисассимилированный код программы с помощью команды `disassemble` начиная с метки `_start`. (рис. 2.8)

```

(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.

```

Рис. 2.8: Дисассимилированный код программы

Переключаюсь на отображение команд с Intel'овским синтаксисом, введя команду `set disassembly-flavor intel`. Разница синтаксиса машинных команд в режимах АТТ и Intel в том, что в АТТ перед всеми значениями и адресами стоят символы \$, а перед регистрами - %, также в АТТ во всех командах `mov` аргументы стоят в обратном порядке. (рис. 2.9)

```
(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.
```

Рис. 2.9: Дисассимилированный код программы с Intel'овским синтаксисом

Далее включаю режим псевдографики для более удобного анализа программы. (рис. 2.10)

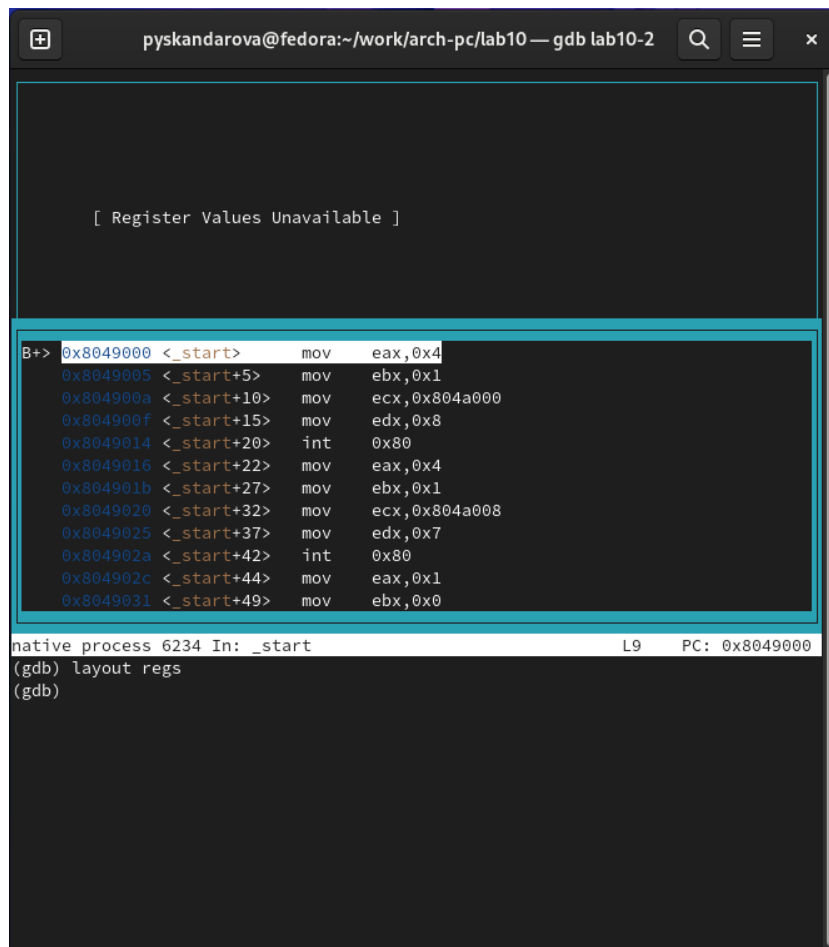


Рис. 2.10: Режим псевдографики gdb

На предыдущих шагах была установлена точка останова по имени метки (`_start`). Проверяю это с помощью команды `info breakpoints` (кратко `i b`). (рис. 2.11)

The screenshot shows a GDB terminal window with the title bar 'pyskandarova@fedora:~/work/arch-pc/lab10 — gdb lab10-2'. The main area displays assembly code for a native process 6631. A table of instructions is shown, with the last instruction highlighted in blue. Below the assembly, the command '(gdb) layout regs' is entered, followed by '(gdb) info breakpoints', which shows a table of breakpoints. The first breakpoint is at address 0x08049000, which is the current PC.

```

[ Register Values Unavailable ]

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

native process 6631 In: _start L9 PC: 0x8049000
(gdb) layout regs
(gdb) info breakpoints
Num   Type             Disp Enb Address      What
1     breakpoint       keep y   0x08049000  lab10-2.asm:9
      breakpoint already hit 1 time
(gdb)

