Отчет по лабораторной работе №8

Дисциплина: архитектура компьютера

Царёв Максим Александрович

Содержание

1 Цель работы

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

2 Задание

1. Программа с вызовом подпрограммы
2. Изменение программы с применением инструкции ret и call
3. Отладка программ с помощью GDB
4. Добавление точек останова
5. Работа с данными программами в GDB
6. Обработка аргументов командной сроки в GDB
7. Выполнение заданий для самостоятельной работы

3 Теоретическое введение

Понятие об отладке Отладка — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа:

• обнаружение ошибки; • поиск её местонахождения; • определение причины ошибки; • исправление ошибки. Можно выделить следующие типы ошибок:

• синтаксические ошибки — обнаруживаются во время трансляции исходного кода и вызваны нарушением ожидаемой формы или структуры языка; • семантические ошибки — являются логическими и приводят к тому, что программа запускается, отрабатывает, но не даёт желаемого результата; • ошибки в процессе выполнения — не обнаруживаются при трансляции и вызывают пре- рывание выполнения программы (например, это ошибки, связанные с переполнением или делением на ноль).

Второй этап — поиск местонахождения ошибки. Некоторые ошибки обнаружить доволь- но трудно. Лучший способ найти место в программе, где находится ошибка, это разбить программу на части и произвести их отладку отдельно друг от друга. Третий этап — выяснение причины ошибки. После определения местонахождения ошибки обычно проще определить причину неправильной работы программы. Последний этап — исправление ошибки. После этого при повторном запуске программы, может обнаружиться следующая ошибка, и процесс отладки начнётся заново

Методы отладки Наиболее часто применяют следующие методы отладки:

• создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных значений на экран — так называемые диагностические сообщения); • использование специальных программ-отладчиков.

Отладчики позволяют управлять ходом выполнения программы, контролировать и из- менять данные. Это помогает быстрее найти место ошибки в программе и ускорить её исправление. Наиболее популярные способы работы с отладчиком — это использование точек останова и выполнение программы по шагам. 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 програм- мы, т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы

Основные возможности отладчика GDB

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. От- ладчик не содержит собственного графического пользовательского интерфейса и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторон- них 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; • изменить программу так, чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

Запуск отладчика GDB; выполнение программы; выход Синтаксис команды для запуска отладчика имеет следующий вид: gdb [опции] [имя_файла | ID процесса] После запуска gdb выводит текстовое сообщение — так называемое «nice GDB logo». В следующей строке появляется приглашение (gdb) для ввода команд. Далее приведён список некоторых команд GDB. Команда run (сокращённо r) — запускает отлаживаемую программу в оболочке GDB. Если точки останова не были установлены, то программа выполняется и выводятся сообщения:

(gdb) run Starting program: test Program exited normally. (gdb)

Если точки останова были заданы, то отладчик останавливается на соответствующей команде и выдаёт номер точки останова, адрес и дополнительную информацию — текущую строку, имя процедуры, и др. Команда kill (сокращённо k) прекращает отладку программы, после чего следует вопрос о прекращении процесса отладки: Kill the program being debugged? (y or n) y Если в ответ введено y (то есть «да»), отладка программы прекращается. Командой run её можно начать заново, при этом все точки останова (breakpoints), точки просмотра (watchpoints) и точки отлова (catchpoints) сохраняются. Для выхода из отладчика используется команда quit (или сокращённо q):

(gdb) q

Дизассемблирование программы Если есть файл с исходным текстом программы, а в исполняемый файл включена информа- ция о номерах строк исходного кода, то

программу можно отлаживать, работая в отладчике непосредственно с её исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом -g. Посмотреть дизассемблированный код программы можно с помощью команды disassemble :

(gdb) disassemble _start

Существует два режима отображения синтаксиса машинных команд: режим Intel, используемый в том числе в NASM, и режим ATT (значительно отличающийся внешне). По умолчанию в дизассемблере GDB принят режим ATT. Переключиться на отображение команд с привычным Intel'овским синтаксисом можно, введя команду set disassembly-flavor intel

