

Отчет по лабораторной работе №9

Дисциплина: Архитектура компьютера

Мутаев Муртазаали Магомедович

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	7
3.1	Реализация подпрограмм в NASM	7
3.2	Отладка программ с помощью GDB	9
3.2.1	Работа с данными программы в GDB	14
3.2.2	Обработка аргументов командной строки в GDB	16
3.3	Задания для самостоятельной работы	18
4	Выводы	21

Список иллюстраций

3.1	Листинг 9.1	7
3.2	Результат Листинга 9.1	8
3.3	Измененный Листинг 9.1	8
3.4	Результат измененного Листинга 9.1	8
3.5	Листинг 9.2	9
3.6	Отладчик gdb	10
3.7	Breakpoint	10
3.8	Дисассимпilirованный код АТТ	11
3.9	Дисассимпilirованный код Intel	11
3.10	Псевдографика asm	12
3.11	Псевдографика regs	12
3.12	Точки останова	13
3.13	Установка нового брейкпоинта	13
3.14	Информация о брейкпоинтах	14
3.15	Содержимое регистров	14
3.16	Значение переменной msg1	15
3.17	Значение переменной msg2	15
3.18	Изменение символа msg1	15
3.19	Изменение символа msg2	15
3.20	Print	16
3.21	Загрузка программы с аргументами в GDB	16
3.22	Изучение стека	17
3.23	Самостоятельная работа 1	18
3.24	Результат самостоятельная работа 1	18
3.25	Работа листинга 9.3	19
3.26	Step 1	19
3.27	Step 2	19
3.28	Step 3	19
3.29	Step 4	20
3.30	Самостоятельная работа 2	20
3.31	Результат самостоятельная работа 2	20

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
 1. Работа с данными программы в GDB
 2. Обработка аргументов командной строки в GDB
3. Задания для самостоятельной работы

3 Выполнение лабораторной работы

3.1 Реализация подпрограмм в NASM

По базе сначала создаем файл lab9-1.asm в каталоге work/arch-pc/lab09

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. Воспользуемся кодом из Листинга 9.1 (рис. 3.1):

```
1 %include 'in_out.asm'
2 SECTION .data
3 msg: DB 'Введите x: ',0
4 result: DB '2x+7=',0
5 SECTION .bss
6 x: RESB 80
7 res: RESB 80
8 SECTION .text
9 GLOBAL _start
10 _start:
11 ;-----
12 ; Основная программа
13 ;-----
14 mov eax, msg
15 call sprint
16 mov ecx, x
17 mov edx, 80
18 call sread
19 mov eax, x
20 call atoi
21 call _calcul ; Вызов подпрограммы _calcul
22 mov eax, result
23 call sprint
24 mov eax, [res]
25 call iprintf
26 call quit
27 ;-----
28 ; Подпрограмма вычисления
29 ; выражения "2x+7"
30 _calcul:
31 mov ebx, 2
32 mul ebx
33 add eax, 7
34 mov [res], eax
35 ret ; выход из подпрограммы
```

Рис. 3.1: Листинг 9.1

Вот такой результат у меня получился (рис. 3.2):

```

mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-1 lab09-1.o
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ ./lab09-1
Введите x: 2
2x+7=11
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ 

```

Рис. 3.2: Результат Листинга 9.1

Попробуем изменить 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.

Вот такая программа у меня получилась (рис. 3.3):

```

1 %include 'in_out.asm'
2 SECTION .data
3 msg: DB "Введите x: ",0
4 result: DB "2(3x-1)+7=",0
5 SECTION .bss
6 x: RESB 80
7 res: RESB 80
8 SECTION .text
9 GLOBAL _start
10 _start:
11 ;-----
12 ; Основная программа
13 ;-----
14 mov eax, msg
15 call sprint
16 mov ecx, x
17 mov edx, 80
18 call sread
19 mov eax, x
20 call atoi
21 call _calcul ; Вызов подпрограммы _calcul
22 mov eax, result
23 call sprint
24 mov eax, [res]
25 call iprintLF
26 call quit
27 ;-----
28 ; Подпрограмма вычисления
29 ; выражения "2x+7"
30 _calcul:
31 call _subcalcul
32 mov ebx, 2
33 mul ebx
34 add eax, 7
35 mov [res], eax
36 ret ; выход из подпрограммы
37
38 _subcalcul:
39 mov ebx, 3
40 mul ebx
41 sub eax, 1
42 mov [res], eax
43 ret ; выход из подпрограммы

```

Рис. 3.3: Измененный Листинг 9.1

У меня получился такой результат (рис. 3.4):

```

mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-1 lab09-1.o
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ ./lab09-1
Введите x: 2
2(3x-1)+7=17
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ 

```

Рис. 3.4: Результат измененного Листинга 9.1

Как мы видим, результат получился верным, значит программа написана правильно.

