

РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ

по лабораторной работе № 10

дисциплина: Архитектура компьютера

Студент: Грязнов Михаил

Группа: НПИбд-01-22

МОСКВА

2022 г.

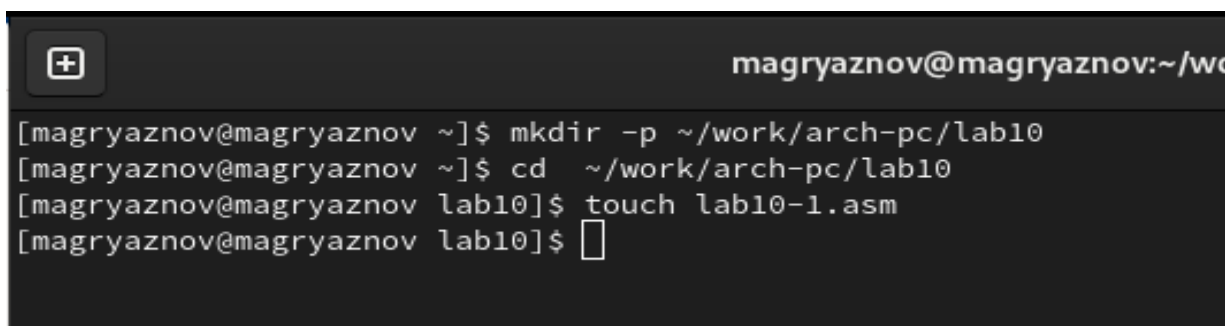
Цель работы:

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

Порядок выполнения лабораторной работы: mcedit

Реализация подпрограмм в NASM.

Создадим каталог для выполнения лабораторной работы № 10, перейдем в него и создадим файл lab10-1.asm:



```
magryaznov@magryaznov:~/work/arch-pc/lab10$ mkdir -p ~/work/arch-pc/lab10
magryaznov@magryaznov:~/work/arch-pc/lab10$ cd ~/work/arch-pc/lab10
magryaznov@magryaznov:~/work/arch-pc/lab10$ touch lab10-1.asm
magryaznov@magryaznov:~/work/arch-pc/lab10$
```

В качестве примера рассмотрим программу вычисления арифметического выражения $f(x) = 2x + 7$ с помощью подпрограммы `_calcul`. В данном примере x вводится с клавиатуры, а само выражение вычисляется в подпрограмме. Внимательно изучите текст программы (рис. 1-3).

```
magryaznov@magryaznov:~/work
lab10-1.asm [----] 7 L: [ 1+32 33/ 33] *(404 / 404)
#include 'in_out.asm'
SECTION .data
    msg:<----->DB 'Enter x: ', 0
    result:<---->DB '2x+7= ', 0
SECTION .bss
    x:<>RESB 80
    rez:<----->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,[rez]
    call iprintLF
    call quit

_calcul:
    mov ebx,2
    mul ebx
    add eax,7
    mov [rez],eax
    ret
```

