

# **Отчёт по лабораторной работе №9**

**Дисциплина: Архитектура компьютера**

Абрамова Ульяна Михайловна

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>5</b>
<b>2</b>	<b>Задание</b>	<b>6</b>
<b>3</b>	<b>Теоретическое введение</b>	<b>7</b>
3.1	Понятие об отладке . . . . .	7
3.2	Методы отладки . . . . .	8
3.3	Основные возможности отладчика GDB . . . . .	8
3.3.1	Запуск отладчика GDB; выполнение программы; выход . . .	9
3.3.2	Дизассемблирование программы . . . . .	10
3.3.3	Точки останова . . . . .	10
3.3.4	Пошаговая отладка . . . . .	11
3.3.5	Работа с данными программы в GDB . . . . .	11
3.3.6	Понятие подпрограммы . . . . .	12
<b>4</b>	<b>Выполнение лабораторной работы</b>	<b>14</b>
4.1	Реализация подпрограмм в NASM . . . . .	14
4.2	Отладка программ с помощью GDB . . . . .	17
4.2.1	Добавление точек останова . . . . .	20
4.2.2	Работа с данными программы в GDB . . . . .	21
4.2.3	Обработка аргументов командной строки в GDB . . . . .	23
4.3	Выполнение заданий для самостоятельной работы . . . . .	24
4.3.1	1 . . . . .	24
4.3.2	2 . . . . .	26
<b>5</b>	<b>Вывод</b>	<b>29</b>
<b>6</b>	<b>Список литературы</b>	<b>30</b>

## Список иллюстраций

4.1	Создание каталога и файла . . . . .	14
4.2	Написание программы . . . . .	15
4.3	Запуск программы . . . . .	15
4.4	Изменения программы . . . . .	16
4.5	Запуск программы . . . . .	17
4.6	Написание программы . . . . .	17
4.7	Запуск программы . . . . .	18
4.8	Установка брейкпоинта . . . . .	18
4.9	Дисассимилированный код программы . . . . .	19
4.10	Отображение команд с Intel'овским синтаксисом . . . . .	19
4.11	Проверка точки останова . . . . .	20
4.12	Установка точки останова . . . . .	21
4.13	Информация о всех точках останова . . . . .	21
4.14	Содержание регистров . . . . .	21
4.15	Содержимое переменной msg1 . . . . .	21
4.16	Содержимое переменной msg2 . . . . .	22
4.17	Изменение первого символа . . . . .	22
4.18	Изменение символа . . . . .	22
4.19	Изменение регистра . . . . .	22
4.20	Завершение и выход . . . . .	23
4.21	Создание исполняемого файла . . . . .	23
4.22	Установка точки останова и запуск . . . . .	23
4.23	Проверка адреса вершины стека . . . . .	24
4.24	Проверка остальных позиций стека . . . . .	24
4.25	Преобразование программы . . . . .	25
4.26	Запуск программы . . . . .	25
4.27	Запуск программы . . . . .	26
4.28	Анализ изменения значений регистров . . . . .	27
4.29	Запуск исправленной команды . . . . .	28

## **Список таблиц**

# 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. Выполнение заданий для самостоятельной работы

## 3 Теоретическое введение

### 3.1 Понятие об отладке

*Отладка* — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа: • обнаружение ошибки; • поиск её местонахождения; • определение причины ошибки; • исправление ошибки. Можно выделить следующие типы ошибок: • синтаксические ошибки — обнаруживаются во время трансляции исходного кода и вызваны нарушением ожидаемой формы или структуры языка; • семантические ошибки — являются логическими и приводят к тому, что программа запускается, отрабатывает, но не даёт желаемого результата; • ошибки в процессе выполнения — не обнаруживаются при трансляции и вызывают прерывание выполнения программы (например, это ошибки, связанные с переполнением или делением на ноль). Второй этап — поиск местонахождения ошибки. Некоторые ошибки обнаружить довольно трудно. Лучший способ найти место в программе, где находится ошибка, это разбить программу на части и произвести их отладку отдельно друг от друга. Третий этап — выяснение причины ошибки. После определения местонахождения ошибки обычно проще определить причину неправильной работы программы. Последний этап — исправление ошибки. После этого при повторном запуске программы, может обнаружиться следующая ошибка, и процесс отладки начнётся заново.