```

Рис. 2.11: Информация о точке останова

Устанавливаю еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции. Определяю адрес предпоследней инструкции (mov ebx,0x0) и устанавливаю точку останова.(рис. 2.12)

The screenshot shows the GDB terminal with the command '(gdb) break *0x8049031' entered, creating a new breakpoint at address 0x8049031. The command '(gdb) i b' is then entered, showing the updated list of breakpoints. The first breakpoint remains at 0x08049000, and a second breakpoint is added at 0x08049031.

```

(gdb) break *0x8049031
Breakpoint 2 at 0x8049031: file lab10-2.asm, line 20.
(gdb) i b
Num   Type             Disp Enb Address      What
1     breakpoint       keep y   0x08049000  lab10-2.asm:9
      breakpoint already hit 1 time
2     breakpoint       keep y   0x08049031  lab10-2.asm:20
(gdb)

```

Рис. 2.12: Новая точка останова и информация о ней

Посмотреть содержимое регистров можно с помощью команды info registers (или i r). (рис. 2.13)

```
(gdb) info registers
eax            0x0            0
ecx            0x0            0
edx            0x0            0
ebx            0x0            0
esp            0xffffd190      0xffffd190
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
es             0x2b           43
fs             0x0            0
gs             0x0            0
```

Рис. 2.13: Результат работы команды info registers

С помощью команды x & можно посмотреть содержимое переменной. Я смотрю значение переменной msg1 по имени. (рис. 2.14)

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

Рис. 2.14: Содержимое переменной msg1

Значение переменной msg2 смотрю по адресу. (рис. 2.15)

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

Рис. 2.15: Содержимое переменной msg1

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). Я изменяю первый символ переменной msg1. (рис. 2.16)

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

Рис. 2.16: Изменение содержимого переменной msg1

Замените любой символ во второй переменной msg2. (рис. 2.17)

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

Рис. 2.17: Изменение содержимого переменной msg2

Вывожу в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра edx. (рис. 2.18)

```
(gdb) p/s $edx
$1 = 7
(gdb) p/i $edx
Format letter "i" is meaningless in "print" command.
(gdb) p/x $edx
$2 = 0x7
(gdb) p/a $edx
$3 = 0x7
```

Рис. 2.18: Значение регистра edx в разных форматах

С помощью команды set изменяю значение регистра ebx. (рис. 2.19)


```
(gdb) p/s $eax
$1 = 0
(gdb) set $ebx='2'
(gdb) p/s $ebx
$2 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$3 = 2
(gdb)
```

Рис. 2.19: Меняющиеся значения регистра ebx

В первый раз в регистре находится символ '2' и программа выводит его код, во второй - значение 2. Завершаю выполнение программы с помощью команды `continue` (сокращенно `c`) или `stepi` (сокращенно `si`) и выхожу из GDB с помощью команды `quit` (сокращенно `q`). (рис. 2.20)

```
(gdb) stepi
(gdb) q
```

Рис. 2.20: Выход из GDB

Копирую файл `lab9-2.asm`, созданный при выполнении лабораторной работы

№9, с программой выводящей на экран аргументы командной строки в файл с именем lab10-3.asm и создаю исполняемый файл. Для загрузки в gdb программы с аргументами необходимо использовать ключ `-args`. Загружаю исполняемый файл в отладчик, указав аргументы. (рис. 2.21)

