

Отчёт по лабораторной работе №9

Дисциплина: архитектура компьютеров и операционные системы

Агаджанян Артур НКАбд-01-23

- 1) Цель работы
- 2) Задание
- 3) Теоретическое введение
- 4) Выполнение лабораторной работы
 - 4.1) Реализация подпрограмм в NASM
 - 4.2) Отладка программ с помощью GDB
 - 4.2.1) Добавление точек останова
 - 4.2.2) Работа с данными программы в GDB
 - 4.2.3) Обработка аргументов командной строки в GDB
 - 4.3) Задания для самостоятельной работы **Ошибка! Закладка не определена.**
- 5) Выводы **Ошибка! Закладка не определена.**
- 6) Список литературы

1 Цель работы

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

2 Задание

1. Реализация подпрограмм в NASM.
2. Отладка программ с помощью GDB.
3. Добавление точек останова.
4. Работа с данными программы в GDB.

5. Обработка аргументов командной строки в GDB.
6. Задания для самостоятельной работы.

3 Теоретическое введение

Отладка — это процесс поиска и исправления ошибок в программе. Отладчики позволяют управлять ходом выполнения программы, контролировать и изменять данные. Это помогает быстрее найти место ошибки в программе и ускорить её исправление. Наиболее популярные способы работы с отладчиком — это использование точек останова и выполнение программы по шагам.

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

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

Команда `run` (сокращённо `r`) — запускает отлаживаемую программу в оболочке GDB.

Команда `kill` (сокращённо `k`) прекращает отладку программы, после чего следует вопрос о прекращении процесса отладки. Если в ответ введено `y` (то есть «да»), отладка программы прекращается. Командой `run` её можно начать заново, при этом все точки останова (breakpoints), точки просмотра (watchpoints) и точки отлова (catchpoints) сохраняются.

Для выхода из отладчика используется команда `quit` (или сокращённо `q`).

Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, то программу можно отлаживать, работая в отладчике непосредственно с её исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом `-g`.

Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать как имя метки или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка».

Информацию о всех установленных точках останова можно вывести командой `info` (кратко `i`).

Для того чтобы сделать неактивной какую-нибудь ненужную точку останова, можно воспользоваться командой `disable`.

Архитектура ЭВМ

Обратно точка останова активируется командой `enable`.

Если же точка останова в дальнейшем больше не нужна, она может быть удалена с помощью команды `delete`.

Для продолжения остановленной программы используется команда `continue (c)`. Выполнение программы будет происходить до следующей точки останова. В качестве аргумента может использоваться целое число `N`, которое указывает отладчику проигнорировать `N – 1` точку останова (выполнение остановится на `N`-й точке).

Команда `stepi` (кратко `sl`) позволяет выполнять программу по шагам, т.е. данная команда выполняет ровно одну инструкцию.

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

Для вызова подпрограммы из основной программы используется инструкция `call`, которая заносит адрес следующей инструкции в стек и загружает в регистр `еір` адрес соответствующей подпрограммы, осуществляя таким образом переход. Затем начинается выполнение подпрограммы, которая, в свою очередь, также может содержать подпрограммы. Подпрограмма завершается инструкцией `ret`, которая извлекает из стека адрес, занесённый туда соответствующей инструкцией `call`, и заносит его в `еір`. После этого выполнение основной программы возобновится с инструкции, следующей за инструкцией `call`.

4 Выполнение лабораторной работы

4.1 Реализация подпрограмм в NASM

Создаю каталог для выполнения лабораторной работы № 9, перехожу в него и создаю файл `lab09-1.asm`. (рис. 32)

```
~]$ mkdir ~/work/arch-pc/lab09
~]$ cd ~/work/arch-pc/lab09
lab09]$ touch lab09-1.asm
```

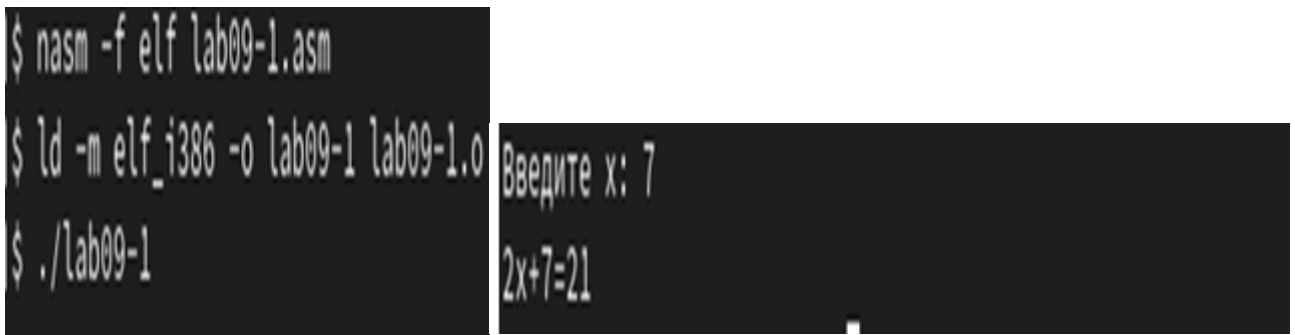
Figure 1: Создание файлов для лабораторной работы

Ввожу в файл lab09-1.asm текст программы с использованием подпрограммы из листинга 9.1. (рис)

```
sazreks@sazreks-System-Product-Name: ~/work/st
lab09-1.asm [----] 0 L: [ 1+ 0 1/ 36] *(0 / 666b)
%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
```

