

Отчёт по лабораторной работе №9

Дисциплина: архитектура компьютера

Черкашина Ангелина Максимовна

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
4	Выполнение лабораторной работы	10
4.1	Реализация подпрограмм в NASM	10
4.2	Отладка программ с помощью GDB	13
4.3	Добавление точек останова	17
4.4	Работа с данными программы в GDB	19
4.5	Обработка аргументов командной строки в GDB	25
4.6	Выполнение заданий для самостоятельной работы	27
5	Выводы	31
6	Список литературы	32

Список иллюстраций

4.1	Создание каталога и файла	10
4.2	Создание копии файла	10
4.3	Редактирование файла	11
4.4	Запуск исполняемого файла	11
4.5	Изменение текста программы	12
4.6	Запуск нового исполняемого файла	13
4.7	Редактирование файла	13
4.8	Получение исполняемого файла	14
4.9	Загрузка исполняемого файла в отладчик	14
4.10	Запуск программы в GDB	14
4.11	Установка брейкпоинта и запуск программы	15
4.12	Дизассемблированный код программы	15
4.13	Дизассемблированный код программы с синтаксисом Intel	16
4.14	Включение режима псевдографики	17
4.15	Просмотр информации о точках останова	18
4.16	Установка брейкпоинта	19
4.17	Просмотр содержимого регистров	20
4.18	Выполнение инструкций с помощью stepi	21
4.19	Просмотр значений переменных	22
4.20	Замена символов переменных	22
4.21	Вывод значения регистра в разных представлениях	23
4.22	Изменение значения регистра	24
4.23	Завершение работы GDB	25
4.24	Создание копии файла	26
4.25	Создание исполняемого файла	26
4.26	Загрузка исполняемого файла с аргументами в отладчик	26
4.27	Установка брейкпоинта и запуск программы	26
4.28	Просмотр значений, введенных в стек	27
4.29	Написание кода программы	28

Список таблиц

1 Цель работы

Целью данной лабораторной работы является 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. Выполнение заданий 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`.

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. 4.1).

```
amcherkashina@dk2n24 ~ $ mkdir ~/work/arch-pc/lab09
amcherkashina@dk2n24 ~ $ cd ~/work/arch-pc/lab09
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ touch lab09-1.asm
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $
```

