

Отчет по лабораторной работе №9

Дисциплина: Архитектура компьютера

Мустафина Аделя Юрисовна

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
3.1	9.2.1. Понятие об отладке	7
3.2	9.2.2. Методы отладки	8
3.3	9.3. Основные возможности отладчика GDB	9
3.4	9.3.1. Запуск отладчика GDB; выполнение программы; выход . . .	9
3.5	9.3.2. Дизассемблирование программы	10
3.6	9.3.3. Точки останова	11
3.7	9.3.4. Пошаговая отладка	12
3.8	9.3.5. Работа с данными программы в GDB	13
3.9	9.3.6. Понятие подпрограммы	14
3.9.1	9.3.6.1. Инструкция call и инструкция ret	14
4	9.4 Выполнение лабораторной работы	15
4.1	9.4.1. Реализация подпрограмм в NASM	15
4.2	9.4.2. Отладка программ с помощью GDB	21
4.2.1	9.4.2.1. Добавление точек останова	26
4.2.2	9.4.2.2. Работа с данными программы в GDB	28
4.2.3	9.4.2.3. Обработка аргументов командной строки в GDB . . .	33
4.3	9.5. Задание для самостоятельной работы	37
5	Выводы	46
6	Список литературы	47

Список иллюстраций

4.1	Создание каталога	15
4.2	Листинг 9.1	18
4.3	Измененный листинг 9.1	18
4.4	Отладка файла	22
4.5	Загрузка файла в отладчик	22
4.6	Проверка работы программы	23
4.7	Брейкпоинт	23
4.8	Дисассимилированный код	24
4.9	Отображение команд	24
4.10	Режим псевдографики.1	25
4.11	Режим псевдографики.2	26
4.12	Проверка точек останова	27
4.13	Точки останова	28
4.14	Содержимое регистров	29
4.15	Значение переменной msg1	30
4.16	Значение переменной msg2	31
4.17	Символ переменной msg1	31
4.18	Символ переменная msg2	32
4.19	Изменение значения регистра ebx	32
4.20	Запуск программы	34
4.21	запуск	34
4.22	run	35
4.23	Адрес вершины стека	35
4.24	Вычисление значения функции f(x)	37
4.25	Неверный результат	39
4.26	Значения регистров	41
4.27	Инструкция	42
4.28	Ошибка	43
4.29	Исправленный вариант	44

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. Порядок выполнения лабораторной работы.
2. Выполнение заданий для самостоятельной работы

3 Теоретическое введение

3.1 9.2.1. 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, отработывает, 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.

3.2 9.2.2. 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 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.

3.3 9.3. 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. 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;
- изменить программу так, чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

3.4 9.3.1. 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
```

3.5 9.3.2. Дизассемблирование программы

Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, то программу можно отлаживать, работая в отладчике непосредственно с её исходным текстом. Чтобы

программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом -g. Посмотреть дизассемблированный код программы можно с помощью команды

```
disassemble <метка/адрес>:  
(gdb) disassemble _start
```

Существует два режима отображения синтаксиса машинных команд: режим Intel, используемый в том числе в NASM, и режим AT&T (значительно отличающийся внешне). По умолчанию в дизассемблере GDB принят режим AT&T. Переключиться на отображение команд с привычным Intel'овским синтаксисом можно, введя команду `set disassembly-flavor intel`

3.6 9.3.3. 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 *<адрес>  
(gdb) b <метка>
```

Информацию о всех установленных точках останова можно вывести командой `info` (кратко `i`):

```
(gdb) info breakpoints  
(gdb) i b
```

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

```
disable breakpoint <номер точки останова>
```

Обратно точка останова активируется командой `enable`:

```
enable breakpoint <номер точки останова>
```

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

```
(gdb) delete breakpoint <номер точки останова>
```

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

```
help breakpoints
```

3.7 9.3.4. Пошаговая отладка

Для продолжения остановленной программы используется команда:

```
continue (c) (gdb)  
с [аргумент]
```

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 `sI`) позволяет выполнять программу по шагам, т.е. данная команда выполняет ровно одну инструкцию:

```
(gdb) si [аргумент]
```

При указании в качестве аргумента целого числа *N* отладчик выполнит команду `step N` 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
```

3.8 9.3.5. 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 `ir`):

