

# **Лабораторная работа номер 9**

**Архитектура компьютера**

Титков Ярослав Максимович

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>4</b>
<b>2</b>	<b>Задание</b>	<b>5</b>
<b>3</b>	<b>Теоретическое введение</b>	<b>6</b>
<b>4</b>	<b>Выполнение лабораторной работы</b>	<b>12</b>
4.1	Реализация подпрограмм в NASM . . . . .	12
4.2	Отладка программ с помощью GDB . . . . .	13
4.3	Обработка аргументов командной строки в GDB . . . . .	17
4.4	Задания для самостоятельной работы: . . . . .	19
<b>5</b>	<b>Выводы</b>	<b>21</b>

# Список иллюстраций

4.1	Создание файла . . . . .	12
4.2	Запуск программы . . . . .	12
4.3	Измененный текст программы . . . . .	13
4.4	Запуск измененной программы . . . . .	13
4.5	Работа файла lab09-2 с отладчиком gdb . . . . .	14
4.6	Установка брейкопонта на метку _start и её запуск . . . . .	14
4.7	Работа с disassemble . . . . .	15
4.8	Режим псевдографики . . . . .	15
4.9	Установка точки основы . . . . .	16
4.10	Значение регистра и msg . . . . .	16
4.11	Использование команды set . . . . .	17
4.12	Изменение регистра ebx . . . . .	17
4.13	Работа с скопированным файлом lab8-2.asm с помощью gdb . . . . .	18
4.14	Остальные позиции стека . . . . .	18
4.15	Преобразовал программу и дал ей имя lab09-4.asm . . . . .	19
4.16	Запустил программу . . . . .	19
4.17	Исправил ошибку . . . . .	20
4.18	Запустил программу и получил нужный результат . . . . .	20

# 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. Задание для самостоятельной работы

### 3 Теоретическое введение

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.

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 про-

граммы, контролиро- вать и из- менять 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. По- шаговое 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 програм- ме, в которых программа- отладчик приостанавли- вает выполнение программы и ждёт команд. Наиболее популярные виды точек останова:

- Breakpoint — точка останова (остановка происходит, когда выпол- нение дохо- дит до определённой строки, адреса или процедуры, отмеченной программы- стом);
- Watchpoint — точка просмотра (выполнение программы приостанавлива- ется, если программа обратилась к определённой переменной: либо считала её значение, либо изменила его).

Точки останова устанавливаются в отладчике на время сеанса работы с кодом програм- мы, т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы

Основные возможности отладчика GDB GDB (GNU Debugger — отладчик проек- та GNU) [1] работает на многих UNIX- подобных системах и умеет производить отладку многих языков программирова- ния. GDB предлагает обширные сред- ства для слежения и контроля за выполнени- ем компьютерных программ. От- ладчик не содержит собственного графического пользовательского интерфей- са и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторон- них графических над- строек, а кроме того, неко- торые 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 поведе- ние;
- остановить программу при указанных условиях;
- исследовать, что случи- лось, когда программа остановилась;
- изменить программу так, чтобы можно

было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других. 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)` у 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` 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 AT&T (значительно отличающийся внешне). По умолчанию в дизассемблере GDB принят режим AT&T. Переключиться на отображение команд с привычным Intel'овским синтаксисом можно, введя команду `set disassembly-flavor intel` Точки останова Установить точку останова можно коман-



дой `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 *` `(gdb) b` 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` Обратная точка останова активируется командой `enable`: `enable breakpoint` Если же точка останова в дальнейшем больше не нужна, она может быть удалена с помощью команды `delete`: `(gdb) delete breakpoint` Ввод этой команды без аргумента удалит все точки останова. Информацию о командах этого раздела можно получить, введя `help breakpoints` Пошаговая отладка Для продолжения остановленной программы используется команда `continue` (`c`) `(gdb) c [аргумент]`. Выполнение программы будет происходить до следующей точки останова. В качестве аргумента может использоваться целое число, которое указывает отладчику проигнорировать — 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 2 отладчик выполнит команду `step` несколько 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`

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` задаёт 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]`. 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. Инструкция `call` и инструкция `ret`