## 3.2 Методы отладки

Наиболее часто применяют следующие методы отладки: • создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных значений на экран — так называемые диагностические сообщения); • использование специальных программ-отладчиков. 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. *Пошаговое выполнение* — это выполнение программы с остановкой после каждой строчки, чтобы программист мог проверить значения переменных и выполнить другие действия. *Точки останова* — это специально отмеченные места в программе, в которых программа-отладчик приостанавливает выполнение программы и ждёт команд. Наиболее популярные виды точек останова: • Breakpoint — точка останова (остановка происходит, когда выполнение доходит до определённой строки, адреса или процедуры, отмеченной программистом); • Watchpoint — точка просмотра (выполнение программы приостанавливается, если программа обратилась к определённой переменной: либо считала её значение, либо изменила его). Точки останова устанавливаются в отладчике на время сеанса работы с кодом программы, т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы.

## 3.3 Основные возможности отладчика GDB

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. GDB может выполнять следующие действия: • начать выполнение программы, задав всё, что может повлиять на её поведение; • остановить программу при указанных условиях; • исследовать, что случилось, когда программа остановилась; • изменить программу так, чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

### 3.3.1 Запуск отладчика GDB; выполнение программы; выход

Синтаксис команды для запуска отладчика имеет следующий вид: `gdb [опции] [имя_файла | ID процесса]` После запуска `gdb` выводит текстовое сообщение — так называемое «nice GDB logo». В следующей строке появляется приглашение (`gdb`) для ввода команд. Далее приведён список некоторых команд GDB. Команда `run` (сокращённо `r`) — запускает отлаживаемую программу в оболочке GDB. Если точки останова не были установлены, то программа выполняется и выводятся сообщения: `(gdb) run Starting program: test Program exited normally.` `(gdb)` Если точки останова были заданы, то отладчик останавливается на соответствующей команде и выдаёт номер точки останова, адрес и дополнительную информацию — текущую строку, имя процедуры, и др. Команда `kill` (сокращённо `k`) прекращает отладку программы, после чего следует вопрос о прекращении процесса отладки: `Kill the program being debugged? (y or n)` `y` 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`): `(gdb) q`

### 3.3.2 Дизассемблирование программы

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`. Посмотреть дизассемблированный код программы можно с помощью команды `disassemble`: `(gdb) disassemble _start` Существует два режима отображения синтаксиса машинных команд: режим Intel, используемый в том числе в NASM, и режим ATT (значительно отличающийся внешне). По умолчанию в дизассемблере GDB принят режим ATT. Переключиться на отображение команд с привычным Intel'овским синтаксисом можно, введя команду `set disassembly-flavor intel`

### 3.3.3 Точки останова

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»: `(gdb) break *<адрес>` `(gdb) b <метка>` Информацию о всех установленных точках останова можно вывести командой `info` (кратко `i`): `(gdb) info breakpoints` `(gdb) i b` Для того чтобы сделать неактивной какую-нибудь ненужную точку останова, можно воспользоваться командой `disable`: `disable breakpoint <номер точки останова>` Обратная точка останова активируется командой `enable`: `enable breakpoint <номер точки останова>` Если же точка останова в дальнейшем больше не нужна, она может быть удалена с помощью команды `delete`: `(gdb) delete breakpoint <номер точки останова>` Ввод этой команды без аргумента удалит все точки останова. Информацию о командах этого раздела можно получить, введя `help breakpoints`

