

Лабораторная работа №9

Понятие подпрограммы.Отладчик GDB.

Четвергова Мария Викторовна

Содержание

1	Цель работы	5
2	Теоретическое введение	6
3	Выполнение лабораторной работы	10
3.1	Добавление точек останова	17
3.2	Работа с данными программы в GDB	19
3.3	Задание для самостоятельной работы	24
4	Выводы	30

Список иллюстраций

3.1	каталог для выполнения лабораторной работы № 9	10
3.2	программу вычисления арифметического выражения $2x + 7$. . .	11
3.3	листинг данной программы	12
3.4	листинг данной программы	13
3.5	результат работы изменённой программы	13
3.6	листинг данной программы	14
3.7	получение исполняемого файла	15
3.8	результат работы программы	15
3.9	запуск программы с брейкпойнтом на <code>_start</code>	15
3.10	запуск программы с брейкпойнтом на <code>_start</code>	16
3.11	запуск программы с брейкпойнтом на <code>_start</code>	16
3.12	Включение режима псевдографики	17
3.13	Проверка с помощью команды <code>info breakpoints</code>	18
3.14	установка одной точки по инструкции	19
3.15	просмотр содержимого регистров с помощью команды <code>info registers</code> .	20
3.16	просмотр значений переменных <code>msg1</code> и <code>msg2</code>	20
3.17	Изменение первого символ переменной <code>msg1</code>	21
3.18	Изменение первого символ переменной <code>msg2</code>	21
3.19	вывод в различных СС значения регистра <code>edx</code> и измените значение регистра <code>ebx</code>	22
3.20	завершение работы в режиме псевдографики	22
3.21	создание исполняемого файла и загрузка ключа <code>-args</code> и установка брейкпойнта перед первой функцией программы	23
3.22	просмотр аргументов под данными адресами	24
3.23	листинг программы до изменений	25
3.24	листинг программы после изменений	26
3.25	работа программа после внесённых изменений	26
3.26	запись листинга программы в редактор	27
3.27	работа данной программы	27
3.28	расстановка брейкпойнтов и рассмотрение каждого шага работы кода	28
3.29	устранение ошибки в листинге программы	29
3.30	Проверка работы исправленного файла	29

Список таблиц

1 Цель работы

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

2 Теоретическое введение

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.

Наиболее часто применяют следующие методы отладки: • создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных значений на экран — так называемые диагностические сообщения); • использование специальных программ-отладчиков. Отладчики позволяют управлять ходом выполнения программы, контролировать и изменять

данные. Это помогает быстрее найти место ошибки в программе и ускорить её исправление. Наиболее популярные способы работы с отладчиком — это использование точек останова и выполнение программы по шагам. Пошаговое выполнение — это выполнение программы с остановкой после каждой строчки, чтобы программист мог проверить значения переменных и выполнить другие действия. Точки останова — это специально отмеченные места в программе, в которых программа-отладчик приостанавливает выполнение программы и ждёт команд. Наиболее популярные виды точек останова: • Breakpoint — точка останова (остановка происходит, когда выполнение доходит до определённой строки, адреса или процедуры, отмеченной программистом); • Watchpoint — точка просмотра (выполнение программы приостанавливается, если программа обратилась к определённой переменной: либо считала её значение, либо изменила его). Точки останова устанавливаются в отладчике на время сеанса работы с кодом программы, т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы

GDB (GNU Debugger — отладчик проекта GNU) [1] работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования. GDB предлагает обширные средства для слежения и контроля за выполнением компьютерных программ. Отладчик не содержит собственного графического пользовательского интерфейса и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторонних графических надстроек, а кроме того, некоторые интегрированные среды разработки используют его в качестве базовой подсистемы отладки. Отладчик GDB (как и любой другой отладчик) позволяет увидеть, что происходит «внутри» программы в момент её выполнения или что делает программа в момент сбоя. GDB может выполнять следующие действия: • начать выполнение программы, задав всё, что может повлиять на её поведение; • остановить программу при указанных условиях; • исследовать, что случилось, когда программа остановилась

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 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 программе можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом -g.

Для продолжения остановленной программы используется команда continue (c) (gdb) с . 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

Чтобы посмотреть значения регистров используется команда print /F (сокращённо 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

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.