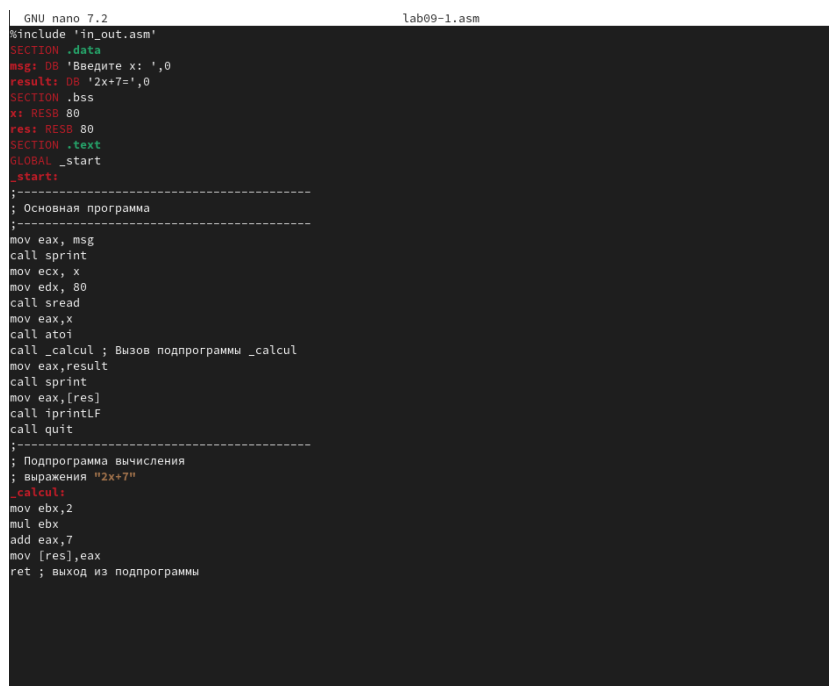
Для вызова подпрограммы из основной программы используется инструкция `call`, которая заносит адрес следующей инструкции в стек и загружает в регистр `esp` адрес соответствующей подпрограммы, осуществляя таким образом переход. Затем начинается выполнение подпрограммы, которая, в свою очередь, также может 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 подпрограммой код, как будто он является её продолжением.

## 4 Выполнение лабораторной работы

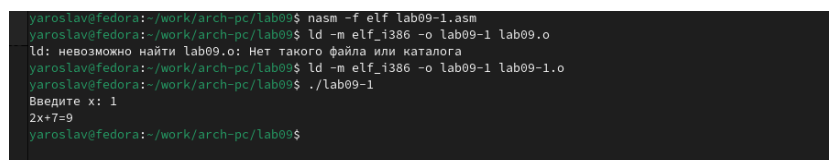
### 4.1 Реализация подпрограмм в NASM

Создал каталог для выполнения лаб. работы номер 9, а затем создал файл lab09-1.asm и загрузил туда данные из Листинга 9.1, после этого запустил файл:



```
GNU nano 7.2 lab09-1.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:
;-----
; 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, [res]
call iprintf
call quit
;-----
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
mov ebx, 2
mul ebx
add eax, 7
mov [res], eax
ret ; выход из подпрограммы
```

Рис. 4.1: Создание файла



```
yaroslav@fedora: ~/work/arch-pc/lab09$ nasm -f elf lab09-1.asm
yaroslav@fedora: ~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09.o
ld: невозможно найти lab09.o: Нет такого файла или каталога
yaroslav@fedora: ~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
yaroslav@fedora: ~/work/arch-pc/lab09$ ./lab09-1
Введите x: 1
2x+7=9
yaroslav@fedora: ~/work/arch-pc/lab09$
```