Точки останова

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»:

(gdb) break * (gdb) b

Информацию о всех установленных точках останова можно вывести командой info (крат- ко i):

(gdb) info breakpoints (gdb) i b

Для того чтобы сделать неактивной какую-нибудь ненужную точку останова, можно вос- пользоваться командой disable:

disable breakpoint

Обратно точка останова активируется командой enable:

enable breakpoint

Если же точка останова в дальнейшем больше не нужна, она может быть удалена с помощью команды delete:

(gdb) delete breakpoint

Ввод этой команды без аргумента удалит все точки останова. Информацию о командах этого раздела можно получить, введя

help breakpoints

Пошаговая отладка

Для продолжения остановленной программы используется команда continue (c) (gdb) c [аргумент]. Выполнение программы будет происходить до следующей точки останова. В качестве аргумента может использоваться целое число □, которое указывает отладчику проигнорировать □ – 1 точку останова (выполнение остановится на □-й точке). Команда stepi (кратко sI) позволяет выполнять

программу по шагам, т.е. данная команда выполняет ровно одну инструкцию: (gdb) si [аргумент]

При указании в качестве аргумента целого числа отладчик выполнит команду step раз при условии, что не будет точек останова или выполнение программы не прервётся по другим причинам. Команда nexti (или ni) аналогична stepi, но вызов процедуры (функции) трактуется отладчиком как одна инструкция:

(gdb) ni [аргумент]

Информацию о командах этого раздела можно получить, введя

(gdb) help running

Работа с данными программы в GDB Как уже упоминалось, отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Посмотреть содержимое регистров можно с помощью команды info registers (или i r):

(gdb) info registers

Для отображения содержимого памяти можно использовать команду x/NFU , выда- ёт содержимое ячейки памяти по указанному адресу. NFU задает формат, в котором выводятся данные

Например, x/4uh 0x63450 — это запрос на вывод четырёх полуслов (h) из памяти в формате беззнаковых десятичных целых (u), начиная с адреса 0x63450. Чтобы посмотреть значения регистров используется команда print /F (сокращен- но p). Перед именем регистра обязательно ставится префикс $. Например, команда p/x e c x в ы в о д и т з н а ч е н и е р е г и с т р а в ш е с т н а д ц а т е р и ч н о м ф о р м а т е . И з м е н и т ь з н а ч е н и е д л я р е г и с т р а и л и я ч е й к и п а м я т и м о ж н о с п о м о щ ь ю к о м а н д ы s e t , з а д а в е й в к а ч е с т в е а р г у м е н т а и м я р е г и с т р а и л и а д р е с . П р и э т о м п е р е д и м е н е м р е г и с т р а с т а в и т с я п р е ф и к с , а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си). Справку о любой команде gdb можно получить, введя

(gdb) help [имя_команды]

Понятие подпрограммы Подпрограмма — это, как правило, функционально законченный участок кода, который можно многократно вызывать из разных мест программы. В отличие от простых переходов из подпрограмм существует возврат на команду, следующую за вызовом. Если в программе встречается одинаковый участок кода, его можно оформить в виде подпрограммы, а во всех нужных местах поставить её вызов. При этом подпрограмма бу- дет содержаться в коде в одном экземпляре, что позволит уменьшить размер кода всей программы.

Инструкция call и инструкция ret Для вызова подпрограммы из основной программы используется инструкция call, кото- рая заносит адрес следующей инструкции в стек и загружает в регистр eip адрес соответству- ющей 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, занесён- ный туда соответствующей инструкцией call, и заносит его в eip. После этого выполнение основной программы возобновится с инструкции, следующей за инструкцией call.