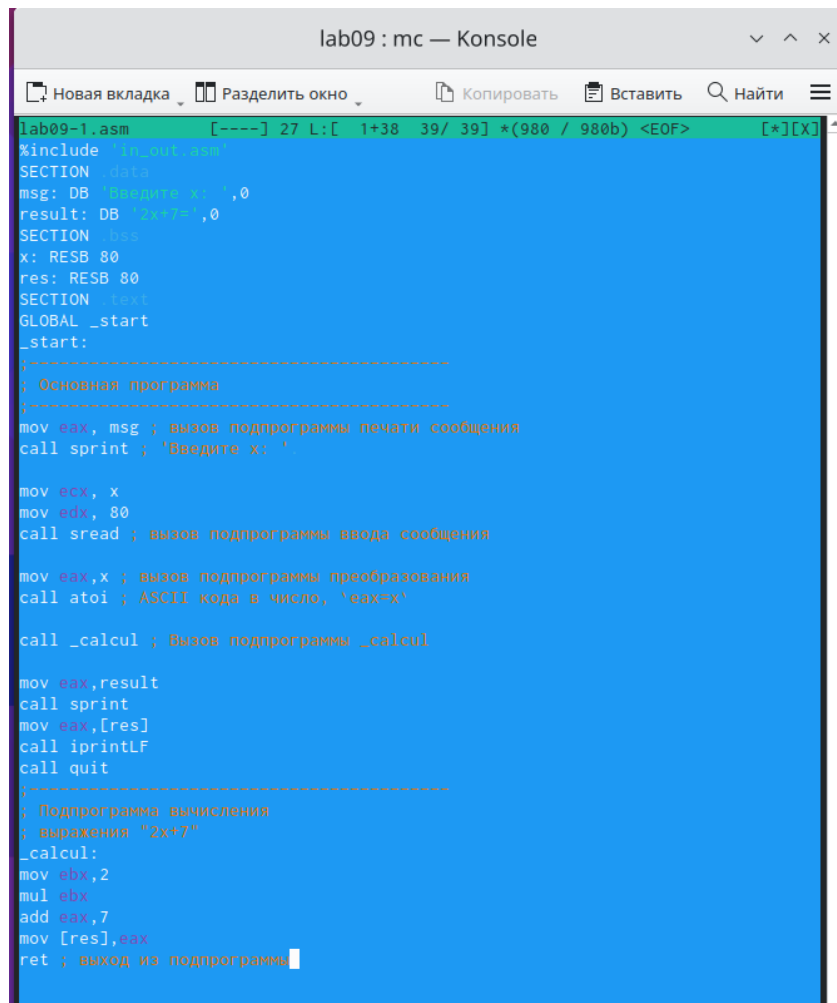
Рис. 4.1: Создание каталога и файла

С помощью команды `cp` копирую в текущий каталог файл `in_out.asm`, т.к. он будет использоваться в программах данной лабораторной работы (рис. 4.2).

```
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ cp ~/Загрузки/in_out.asm in_out.asm
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $
```

Рис. 4.2: Создание копии файла

Ввожу в файл `lab09-1.asm` текст программы с использованием вызова подпрограммы из листинга 9.1 (рис. 4.3).



```
lab09-1.asm [----] 27 L: [ 1+38 39/ 39] *(980 / 980b) <EOF> [*][X]
%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 ; вызов подпрограммы печати сообщения
call sprint ; "Введите x: "

mov ecx, x
mov edx, 80
call sread ; вызов подпрограммы ввода сообщения

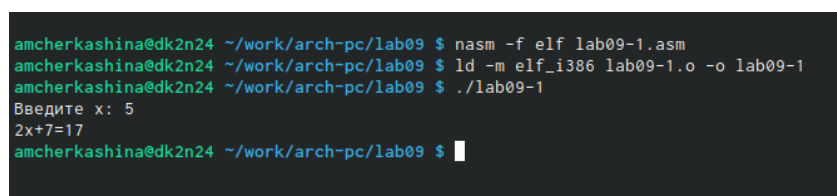
mov eax, x ; вызов подпрограммы преобразования
call atoi ; ASCII кода в число, 'eax=x'

call _calcul ; Вызов подпрограммы _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 ; выход из подпрограммы
```

Рис. 4.3: Редактирование файла

Создаю исполняемый файл и запускаю его (рис. 4.4).



```
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ nasm -f elf lab09-1.asm
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ ld -m elf_i386 lab09-1.o -o lab09-1
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ ./lab09-1
Введите x: 5
2x+7=17
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $
```

Рис. 4.4: Запуск исполняемого файла

Изменяю текст программы, добавив подпрограмму `_subcalcul` в подпрограмму `_calcul`, для вычисления выражения $f(g(x))$, где \boxtimes вводится с клавиатуры, $f(x) = 2x$

+ 7, $g(x) = 3x - 1$. Т.е. \square передается в подпрограмму `_calcul` из нее в подпрограмму `_subcalcul`, где вычисляется выражение $g(x)$, результат возвращается в `_calcul` и вычисляется выражение $f(g(x))$. Результат возвращается в основную программу для вывода результата на экран (рис. 4.5).

```

lab09-1.asm  [----]  3  L: [ 1+45  46/ 46] *(1090/1090b) <EOF>  [*][X]
%include "io.inc.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 ; вызов подпрограммы печати сообщения
call sprint ; 'Введите x: '

mov ecx, x
mov edx, 80
call sread ; вызов подпрограммы ввода сообщения

mov eax, x ; вызов подпрограммы преобразования
call atoi ; ASCII коды в число, 'eax=x'

call _calcul ; Вызов подпрограммы _calcul

mov eax, result
call sprint
mov eax, [res]
call iprintLF
call quit
; =====
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
call _subcalcul ; Вызов подпрограммы _subcalcul
mov ebx, 2
mul ebx
add eax, 7
mov [res], eax
ret ; выход из подпрограммы

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

Рис. 4.5: Изменение текста программы

Создаю исполняемый файл и проверяю его работу (рис. 4.6).