Рис. 4.2: Запуск программы

Изменил текст программы, добавив подпрограмму `_subcalcul` в подпрограмму `_calcul`, для вычисления выражения  $f(g(x))$ , где  $x$  вводится с клавиатуры,  $f(x) = 2x + 7$ ,  $g(x) = 3x - 1$ . Т.е.  $x$  передается в подпрограмму `_calcul` из нее в подпрограмму `_subcalcul`, где вычисляется выражение  $g(x)$ , результат возвращается в `_calcul` и вычисляется выражение  $f(g(x))$

```
%include 'in_out.asm'

SECTION .data
msg: DB 'Введите x: ',0
result: DB 'f(g(x)) = 2(3x-1) + 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
call sprint

mov eax, [res]
call iprintf

call quit

;-----
; Подпрограмма вычисления
; выражения "2(3x-1) + 7"
_calcul:
call _subcalcul ; Вызов подпрограммы _subcalcul
```

Рис. 4.3: Измененный текст программы

```
yaroslav@fedora:~/work/arch-pc/lab09$ nano lab09-1.asm
yaroslav@fedora:~/work/arch-pc/lab09$ nasm -f elf lab09-1.asm
yaroslav@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
yaroslav@fedora:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 2
f(g(x)) = 2(3x-1) + 7 = 17
yaroslav@fedora:~/work/arch-pc/lab09$
```

Рис. 4.4: Запуск измененной программы

## 4.2 Отладка программ с помощью GDB

Создал файл `lab09-2.asm` с текстом программы из Листинга 9.2. Получил исполняемый файл, загрузил его в отладчик `gdb`, проверил его работу.

```

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

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.
Hello, world!
Inferior 1 (process 4501) exited normally
(gdb)

```

Рис. 4.5: Работа файла lab09-2 с отладчиком gdb

Для более подробного анализа программы установил брейкпоинт на метку `_start`, с которой начинается выполнение любой ассемблерной программы, и запустил её.

```

Inferior 1 (process 4501) exited normally
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /home/yaroslav/work/arch-pc/lab09/lab09-2

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

```

Рис. 4.6: Установка брейкпоинта на метку `_start` и её запуск

Посмотрел дисассимилированный код программы с помощью команды `disassemble` начиная с метки `_start`, затем переключился на отображение команд с Intel'овским синтаксисом, введя команду `set disassembly-flavor intel`

```

(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>: mov $0x4,%eax
0x08049005 <+5>: mov $0x1,%ebx
0x0804900a <+10>: mov $0x04a000,%ecx
0x0804900f <+15>: mov $0x8,%edx
0x08049014 <+20>: int $0x80
0x08049016 <+22>: mov $0x4,%eax
0x0804901b <+27>: mov $0x1,%ebx
0x08049020 <+32>: mov $0x04a008,%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,0x04a000
0x0804900f <+15>: mov edx,0x8
0x08049014 <+20>: int 0x80
0x08049016 <+22>: mov eax,0x4
0x0804901b <+27>: mov ebx,0x1
0x08049020 <+32>: mov ecx,0x04a008
0x08049025 <+37>: mov edx,0x7
0x0804902a <+42>: int 0x80
0x0804902c <+44>: mov eax,0x1
0x08049031 <+49>: mov ebx,0x0
0x08049036 <+54>: int 0x80
End of assembler dump.
(gdb)

```

Рис. 4.7: Работа с disassemble

Ответ на вопрос: Перечислите различия отображения синтаксиса машинных команд в режимах АТТ и Intel: В режиме АТТ операнды указываются в порядке “источник, назначение” и используют префиксы размеров данных (например, movl), в то время как в режиме Intel операнды указываются в порядке “назначение, источник” и размер данных указывается в имени команды.

Включил режим псевдографики для более удобного анализа программы

```

--Register group: general--
eax    0x0      0      ecx    0x0      0
edx    0x0      0      ebx    0x0      0
esp    0xffffd090 0xffffd090  ebp    0x0      0x0
esi    0x0      0      edi    0x0      0
eip    0x08049000 0x08049000 <_start>  eflags 0x202    [ IF ]
cs     0x23     35      ss     0x2b     43
ds     0x2b     43      es     0x2b     43
fs     0x0      0      gs     0x0      0

0x08049000 <_start> mov    eax,0x4
0x08049005 <_start+5> mov    ebx,0x1
0x0804900a <_start+10> mov    ecx,0x04a000
0x0804900f <_start+15> mov    edx,0x8
0x08049014 <_start+20> int    0x80
0x08049016 <_start+22> mov    eax,0x4
0x0804901b <_start+27> mov    ebx,0x1
0x08049020 <_start+32> mov    ecx,0x04a008
0x08049025 <_start+37> mov    edx,0x7
0x0804902a <_start+42> int    0x80
0x0804902c <_start+44> mov    eax,0x1
0x08049031 <_start+49> mov    ebx,0x0
0x08049036 <_start+54> int    0x80
0x08049038      add    BYTE PTR [eax],al

native process 4536 (asm) In: _start          L9    PC: 0x08049000
(gdb) layout regs
(gdb)