Подпрограмма может вызываться как из внешнего файла, так и быть частью основной программы. Основные моменты выполнения подпрограммы иллюстрируются на рис. 9.1. Важно помнить, что если в подпрограмме занести что-то в стек и не извлечь, то на вершине стека окажется не адрес возврата и это приведёт к ошибке выхода из подпрограммы. Кроме того, надо помнить, что подпрограмма без команды возврата не вернётся в точку вызова, а будет выполнять следующий за подпрограммой код, как будто он является её продолжением.

4 Выполнение лабораторной работы

4.1 Реализация программ в NASM

Создал каталог для выполнения лабораторной работы № 9, перешел в него и создал файл lab09-1.asm.Ввел туда программу с листинга 9.1 и запустил ее.

```
%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]
```

Программа действительно работает верно

```
/lab09$ nasm -f elf lab09-1.asm
/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
/lab09$ ./lab09-1
```

```
Введите x: 1
2x+7=9
```

4.2 Отладка программам с помощью GDB

Создаю файл lab09-2.asm с текстом программы из листинга 9.2

```
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
```

Получаю исполняемый файл. Для работы с GDB в исполняемый файл добавляю отладочную информацию, для этого трансляцию программ провожу с ключом ‘-g’.

```
09$ nasm -f elf -g -l lab09-2.lst lab09-2.asm
09$ ld -m elf_i386 -o lab09-2 lab09-2.o
09$ gdb lab09-2
```

Загружаю исполняемый файл в отладчик gdb

```
(gdb) run
Starting program: study_2024-2025_arhpc/labs/lab09/lab09/lab09-2
Hello, world!
[Inferior 1 (process 53675) exited normally]
(gdb)
```

Для более подробного анализа программы установливаю брейкпоинт на метку _start, с которой начинается выполнение любой ассемблерной программы, и запускаю её.

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: study_2024-2025_arhpc/labs/lab09/lab09/lab09-2

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

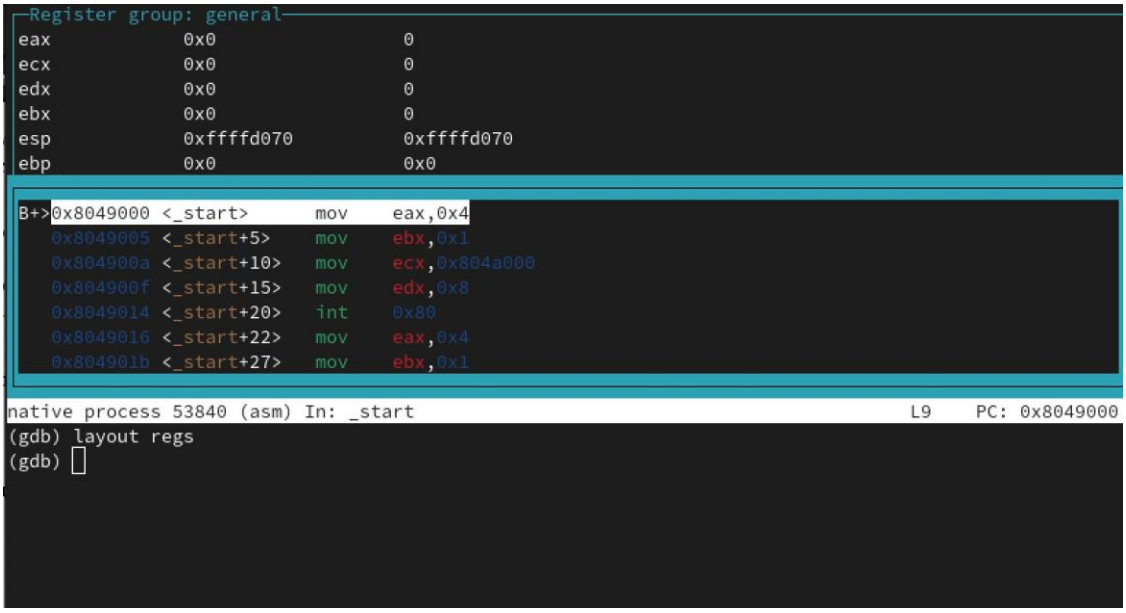
Просматриваю дисассимилированный код программы с помощью команды disassemble начиная с метки _start

```
(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)
```