Figure 2: Ввод текста программы из листинга 9.1

Создаю исполняемый файл и проверяю его работу. (рис. 32)

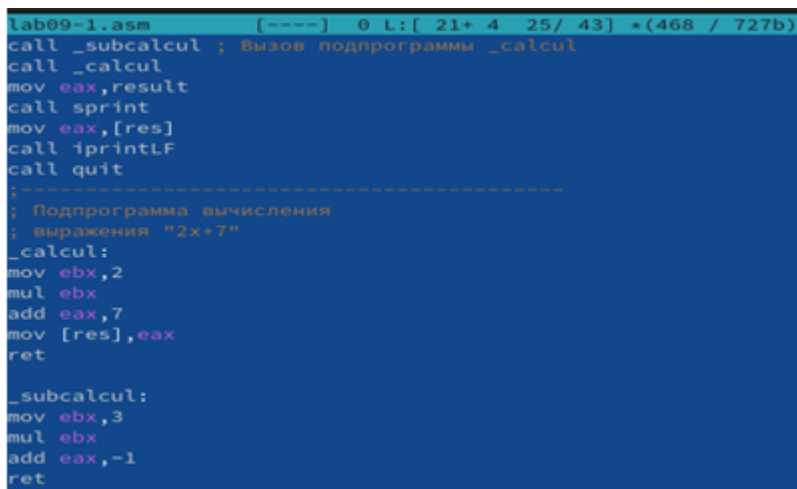


```
$ nasm -f elf lab09-1.asm
$ ld -m elf_i386 -o lab09-1 lab09-1.o
$ ./lab09-1
```

Введите x: 7
2x+7=21

Figure 3: Запуск исполняемого файла

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.)



```
lab09-1.asm [----] 0 L: [ 21+ 4 25/ 43] *(468 / 727b)
call _subcalcul ; Вызов подпрограммы _calcul
call _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
_subcalcul:
mov ebx,3
mul ebx
add eax,-1
ret
```

Figure 4: Изменение текста программы согласно заданию

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



```
$ nasm -f elf lab09-1.asm
$ ld -m elf_i386 -o lab09-1 lab09-1.o
$ ./lab09-1
```

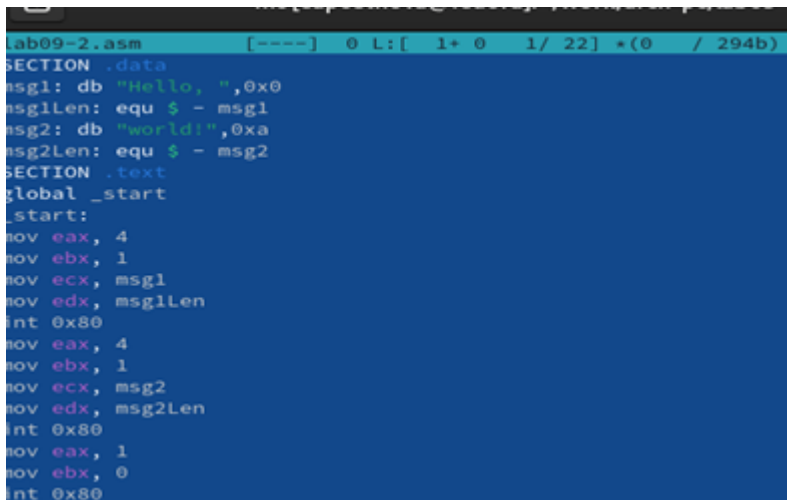
Введите x: 7
2x+7=47

Figure 5: Запуск исполняемого файла

4.2 Отладка программ с помощью GDB

Создаю файл `lab09-2.asm` с текстом программы из Листинга 9.2. (рис)

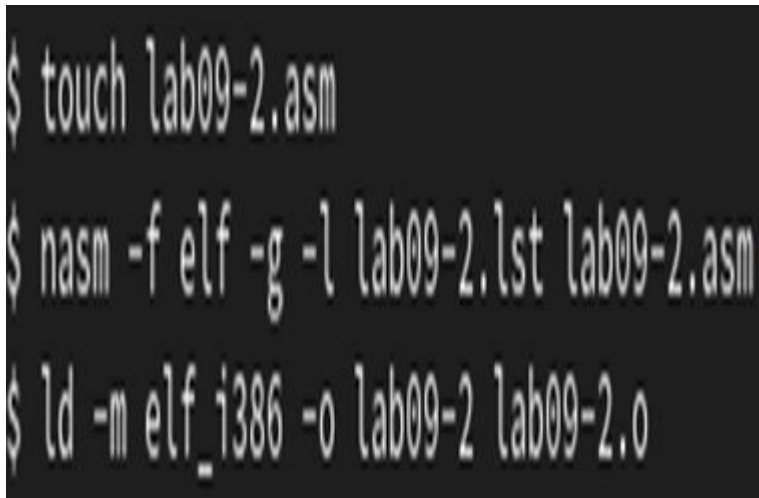
Архитектура ЭВМ



```
lab09-2.asm [----] 0 L: [ 1+ 0 1/ 22] *(0 / 294b)
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
```

Figure 6: Ввод текста программы из листинга 9.2

Получаю исполняемый файл для работы с GDB с ключом '-g'. (рис.)



```
$ touch lab09-2.asm
$ nasm -f elf -g -l lab09-2.lst lab09-2.asm
$ ld -m elf_i386 -o lab09-2 lab09-2.o
```