```

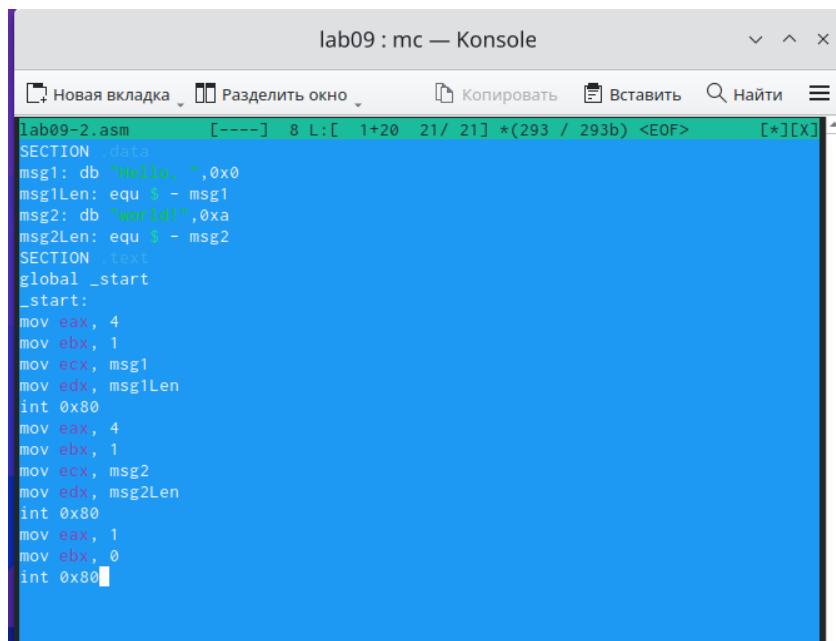
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ nasm -f elf lab09-1.asm
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ ld -m elf_i386 lab09-1.o -o lab09-1
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ ./lab09-1
Введите x: 5
2x+7=35
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $

```

Рис. 4.6: Запуск нового исполняемого файла

4.2 Отладка программ с помощью GDB

Создаю файл lab09-2.asm и ввожу в него текст программы вывода сообщения Hello world! из листинга 9.2 (рис. 4.7).



```

lab09-2.asm [----] 8 L: [ 1+20 21/ 21] *(293 / 293b) <EOF> [*][X]
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. 4.7: Редактирование файла

Получаю исполняемый файл для работы с GDB добавив в него отладочную информацию. Для этого трансляцию программ провожу с ключом -g (рис. 4.8).

```

amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ touch lab09-2.asm
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ mc
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ nasm -f elf -g -l lab09-2.lst lab09-2.asm
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-2 lab09-2.o
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $

```

Рис. 4.8: Получение исполняемого файла

Загружаю исполняемый файл в отладчик gdb (рис. 4.9).

```

amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ gdb lab09-2
GNU gdb (Gentoo 13.2 vanilla) 13.2
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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
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)

```

Рис. 4.9: Загрузка исполняемого файла в отладчик

Проверяю работу программы, запустив ее в оболочке GDB с помощью команды run (рис. 4.10).

```

(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/a/m/amcherkashina/work/arch-pc/lab09/lab09-2
Hello, world!
[Inferior 1 (process 5951) exited normally]
(gdb)

```

Рис. 4.10: Запуск программы в GDB

Для более подробного анализа программы устанавливаю брейкпоинт на метку `_start`, с которой начинается выполнение ассемблерной программы, и запускаю её (рис. 4.11).

```
(gdb) break _start
Breakpoint 1 at 0x08049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/a/m/amcherkashina/work/arch-pc/lab09/lab09-2

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

Рис. 4.11: Установка брейкпоинта и запуск программы

Просматриваю дизассимилированный код программы с помощью команды `disassemble`, начиная с метки `_start` (рис. 4.12).