```

Рис. 4.8: Режим псевдографики

С помощью команды `info breakpoint` узнал о точке основы, а затем определил адрес предпоследней инструкции и установил ещё одну точку основы

```

Register group: general
eax      0x0      0      ecx      0x0      0
edx      0x0      0      ebx      0x0      0
esp      0xffffd090 0xffffd090  ebp      0x0      0
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

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
0x8049038 add     BYTE PTR [eax],al
0x804903a add     BYTE PTR [eax],al

native process 5114 (asm) In: _start L9 PC: 0x8049000
(gdb) layout regs
(gdb) info breakpoints
Num   Type             Disp Enb Address      What
1     breakpoint       keep y  0x8049000 lab09-2.asm:9
      breakpoint already hit 1 time
(gdb) break *0x8049031
Breakpoint 2 at 0x8049031: file lab09-2.asm, line 20.
(gdb) i b
Num   Type             Disp Enb Address      What
1     breakpoint       keep y  0x8049000 lab09-2.asm:9
      breakpoint already hit 1 time
2     breakpoint       keep y  0x8049031 lab09-2.asm:20
(gdb)

```

Рис. 4.9: Установка точки основы

Ответ на вопрос: Выполните 5 инструкций с помощью команды `stepi`. Значение каких регистров меняются? изменяются регистры RAX, RBX, RCX, RDX, RSI, RDI, RSP.

Посмотрел содержание регистров, а затем значение переменной `msg1` и значение `msg2` по адресу.

```

rsi      0x0      0
rdi      0x0      0
eip      0x8049000 0x8049000 <_start>
eflags   0x202    [ IF ]
fs       0x23     35
ss       0x2b     43
ds       0x2b     43
es       0x2b     43
fs       0x0      0
gs       0x0      0
--Type <RET> for more, q to quit, c to continue without paging--
(gdb) x/1sb &msg1
0x804a000 <msg1>: "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>: "world!\n034"
(gdb)

```

Рис. 4.10: Значение регистра и msg



С помощью команды set изменил значения для переменной msg1 и изменил символ для переменной msg2.

```
(gdb) set {char}&msg1='h'
(gdb) set {char}0x804a001='h'
(gdb) x/1sb &msg1
0x804a001: "hhlllo, "
(gdb) set {char}0x804a008='L'
(gdb) set {char}0x804a00b=' '
(gdb) x/1sb &msg2
0x804a008: "Lor d!\n\034"
(gdb)
```

Рис. 4.11: Использование команды set

Изменил значения регистра ebx

```
(gdb) set $ebx='2'
(gdb) set $ebx='2'
(gdb) set $ebx=2
(gdb) p/s $ebx
$6 = 2
(gdb)
```

Рис. 4.12: Изменение регистра ebx

Ответ на вопрос: Объясните разницу вывода команд p/s \$ebx: При set \$ebx='2' GDB интерпретирует '2' как символ с ASCII-кодом 50, а при set \$ebx=2 — как число 2.