Figure 7: Получение исполняемого файла

Загружаю исполняемый файл в отладчик gdb. (рис)

Архитектура ЭВМ

```
GNU gdb (GDB) Fedora Linux 13.2-3.fc38
Copyright (C) 2023 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 lab09-2...
```

Figure 8: Загрузка исполняемого файла в отладчик

Проверяю работу программы, запустив ее в оболочке GDB с помощью команды `run`. (рис.)

```
(gdb) run
Starting program: /home/eapostnova/work/arch-pc/lab09/lab09-2
Downloading separate debug info for system-supplied DSO at 0xf7ffc000
Download failed: Нет маршрута до узла. Continuing without separate debug info
for system-supplied DSO at 0xf7ffc000.
Hello, world!
[Inferior 1 (process 6425) exited normally]
```

Figure 9: Проверка работы файла с помощью команды `run`

Для более подробного анализа программы устанавливаю брейкпоинт на метку `_start` и запускаю её. (рис.)

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /home/eapostnova/work/arch-pc/lab09/lab09-2

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

Figure 10: Установка брейкпоинта и запуск программы

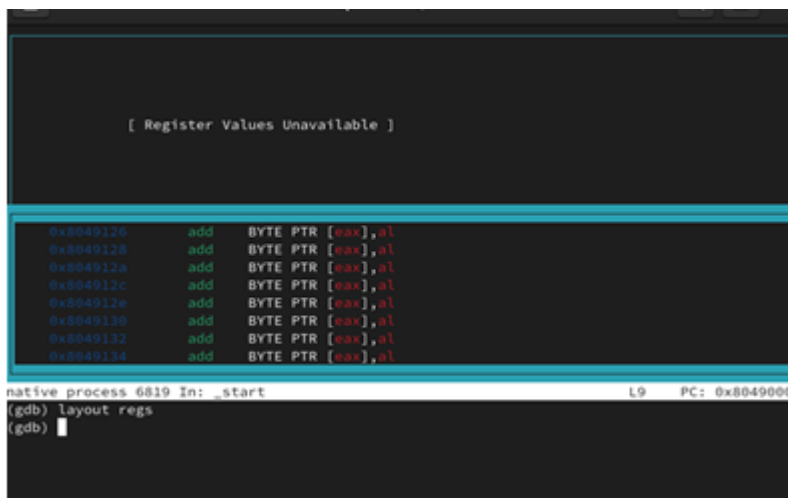
Просматриваю дисассимилированный код программы с помощью команды `disassemble`, начиная с метки `_start`, и переключаюсь на отображение команд с синтаксисом Intel, введя команду `set disassembly-flavor intel`. (рис.)

```
(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) 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.
```

Figure 11: Использование команд `disassemble` и `disassembly-flavor intel`

В режиме АТТ имена регистров начинаются с символа `%`, а имена операндов с `$`, в то время как в Intel используется привычный нам синтаксис.

Включаю режим псевдографики для более удобного анализа программы с помощью команд `layout asm` и `layout regs`. (рис.)



```
[ Register Values Unavailable ]

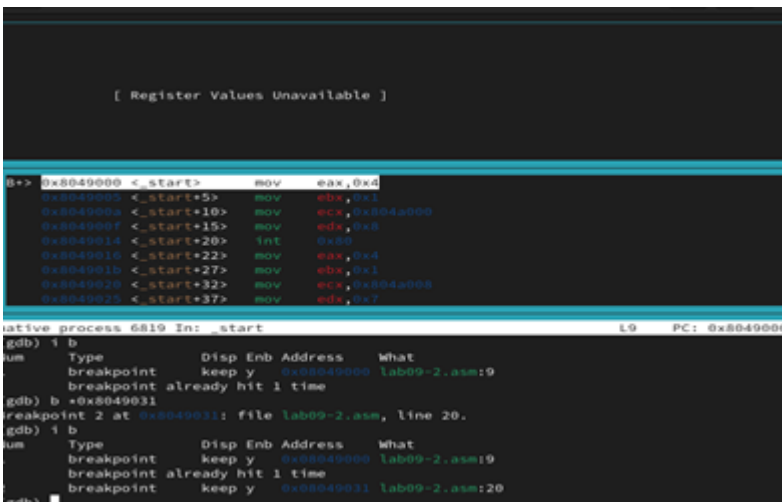
0x8049126 add BYTE PTR [eax],al
0x8049128 add BYTE PTR [eax],al
0x804912a add BYTE PTR [eax],al
0x804912c add BYTE PTR [eax],al
0x804912e add BYTE PTR [eax],al
0x8049130 add BYTE PTR [eax],al
0x8049132 add BYTE PTR [eax],al
0x8049134 add BYTE PTR [eax],al

native process 6819 In: _start L9 PC: 0x8049000
(gdb) layout regs
(gdb)
```