3.2 Отладка программ с помощью GDB

Воспользуемся кодом из Листинга 9.2 (программы для вывода Hello world!) (рис. 3.5):

```
1 SECTION .data
2 msg1: db "Hello, ",0x0
3 msg1Len: equ $ - msg1
4 msg2: db "world!",0xa
5 msg2Len: equ $ - msg2
6 SECTION .text
7 global _start
8 _start:
9 mov eax, 4
10 mov ebx, 1
11 mov ecx, msg1
12 mov edx, msg1Len
13 int 0x80
14 mov eax, 4
15 mov ebx, 1
16 mov ecx, msg2
17 mov edx, msg2Len
18 int 0x80
19 mov eax, 1
20 mov ebx, 0
21 int 0x80
```

Рис. 3.5: Листинг 9.2

Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию программ необходимо проводить с ключом ‘-g’. После этого я загрузил исполняемый файл в отладчик gdb и запустил его (рис. 3.6):

```

mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ touch lab09-2.asm
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ nasm -f elf -g -l lab09-2.lst lab09-2.asmmmutaev@dk8n64 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-2 lab09-2.o
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ gdb lab09-2
GNU gdb (Gentoo 14.2 vanilla) 14.2
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) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/mmmutaev/work/arch-pc/lab09/lab09-2
Hello, world!

```

Рис. 3.6: Отладчик gdb

Для более подробного анализа программы установим брейкпоинт на метку `_start`, с которой начинается выполнение любой ассемблерной программы, и запустим её (рис. 3.7):

```

[Inferior 1 (process 4877) exited normally]
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) r
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/mmmutaev/work/arch-pc/lab09/lab09-2

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

```

Рис. 3.7: Breakpoint

Далее посмотрим дисассимилированный код программы с помощью команды `disassemble` начиная с метки `_start` (рис. 3.8):

```

(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. 3.8: Дисассимпированный код АТТ

Переключимся на отображение команд с Intel'овским синтаксисом, введя команду `set disassembly-flavor intel` (рис. 3.9):

```

(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. 3.9: Дисассимпированный код Intel

Есть некоторые различия в отображениях в этих режимах, а именно в виде ко-

лонки с текстом программы: в АТТ'е она выглядит, как “\$0x{операнд},{%{регистр}}”, а в Intel - “{регистр},0x{операнд}”

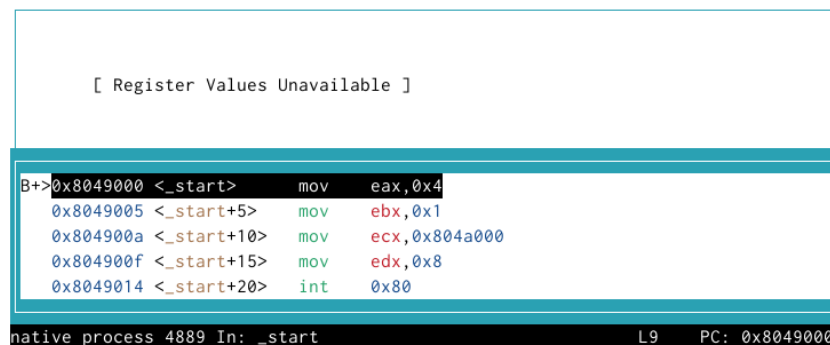
Теперь включим режим псевдографики для более удобного анализа программы:



```
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

native process 4889 In: _start                                L9    PC: 0x8049000
```

Рис. 3.10: Псевдографика asm



```
[ 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

native process 4889 In: _start                                L9    PC: 0x8049000
```

Рис. 3.11: Псевдографика regs

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. 3.12):

```
(gdb) i b
Num      Type      Disp Enb Address      What
1        breakpoint keep y  0x08049000 lab09-2.asm:9
breakpoint already hit 1 time
(gdb) 
```

Рис. 3.12: Точки останова

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`) и установим точку останова (рис. 3.13):

```

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
0x804903a              add    BYTE PTR [eax],al
native process 4533 In: _start L9
(gdb) layout regs
(gdb) i b
Num      Type      Disp Enb Address      What
1        breakpoint keep y  0x08049000 lab09-2.asm:9
breakpoint already hit 1 time
(gdb) break *0x8049031
Breakpoint 2 at 0x8049031: file lab09-2.asm, line 20.
(gdb) 
```

Рис. 3.13: Установка нового брейкпоинта

Адрес инструкции находился в средней части экрана в левом столбце соответствующей инструкции. Можно заметить, что слева от адреса появился значок *b+*. Вероятно он означает, что здесь поставлен брейкпоинт.