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

3 Выполнение лабораторной работы

1. Создайте каталог для выполнения лабораторной работы № 9, перейдите в него и создайте файл lab09-1.asm:

```
mvchetvergova@dk4n65 ~ $ cd ~/work/arch-pc
mvchetvergova@dk4n65 ~/work/arch-pc $ mkdir lab09
mvchetvergova@dk4n65 ~/work/arch-pc $ ls
lab04 lab05 lab06 lab07 lab08 lab09 lab10
mvchetvergova@dk4n65 ~/work/arch-pc $ touch lab09-1.asm
mvchetvergova@dk4n65 ~/work/arch-pc $
```

Рис. 3.1: каталог для выполнения лабораторной работы № 9

2. В качестве примера рассмотрим программу вычисления арифметического выражения $\text{X}(\text{X}) = 2\text{X} + 7$ с помощью подпрограммы `_calcul`. В данном примере `X` вводится с клавиатуры, а само выражение вычисляется в подпрограмме. Внимательно изучите текст программы (Листинг 9.1).

```

lab09-1.asm      [----] 0 L: [ 1+ 0 1/ 35] *(0 / 707b) 0037 0x025
#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 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. 3.2: программу вычисления арифметического выражения $2x + 7$

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 инструкции подпрограммы

```
mov ebx, 2 mul ebx add eax, 7 mov [res], eax ret
```

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 lab09-1.asm текст программы

из листинга 9.1. Создайте исполняемый файл и проверьте его работу.

```
mvchetvergova@dk4n65 ~/work/arch-pc/lab09 $  
mvchetvergova@dk4n65 ~/work/arch-pc/lab09 $ nasm -f elf lab09-1.asm  
mvchetvergova@dk4n65 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-1 lab09-1.o  
mvchetvergova@dk4n65 ~/work/arch-pc/lab09 $ ./lab09-1  
Введите x: 5  
2x+7=17
```

Рис. 3.3: листинг данной программы

Измените текст программы, добавив подпрограмму `_subcalcul` в подпрограмму `_calcul`, для вычисления выражения $\text{f}(\text{f}(\text{f}(x)))$, где x вводится с клавиатуры, $\text{f}(x) = 2x + 7$, $\text{f}(x) = 3x - 1$. Т.е. x передается в подпрограмму `_calcul` из нее в подпрограмму `_subcalcul`, где вычисляется выражение $\text{f}(x)$, результат возвращается в `_calcul` и вычисляется выражение $\text{f}(\text{f}(x))$. Результат возвращается в основную программу для вывода результата на экран.

```

lab09-1.asm [----] 18 L: [ 1+ 3 4/ 44] *(85 / 787b) 0041 0x029
#include "in_out.asm"
SECTION .data
msg: DB "Введите x: ",0
result: DB "2(3x-1)+7=",0
SECTION .bss
res: 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 iprintLF
    call quit
-----
    Подпрограмма вычисления
    выражения "2x+7"
calcul:
    call _subcalcul
    mov ebx, 2
    mul ebx
    add eax, 7
    jmp _end

subcalcul:
    mov ebx, 3
    mul ebx
    add eax, -1
end:

```

Рис. 3.4: листинг данной программы

```

mvchetvergova@dk8n67 ~/work/arch-pc/lab09 $ mcedit lab09-1.asm

mvchetvergova@dk8n67 ~/work/arch-pc/lab09 $ nasm -f elf lab09-1.asm
mvchetvergova@dk8n67 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-1 lab09-1.o
mvchetvergova@dk8n67 ~/work/arch-pc/lab09 $ ./lab09-1
Введите x: 4
2(3x-1)+7=29
mvchetvergova@dk8n67 ~/work/arch-pc/lab09 $ mcedit lab09-1.asm