Figure 12: Включение режима псевдографики

4.2.1 Добавление точек останова

Проверяю, что точка останова по имени метки `_start` установлена с помощью команды `info breakpoints` и устанавливаю еще одну точку останова по адресу инструкции `mov ebx,0x0`. Просматриваю информацию о всех установленных точках останова. (рис.13)



```
[ 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,0x0
0x8049014 <_start+20> int 0x0
0x8049016 <_start+22> mov eax,0x4
0x804901b <_start+27> mov ebx,0x1
0x8049020 <_start+32> mov ecx,0x804a000
0x8049025 <_start+37> mov edx,0x7

native process 6819 In: _start L9 PC: 0x8049000
(gdb) i b
um Type Disp Enb Address What
breakpoint keep y 0x8049000 lab09-2.asm:9
breakpoint already hit 1 time
(gdb) b *0x8049031
Breakpoint 2 at 0x8049031: file lab09-2.asm, line 20.
(gdb) i b
um Type Disp Enb Address What
breakpoint keep y 0x8049000 lab09-2.asm:9
breakpoint already hit 1 time
breakpoint keep y 0x8049031 lab09-2.asm:20
(gdb)
```

Figure 13: Установление точек останова и просмотр информации о них

4.2.2 Работа с данными программы в GDB

Выполняю 5 инструкций с помощью команды `stepi` и слежу за изменением значений регистров. (рис.)

Архитектура ЭВМ

```
--Register group: general--
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd160 0xffffd160
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049000 0x8049000 <_start>

B> 0x8049000 <_start> mov    eax,0x4
0x8049005 <_start+5> mov    ebx,0x1
0x804900a <_start+10> mov    ecx,0x804a000
0x804900f <_start+15> mov    edx,0x8
0x8049014 <_start+20> int     0x80
0x8049016 <_start+22> mov    eax,0x4
0x804901b <_start+27> mov    ebx,0x1
0x8049020 <_start+32> mov    ecx,0x804a000

Active process 7430 In: _start L9 PC: 0x8049000
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd160 0xffffd160
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049000 0x8049000 <_start>
eflags   0x202    [ IF ]
--Type <RET> for more, q to quit, c to continue without paging--
```

Figure 14: До использования команды `stepi`

(рис.)

```
Register group: general--
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd160 0xffffd160
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049016 0x8049016 <_start+22>

B> 0x8049000 <_start> mov    eax,0x4
0x8049005 <_start+5> mov    ebx,0x1
0x804900a <_start+10> mov    ecx,0x804a000
0x804900f <_start+15> mov    edx,0x8
0x8049014 <_start+20> int     0x80
> 0x8049016 <_start+22> mov    eax,0x4
0x804901b <_start+27> mov    ebx,0x1
0x8049020 <_start+32> mov    ecx,0x804a000

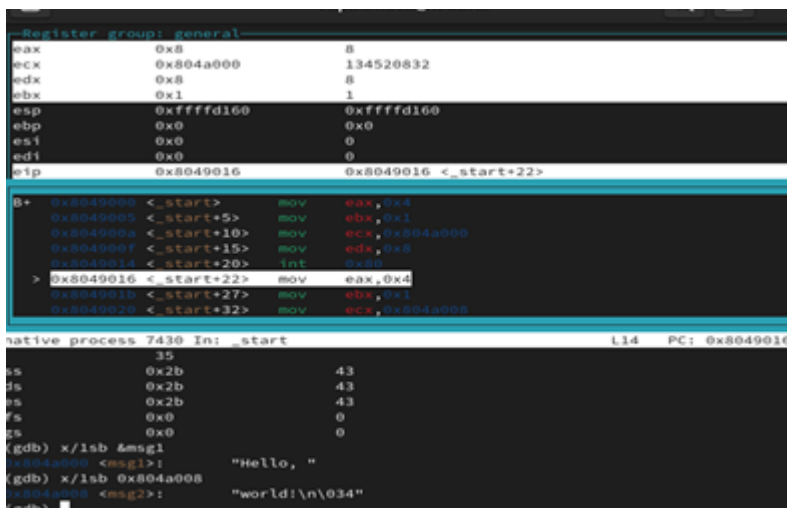
Active process 7430 In: start L14 PC: 0x8049016
p      0x8049000 0x8049000 <_start>
lags   0x202    [ IF ]
Type <RET> for more, q to quit, c to continue without paging--ccs      0x2
35
0x2b      43
0x2b      43
0x2b      43
0x0       0
0x0       0
```

Figure 15: После использования команды `stepi`

Изменились значения регистров `eax`, `ecx`, `edx` и `ebx`.

Просматриваю значение переменной `msg1` по имени с помощью команды `x/1sb &msg1` и значение переменной `msg2` по ее адресу. (рис.)

Архитектура ЭВМ



The screenshot shows a debugger window with two main panes. The top pane displays the state of various CPU registers. The bottom pane shows a list of assembly instructions with their corresponding memory addresses and disassembled code.

Register group: general		
eax	0x0	0
ecx	0x804a000	134520832
edx	0x8	8
ebx	0x1	1
esp	0xffffd160	0xffffd160
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x8049016	0x8049016 <_start+22>

Address	Disassembly
0x8049000 <_start>	mov eax, 0x4
0x8049005 <_start+5>	mov ebx, 0x1
0x804900a <_start+10>	mov ecx, 0x804a000
0x804900f <_start+15>	mov edx, 0x8
0x8049014 <_start+20>	int 0x80
0x8049016 <_start+22>	mov eax, 0x4
0x804901b <_start+27>	mov ebx, 0x1
0x8049020 <_start+32>	mov ecx, 0x804a000