### 3.3.4 Пошаговая отладка

Для продолжения остановленной программы используется команда `continue` (с) (gdb) с [аргумент]. 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 `si`) позволяет выполнять программу по шагам, т.е. данная команда выполняет ровно одну инструкцию: (gdb) `si` [аргумент] При указании в качестве аргумента целого числа  $N$  отладчик выполнит команду `step`  $N$  раз при условии, что не будет точек останова или выполнение программы не прервётся по другим причинам. Команда `nexthi` (или `ni`) аналогична `stepi`, но вызов процедуры (функции) трактуется отладчиком как одна инструкция: (gdb) `ni` [аргумент] Информацию о командах этого раздела можно получить, введя (gdb) `help running`

### 3.3.5 Работа с данными программы в GDB

Как уже упоминалось, отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Посмотреть содержимое регистров можно с помощью команды `info registers` (или `i r`): (gdb) `info registers` Для отображения содержимого памяти можно использовать команду `x/NFU`, выдаёт содержимое ячейки памяти по указанному адресу. `NFU` задает формат, в котором выводятся данные. Например, `x/4uh 0x63450` — это запрос на вывод четырёх полуслов (`h`) из памяти в формате беззнаковых десятичных целых (`u`), начиная с адреса `0x63450`. Чтобы посмотреть значения регистров используется команда `print /F` (сокращенно `p`). Перед именем регистра обязательно ставится префикс `$`. Например, команда `p/x $ecx` выводит значение регистра в шестнадцатеричном формате. 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). Справку о любой команде gdb можно получить, введя `(gdb) help [имя_команды]`

### 3.3.6 Понятие подпрограммы

*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.

#### 3.3.6.1 Инструкция call и инструкция ret

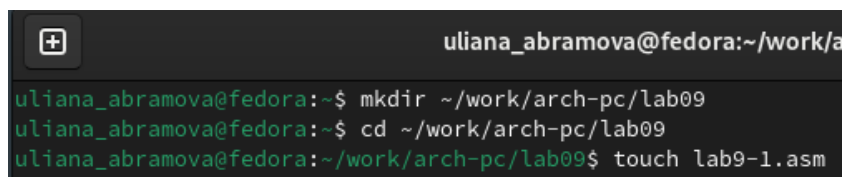
Для вызова подпрограммы из основной программы используется инструкция `call`, которая заносит адрес следующей инструкции в стек и загружает в регистр `esp` адрес соответствующей подпрограммы, осуществляя таким образом переход. Затем начинается выполнение подпрограммы, которая, в свою очередь, также может содержать подпрограммы. Подпрограмма завершается инструкцией `ret`, которая извлекает из стека адрес, занесённый туда соответствующей инструкцией `call`, и заносит его в `esp`. После этого выполнение основной программы возобновится с инструкции, следующей за инструкцией `call`. Подпрограмма может вызываться как из внешнего файла, так и быть частью основной программы. Важно помнить, что если в подпрограмме занести что-то в стек и не извлечь, то на вершине стека окажется не адрес возврата и это приведёт к ошибке выхода из подпрограммы. Кроме того, надо помнить, что подпрограмма без команды возврата не вернётся в точку вызова, а будет выполнять следующий за подпро-

граммой код, как будто он является её продолжением.

## 4 Выполнение лабораторной работы

### 4.1 Реализация подпрограмм в NASM

Создаю каталог для выполнения лабораторной работы №9, перейдя в него, создаю файл lab9-1.asm (рис. 4.1).