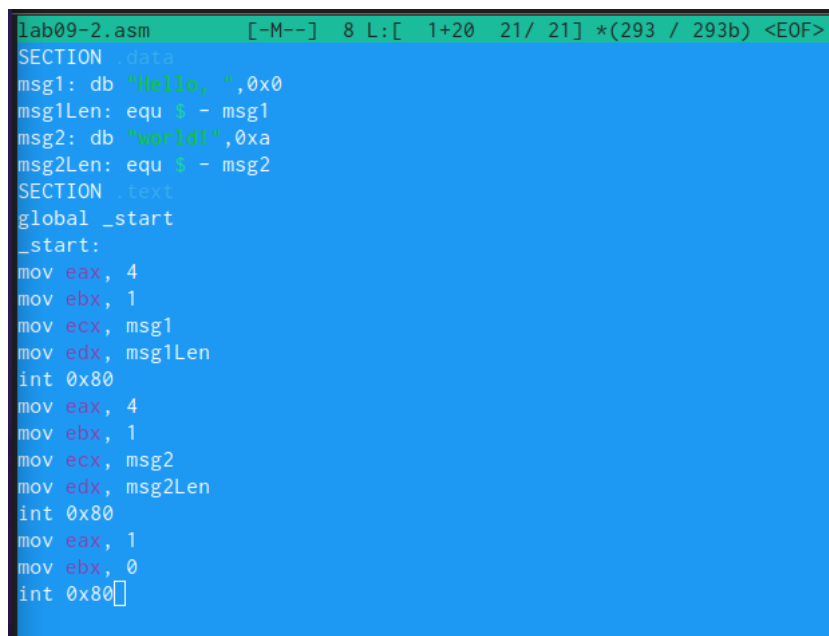
```

Рис. 3.5: результат работы изменённой программы

##Отладка программ с помощью GDB Создайте файл lab09-2.asm с текстом программы из Листинга 9.2. (Программа печати сообщения Hello world!)

Листинг 9.2. Программа вывода сообщения Hello world!

```
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
```



```
lab09-2.asm [-M--] 8 L: [ 1+20 21/ 21] *(293 / 293b) <EOF>
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. 3.6: листинг данной программы

Получите 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.

```

mvchetvergova@dk8n67 ~/work/arch-pc/lab09 $ nasm -f elf -g -l lab09-2.lst lab09-2.asm
mvchetvergova@dk8n67 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-2 lab09-2.o
mvchetvergova@dk8n67 ~/work/arch-pc/lab09 $ gdb lab09-2
GNU gdb (Gentoo 12.1 vanilla) 12.1
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-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. 3.7: получение исполняемого файла

Проверим работу программы, запустив ее в оболочке GDB с помощью команды `run` (со- кращённо `r`):

```

(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/v/mvchetvergova/work/arch-pc/lab09/lab09-2
Hello, world!
[Inferior 1 (process 4724) exited normally]
(gdb)

```

Рис. 3.8: результат работы программы

Для более подробного анализа программы установите брейкпоинт на метку `_start`, с которой начинается выполнение любой ассемблерной программы, и запустите её:

```

[Inferior 1 (process 4724) exited normally]
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/v/mvchetvergova/work/arch-pc/lab09/lab09-2

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

```

Рис. 3.9: запуск программы с брейкпоинтом на `_start`

Посмотрите дисассимилированный код программы с помощью команды `disassemble` начиная с метки `_start` и переключитесь на отображение команд с Intel'овским синтаксисом, введя команду `set`