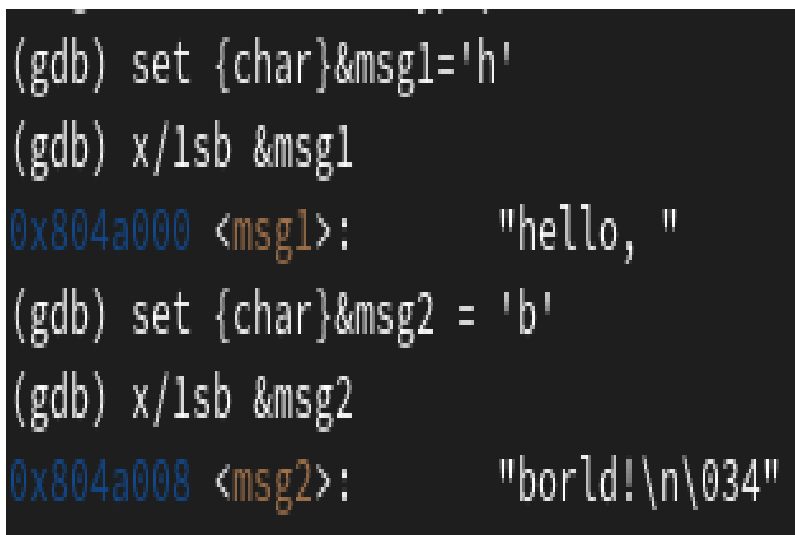
Native process 7430 in: _start L14 PC: 0x8049016

35
%s 0x2b 43
%s 0x2b 43
%s 0x2b 43
%s 0x0 0
%s 0x0 0

(gdb) x/1sb &msg1
0x804a000 <msg1>: "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>: "world!\n\034"

Figure 16: Просмотр значений переменных

С помощью команды set изменяю первый символ переменной msg1 и заменяю первый символ в переменной msg2. (рис.)



The screenshot shows a series of commands entered in a debugger's command window and the resulting output. The first command changes the first character of msg1 to 'h'. The second command displays the current value of msg1. The third command changes the first character of msg2 to 'b'. The fourth command displays the current value of msg2.

```
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>: "hello, "
(gdb) set {char}&msg2 = 'b'
(gdb) x/1sb &msg2
0x804a008 <msg2>: "borld!\n\034"
```

Figure 17: Использование команды set

Вывожу в шестнадцатеричном формате, в двоичном формате и в символьном виде соответственно значение регистра edx с помощью команды print p/F \$val. (рис.)

Архитектура ЭВМ

```
Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd160 0xffffd160
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049016 0x8049016 <_start+22>
eflags   0x202    [ IF ]

0x8049000 <_start>    mov     eax,0x4
0x8049005 <_start+5>  mov     ebx,0x1
0x804900a <_start+10> mov     ecx,0x804a000
0x804900f <_start+15> mov     edx,0x8
0x8049014 <_start+20> int     0x80
> 0x8049016 <_start+22> mov     eax,0x4
0x804901b <_start+27> mov     ebx,0x1
0x8049020 <_start+32> mov     ecx,0x804a000
0x8049025 <_start+37> mov     edx,0x7
0x804902a <_start+42> int     0x80

native process 7430 In: _start L14 PC: 0x8049016
(gdb) p/x $edx
$ = 0x8
(gdb) p/t $edx
9 = 1000
(gdb) p/c $edx
10 = 8 '\b'
```

Figure 18: Вывод значения регистра в разных представлениях

С помощью команды set изменяю значение регистра ebx в соответствии с заданием. (рис.)

```
Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x2      2
esp      0xffffd160 0xffffd160
ebp      0x0      0x0

0x804900a <_start+10> mov     ecx,0x804a000
0x804900f <_start+15> mov     edx,0x8
0x8049014 <_start+20> int     0x80
> 0x8049016 <_start+22> mov     eax,0x4
0x804901b <_start+27> mov     ebx,0x1
0x8049020 <_start+32> mov     ecx,0x804a000

native process 19345 In: _start L14 PC: 0x8049016
(gdb) set $ebx='2'
(gdb) p/s $ebx
$1 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$2 = 2
(gdb)
```

Figure 19: Использование команды set для изменения значения регистра

Разница вывода команд p/s \$ebx отличается тем, что в первом случае мы переводим символ в его строковый вид, а во втором случае число в строковом виде не изменяется.

Завершаю выполнение программы с помощью команды continue и выхожу из GDB с помощью команды quit. (рис.)

Архитектура ЭВМ

```
Register group: general
rax      0x1      1
rcx      0x804a008 134520840
rdx      0x7      7
rbx      0x1      1
rsp      0xffffd160 0xffffd160
rbp      0x0      0x0
rsi      0x0      0
rdi      0x0      0
rip      0x8049031 0x8049031 <_start+49>

0x8049020 <_start+32> mov     ecx,0x804a008
0x8049025 <_start+37> mov     edx,0x7
0x804902a <_start+42> int     0x80
0x804902e <_start+44> mov     eax,0x1
0x8049031 <_start+49> mov     ebx,0x0
0x8049035 <_start+54> int     0x80
0x8049038 add     BYTE PTR [eax],al
0x804903a add     BYTE PTR [eax],al

Active process 7430 In: _start L20 PC: 0x804903
4 = 2
(gdb) c
Continuing.
Breakpoint 2, _start () at lab09-2.asm:20
(gdb) q
debugging session is active.

Inferior 1 [process 7430] will be killed.
```