```
(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) █
```

Рис. 4.12: Дизассемблированный код программы

Переключаюсь на отображение команд с синтаксисом Intel и снова просматриваю дизассимилированный код программы (рис. 4.13).

```

(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.
(gdb) █

```

Рис. 4.13: Дизассемблированный код программы с синтаксисом 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. 4.14).

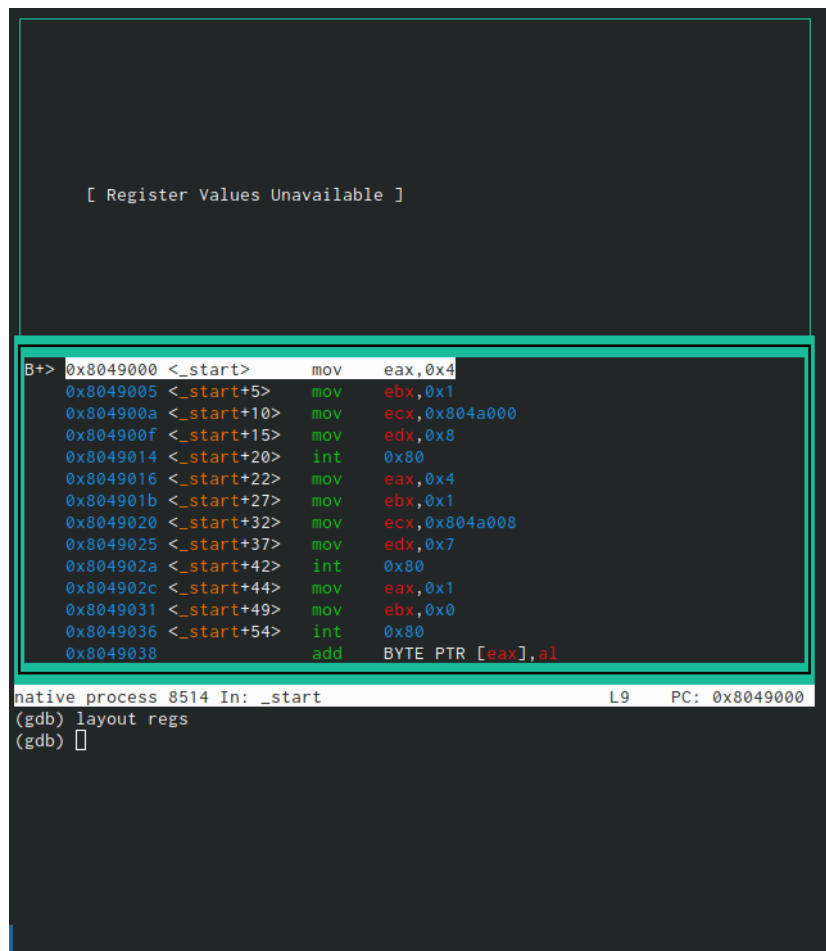


Рис. 4.14: Включение режима псевдографики

В этом режиме есть три окна: • В верхней части видны названия регистров и их текущие значения; • В средней части виден результат дисассимилирования программы; • Нижняя часть доступна для ввода команд

4.3 Добавление точек останова

Ранее мной была установлена точка останова по имени метки _start. Проверяю это с помощью команды info breakpoints (кратко i b) (рис. 4.15).

```

[ Register Values Unavailable ]

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

native process 8514 In: _start L9 PC: 0x8049000
(gdb) layout regs
(gdb) info breakpoints
Num      Type      Disp Enb Address      What
1        breakpoint keep y 0x08049000 lab09-2.asm:9
breakpoint already hit 1 time
(gdb) 

```

Рис. 4.15: Просмотр информации о точках останова

Устанавливаю еще одну точку останова по адресу предпоследней инструкции (`mov ebx,0x0`). Адрес инструкции нахожу в средней части экрана в левом столбце соответствующей инструкции. Просматриваю информацию о всех установленных точках останова с помощью `i b` - сокращения команды `info breakpoints` (рис. 4.16).