A terminal window with a dark background. The title bar shows a window icon and the text 'uliana\_abramova@fedora:~/work/a'. The terminal contains three lines of text: the first line shows the user 'uliana\_abramova@fedora' at the prompt '~\$' running the command 'mkdir ~/work/arch-pc/lab09'; the second line shows the user at the prompt '~\$' running the command 'cd ~/work/arch-pc/lab09'; the third line shows the user at the prompt '~/work/arch-pc/lab09\$' running the command 'touch lab9-1.asm'.

```
uliana_abramova@fedora:~$ mkdir ~/work/arch-pc/lab09
uliana_abramova@fedora:~$ cd ~/work/arch-pc/lab09
uliana_abramova@fedora:~/work/arch-pc/lab09$ touch lab9-1.asm
```

Рис. 4.1: Создание каталога и файла

В созданный файл ввожу программу с вызовом подпрограммы (рис. 4.2).

```
uliana_abramova@fedora:~/work/arch-pc/lab09$ nano lab9-1.asm
GNU nano 7.2 lab9-1.asm
#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:

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:
```

Рис. 4.2: Написание программы

Создаю исполняемый файл и проверяю его работу (рис. 4.3).

```
uliana_abramova@fedora:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
uliana_abramova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
uliana_abramova@fedora:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 5
2x+7=17
```

Рис. 4.3: Запуск программы

Изменяю текст программы, добавив подпрограмму `_subcalcul` в подпрограмму `_calcul` для вычисления выражения  $f(g(x))$  (рис. 4.4).

```
uliana_abramova@fedora:~/work/arch-pc/
GNU nano 7.2 lab9-1.asm
#include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
prim1: DB '2x+7=',0
prim2: DB '3x-1=',0
result: DB '2(3x-1)+7=',0
SECTION .bss
x: RESB 80
res: RESB 80
SECTION .text
GLOBAL _start
_start:

mov eax, prim1

mov eax, prim2

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
```

Рис. 4.4: Изменения программы

Создаю исполняемый файл и проверяю его работу (рис. 4.5).



```

uliana_abramova@fedora:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
uliana_abramova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
uliana_abramova@fedora:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 5
2(3x-1)+7=35

```

Рис. 4.5: Запуск программы

## 4.2 Отладка программ с помощью GDB

Создаю файл lab9-2.asm с текстом программы вывода сообщения Hello world!  
(рис. 4.6)

```

uliana_abramova@fedora:~/work/arch-pc/lab09
GNU nano 7.2 lab9-2.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

```

Рис. 4.6: Написание программы

Получаю исполняемый файл и загружаю его в отладчик gdb, а также проверяю работу программы, запустив ее в оболочке gdb с помощью команды run (сокращенно r) (рис. 4.7)

```

uliana_abramova@fedora:~/work/arch-pc/lab09$ nasm -f elf -g -l lab9-2.lst lab9-2.asm
uliana_abramova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-2 lab9-2.o
uliana_abramova@fedora:~/work/arch-pc/lab09$ gdb lab9-2
GNU gdb (Fedora Linux) 15.2-3.fc40
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.
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 lab9-2...
(gdb) run
Starting program: /home/uliana_abramova/work/arch-pc/lab09/lab9-2

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 47.72 K separate debug info for system-supplied DSO at 0xf7ffc000
Hello, world!
[Inferior 1 (process 4458) exited normally]
(gdb)

```

Рис. 4.7: Запуск программы

Для более подробного анализа программы устанавливаю брейкпоинт на метку `_start`, с которой начинается выполнение любой ассемблерной программы, и запускаю её (рис. 4.8)

```

(gdb) break _start
Breakpoint 1 at 0x8049000: file lab9-2.asm, line 9.
(gdb) run
Starting program: /home/uliana_abramova/work/arch-pc/lab09/lab9-2

Breakpoint 1, _start () at lab9-2.asm:9
9      mov eax, 4
(gdb)

```

Рис. 4.8: Установка брейкпоинта

С помощью команды `disassemble`, начиная с метки `_start`, смотрю дисассимблированный код программы (рис. 4.9)

```

(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. 4.9: Дисассимилированный код программы

Переключаюсь на отображение команд с Intel'овским синтаксисом, введя команду `set disassembly-flavor intel` (рис. 4.10)

```

(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. 4.10: Отображение команд с Intel'овским синтаксисом

Отличие заключается в командах: в дисассимилированном отображении в ко-

мандах используют % и \$, а в Intel отображение эти символы не используются.