## 4.3 Обработка аргументов командной строки в GDB

Скопировал файл lab8-2.asm в файл с именем lab09-3.asm, создал исполняемый файл, загрузил исполняемый файл в отладчик, установил точку основы и запустил её

```

yaroslav@fedora:~/work/arch-pc/lab09$ cp ~/work/arch-pc/lab08/lab8-2.asm ~/work/arch-pc/lab09/lab09-3.asm
yaroslav@fedora:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-3.lst lab09-3.asm
yaroslav@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-3 lab09-3.o
yaroslav@fedora:~/work/arch-pc/lab09$ gdb --args lab09-3 аргумент1 аргумент 2 'аргумент 3'
GNU gdb (Fedora Linux) 15.2-2.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 lab09-3...
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 5.
(gdb) run
Starting program: /home/yaroslav/work/arch-pc/lab09/lab09-3 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]) 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
5      pop ecx ; Извлекаем из стека в 'ecx' количество аргументов (первое значение в стеке)
(gdb) x/x $esp
0xffffd040: 0x00000005
(gdb)

```

Рис. 4.13: Работа с скопированным файлом lab8-2.asm с помощью gdb

## Посмотрел остальные позиции стека

```

(gdb) x/x $esp
0xffffd030: 0x00000005
(gdb) x/s *(void**)(esp + 4)
0xffffd1f3: "/home/yaroslav/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)(esp + 8)
0xffffd21d: "аргумент1"
(gdb) x/s *(void**)(esp + 12)
0xffffd22f: "аргумент"
(gdb) x/s *(void**)(esp + 16)
0xffffd240: "2"
(gdb) x/s *(void**)(esp + 20)
0xffffd242: "аргумент 3"
(gdb) x/s *(void**)(esp + 24)
0x0: <error: Cannot access memory at address 0x0>
(gdb)

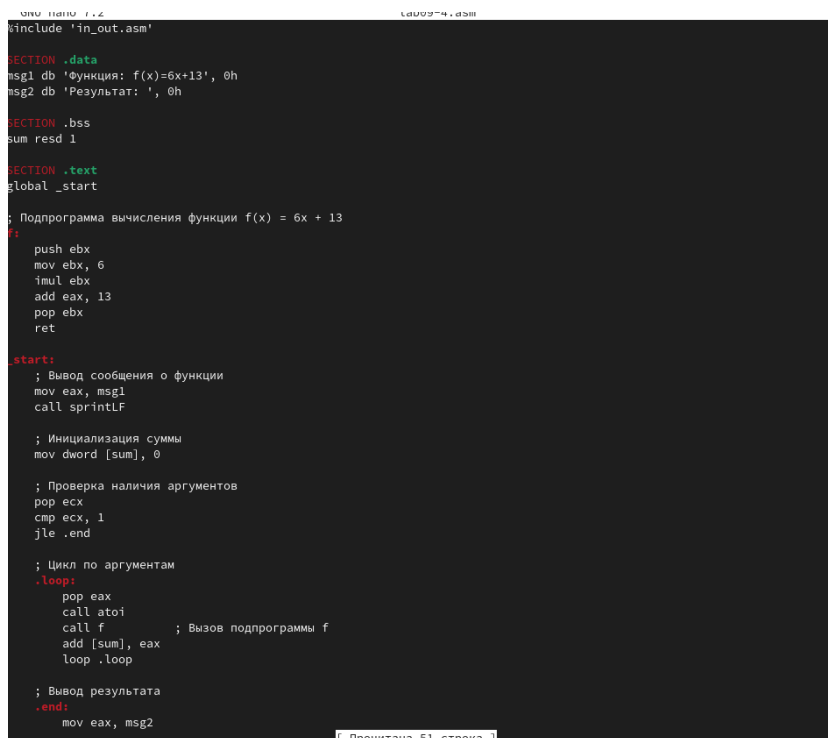
```

Рис. 4.14: Остальные позиции стека

Ответ на вопрос: Объясните, почему шаг изменения адреса равен 4 ([esp+4], [esp+8], [esp+12] и т.д.): Шаг изменения адреса равен 4, потому что указатели занимают 4 байта в 32-битной архитектуре.