```

B+> 0x08049000 <_start>    mov     eax,0x4
0x08049005 <_start+5>    mov     ebx,0x1
0x0804900a <_start+10>   mov     ecx,0x804a000
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,0x804a008
0x08049025 <_start+37>   mov     edx,0x7
0x0804902a <_start+42>   int     0x80
0x0804902c <_start+44>   mov     eax,0x1
b+ 0x08049031 <_start+49> mov     ebx,0x0
0x08049036 <_start+54>   int     0x80
0x08049038                add     BYTE PTR [eax],al

native process 8514 In: _start                                L9    PC: 0x08049000
(gdb) layout regs
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint       keep y 0x08049000 lab09-2.asm:9
          breakpoint already hit 1 time
(gdb) break *0x08049031
Breakpoint 2 at 0x08049031: file lab09-2.asm, line 20.
(gdb) i b
Num      Type             Disp Enb Address      What
1        breakpoint       keep y 0x08049000 lab09-2.asm:9
          breakpoint already hit 1 time
2        breakpoint       keep y 0x08049031 lab09-2.asm:20
(gdb) 

```

Рис. 4.16: Установление брейкпоинта

4.4 Работа с данными программы в GDB

Просматриваю содержимое регистров с помощью команды info registers (i r) (рис. 4.17).

```
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
b+ 0x8049031 <_start+49> mov    ebx,0x0
0x8049036 <_start+54> int    0x80
0x8049038 add    BYTE PTR [eax],al

native process 14036 In: _start L9 PC: 0x8049000
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffc1e0 0xffffc1e0
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
--Type <RET> for more, q to quit, c to continue without paging--
```

Рис. 4.17: Просмотр содержимого регистров

Выполняю 5 инструкций с помощью команды stepi (si) (рис. 4.18).

```

Register group: general
eax      0x4      4
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffc1e0 0xffffc1e0
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x804901b 0x804901b <_start+27>
eflags   0x202    [ IF ]
cs       0x23     35
ss       0x2b     43
ds       0x2b     43
es       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
b+ 0x8049031 <_start+49> mov     ebx,0x0
0x8049036 <_start+54>   int     0x80
0x8049038          add     BYTE PTR [eax],al

native process 5148 In: _start L15 PC: 0x804901b
esp      0xffffc1e0 0xffffc1e0
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
--Type <RET> for more, q to quit, c to continue without paging--cgs
0x0      0
(gdb) stepi
(gdb) stepi 5
(gdb)

```

Рис. 4.18: Выполнение инструкций с помощью stepi

Изменились значения регистров ecx, edx и ebx.

Просматриваю значение переменной msg1 по имени с помощью команды x/1sb &msg1, а затем просматриваю значение переменной msg2 по ее адресу(x/1sb). Адрес переменной определяю по дизассемблированной инструкции (рис. 4.19).

```

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
b+ 0x8049031 <_start+49> mov     ebx,0x0
    0x8049036 <_start+54> int      0x80
    0x8049038          add     BYTE PTR [eax],a

native process 5148 In: _start L15 PC: 0x804901b
eip      0x8049000      0x8049000 <_start>
eflags   0x202          [ IF ]
cs       0x23           35
ss       0x2b           43
ds       0x2b           43
es       0x2b           43
fs       0x0            0
--Type <RET> for more, q to quit, c to continue without paging--cgs
0x0
(gdb) stepi
(gdb) stepi 5
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n\034"
(gdb)

```

Рис. 4.19: Просмотр значений переменных

С помощью команды `set` заменяю первый символ переменной `msg1` на 'h', а далее изменяю один символ второй переменной `msg2` (рис. 4.20).

```

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

```

Рис. 4.20: Замена символов переменных

С помощью команды `print print/F $val` вывожу значение регистра `edx` в трех различных форматах: в шестнадцатеричном формате, в двоичном формате и в символьном виде (рис. 4.21).