Включаю режим псевдографики для более удобного анализа программы (рис.

```
uliana_abramova@fedora:~/work/arch-pc
Register group: general
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd000 0xffffd000
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049000 0x8049000 <_start>
eflags   0x202    [ IF ]
cs       0x23     35
ss       0x2b     43

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,0x804a008
0x8049025 <_start+37>  mov     edx,0x7
0x804902a <_start+42>  int     0x80
0x804902c <_start+44>  mov     eax,0x1
0x8049031 <_start+49>  mov     ebx,0x0
0x8049036 <_start+54>  int     0x80

native process 3959 (asm) In: _start
(gdb) layout regs
(gdb)
??)
```

#### 4.2.1 Добавление точек останова

На предыдущих шагах была установлена точка останова по имени метки (\_start). Проверяю это с помощью команды info breakpoints (кратко i b) (рис. 4.11)

```
(gdb) i b
Num      Type           Disp Enb Address      What
1        breakpoint     keep y   0x08049000 lab9-2.asm:9
breakpoint already hit 1 time
(gdb)
```

Рис. 4.11: Проверка точки останова

Устанавливаю еще одну точку останова по адресу инструкции и смотрю информацию о всех установленных точках останова (рис. 4.12, 4.13)

```
(gdb) b *0x8049031
Breakpoint 2 at 0x8049031: file lab9-2.asm, line 20.
(gdb) i b
Num      Type           Disp Enb Address      What
1        breakpoint     keep y   0x08049000 lab9-2.asm:9
          breakpoint already hit 1 time
2        breakpoint     keep y   0x08049031 lab9-2.asm:20
(gdb)
```

Рис. 4.12: Установка точки останова

Информация о всех точках останова

Рис. 4.13: Информация о всех точках останова

## 4.2.2 Работа с данными программы в GDB

С помощью команды info registers (i r) смотрю содержимое регистров (рис. 4.14)

```
edx          0x0          0
ebx          0x0          0
esp          0xffffd000    0xffffd000
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
k0           0x0          0
--Type <RET> for more, q to quit, c to continue without paging--q
Quit
```

Рис. 4.14: Содержание регистров

С помощью команды x/NFU смотрю содержимое переменной msg1 (рис. 4.15)

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

Рис. 4.15: Содержимое переменной msg1

Также смотрю значение переменной msg2 по адресу (рис. 4.16)

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

Рис. 4.16: Содержимое переменной msg2

Используя команду set, изменяю первый символ переменной msg1 (рис. 4.17)

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

Рис. 4.17: Изменение первого символа

Теперь изменяю символ переменной msg2 (рис. 4.18)

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

Рис. 4.18: Изменение символа

С помощью той же команды изменяю значение регистра ebx (рис. 4.19)

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

Рис. 4.19: Изменение регистра

Команда выводит два разных значения так как в первый раз мы вносим значение 2, а во второй раз регистр равен двум.

Завершаю выполнение программы с помощью команды `continue` (сокращенно `c`) и выхожу из `gdb` с помощью команды `quit` (сокращенно `q`) (рис. 4.20)

```
(gdb) c
Continuing.
hello, Korld!

Breakpoint 2, _start () at lab9-2.asm:20
(gdb)
```

Рис. 4.20: Завершение и выход

### 4.2.3 Обработка аргументов командной строки в GDB

Копирую файл `lab8-2.asm` в файл с именем `lab9-3.asm`. Создаю исполняемый файл и для загрузки в `gdb` использую ключ `-args`, так как программа содержит аргументы (рис. 4.21)

```
uliana_abramova@fedora:~/work/arch-pc/lab09$ gdb --args lab9-3 аргумент1 аргумент 2 'аргумент 3'
GNU gdb (Fedora Linux) 15.2-3.fc40
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.
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 lab9-3....
```