## 4.4 Задания для самостоятельной работы:

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



```
include 'in_out.asm'

SECTION .data
msg1 db 'Функция: f(x)=6x+13', 0h
msg2 db 'Результат: ', 0h

SECTION .bss
sum resd 1

SECTION .text
global _start

; Подпрограмма вычисления функции f(x) = 6x + 13
f:
    push ebx
    mov ebx, 6
    imul ebx
    add eax, 13
    pop ebx
    ret

_start:
    ; Вывод сообщения о функции
    mov eax, msg1
    call sprintf

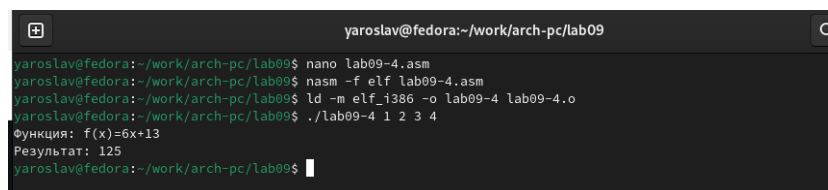
    ; Инициализация суммы
    mov dword [sum], 0

    ; Проверка наличия аргументов
    pop ecx
    cmp ecx, 1
    jle .end

    ; Цикл по аргументам
.loop:
    pop eax
    call atoi
    call f
    add [sum], eax
    loop .loop

    ; Вывод результата
.end:
    mov eax, msg2
```

Рис. 4.15: Преобразовал программу и дал ей имя lab09-4.asm



```
yaroslav@fedora:~/work/arch-pc/lab09$ nano lab09-4.asm
yaroslav@fedora:~/work/arch-pc/lab09$ nasm -f elf lab09-4.asm
yaroslav@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-4 lab09-4.o
yaroslav@fedora:~/work/arch-pc/lab09$ ./lab09-4 1 2 3 4
Функция: f(x)=6x+13
Результат: 125
yaroslav@fedora:~/work/arch-pc/lab09$
```

Рис. 4.16: Запустил программу

В листинге 9.3 приведена программа вычисления выражения  $(3 + 2) * 4 + 5$ . При запуске данная программа дает неверный результат. Проверьте это. С помощью отладчика GDB, анализируя изменения значений регистров, определите ошибку и исправьте ее.



```
GNU nano 7.2 lab09-5.asm
#include 'in_out.asm'

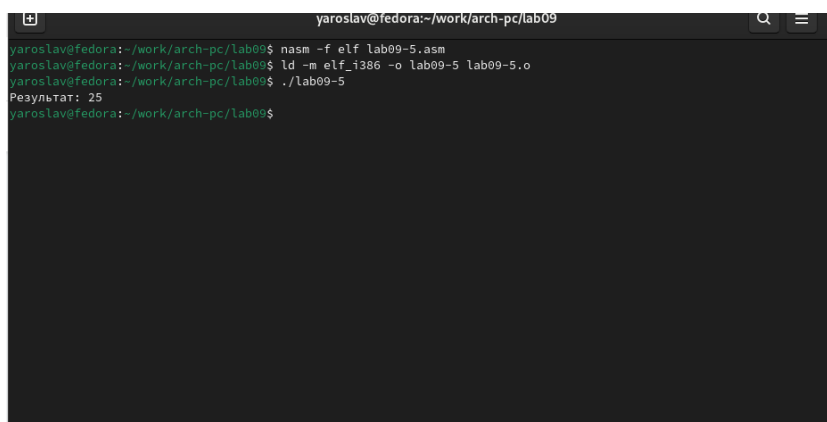
SECTION .data
div: DB 'Результат: ',0

SECTION .text
GLOBAL _start

_start:
mov eax, 3
add eax, 2
mov ebx, eax
mov ecx, 4
mul ecx
add eax, 5
mov edi, eax

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

Рис. 4.17: Исправил ошибку



```
yaroslav@fedora:~/work/arch-pc/lab09
yaroslav@fedora:~/work/arch-pc/lab09$ nasm -f elf lab09-5.asm
yaroslav@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-5 lab09-5.o
yaroslav@fedora:~/work/arch-pc/lab09$ ./lab09-5
Результат: 25
yaroslav@fedora:~/work/arch-pc/lab09$
```

Рис. 4.18: Запустил программу и получил нужный результат

## **5 Выводы**

В ходе работы приобрел навыки написания программ с использованием подпрограмм. Познакомился с методами отладки при помощи GDB и его основными возможностями