Figure 20: Завершение работы GDB

4.2.3 Обработка аргументов командной строки в GDB

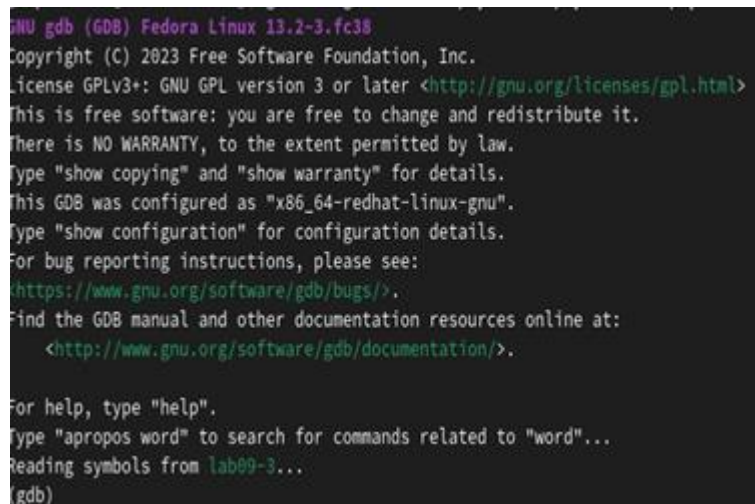
Копирую файл lab8-2.asm с программой из листинга 8.2 в файл с именем lab09-3.asm и создаю исполняемый файл. (рис.)

```
$ cp ~/work/arch-pc/lab08/lab8-2.asm ~/work/arch-pc/lab09/lab09-3.asm
$ nasm -f elf -g -l lab09-3.lst lab09-3.asm
$ ld -m elf_i386 -o lab09-3 lab09-3.o
```

Figure 21: Создание файла

Загружаю исполняемый файл в отладчик gdb, указывая необходимые аргументы с использованием ключа -args. (рис.)

Архитектура ЭВМ

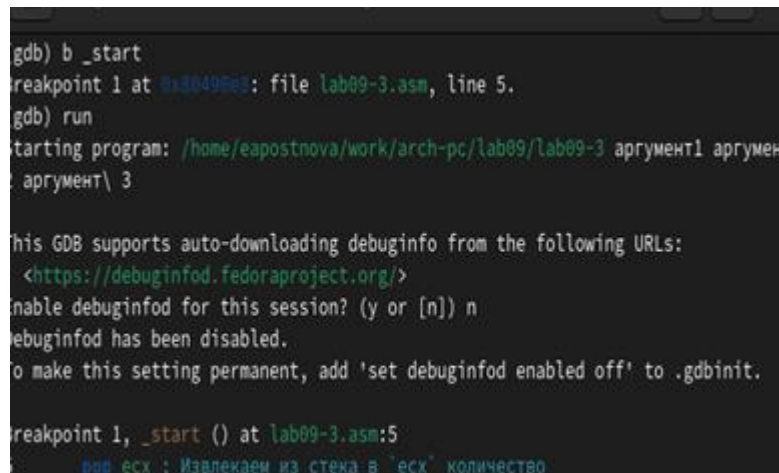
A screenshot of the GNU Debugger (GDB) startup screen. The text is displayed in a monospaced font on a dark background. It includes the version number 'GNU gdb (GDB) Fedora Linux 13.2-3.fc38', copyright information for the Free Software Foundation, Inc. (2023), and the GNU GPL license. It also provides links to the GDB website for bug reporting and documentation, and instructions on how to use the 'help' and 'apropos' commands. The prompt '(gdb)' is visible at the bottom.

```
GNU gdb (GDB) Fedora Linux 13.2-3.fc38
Copyright (C) 2023 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 lab09-3...
(gdb)
```

Figure 22: Загрузка файла с аргументами в отладчик

Устанавливаю точку останова перед первой инструкцией в программе и запускаю ее. (рис.)

A screenshot of the GDB terminal showing the process of setting a breakpoint and running a program. The user enters 'b _start' to set a breakpoint at the start of the program. Then, they enter 'run' to execute the program. The output shows the program's path and arguments. Below this, there is a message about debuginfo and a confirmation to disable it. Finally, the breakpoint is reached, and the program's entry point is shown as '_start () at lab09-3.asm:5'. The prompt '(gdb)' is visible at the bottom.

```
(gdb) b _start
Breakpoint 1 at 0x30496e3: file lab09-3.asm, line 5.
(gdb) run
Starting program: /home/eaostnova/work/arch-pc/lab09/lab09-3 аргумент1 аргумент2 аргумент3

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) n
debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.

Breakpoint 1, _start () at lab09-3.asm:5
    pop ecx ; Извлекаем из стека в 'ecx' количество
(gdb)
```

Figure 23: Установление точки останова и запуск программы

Посматриваю вершину стека и позиции стека по их адресам. (рис.)

```
(gdb) x/x $esp
0xffffd120: 0x00000005
(gdb) x/s *(void**)($esp + 4)
0xffffd2e2: "/home/eaipostnova/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)($esp + 8)
0xffffd30e: "аргумент1"
(gdb) x/s *(void**)($esp + 12)
0xffffd320: "аргумент"
(gdb) x/s *(void**)($esp + 16)
0xffffd331: "2"
(gdb) x/s *(void**)($esp + 20)
0xffffd333: "аргумент 3"
(gdb) x/s *(void**)($esp + 24)
0x0: <error: Cannot access memory at address 0x0>
```

Figure 24: Просмотр значений, введенных в стек

Шаг изменения адреса равен 4, т.к количество аргументов командной строки равно 4.