Рис. 4.21: Создание исполняемого файла

Для начала устанавливаю точку останова и запускаю программу (рис. 4.22)

```
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab9-3.asm, line 5.
(gdb) run
Starting program: /home/uliana_abramova/work/arch-pc/lab09/lab9-3 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 lab9-3.asm:5
5      pop ecx
(gdb) █
```

Рис. 4.22: Установка точки останова и запуск

Далее я проверила адрес вершины стека и убедилась, что там хранится 5 элементов. (рис. 4.23)

```
(gdb) x/x $esp
0xffffcfb0:      0x00000005
(gdb)
```

Рис. 4.23: Проверка адреса вершины стека

Затем просмотрела остальные позиции стека (рис. 4.24)

```
(gdb) x/s *(void**)(esp + 4)
0xffffd178:      "/home/uliana_abramova/work/arch-pc/lab09/lab9-3"
(gdb) x/s *(void**)(esp + 8)
0xffffd1a8:      "аргумент1"
(gdb) x/s *(void**)(esp + 12)
0xffffd1ba:      "аргумент"
(gdb) x/s *(void**)(esp + 16)
0xffffd1cb:      "2"
(gdb) x/s *(void**)(esp + 20)
0xffffd1cd:      "аргумент 3"
(gdb) x/s *(void**)(esp + 24)
0x0:      <error: Cannot access memory at address 0x0>
(gdb)
```

Рис. 4.24: Проверка остальных позиций стека

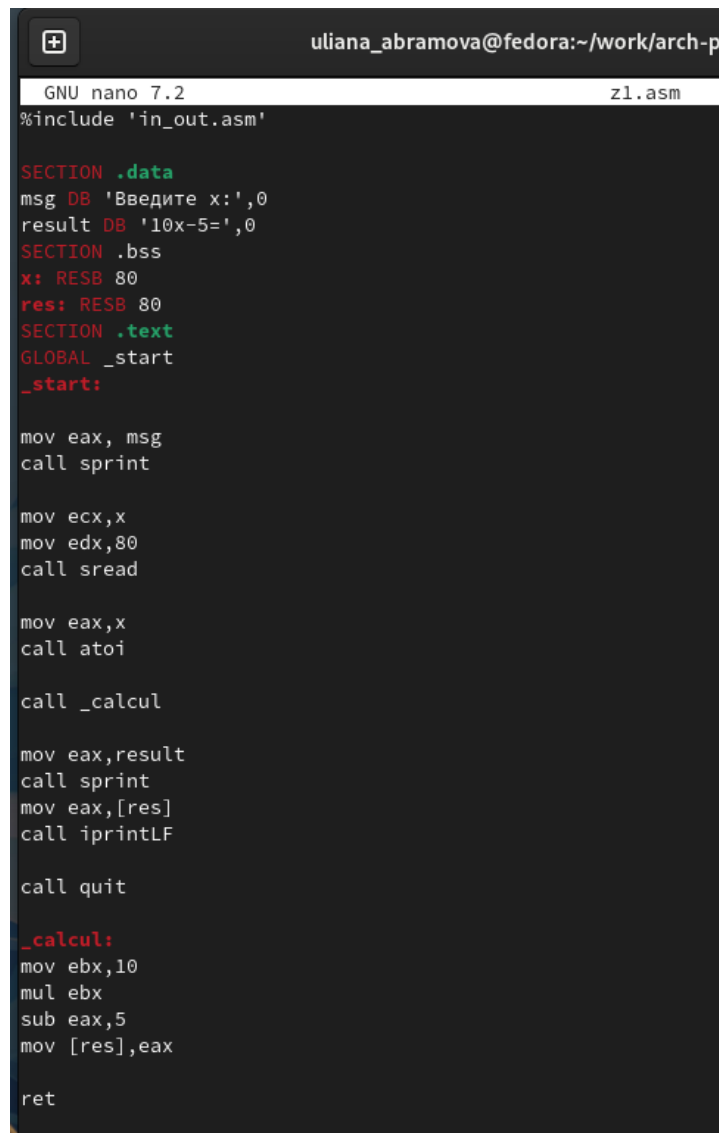
Элементы расположены с интервалом в 4 единицы, так как стек может хранить до 4 байт, и для того чтобы данные сохранялись нормально и без помех, компьютер использует новый стек для новой информации.