Переключяюсь на отображение команд с Intel’овским синтаксисом, введя команду set disassembly-flavor intel

```
(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.
```

Перечислил различия отображения синтаксиса машинных команд в режимах ATT и Intel. Включил режим псевдографики для более удобного анализа программы



На предыдущих шагах была установлена точка останова по имени метки (_start). Проверяю это с помощью команды info breakpoints

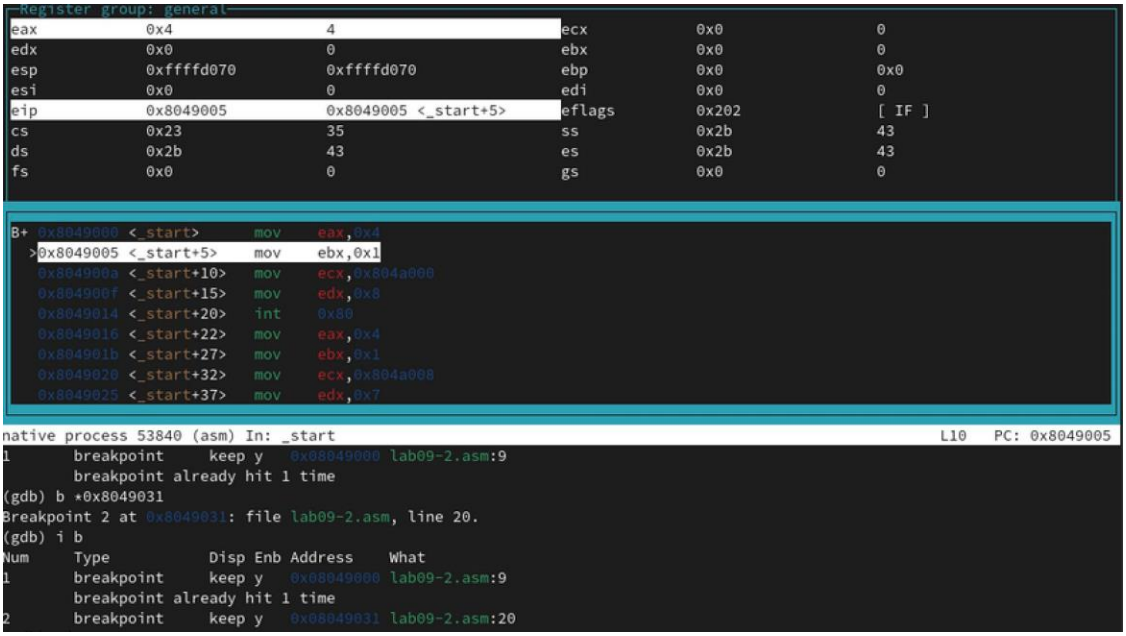
```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08049000 lab09-2.asm:9
        breakpoint already hit 1 time
(gdb)
```

Установил еще одну точку останова по адресу инструкции. Адрес инструкции можно Определил адрес предпоследней инструкции (mov ebx,0x0) и установил точку останова. Посмотрел информацию о всех установленных точках останова

```
(gdb) b *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
```