```

(gdb) disassembly-flavor intel
Undefined command: "disassembly-flavor". Try "help".
(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. 3.10: запуск программы с брейкпоинтом на _start

```

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. 3.11: запуск программы с брейкпоинтом на _start

Перечислите различия отображения синтаксиса машинных команд в режимах АТТ и Intel. различия заключаются в записи последнего столбца: регистры и машинный код стоят на разных местах. также, в INTEL-овском варианте запись визуально воспринимается проще: машинный код выделен синим цветом и не сливается со столбиком регистров и в нём нет посторонних символов типа “%”

или “\$”.

Включите режим псевдографики для более удобного анализа программы (рис. 9.2):

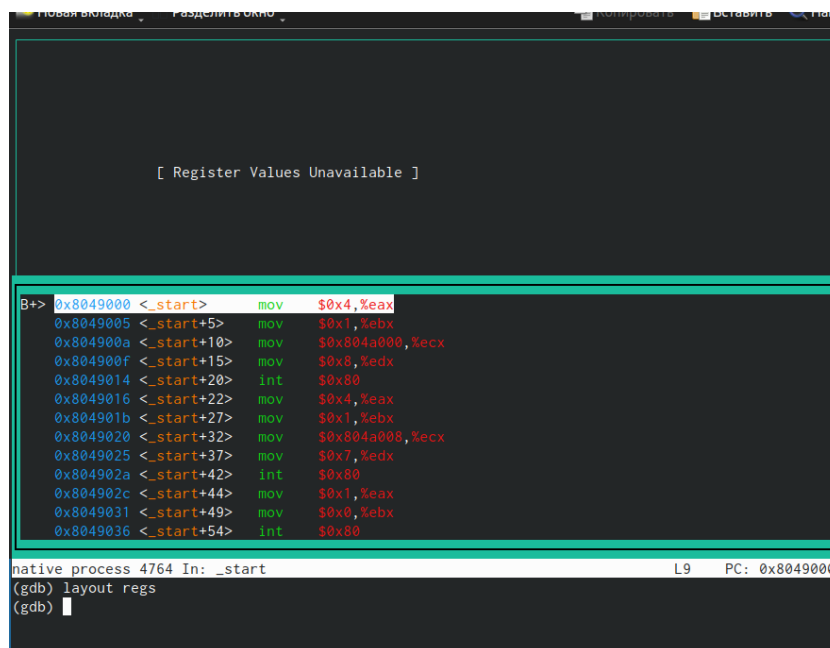


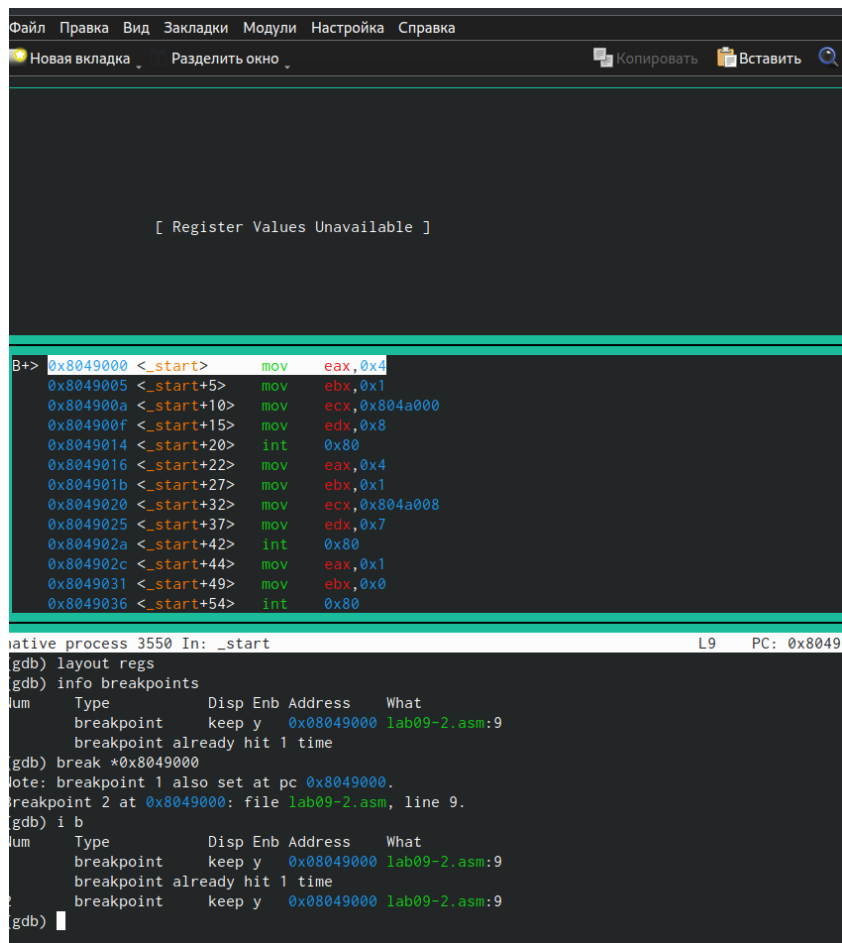
Рис. 3.12: Включение режима псевдографики

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.

3.1 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`). Про- верьте

это с помощью команды `info breakpoints` (кратко `i b`):



The screenshot shows a debugger window with a menu bar (Файл, Правка, Вид, Закладки, Модули, Настройка, Справка) and a toolbar (Новая вкладка, Разделить окно, Копировать, Вставить, search). The main area displays assembly code starting at address 0x8049000. Below the assembly code, the command prompt shows the execution of `gdb) info breakpoints`, which lists two breakpoints at address 0x8049000. The first breakpoint is at line 9 of `lab09-2.asm` and has been hit once. The second breakpoint is also at line 9 of `lab09-2.asm` and has also been hit once. The status bar at the bottom indicates the current process is 3550, the instruction pointer (PC) is 0x8049000, and the current instruction is at address 0x8049000.