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

Рис. 2.21: Загрузка файла lab10-3 в GDB

Как отмечалось в предыдущей лабораторной работе, при запуске программы аргументы командной строки загружаются в стек. Исследую расположение аргументов командной строки в стеке после запуска программы с помощью gdb для начала установив точку останова перед первой инструкцией в программе и запустив ее. (рис. 2.22)

```
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab10-3.asm, line 5.
(gdb) run
Starting program: /home/pyskandarova/work/arch-pc/lab10/lab10-3 аргумент1 аргумент 2 аргумент\ 3

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.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:5
5      pop есх ; Извлекаем из стека в 'есх' количество
```

Рис. 2.22: Запуск файла lab10-3 в GDB

Адрес вершины стека храниться в регистре `esp` и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы). Смотрите остальные позиции стека – по адресу `[esp+4]` располагается адрес в памяти где находится имя программы, по адресу `[esp+8]` храниться адрес первого аргумента, по адресу `[esp+12]` – второго и т.д.(рис. 2.23)

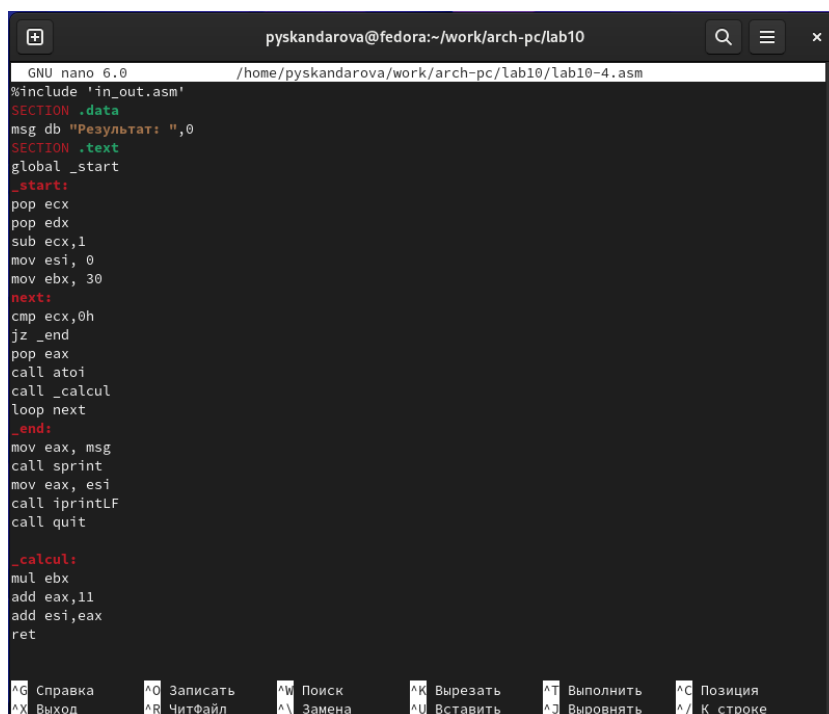
```
(gdb) x/x $esp
0xffffd140:    0x00000005
(gdb) x/s *(void**)(esp + 4)
0xffffd2fd:    "/home/pyskandarova/work/arch-pc/lab10/lab10-3"
(gdb) x/s *(void**)(esp + 8)
0xffffd32b:    "аргумент1"
(gdb) x/s *(void**)(esp + 12)
0xffffd33d:    "аргумент"
(gdb) x/s *(void**)(esp + 16)
0xffffd34e:    "2"
(gdb) x/s *(void**)(esp + 20)
0xffffd350:    "аргумент 3"
(gdb) x/s *(void**)(esp + 24)
0x0:    <error: Cannot access memory at address 0x0>
(gdb)
```

Рис. 2.23: Аргументы программы по их адресам

Шаг равен 4, так как каждый адрес занимает 4 байта.