```

Register group: general
eax      0x4      4
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffc1e0 0xffffc1e0
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x804901b 0x804901b <_start+27>
eflags   0x202    [ IF ]
cs       0x23     35
ss       0x2b     43
ds       0x2b     43
es       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
b+ 0x8049031 <_start+49> mov     ebx,0x0
    0x8049036 <_start+54> int     0x80
    0x8049038          add     BYTE PTR [eax],al

native process 5148 In: _start L15 PC: 0x804901b
0x804a008 <msg2>: "world!\n\034"
(gdb) set {char}msg1='h'
'msg1' has unknown type; cast it to its declared type
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>: "hello, "
(gdb) set {char}&msg2='W'
(gdb) x/1sb &msg2
0x804a008 <msg2>: "World!\n\034"
(gdb) print/x $edx
$1 = 0x8
(gdb) print/t $edx
$2 = 1000
(gdb) print/c $edx
$3 = 8 '\b'
(gdb)

```

Рис. 4.21: Вывод значения регистра в разных представлениях

С помощью команды set изменяю значение регистра ebx в соответствии с заданием и вывожу его значения (рис. 4.22).

```

Register group: general
eax      0x4      4
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x2      2
esp      0xffffc1e0 0xffffc1e0
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x804901b 0x804901b <_start+27>
eflags   0x202    [ IF ]
cs       0x23     35
ss       0x2b     43
ds       0x2b     43
es       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
b+ 0x8049031 <_start+49> mov    ebx,0x0
0x8049036 <_start+54>   int     0x80
0x8049038              add    BYTE PTR [eax],al

native process 5148 In: _start L15 PC: 0x804901b
(gdb) print/x $edx
$1 = 0x8
(gdb) print/t $edx
$2 = 1000
(gdb) print/c $edx
$3 = 8 '\b'
(gdb) set $ebx='2'
(gdb) p/s &ebx
No symbol "ebx" in current context.
(gdb) set $ebx='2'
(gdb) print/s $ebx
$4 = 50
(gdb) set $ebx=2
(gdb) print/s $ebx
$5 = 2
(gdb) 

```

Рис. 4.22: Изменение 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. 4.23).


```

eax      0x4      4
ecx      0x1      1
ecx      0x804a008 134520840
edx      0x7      7
esp      0xffffffff 0xffffffff
ebp      0x0      0
esi      0x0      0
edi      0x0      0
eip      0x804901b 0x804901b <_start+27>
eip      0x8049031 0x8049031 <_start+49>
cs       0x23     35
ss       0x2b     43
ds       0x2b     43
es       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
0x804901b <_start+27>  mov     ebx,0x104a008
0x8049025 <_start+37>   mov     edx,0x7
0x804902a <_start+42>   int     0x80
0x804902c <_start+44>   mov     eax,0x1
b+ 0x8049031 <_start+49> mov     ebx,0x0
B > 0x8049031 <_start+49> mov     ebx,0x0
0x8049038 add     BYTE PTR [eax],al

native process 5148 In: _start L15 PC: 0x804901b
(gdb) print/t $edx 20
(gdb) print/s $ebx 31
$4 = 50
(gdb) set $ebx=2
(gdb) print/s $ebx
$5 = 2
(gdb) continue
Continuing.
World!

Breakpoint 2, _start () at lab09-2.asm:20
(gdb) quit
A debugging session is active.

    Inferior 1 [process 5148] will be killed.

Quit anyway? (y or n)

```

Рис. 4.23: Завершение работы GDB

4.5 Обработка аргументов командной строки в GDB

Копирую файл lab8-2.asm, созданный при выполнении лабораторной работы №8, с программой выводящей на экран аргументы командной строки в файл с именем lab09-3.asm (рис. 4.24).

```
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ cp ~/work/arch-pc/lab08/lab8-2.asm ~/work/arch-pc/lab09/lab09-3.asm
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $
```

Рис. 4.24: Создание копии файла

Создаю исполняемый файл (рис. 4.25).

```
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ nasm -f elf -g -l lab09-3.lst lab09-3.asm
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-3 lab09-3.o
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $
```

Рис. 4.25: Создание исполняемого файла

Загружаю исполняемый файл программы с аргументами в отладчик, используя для этого ключ `--args` (рис. 4.26).