```
gdb) info breakpoints
Num   Type             Disp Enb Address      What
-----
0      breakpoint         keep y  0x08049000 lab09-2.asm:9
      breakpoint already hit 1 time
gdb) break *0x8049000
Note: breakpoint 1 also set at pc 0x8049000.
Breakpoint 2 at 0x8049000: file lab09-2.asm, line 9.
gdb) i b
Num   Type             Disp Enb Address      What
-----
0      breakpoint         keep y  0x08049000 lab09-2.asm:9
      breakpoint already hit 1 time
1      breakpoint         keep y  0x08049000 lab09-2.asm:9
gdb)
```

Рис. 3.13: Проверка с помощью команды `info breakpoints`

Установим еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции (см. рис. 9.3). Определите адрес предпоследней инструкции (`mov ebx,0x0`) и установите точку останова. Посмотрите информацию о всех установленных точках останова:

The screenshot shows the GDB interface with a menu bar (Файл, Правка, Вид, Закладки, Модули, Настройка, Справка) and a toolbar (Новая вкладка, Разделить окно, Копировать, Вставить). The main window displays assembly code starting from address 0x8049000. The code includes instructions like `mov eax, 0x4`, `mov ebx, 0x1`, `mov ecx, 0x804a000`, `mov edx, 0x8`, `int 0x80`, `mov eax, 0x4`, `mov ebx, 0x1`, `mov ecx, 0x804a008`, `mov edx, 0x7`, `int 0x80`, `mov eax, 0x1`, `mov ebx, 0x0`, and `int 0x80`. The status bar at the bottom indicates 'native process 3550 In: _start' and 'L9 PC: 0x8049000'. The command window shows the following commands and output:

```
(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 *0x8049000
Note: breakpoint 1 also set at pc 0x8049000.
Breakpoint 2 at 0x8049000: file lab09-2.asm, line 9.
(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   0x08049000 lab09-2.asm:9
(gdb)
```

Рис. 3.14: установка одной точки по инструкции

3.2 Работа с данными программы в GDB

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Выполните 5 инструкций с помощью команды `stepi` (или `si`) и проследите за изменением значений регистров. Значения каких регистров изменяются? изменяются значения регистров определённых элементов. Посмотреть содержимое регистров также можно с помощью команды `info registers` (или `i r`).

```

native process 3550 In: _start
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffc2e0 0xffffc2e0
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
--Type <RET> for more, q to quit, c to continue without paging--

```

Рис. 3.15: просмотр содержимого регистров с помощью команды info registers

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. Посмотрите значение переменной `msg1` по имени и значение переменной `msg2` по адресу.

```

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

```

```

native process 3550 In: _start L9 PC: 0x8049000
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
gs       0x0      0
--Type <RET> for more, q to quit, c to continue without paging--cfs 0x0
(gdb) x/1sb &msg1
0x804a000 <msg1>: "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>: "world!\n\034"
(gdb)

```

Рис. 3.16: просмотр значений переменных `msg1` и `msg2`

Адрес переменной можно определить по дизассемблированной инструкции.

Посмотрите инструкцию `mov ecx,msg2` которая записывает в регистр `ecx` адрес 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). Измените первый символ переменной `msg1`

```
(gdb) x/1sb &msg1
0x804a000 <msg1>: "Hello, "
(gdb) set{char}&msg1 = "h"
Unterminated string in expression.
(gdb) set{char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>: "hello, "
(gdb) █
```

Рис. 3.17: Изменение первого символ переменной `msg1`

Заменяем первый символ во второй переменной `msg2`

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

Рис. 3.18: Изменение первого символ переменной `msg2`

Чтобы посмотреть значения регистров используется команда `print` (перед именем регистра обязательно ставится префикс `$`) Выведете в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра `edx`. С помощью команды `set` измените значение регистра `ebx`:

```

(gdb) p/t $ebx
$2 = 110010
(gdb) p/x $ecx
$3 = 0x0
(gdb) set $ebx=2
(gdb) p/s $ebx
$4 = 2
(gdb)

```

Рис. 3.19: вывод в различных СС значения регистра `edx` и измените значение регистра `ebx`

Объясните разницу вывода команд `p/s $ebx`: Через `p/s` выводится только значение регистра и его значение в указанной СС. При выводе через `x/1sb` выводится адрес переменной, её название и машинный код.

Завершите выполнение программы с помощью команды `continue` (сокращенно `c`) или `stepi` (сокращенно `si`) и выйдите из GDB с помощью команды `quit` (сокращенно `q`).