3 Задание для самостоятельной работы

1. Преобразую программу из лабораторной работы №9 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции $f(x)$ как подпрограмму. (рис. 3.1)

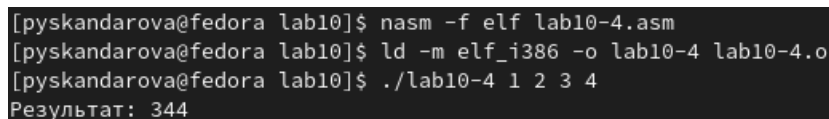


```
GNU nano 6.0 /home/pyskandarova/work/arch-pc/lab10/lab10-4.asm
#include 'in_out.asm'
SECTION .data
msg db "Результат: ",0
SECTION .text
global _start
_start:
pop ecx
pop edx
sub ecx,1
mov esi, 0
mov ebx, 30
next:
cmp ecx,0h
jz _end
pop eax
call atoi
call _calcul
loop next
_end:
mov eax, msg
call sprint
mov eax, esi
call iprintLF
call quit

_calcul:
mul ebx
add eax,11
add esi,eax
ret
```

Рис. 3.1: Изменённая программа в файле

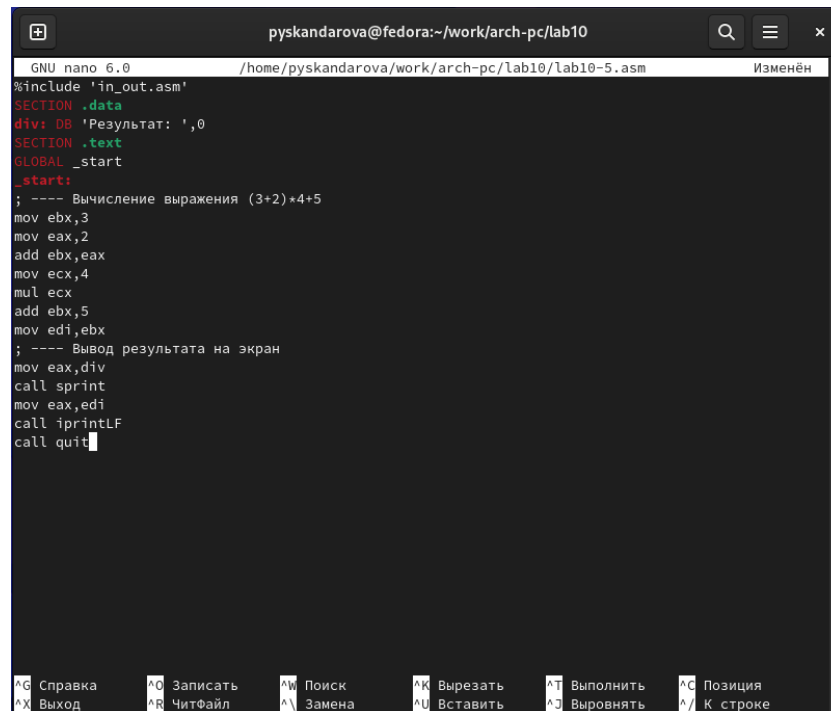
И проверяю её работу. (рис. 3.2)



```
[pyskandarova@fedora lab10]$ nasm -f elf lab10-4.asm
[pyskandarova@fedora lab10]$ ld -m elf_i386 -o lab10-4 lab10-4.o
[pyskandarova@fedora lab10]$ ./lab10-4 1 2 3 4
Результат: 344
```

Рис. 3.2: Проверка работы программы

2. В листинге приведена программа вычисления выражения $(3 + 2) * 4 + 5$. При запуске данная программа дает неверный результат. Проверяю это и с помощью отладчика GDB, анализируя изменения значений регистров, определяю ошибку и исправляю ее. (рис. 3.3)



```
GNU nano 6.0 /home/pyskandarova/work/arch-pc/lab10/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 ebx,eax
mov ecx,4
mul ecx
add ebx,5
mov edi,ebx
; --- Вывод результата на экран
mov eax,div
call sprint
mov eax,edi
call iprintLF
call quit
```