4.3 Задания для самостоятельной работы

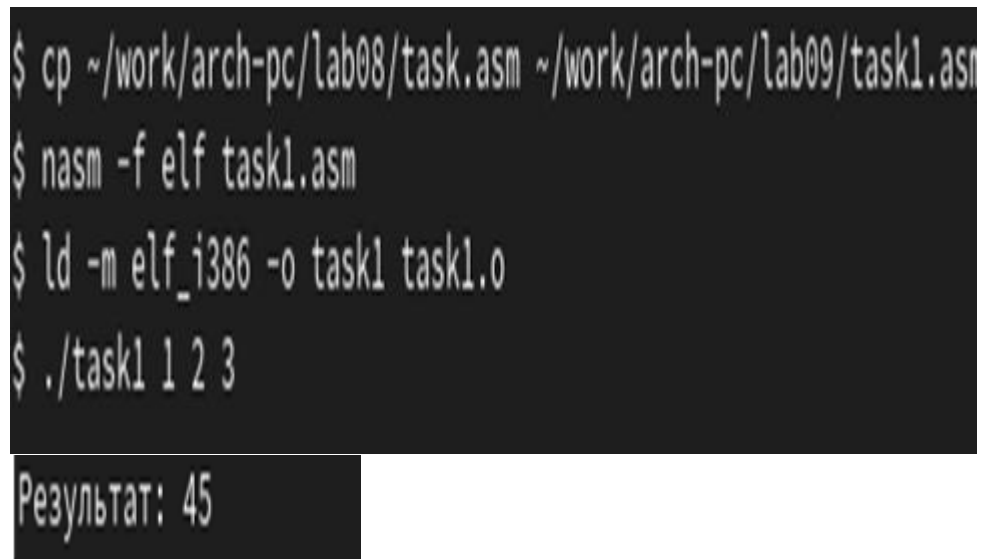
1. Преобразовываю программу из лабораторной работы №8 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции $f(x)$ как подпрограмму. (рис)

```
task1.asm [----] 0 L: [ 14+ 7 21/ 31] *(229 / 323b) 0106 0x06A
.next:
pop eax
call atoi
add eax,2
mul edi
add esi,eax
cmp ecx,0h
jz .done
loop .next

.done:
mov eax,msg
call sprint
mov eax,esi
call iprintf
call quit
ret
```

Figure 25: Написание кода подпрограммы

Запускаю код и проверяю, что она работает корректно. (рис.)



A terminal window with a dark background and light green text. It shows the following commands and their outputs:

```
$ cp ~/work/arch-pc/lab08/task.asm ~/work/arch-pc/lab09/task1.asm
$ nasm -f elf task1.asm
$ ld -m elf_i386 -o task1 task1.o
$ ./task1 1 2 3
```

Below the terminal window, a separate box displays the output of the program:

```
Результат: 45
```

Figure 26: Запуск программы и проверка его вывода

Код программы:

```
%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 edi,5
call .next

.next:
pop eax
call atoi
add eax,2
mul edi
add esi,eax
```


Архитектура ЭВМ

```
cmp ecx,0h
```

```
jz .done
```

```
loop .next
```

```
.done:
```

```
mov eax, msg
```

```
call sprint
```

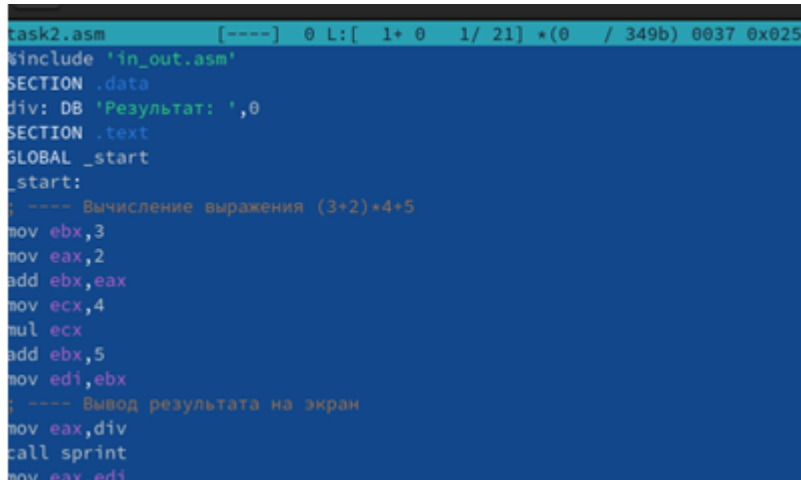
```
mov eax, esi
```

```
call iprintLF
```

```
call quit
```

```
ret
```

2. Ввожу в файл task1.asm текст программы из листинга 9.3. (рис.)



```
task2.asm [----] 0 L:[ 1+ 0 1/ 21] *(0 / 349b) 0037 0x025
#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
```

Figure 27: Ввод текста программы из листинга 9.3

При корректной работе программы должно выводиться “25”. Создаю исполняемый файл и запускаю его. (рис.)

Архитектура ЭВМ

```
lab09]$ touch task2.asm
lab09]$ nasm -f elf task2.asm
lab09]$ ld -m elf_i386 -o task2 task2.o
lab09]$ ./task2
```

Результат: 10

Figure 28: Создание и запуск исполняемого файла

Видим, что в выводе мы получаем неправильный ответ.