Рис. 1. Пример программы с использованием вызова подпрограммы

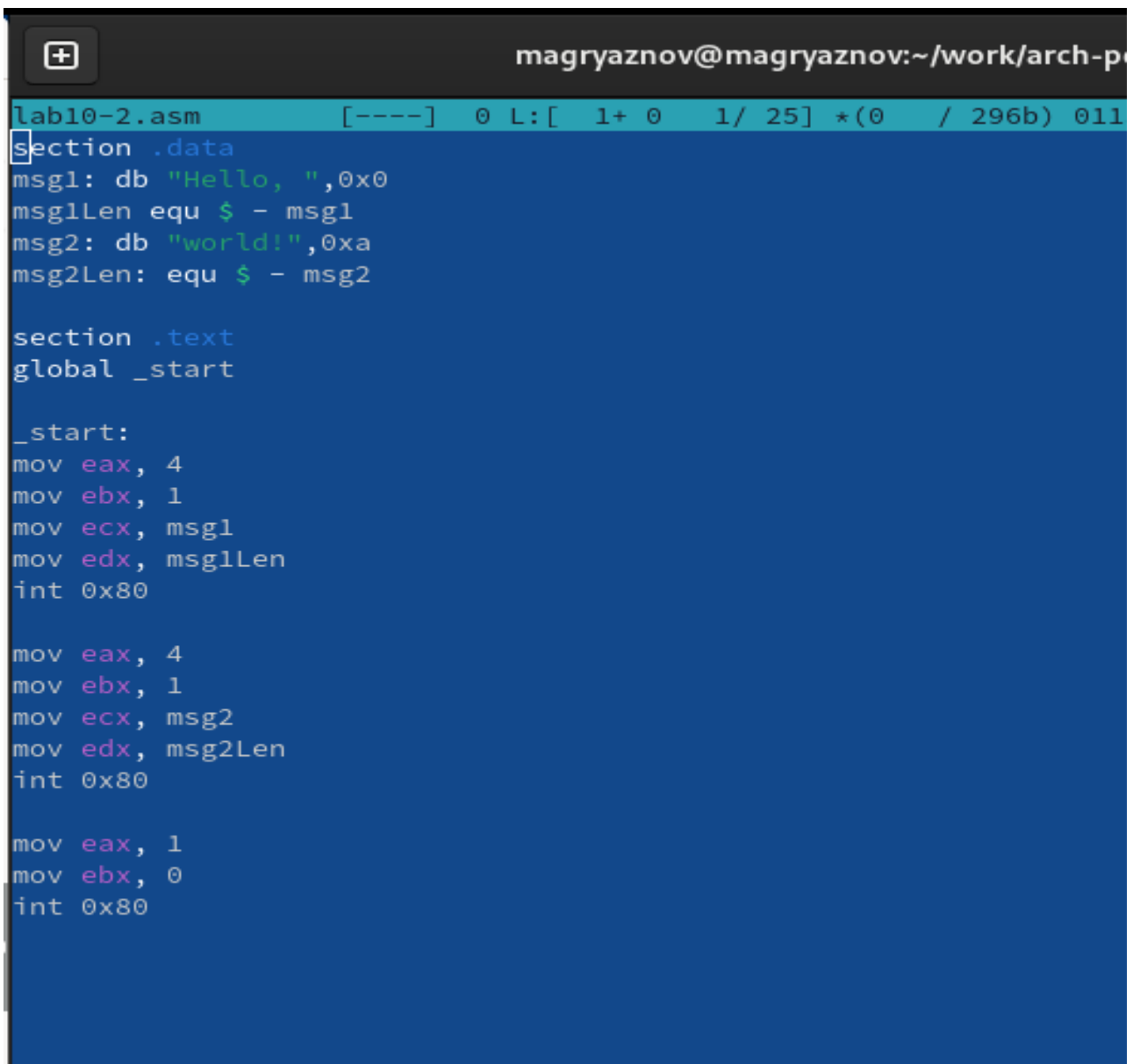
Первые строки программы отвечают за вывод сообщения на экран (call sprint), чтение данных введенных с клавиатуры (call sread) и преобразования введенных данных из символьного вида в численный (call atoi).

После следующей инструкции call _calcul, которая передает управление подпрограмме _calcul, будут выполнены следующие инструкции подпрограммы

Инструкция `ret` является последней в подпрограмме и ее исполнение приводит к возвращению в основную программу к инструкции, следующей за инструкцией `call`, которая вызвала данную подпрограмму. Последние строки программы реализуют вывод сообщения (`call sprint`), результата вычисления (`call iprintLF`) и завершение программы (`call quit`). Введите в файл `lab10-1.asm` текст программы из листинга 10.1. Создайте исполняемый файл и проверьте его работу. Измените 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 $f(g(x))$. Результат возвращается в основную программу для вывода результата на экран.

Отладка программ с помощью GDB.

Создадим файл `lab10-2.asm` с текстом программы из рис. 4 (Программа печати сообщения `Hello world!`).



```
magryaznov@magryaznov:~/work/arch-p  
lab10-2.asm [----] 0 L: [ 1+ 0 1/ 25] *(0 / 296b) 011  
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. Текст программы печати сообщения Hello world!

Получим исполняемый файл. Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию программ необходимо проводить с ключом '-g'. Затем загрузим исполняемый файл в отладчик gdb. И проверим работу программы, запустив ее в оболочке GDB с помощью команды run (сокращённо r) (рис. 5).