## 4.3 Выполнение заданий для самостоятельной работы

### 4.3.1 1

Я преобразовала программу из лабораторной работы №8 и реализовала вычисления как подпрограмму (рис. 4.25)





```
uliana_abramova@fedora:~/work/arch-pc
GNU nano 7.2 z1.asm
#include 'in_out.asm'

SECTION .data
msg DB 'Введите x:',0
result DB '10x-5=',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

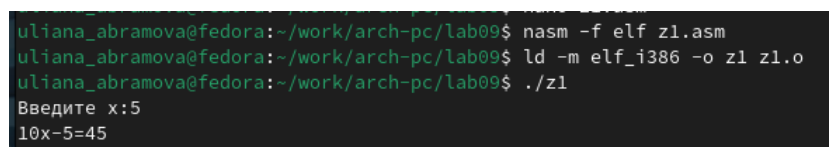
call quit

_calcul:
mov ebx, 10
mul ebx
sub eax, 5
mov [res], eax

ret
```

Рис. 4.25: Преобразование программы

Создаю исполняемый файл и проверяю корректность работы программы (рис. 4.26)



```
uliana_abramova@fedora:~/work/arch-pc/lab09$ nasm -f elf z1.asm
uliana_abramova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o z1 z1.o
uliana_abramova@fedora:~/work/arch-pc/lab09$ ./z1
Введите x:5
10x-5=45
```

Рис. 4.26: Запуск программы

### 4.3.2 2

Переписав программу из Листинга 9-3, я запустила ее, чтобы увидеть арифметическую ошибку: вместо 25 программа выводила 10 (рис. 4.27)

```
uliana_abramova@fedora:~/work/arch-pc/lab09$ nasm -f elf z2.asm
uliana_abramova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o z2 z2.o
uliana_abramova@fedora:~/work/arch-pc/lab09$ ./z2
Результат: 10
```

Рис. 4.27: Запуск программы

Я открыла регистры и проанализировала их: некоторые регистры стоят не на своих местах (рис. 4.28)

```

uliana_abramova@fedora:~/work/arch-pc/lab09
Register group: general
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd000 0xffffd000
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x80490e8 0x80490e8 <_start>
eflags   0x202    [ IF ]
cs       0x23     35
ss       0x2b     43
ds       0x2b     43
es       0x2b     43
fs       0x0      0
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 <iprintLF>
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

native process 8815 (asm) In: _start
(gdb) layout regs

```

Рис. 4.28: Анализ изменения значений регистров

Определив ошибку, я исправила, и теперь команда отработана верно (рис. 4.29)

```

uliana_abramova@fedora:~/work/arch-pc/lab09$ nasm -felf -g -l z2.lst z2.asm
uliana_abramova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o z2 z2.o
uliana_abramova@fedora:~/work/arch-pc/lab09$ gdb z2
GNU gdb (Fedora Linux) 15.2-3.fc40
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.
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 z2...
(gdb) r
Starting program: /home/uliana_abramova/work/arch-pc/lab09/z2

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.
Результат: 25
[Inferior 1 (process 12096) exited normally]

```

Рис. 4.29: Запуск исправленной команды

## 5 Вывод

Я приобрела навыки написания программ использованием подпрограмм. Познакомилась с методами отладки при помощи GDB и его основными возможностями.

## **6 Список литературы**

1. Архитектура ЭВМ