```

(gdb) set $ebx=2
(gdb) p/s $ebx
$4 = 2
(gdb) continue
Continuing.
hello, jorld!
[Inferior 1 (process 3550) exited normally]
(gdb) quit

```

Рис. 3.20: завершение работы в режиме псевдографики

##Обработка аргументов командной строки в GDB Скопируйте файл `lab8-2.asm`, созданный при выполнении лабораторной работы No8, с программой выводящей на экран аргументы командной строки (Листинг 8.2) в файл с именем `lab09-3.asm` и создайте исполняемый файл. Для загрузки в `gdb` программы с аргументами необходимо использовать ключ `-args`. Загрузите исполняемый файл в отладчик, указав аргументы:

```

mvchettergova@dk4n60 ~/work/arch-pc/lab09 $ nasm -f elf -g -l lab09-3.lst lab09-3.asm
mvchettergova@dk4n60 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-3 lab09-3.o
mvchettergova@dk4n60 ~/work/arch-pc/lab09 $ gdb --args lab09-3 1 2 '3'
GNU gdb (Gentoo 12.1 vanilla) 12.1
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-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) b _start
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 9.
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/v/mvchettergova/work/arch-pc/lab09/lab09-3 1 2 3

Breakpoint 1, _start () at lab09-3.asm:9
9      pop ecx                ; Извлекаем из стека в 'ecx' количество
(gdb)

```

Рис. 3.21: создание исполняемого файла и загрузка ключа `--args` и установка брейкпоинта перед первой функцией программы

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`. Для начала установим точку останова перед первой инструкцией в программе и запустим ее (см. рис. 20).

Адрес вершины стека храниться в регистре `esp` и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы) Число аргументов равно 5 – это имя программы `lab09-3` и непосредственно аргументы: аргумент1, аргумент, 2 и ‘аргумент 3’. Посмотрите 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.

```

Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/v/mvchetvergova/work/arch-pc/lab09/lab09-3 1 2 3
Breakpoint 1, _start () at lab09-3.asm:9
9      pop ecx                ; Извлекаем из стека в 'ecx' количество
(gdb) x/x $esp
0xffffc2d0: 0x00000004
(gdb) x/s *(void**)(esp+4)
0xffffc567: "/afs/.dk.sci.pfu.edu.ru/home/m/v/mvchetvergova/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)(esp+8)
0xffffc5b1: "1"
(gdb) x/s *(void**)(esp+12)
0xffffc5b3: "2"
(gdb) x/s *(void**)(esp+16)
0xffffc5b5: "3"

```

Рис. 3.22: просмотр аргументов под данными адресами

3.3 Задание для самостоятельной работы

1. Преобразуйте программу из лабораторной работы No8 (Задание No1 для самостоятельной работы), реализовав вычисление значения функции $\boxtimes(\boxtimes)$ как подпрограмму.

шаг 1 - рассмотрим листинг программы до изменений.


```

lab09-4.asm      [----]  0 L: [ 1+38 39/ 39] *(1485/1536b) 0099 0x063
#include "in_out.asm"
SECTION .data
msg1 db "    Результат: ",0h
msg2 db "Функция: f(x)=3(x+2)",0h
SECTION .text
global _start
_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
add eax,2
mov ebx, 3
mul ebx
add esi,eax ; добавляем к промежуточной сумме
; след. аргумент 'esi=esi+eax'
loop next ; переход к обработке следующего аргумента

_end:
mov eax, msg2
call sprint
mov eax, msg1 ; вывод сообщения "Результат: "
call sprint
mov eax, esi ; записываем сумму в регистр 'eax'
call iprintLF ; печать результата
call quit ; завершение программы

```

Рис. 3.23: листинг программы до изменений

шаг 2 - создадим подпрограмму `_steps`, которая будет производить вычисления. После того, как программа берёт следующий элемент из стека, этот элемент проходит через подпрограмму `_steps`, а затем записывается в сумму всех предыдущих элементов (`esi`). Подпрограмма записанан в самом конце.