```

[magryaznov@magryaznov lab10]$ mcedit lab10-2.asm

[magryaznov@magryaznov lab10]$
[magryaznov@magryaznov lab10]$ nasm -f elf lab10-2.asm -l lab10-2.lst
[magryaznov@magryaznov lab10]$ ld -m elf_i386 lab10-2.o -o lab10-2
[magryaznov@magryaznov lab10]$ gdb lab10-2
GNU gdb (GDB) Fedora 12.1-1.fc35
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab10-2...
(No debugging symbols found in lab10-2)
(gdb) run
Starting program: /home/magryaznov/work/arch-pc/lab10/lab10-2

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n]) y

```

Рис. 5. Результат работы программы

Для более подробного анализа программы установим брейкпоинт на метку `_start`, с которой начинается выполнение любой ассемблерной программы, и запустим её (рис. 6).

```

^C Cancelling download of separate debug info for system-supplied DSO at 0xf7ffc000...
Hello, world!
[Inferior 1 (process 9947) exited normally]
(gdb) break _start
Breakpoint 1 at 0x8049000
(gdb) run
Starting program: /home/magryaznov/work/arch-pc/lab10/lab10-2

Breakpoint 1, 0x08049000 in _start ()
(gdb) █
%
```

Рис. 6. Breakpoint

Посмотрим дисассимилированный код программы с помощью команды `disassemble`, начиная с метки `_start` (рис. 7).

```
Breakpoint 1, 0x08049000 in _start ()
(gdb)
(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. 7. Дисассимилированный код программы

Переключимся на отображение команд с Intel'овским синтаксисом, введя команду `set disassembly-flavor intel` (рис. 8).

```
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:      mov     eax,0x4
    0x08049005 <+5>:      mov     ebx,0x1
    0x0804900a <+10>:     mov     ecx,0x804a000
    0x0804900f <+15>:     mov     edx,0x8
    0x08049014 <+20>:     int     0x80
    0x08049016 <+22>:     mov     eax,0x4
    0x0804901b <+27>:     mov     ebx,0x1
    0x08049020 <+32>:     mov     ecx,0x804a008
    0x08049025 <+37>:     mov     edx,0x7
    0x0804902a <+42>:     int     0x80
    0x0804902c <+44>:     mov     eax,0x1
    0x08049031 <+49>:     mov     ebx,0x0
    0x08049036 <+54>:     int     0x80
End of assembler dump.
(gdb) □
```

Рис. 8. Дисассимилированный код программы

Включим режим псевдографики для более удобного анализа программы (рис. 9-10).

(gdb) layout asm

(gdb) layout regs

```
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 <_start+56>     add     BYTE PTR [eax],al
0x804903a <_start+58>     add     BYTE PTR [eax],al
[ 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 <_start+56>     add     BYTE PTR [eax],al
0x804903a <_start+58>     add     BYTE PTR [eax],al
```

native process 9956 In: _start

(gdb) layout regs

(gdb)

Рис. 9-10. Режим псевдографики

Добавление точек останова.

Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать или как номер строки программы (имеет смысл, если есть исходный файл, а программа компилировалась с информацией об отладке), или как имя метки, или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка».

На предыдущих шагах была установлена точка останова по имени метки (`_start`). Проверим это с помощью команды `info breakpoints` (кратко `i b`) (рис. 10).

```
Breakpoint 1 at 0x8049000
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint      keep y   0x08049000  <_start>
(gdb) █
```

рис. 10. Точка останова `_start`

Установим еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции. Определим адрес предпоследней инструкции (`mov ebx,0x0`), установим точку останова и посмотрим информацию о всех установленных точках останова

Работа с данными программы в GDB.

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Выполним 5 инструкций с помощью команды `stepi` (или `si`) и проследите за изменением значений регистров.