Теперь снова посмотрим информацию о всех установленных точках останова (рис. 3.14):

```

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
0x804903a              add     BYTE PTR [eax],al

native process 4533 In: _start L9
(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   0x08049000 lab09-2.asm:9
          breakpoint already hit 1 time
2        breakpoint      keep y   0x08049031 lab09-2.asm:20
(gdb) 

```

Рис. 3.14: Информация о брейкпоинтах

Как мы видим, информация обновилась.

3.2.1 Работа с данными программы в GDB

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Посмотрим содержимое регистров также можно с помощью команды `info registers` (рис. 3.15):

```

edx      0x0      0
ebx      0x0      0
esp      0xffffc480 0xffffc480
ebp      0x0      0x0
esi      0x0      0
--Type <RET> for more, q to quit, c to continue without paging--
Quit
(gdb) 

```

Рис. 3.15: Содержимое 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. 3.16):

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

Рис. 3.16: Значение переменной msg1

Посмотрим значение переменной msg2 по адресу. Адрес переменной можно определить по дизассемблированной инструкции. Посмотрите инструкцию mov ecx,msg2 которая записывает в регистр ecx адрес переменной msg2 (рис. 3.17):

```
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n"<error: Cannot access memory at address 0x804a00f>
(gdb) 
```

Рис. 3.17: 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. 3.18):

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

Рис. 3.18: Изменение символа msg1

Заменим любой символ в переменной msg2 (рис. 3.19):

```
(gdb) set {char}0x804a00d='?'
(gdb) x/1sb &msg2
0x804a008 <msg2>:      "World?\n\034"
(gdb) 
```

Рис. 3.19: Изменение символа msg2

Поздравляю, теперь наша программа задает вопрос “hello, World?”

Далее изменим значение регистра `ebx` и выведем Обработка аргументов командной строки в GDB его значение с помощью `print` (рис. 3.20):

```
(gdb) set $ebx='2'
(gdb) p/s $ebx
$9 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$10 = 2
(gdb) 
```

Рис. 3.20: Print

Можно заметить разницу, в зависимости от введенного значения. В первом случае мы в `ebx` записываем символ “2”, поэтому принт выводит номер этого символа в таблице ASCII, а во втором случае мы присваиваем `ebx` значение 2, поэтому он выводит 2.

3.2.2 Обработка аргументов командной строки в GDB

Скопируем файл `lab8-2.asm` в нашу папку и назовем ее `lab09-3.asm`. Создадим исполняемый файл и загрузим в `gdb` программу. Для загрузки в `gdb` программы с аргументами необходимо использовать ключ `-args` (рис. 3.21):

```
mmmtaev@dk8n64 ~/work/arch-pc/lab09 $ gdb --args lab09-3 аргумент1 аргумент 2 '
аргумент 3'
GNU gdb (Gentoo 14.2 vanilla) 14.2
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. 3.21: Загрузка программы с аргументами в GDB

Как отмечалось в предыдущей лабораторной работе, при запуске программы аргументы командной строки загружаются в стек. Исследуем расположение аргументов командной строки в стеке после запуска программы с помощью gdb. Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки (рис. 3.22)

Как видно, число аргументов равно 5 – это имя программы lab09-3 и непосредственно аргументы: аргумент1, аргумент, 2 и ‘аргумент 3’. Посмотрим остальные позиции стека – по адресу [esp+4] располагается адрес в памяти где находится имя программы, по адресу [esp+8] храниться адрес первого аргумента, по адресу [esp+12] – второго и т.д

```
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 5.
(gdb) r
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/m/mmmuteaev/work/arch-pc/lab09/1
ab09-3 аргумент1 аргумент 2 аргумент\ 3

Breakpoint 1, _start () at lab09-3.asm:5
5      pop ecx ; Извлекаем из стека в `ecx` количество
(gdb) x/x $esp
0xffffc430: 0x00000005
(gdb) x/s *(void**)(esp+4)
0xffffc68b: "/afs/.dk.sci.pfu.edu.ru/home/m/m/mmmuteaev/work/arch-pc/lab09/1a
b09-3"
(gdb) x/s *(void**)(esp+8)
0xffffc6d0: "аргумент1"
(gdb) x/s *(void**)(esp+12)
0xffffc6e2: "аргумент"
(gdb) x/s *(void**)(esp+16)
0xffffc6f3: "2"
(gdb) x/s *(void**)(esp+20)
0xffffc6f5: "аргумент 3"
(gdb) x/s *(void**)(esp+24)
0x0: <error: Cannot access memory at address 0x0>
(gdb) □
```

Рис. 3.22: Изучение стека

Предполагаю, что шаг изменения адреса равен 4, потому что на каждый аргумент выделено 4 байта.

3.3 Задания для самостоятельной работы

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

Легко! Просто переносим часть с нахождением значения $f(x)$ в подпрограмму. Вот код, который у меня получился (рис. 3.23):