```

lab09-4.asm      [----]  7 L: [ 6+32 38/ 48] *(1285/1356b) 0010 0x00A
global _start
_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
;

next:
cmp ecx,0h ; проверяем, есть ли еще аргументы

jz _end ; если аргументов нет выходим из цикла
; (переход на метку '_end')
pop eax ;извлекаем следующий аргумент из стека
call atoi ; преобразуем символ в число
call _steps
loop next

_end:
mov eax, msg2
call sprint
mov eax, msg1 ; вывод сообщения "Результат: "
call sprint
mov eax, esi ; записываем сумму в регистр 'eax'
call iprintLF ; печать результата
call quit ; завершение программы

_steps:
add eax,2
mov ebx, 3
mul ebx
add esi, eax
ret

```

Рис. 3.24: листинг программы после изменений

```

ivchettergova@dk2n24 ~/work/arch-pc/lab09 $ nasm -f elf lab09-4.asm
ivchettergova@dk2n24 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-4 lab09-4.o
ivchettergova@dk2n24 ~/work/arch-pc/lab09 $ ./lab09-4 1 2 3 4
Функция: f(x)=3(x+2)    Результат: 54
ivchettergova@dk2n24 ~/work/arch-pc/lab09 $ mcedit lab09-4.asm

```

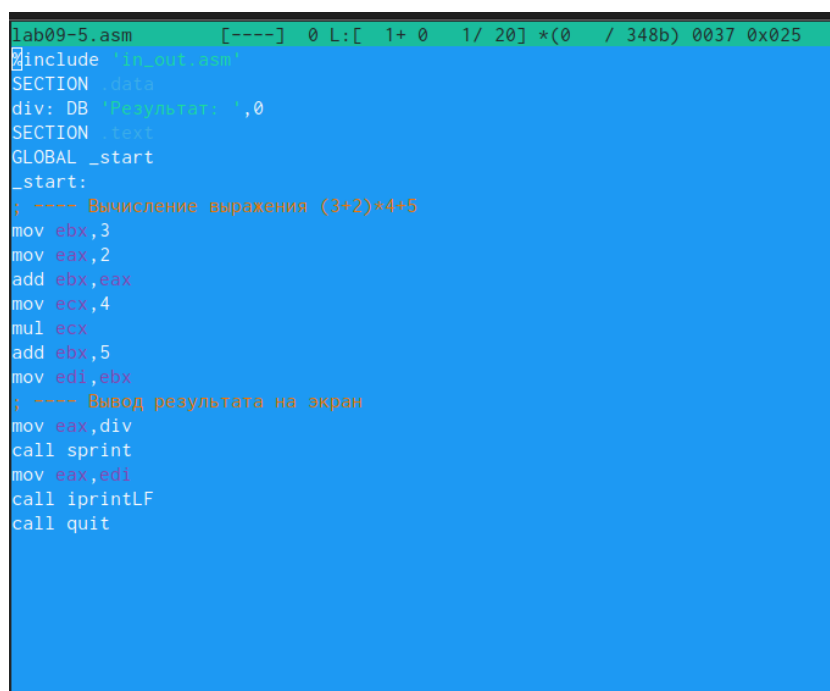
Рис. 3.25: работа программа после внесённых изменений

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

шаг 1 - запишем листинг в редактор NASM и проверим работу этого файла. Действительно, в коде допущена ошибка: программа должна выводит число 25, но выводится число 54.

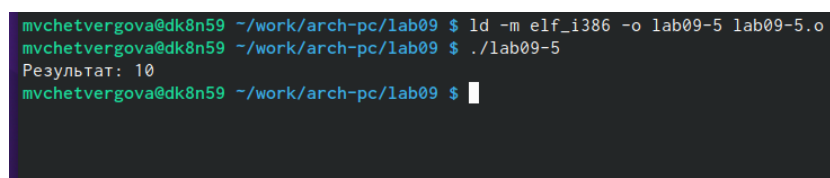
листинг программы:

```
%include 'in_out.asm' SECTION .data div: DB 'Результат:',0 SECTION .text GLOBAL _start _start: ; --- Вычисление выражения (3+2)*4+5 mov ebx,3 mov eax,2 add ebx,eax mov ecx,4 mul ecx add ebx,5 mov edi,ebx ; --- Вывод результата на экран mov eax,div call sprint mov eax,edi call iprintLF call quit
```



```
lab09-5.asm [----] 0 L: [ 1+ 0 1/ 20] *(0 / 348b) 0037 0x025
include 'in_out.asm'
SECTION .data
div: DB 'Результат:',0
SECTION .text
GLOBAL _start
_start:
; ---- Вычисление выражения (3+2)*4+5
mov ebx,3
mov eax,2
add ebx,eax
mov ecx,4
mul ecx
add ebx,5
mov edi,ebx
; ---- Вывод результата на экран
mov eax,div
call sprint
mov eax,edi
call iprintLF
call quit
```

Рис. 3.26: запись листинга программы в редактор



```
mvchetvergova@dk8n59 ~/work/arch-pc/lab09 $ ld -m elf_i386 -o lab09-5 lab09-5.o
mvchetvergova@dk8n59 ~/work/arch-pc/lab09 $ ./lab09-5
Результат: 10
mvchetvergova@dk8n59 ~/work/arch-pc/lab09 $
```

Рис. 3.27: работа данной программы

шаг 2 - откроем режим псевдографики и расставим брейкпоинты на каждом

арифметическом действии. С помощью команды `continue` пройдемся по каждому шагу, чтобы понять, в какой момент произошла ошибка.

```

edi      0x0      0
eip      0x80490fb 0x80490fb <_start+19>
eflags   0x202    [ IF ]
cs       0x23     35
ss       0x2b     43

0x80490e8 <_start>    mov     ebx,0x3
0x80490ed <_start+5>   mov     eax,0x2
B+ 0x80490f2 <_start+10> add     ebx,eax
0x80490f4 <_start+12>   mov     ecx,0x4
B+ 0x80490f9 <_start+17> mul     ecx
B+ 0x80490fb <_start+19> add     ebx,0x5
0x80490fe <_start+22>   mov     edi,ebx
0x8049100 <_start+24>   mov     eax,0x804a000
0x8049105 <_start+29>   call    0x804900f <sprint>
0x804910a <_start+34>   mov     eax,edi
0x804910c <_start+36>   call    0x8049086 <iprintf>
0x8049111 <_start+41>   call    0x80490db <quit>

exec No process in:
Start it from the beginning? (y or n) yStarti
gova/work/arch-pc/lab09/lab09-5 _start

Breakpoint 8, 0x080490f2 in _start ()
(gdb) c
Continuing.

Breakpoint 9, 0x080490f9 in _start ()
(gdb) c
Continuing.

Breakpoint 10, 0x080490fb in _start ()
(gdb) c
Continuing.
[Inferior 1 (process 17461) exited normally]
(gdb)

```

Рис. 3.28: расстановка брейкпоинтов и рассмотрение каждого шага работы кода

шаг 3 - ошибка заключается в том, что в строке под адресом 0x804490f2 значение `eax` записывается в `ebx`, хотя должно быть наоборот. Исправим это в редакторе NASM. Проверим работу исправного файла.

```

lab09-5.asm      [----]  9 L:[ 1+19 20/ 20] *(348 / 348b) <EOF>
#include "in_out.asm"
SECTION .data
div: DB "Результат: ",0
SECTION .text
GLOBAL _start
_start:
; ----- Вычисление выражения (3+2)*4+5
mov ebx,3
mov eax,2
add eax,ebx
mov ecx,4
mul ecx
add eax,5
mov edi,eax
; ----- Вывод результата на экран
mov eax,div
call sprint
mov eax,edi
call iprintLF
call quit

```

Рис. 3.29: устранение ошибки в листинге программы

```

(gdb) layout regs
(gdb) si
0x080490ed in _start ()
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) yStarting program: /afs/.dk.sci.pfu.edu.ru/home/m/v/mvchetvergova/
ork/arch-pc/lab09/lab09-5

Breakpoint 1, 0x080490e8 in _start ()
(gdb) c
Continuing.
Результат: 25
(gdb) qor 1 (process 20992) exited normally]

```

Рис. 3.30: Проверка работы исправленного файла

4 Выводы

Здесь кратко описываются итоги проделанной работы.

{.unnumbered}