Для отображения содержимого памяти можно использовать команду `x`, которая выдаёт содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: `x/NFU`. С помощью команды `x &` также можно посмотреть содержимое переменной. Посмотрим значение переменной `msg1` по имени (рис. 12)

```
(gdb) x/1sb &msg1
```

```
0x804a000 : "Hello, "
```

```
Breakpoint 1, 0x08049000 in _start ()
(gdb) si
0x08049005 in _start ()
(gdb) x/1sb &msg1
0x804a000:      "Hello, "
(gdb) █
```


Рис. 12. Значение переменной *msg1*

Посмотрим значение переменной *msg2* по адресу. Адрес переменной можно определить по дизассемблированной инструкции. Посмотрим инструкцию `mov ecx,msg2` которая записывает в регистр *ecx* адрес переменной *msg2*

```
(gdb) x/lsb 0x804a000
0x804a000:      "Hello, "
(gdb) x/lsb 0x804a008
0x804a008:      "world!\n"
(gdb) □
```

Рис. 13. Значение переменной *msg2*

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). Изменим первый символ переменной *msg1* (рис. 14).

```
(gdb)
0x804a013:      ""
(gdb) set {char}&msg1='h'
(gdb) set {char}0x804a001='h'
(gdb) x/lsb &msg1
0x804a000:      "hhlllo, "
(gdb) □
```

Рис. 14. Примеры использования команды *set*

Обработка аргументов командной строки в GDB.

Скопируем файл `lab9-2.asm`, созданный при выполнении лабораторной работы №9, с программой выводящей на экран аргументы командной строки в файл с именем `lab10-3.asm`:

```
cp ~/work/arch-pc/lab09/lab9-2.asm ~/work/arch-pc/lab10/lab10-3.asm
```

Создадим исполняемый файл:

```
nasm -f elf -g -l lab10-3.lst lab10-3.asm
```

```
ld -m elf_i386 lab10-3.o -o lab10-3
```

Для загрузки в `gdb` программы с аргументами необходимо использовать ключ `--args`. Загрузим исполняемый файл в отладчик, указав аргументы:

```
gdb --args lab10-3 аргумент1 аргумент 2 'аргумент 3'
```

Для начала установим точку останова перед первой инструкцией в программе и запустим ее.

```
(gdb) b _start
```

```
(gdb) run
```

Адрес вершины стека храниться в регистре `esp` и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы):

```
(gdb) x/x $esp
```

```
0xffffd200: 0x05
```

Как видно, число аргументов равно 5 – это имя программы `lab10-3` и непосредственно аргументы: `аргумент1`, `аргумент, 2` и `'аргумент 3'`

Порядок выполнения самостоятельной работы:

На рис. 15 приведена программа вычисления выражения $3x-1$. При запуске данная программа дает неверный результат. Проверьте это. С помощью отладчика GDB, анализируя изменения значений регистров, определите ошибку и исправьте ее.

```
magryaznov@magryaznov:~/work/arch-pc/lab10 — mcedit lab10-4.asm
lab10-4.asm  [----]  9 L:[ 1+10 11/ 23] *(127 / 242b) 0010 0x00A
#include 'in_out.asm'

section .data
div: db 'Результат: ',0

section .text
global _start
_start:

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

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

call quit
```

Рис. 15. Программа с ошибкой

```
[magryaznov@magryaznov lab10]$ ls
in_out.asm  lab10-1.asm  lab10-2  lab10-2.asm  lab10-2.lst  lab10-2.o  lab10-4.asm
[magryaznov@magryaznov lab10]$ mcedit lab10-4.asm

[magryaznov@magryaznov lab10]$ nasm -f elf lab10-4.asm
[magryaznov@magryaznov lab10]$ ld -m elf_i386 lab10-4.o -o lab10-4
[magryaznov@magryaznov lab10]$ ./lab10-4
Результат: 25
[magryaznov@magryaznov lab10]$
```

Рис. 16 Результат работы программы

Вывод:

Во время выполнения лабораторной работы были приобретены навыки написания программ с использованием подпрограмм. Также были изучены методы отладки при помощи GDB и его основные возможности.