```
6 global _start
7 _start:
8
9     pop ecx
10    pop edx
11    sub ecx, 1
12
13    mov esi, 0
14
15next:
16    cmp ecx, 0h
17    jz _end
18    pop eax
19    call atoi
20    call f
21    add esi, eax
22    loop next
23
24_end:
25    mov eax, msg1
26    call sprintf
27    mov eax, msg2
28    call sprintf
29    mov eax, esi
30    call sprintf
31    call quit
32
33f:
34    add eax, 10
35    mov ebx, 3
36    mul ebx
37    ret
```

Рис. 3.23: Самостоятельная работа 1

А вот результат (рис. 3.24):

```
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ nasm -f elf SR1.asm
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o SR1 SR1.o
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ ./SR1 1 2 3
Функция f(x) = 3(10+x)
Результат: 108
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $
```

Рис. 3.24: Результат самостоятельная работа 1

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

Ответ действительно неправильный: программа должна выводить 25, а выводит 10

```

Reading symbols from SR2...
(gdb) r
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/m/mmmutaev/work/arch-pc/lab09/S
R2
Результат: 10
[Inferior 1 (process 8218) exited normally]
(gdb) 

```

Рис. 3.25: Работа листинга 9.3

Так как в результат записывается значение регистра ebx, отследим его изменение и изменение других регистров. Изначально ebx = 3

Register group: general		
eax	0x0	0
ecx	0x0	0
edx	0x0	0
ebx	0x3	3
esp	0xffffc480	0xffffc480
ebp	0x0	0x0

Рис. 3.26: Step 1

Далее мы прибавили к ebx eax, и у нас ebx = 5

Register group: general		
eax	0x2	2
ecx	0x0	0
edx	0x0	0
ebx	0x5	5
esp	0xffffc480	0xffffc480
ebp	0x0	0x0

Рис. 3.27: Step 2

Далее мы присвоили ecx значение 4 и умножили eax на ecx. Возможно, в этом и есть ошибка

Register group: general		
eax	0x8	8
ecx	0x4	4
edx	0x0	0
ebx	0x5	5
esp	0xffffc480	0xffffc480
ebp	0x0	0x0

0x80490f2	<_start+10>	add	%eax,%ebx
0x80490f4	<_start+12>	mov	\$0x4,%ecx
0x80490f9	<_start+17>	mul	%ecx
>0x80490fb	<_start+19>	add	\$0x5,%ebx
0x80490fe	<_start+22>	mov	%ebx,%edi
0x8049100	<_start+24>	mov	\$0x804a000,%eax

Рис. 3.28: Step 3

Далее к ebx прибавили 5 (теперь ebx = 10) и записали его значение в edi

Register group: general		
eax	0x8	8
ecx	0x4	4
edx	0x0	0
ebx	0xa	10
esp	0xffffc480	0xffffc480
ebp	0x0	0x0

0x80490f4	<_start+12>	mov	\$0x4,%ecx
0x80490f9	<_start+17>	mul	%ecx
0x80490fb	<_start+19>	add	\$0x5,%ebx
>0x80490fe	<_start+22>	mov	%ebx,%edi
0x8049100	<_start+24>	mov	\$0x804a000,%eax
0x8049105	<_start+29>	call	0x804900f <sprint>

Рис. 3.29: Step 4

Пошагово проанализировав изменение значений регистров, я выяснил, что ошибкой является то, что мы умножили eax на 4, а должны были ebx. Попытаемся исправить эту ошибку. Вот такая программа у меня получилась (рис. 3.30):

```
1 %include 'in_out.asm'
2 SECTION .data
3 div: DB 'Результат: ',0
4 SECTION .text
5 GLOBAL _start
6 _start:
7 ; --- Вычисление выражения (3+2)*4+5
8 mov ebx,3
9 mov eax,2
10 add eax, ebx
11 mov ecx,4
12 mul ecx
13 add eax, 5
14 mov edi,eax
15 ; --- Вывод результата на экран
16 mov eax,div
17 call sprint
18 mov eax,edi
19 call iprintLF
20 call quit
```

Рис. 3.30: Самостоятельная работа 2

И вот такой результат (рис. 3.31):

```
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ nasm -f elf SR2.asm
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o SR2 SR2.o
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $ ./SR2
Результат: 25
mmmutaev@dk8n64 ~/work/arch-pc/lab09 $
```

Рис. 3.31: Результат самостоятельная работа 2

4 Выводы

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