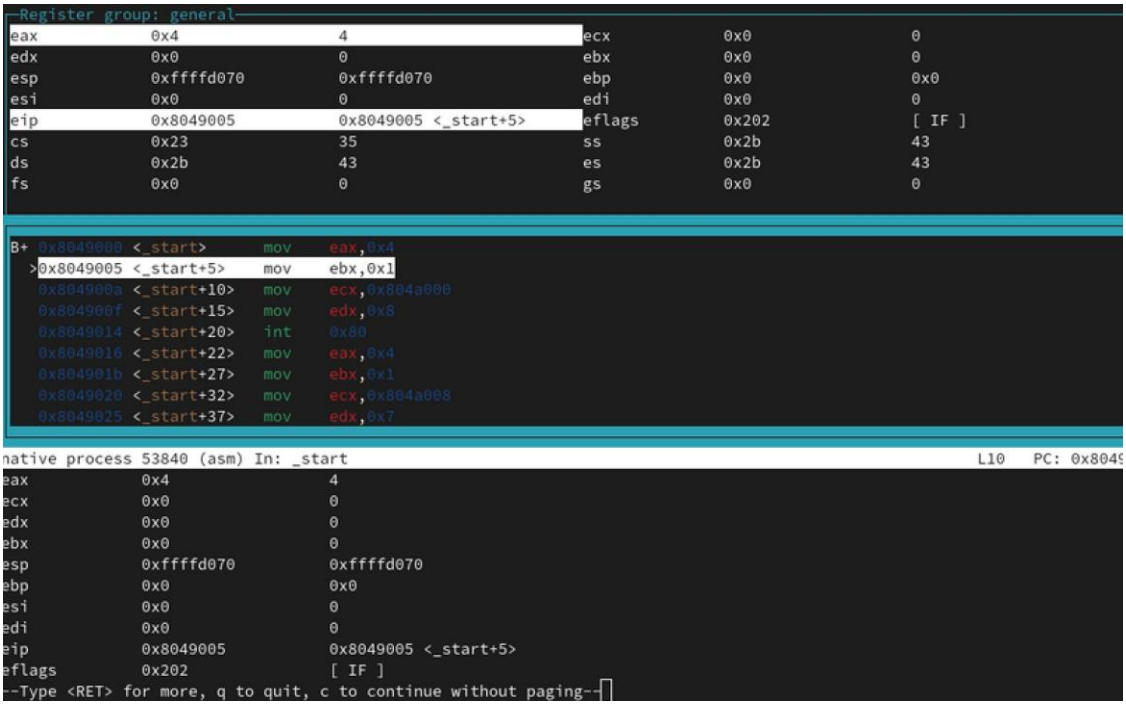
4.3 Работа с данными программы в GDB

Выполнил 5 инструкций с помощью команды si и проследил за изменением значений регистров



изменяются следующие регистры: ECX: уменьшается при pop ecx и dec ecx. EDX: изменяется при pop edx. EAX: изменяется при pop eax, call atoi (где функция записывает результат в eax), и при выполнении арифметических операций (imul, sub). ESI: изменяется при сложении в add esi, eax

Посмотрел значение переменной msg1 по имени



Посмотрел значение переменной msg2 по адресу.

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

Изменил первый символ переменной msg1

```
(gdb) x/1sb &msg1
0x804a000 <msg1>:       "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:       "world!\n\034"
```

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

Заменяю первый символ в переменной msg2

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

С помощью команды set измените значение регистра ebx:

```
(gdb) set $ebx='2'
(gdb) p/s $ebx
$1 = 50
```

```
(gdb) p/s $eax
$2 = 4
(gdb) p/t $eax
$3 = 100
(gdb) p/s $ecx
$4 = 0
(gdb) p/x $ecx
$5 = 0x0
```

Разница заключается в интерпретации значения регистра: p $ebx выводит его как число, а p/s $ebx трактует его как указатель на строку.

4.4 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 (Листинг 8.2) в файл с именем lab09-3.asm.Создаю исполняемый файл.

```
lab09$ nasm -f elf -g -l lab09-3.lst lab09-3.asm
lab09$ ld -m elf_i386 -o lab09-3 lab09-3.o
lab09$ gdb --args lab09-3 1 2 3
```

```
GNU gdb (Fedora Linux) 15.1-1.fc40
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab09-3...
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 5.
```

Устанавливаю точку останова и запускаю программу.

```
Starting program: study_2024-2025_arhpc/labs/lab09/lab09/lab09-2            1 2 3

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.

Breakpoint 1, _start () at lab09-3.asm:5
5          pop ecx ; Извлекаем из стека в `ecx` количество
```

Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки,число аргументов равно 5

```
(gdb) x/x $esp
0xffffd060:     0x00000004
```

Посмотрел остальные позиции стека

```
(gdb) x/s *(void**)($esp + 4)
0xffffd226:     "/home/emil/study_2024-2025_arhpc/labs/lab09/lab09/lab09-3"
(gdb) x/s *(void**)($esp + 8)
0xffffd260:     "1"
(gdb) x/s *(void**)($esp + 12)
0xffffd262:     "2"
(gdb) x/s *(void**)($esp + 16)
0xffffd264:     "3"
```

Шаг изменения адреса равен 4 байта ([esp+4], [esp+8], [esp+12] и т.д.) потому, что каждая ячейка стека занимает 4 байта. Это связано с тем, что программа работает в 32-битной архитектуре, где один элемент типа int или указатель занимает 4 байта в памяти.

5 Выполнение заданий для самостоятельной работы

Копирую файл lab8-4.asm, переименовываю в lab09-4.asm, реализовываю вычисление функции как подпрограмму

```
    call compute_fx

    add esi, eax        ; Добавляем f(x) к промежуточной сумме `esi`
    dec ecx             ; Уменьшаем `ecx` на 1
    jmp next            ; Переход к обработке следующего аргумента

_end:
    mov eax, msg        ; Выводим сообщение "Результат: "
    call sprint
    mov eax, esi        ; Записываем сумму в регистр `eax`
    call iprintLF       ; Печать результата
    call quit           ; Завершение программы

; Подпрограмма для вычисления f(x) = 15 * eax - 9
compute_fx:
    push ebx            ; Сохраняем значение `ebx` на стеке (для сохранения данных)
    mov ebx, 15         ; Устанавливаем коэффициент 15
    imul eax, ebx       ; eax = eax * 15
    sub eax, 9          ; eax = eax - 9 (получаем f(x))
    pop ebx             ; Восстанавливаем значение `ebx` из стека
    ret                 ; Возврат из подпрограммы (результат в `eax`)
```

Проверил работу программы,программа работает корректно

```
@fedora:~/study_2024-2025_arhpc/labs/lab09/lab09$ nasm -f elf lab09-4.asm
@fedora:~/study_2024-2025_arhpc/labs/lab09/lab09$ ld -m elf_i386 -o lab09-4 lab09-4.o
@fedora:~/study_2024-2025_arhpc/labs/lab09/lab09$ ./lab09-4 1 2 3 4
льтат: 114
```

Создаю файл с именем lab09-5.asm,при помощи отладчика GDB устраняю ошибку,теперь программа работает корректно.

```
%include 'in_out.asm'

SECTION .data
msg db 'Результат: ', 0

SECTION .text
global _start

_start:
    ; ---- Вычисление выражения (3 + 2) * 4 + 5
    mov eax, 3          ; eax = 3
    add eax, 2          ; eax = eax + 2 = 5
    mov ebx, 4          ; ebx = 4
    imul eax, ebx       ; eax = eax * ebx = 5 * 4 = 20
    add eax, 5          ; eax = eax + 5 = 25

    ; ---- Вывод результата на экран
    mov edi, eax        ; Сохраняем результат в `edi` для вывода
    mov eax, msg        ; Устанавливаем `eax` на строку сообщения
    call sprint         ; Печатаем "Результат: "
    mov eax, edi        ; Перемещаем результат в `eax` для печати
```

```
@fedora:~/study_2024-2025_arhpc/labs/lab09/lab09$ nasm -f elf lab09-5.asm
@fedora:~/study_2024-2025_arhpc/labs/lab09/lab09$ ld -m elf_i386 -o lab09-5 lab09-5.o
@fedora:~/study_2024-2025_arhpc/labs/lab09/lab09$ ./lab09-5
льтат: 25
```

6 Выводы

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