```
amcherkashina@dk2n24 ~/work/arch-pc/lab09 $ gdb --args lab09-3 аргумент1 аргумент 2 'аргумент 3'
GNU gdb (Gentoo 13.2 vanilla) 13.2
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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
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)
```

Рис. 4.26: Загрузка исполняемого файла с аргументами в отладчик

Устанавливаю точку останова перед первой инструкцией в программе и запускаю ее (рис. 4.27).

```
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 8.
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/a/m/amcherkashina/work/arch-pc/lab09/lab09-3 аргумент1 аргумент 2 аргумент\ 3
Breakpoint 1, _start () at lab09-3.asm:8
8      pop ecx ; Извлекаем из стека в 'ecx' количество
(gdb)
```

Рис. 4.27: Установка брейкпоинта и запуск программы

Посматриваю вершину стека и остальные позиции стека по их адресам (по адресу `[esp+4]` находится имя программы, по адресу `[esp+8]` хранится адрес

первого аргумента, по адресу [esp+12] – второго и т.д) (рис. 4.28).

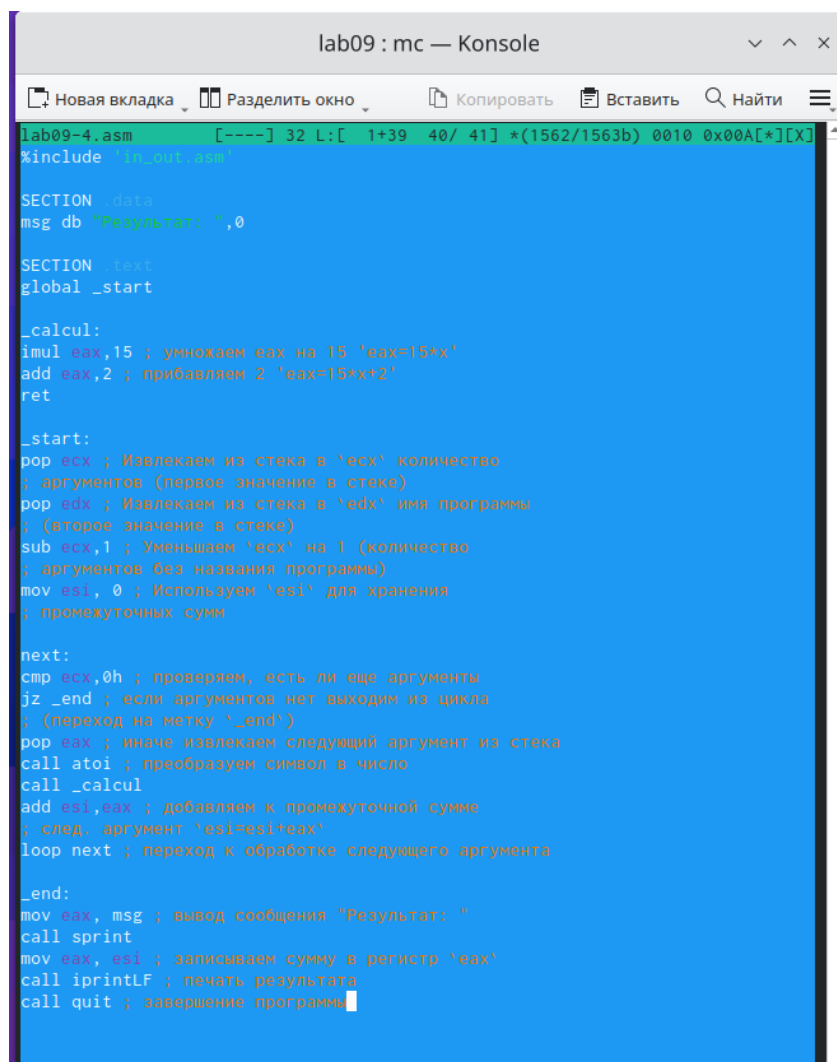
```
(gdb) x/x $esp
0xfffff1a0: 0x00000005
(gdb) x/s *(void**)(esp + 4)
0xfffff433: "/afs/.dk.sci.pfu.edu.ru/home/a/m/amcherkashina/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)(esp + 8)
0xfffff47d: "аргумент1"
(gdb) x/s *(void**)(esp + 12)
0xfffff48f: "аргумент"
(gdb) x/s *(void**)(esp + 16)
0xfffff4a0: "2"
(gdb) x/s *(void**)(esp + 20)
0xfffff4a2: "аргумент 3"
(gdb) x/s *(void**)(esp + 24)
0x0: <error: Cannot access memory at address 0x0>
(gdb)
```

Рис. 4.28: Просмотр значений, введенных в стек

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

4.6 Выполнение заданий для самостоятельной работы

1. Копирую файл lab8-4.asm, созданный при выполнении самостоятельного задания лабораторной работы №8, в файл с именем lab09-4.asm Преобразовываю программу из лабораторной работы №8 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции $f(x)$ как подпрограмму. У меня был 11 вариант: $f(x) = 15 \cdot x + 2$ (рис. 4.29).