Получаю исполняемый файл для работы с GDB, запускаю его и ставлю брейкпоинты для каждой инструкции, связанной с вычислениями. С помощью команды `continue` прохожусь по каждому брейкпоинту и слежу за изменениями значений регистров.

При выполнении инструкции `mul ecx` происходит умножение `ecx` на `eax`, то есть 4 на 2, вместо умножения 4 на 5 (регистр `ebx`). Происходит это из-за того, что стоящая перед `mov ecx,4` инструкция `add ebx,eax` не связана с `mul ecx`, но связана инструкция `mov eax,2`. (рис.)

```
--Register group: general--
eax      0x8      8
ecx      0x4      4
edx      0x0      0
ebx      0x5      5
esp      0xffffd160 0xffffd160
ebp      0x0      0x0

B* 0x80490f4 <_start+12> mov    ecx,4
B* 0x80490f9 <_start+17> mul    ecx
B* 0x80490fb <_start+19> add    ebx,eax
b* 0x80490fe <_start+22> mov    edi,ebx
b* 0x8049100 <_start+24> mov    eax,0x504a000

native process 14573 In: _start L13 PC: 0x80490f
Continuing.

Breakpoint 5, _start () at task2.asm:12
(gdb) c
Continuing.

Breakpoint 6, _start () at task2.asm:13
(gdb)
```

Figure 29: Нахождение причины ошибки

Из-за этого мы получаем неправильный ответ. (рис.)

Архитектура ЭВМ

```
Register group: general
eax      0x8      8
ecx      0x4      4
edx      0x0      0
ebx      0xa      10
esp      0xffffd160 0xffffd160
ebp      0x0      0x0

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

native process 14573 In: _start L14 PC: 0x80490fe
Continuing.

Breakpoint 6, _start () at task2.asm:13
(gdb) c
Continuing.

Breakpoint 7, _start () at task2.asm:14
(gdb)
```

Figure 30: Неверное изменение регистра

Исправляем ошибку, добавляя после add ebx,eax mov eax,ebx и заменяя ebx на eax в инструкциях add ebx,5 и mov edi,ebx. (рис.)

```
task2.asm [-M--] 11 L: [ 1+14 15/ 22] *(245 / 361b) 0010 0x00
#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 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 iprintfLF
call quit
```

Figure 31: Исправление ошибки

Также, вместо того, чтобы изменять значение eax, можно было изменять значение неиспользованного регистра edx.

Создаем исполняемый файл и запускаем его. Убеждаемся, что ошибка исправлена. (рис.)

```
.ab09]$ nasm -f elf task2.asm  
.ab09]$ ld -m elf_i386 -o task2 task2.  
.ab09]$  
.ab09]$ ./task2
```

```
Результат: 25
```

Figure 32: Ошибка исправлена

Код программы:

```
%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 eax,ebx  
mov ecx,4  
mul ecx  
add eax,5  
mov edi,eax  
; --- Вывод результата на экран  
mov eax,div
```

Архитектура ЭВМ

call sprint

mov eax,edi

call iprintLF

call quit

5 Выводы

Во время выполнения данной лабораторной работы я приобрела навыки написания программ с использованием подпрограмм и ознакомилась с методами отладки при помощи GDB и его основными возможностями.

6 Список литературы

1. GDB: The GNU Project Debugger. — URL: <https://www.gnu.org/software/gdb/>.
2. GNU Bash Manual. — 2016. — URL: <https://www.gnu.org/software/bash/manual/>.
3. Midnight Commander Development Center. — 2021. — URL: <https://midnight-commander.org/>.
4. NASM Assembly Language Tutorials. — 2021. — URL: <https://asmtutor.com/>.
5. Newham C. Learning the bash Shell: Unix Shell Programming. — O'Reilly Media, 2005 — 354 с. — (In a Nutshell). — ISBN 0596009658. — URL: <http://www.amazon.com/Learningbash-Shell-Programming-Nutshell/dp/0596009658>.
6. Robbins A. Bash Pocket Reference. — O'Reilly Media, 2016. — 156 с. — ISBN 978-1491941591.
7. The NASM documentation. — 2021. — URL: <https://www.nasm.us/docs.php>.
8. Zarrelli G. Mastering Bash. — Packt Publishing, 2017. — 502 с. — ISBN 9781784396879.
9. Колдаев В. Д., Лупин С. А. Архитектура ЭВМ. — М. : Форум, 2018.
10. Куляс О. Л., Никитин К. А. Курс программирования на ASSEMBLER. — М. : Солон-Пресс, 2017.
11. Новожилов О. П. Архитектура ЭВМ и систем. — М. : Юрайт, 2016.
12. Расширенный ассемблер: NASM. — 2021. — URL: <https://www.opennet.ru/docs/RUS/nasm/>.
13. Робачевский А., Немнюгин С., Стесик О. Операционная система UNIX. — 2-е изд. — БХВПетербург, 2010. — 656 с. — ISBN 978-5-94157-538-1.
14. Столяров А. Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011. — URL: http://www.stolyarov.info/books/asm_unix.
15. Таненбаум Э. Архитектура компьютера. — 6-е изд. — СПб. : Питер, 2013. — 874 с. — (Классика Computer Science).
16. Таненбаум Э., Бос Х. Современные операционные системы. — 4-е изд. — СПб. : Питер, 2015. — 1120 с. — (Классика Computer Science).