Рис. 3.3: Программа из листинга в файле

Результат сложения оказывается не в том регистре и не участвует в умножении, сложение и вывод происходят с регистром, где умножения не происходило. (рис. 3.4) (рис. 3.5)

```
0x080490f4 in _start ()
(gdb) p/s $eax
$1 = 2
(gdb) p/s $ebx
$2 = 5
(gdb) si
0x080490f9 in _start ()
(gdb) si
0x080490fb in _start ()
(gdb) p/s $eax
$3 = 8
(gdb) p/s $ebx
$4 = 5
```

Рис. 3.4: Значения регистров в процессе выполнения программы

```
0x80490fd <_start+19> add    ebx,0x3
> 0x80490fe <_start+22> mov    edi,ebx
0x8049100 <_start+24> mov    eax,0x804
0x8049105 <_start+29> call   0x804900f
0x804910a <_start+34> mov    eax,edi
0x804910c <_start+36> call   0x8049086
0x8049111 <_start+41> call   0x80490db
0x8049116 add    BYTE PTR
0x8049118 add    BYTE PTR

native process 8146 In: _start
(gdb) p/s $ebx
$5 = 10
(gdb) p/s $eax
$6 = 8
```

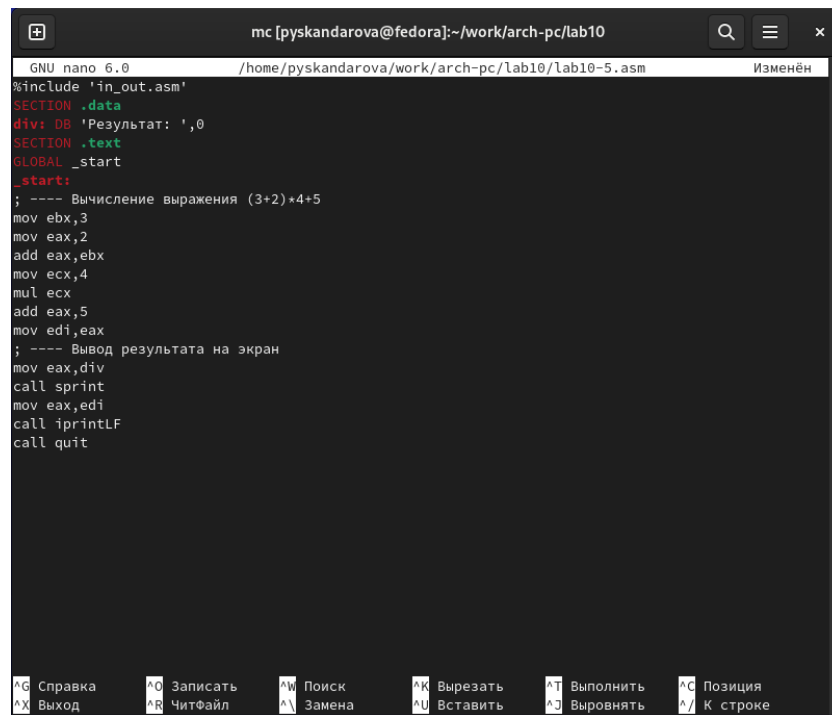
Рис. 3.5: В выводе участвует регистр ebx вместо регистра eax

Исправляю программу.(рис. 3.6)

```
mov edi,ebx
; ---- Вывод
mov eax,div
call sprint
mov eax,edi
```

Рис. 3.6: Исправленная программа в файле

Теперь вывод корректен. (рис. 3.7)



```
GNU nano 6.0 /home/pyskandarova/work/arch-pc/lab10/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
```

^G Справка ^O Записать ^W Поиск ^K Вырезать ^T Выполнить ^C Позиция
^X Выход ^R ЧитФайл ^\ Замена ^U Вставить ^D Выводить ^/_ К строке

Рис. 3.7: Результат выполнения исправленной программы

4 Выводы

В ходе выполнения лабораторной работы приобретены навыки написания программ с использованием подпрограмм. Знакомство с методами отладки при помощи GDB и его основными возможностями прошло успешно.