```
lab09-4.asm [----] 32 L: [ 1+39 40/ 41] *(1562/1563b) 0010 0x00A[*][X]
#include 'in_out.asm'

SECTION .data
msg db "Результат: ",0

SECTION .text
global _start

_calcul:
imul eax,15 ; умножаем eax на 15 'eax=15*x'
add eax,2 ; прибавляем 2 'eax=15*x+2'
ret

_start:
pop ecx ; Извлекаем из стека в 'ecx' количество
; аргументов (первое значение в стеке)
pop edx ; Извлекаем из стека в 'edx' имя программы
; (второе значение в стеке)
sub ecx,1 ; Уменьшаем 'ecx' на 1 (количество
; аргументов без названия программы)
mov esi, 0 ; Используем 'esi' для хранения
; промежуточных сумм

next:
cmp ecx,0h ; проверяем, есть ли еще аргументы
jz _end ; если аргументов нет выходим из цикла
; (переход на метку '_end')
pop eax ; иначе извлекаем следующий аргумент из стека
call atoi ; преобразуем символ в число
call _calcul
add esi,eax ; добавляем к промежуточной сумме
; след. аргумент 'esi=esi+eax'
loop next ; переход к обработке следующего аргумента

_end:
mov eax, msg ; вывод сообщения "Результат: "
call sprint
mov eax, esi ; записываем сумму в регистр 'eax'
call iprintLF ; печать результата
call quit ; завершение программы
```

Рис. 4.29: Написание кода программы

Листинг 4.1. Программа нахождения суммы значений функции $f(x)=15*x+2$

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

```
SECTION .data
```

```
msg db "Результат: ", 0
```

```
SECTION .text
```

```

global _start

_calcul:
    imul eax, 15    ; умножаем eax на 15 'eax=15*x'
    add  eax, 2     ; прибавляем 2 'eax=15*x+2'
    ret

_start:
    pop ecx        ; Извлекаем из стека в ecx количество
                  ; аргументов (первое значение в стеке)
    pop edx        ; Извлекаем из стека в edx имя программы
                  ; (второе значение в стеке)
    sub  ecx, 1     ; Уменьшаем ecx на 1 (количество
                  ; аргументов без названия программы)
    mov  esi, 0     ; Используем esi для хранения
                  ; промежуточных сумм

next:
    cmp  ecx, 0h    ; проверяем, есть ли еще аргументы
    jz   _end       ; если аргументов нет, выходим из цикла
                  ; (переход на метку _end)
    pop  eax        ; иначе извлекаем следующий аргумент из стека
    call atoi       ; преобразуем символ в число
    call _calcul    ; вызываем подпрограмму для вычисления f(x)
    add  esi, eax    ; добавляем значение функции для
                  ; конкретного аргумента к промежуточной сумме
    loop next       ; переход к обработке следующего аргумента

_end:

```

```
mov eax, msg      ; вывод сообщения "Результат: "  
call sprint  
mov eax, esi      ; записываем сумму в регистр eax  
call iprintLF     ; печать результата  
call quit         ; завершение программы
```

5 Выводы

В ходе 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. Архитектура ЭВМ