```
(gdb) info registers
```

Для отображения содержимого памяти можно использовать команду `x/NFU`, выдаёт содержимое ячейки памяти по указанному адресу. `NFU` задает формат, в котором выводятся данных. 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 $ecx` выводит значение регистра в шестнадцатеричном формате. Изменить значение для регистра или ячейки памяти можно с помощью команды `set`, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс `$`, 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

```
(gdb) help [имя_команды]
```

3.9 9.3.6. 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 будет содержаться в коде в одном экземпляре, что позволит уменьшить размер кода всей программы.

3.9.1 9.3.6.1. Инструкция call и инструкция ret

Для вызова подпрограммы из основной программы используется инструкция call, которая заносит адрес следующей инструкции в стек и загружает в регистр `esp` адрес соответствующей подпрограммы, осуществляя таким образом переход. Затем начинается выполнение подпрограммы, которая, в свою очередь, также может содержать подпрограммы. Подпрограмма завершается инструкцией `ret`, которая извлекает из стека адрес, занесённый туда соответствующей инструкцией `call`, и заносит его в `esp`. 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.

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 9.4 Выполнение лабораторной работы

4.1 9.4.1. Реализация подпрограмм в NASM

Создаю каталог для выполнения лабораторной работы №9, и создаю файл lab09-1.asm (рис. 4.1).

```
aymustafina@vbox:~$ mkdir ~/work/arch-pc/lab09
aymustafina@vbox:~$ cd ~/work/arch-pc/lab09
aymustafina@vbox:~/work/arch-pc/lab09$ touch lab09-1.asm
```

Рис. 4.1: Создание каталога

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

Листинг 9.1. Пример программы с использованием вызова подпрограммы

```
%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 "2х+7"
```

```
_calcul:
mov ebx,2
mul ebx
add eax,7
mov [res],eax
ret ; выход из 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).

```
mov eax, msg          ; вызов подпрограммы печати сообщения
call sprint           ; 'Введите х: '

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

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

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`).

Ввела в файл `lab09-1.asm` текст программы из листинга 9.1. Создала исполняемый файл и проверила его работу (рис. 4.2).

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

Рис. 4.2: Листинг 9.1

Изменила 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 (рис. 4.3).

```
aymustafina@vbox:~/work/arch-pc/lab09$ nasm -f elf lab09-1.asm
aymustafina@vbox:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-1 lab09-1.o
aymustafina@vbox:~/work/arch-pc/lab09$ ./lab09-1
Введите x: 5
f(g(x)) = 203
aymustafina@vbox:~/work/arch-pc/lab09$ █
```

Рис. 4.3: Измененный листинг 9.1

Листинг 9.1. Вычисление выражения $f(g(x))$

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

```
SECTION .data
```

```
msg: DB 'Введите x: ',0
result: DB 'f(g(x)) = ',0
```

```
SECTION .bss
```

```
x: RESB 80
```

```
res: RESB 80
```

```
SECTION .text
```

```
GLOBAL _start
```

```
_start:
```

```
;-----
```

```
; Основная программа
```

```
;-----
```

```
    mov eax, msg
```

```
    call sprintf          ; 'Введите x: '
```

```
    mov ecx, x
```

```
    mov edx, 80
```

```
    call sread           ; Ввод x с клавиатуры
```

```
    mov eax, x
```

```
    call atoi            ; Преобразование ASCII кода в число, eax=x
```

```
    call _calcul         ; Вызов подпрограммы _calcul
```

```
    mov eax, result
```

```
    call sprintf         ; 'f(g(x)) = '
```

```

    mov eax, [res]          ; Получаем результат из памяти
    call iprintLF           ; Выводим результат

    call quit               ; Завершение программы

;-----
; Подпрограмма вычисления выражения "f(g(x))"
;  $f(x) = 2x + 7$ ,  $g(x) = 3x - 1$ 
;-----
_calcul:
    push eax                ; Сохраняем x на стеке
    call _subcalcul         ; Вызываем подпрограмму _subcalcul
    pop ebx                 ; Восстанавливаем x из стека (если нужно)

    ; Теперь eax содержит g(x), вычисляем f(g(x))
    mov ebx, 2
    mul eax                 ; Умножаем g(x) на 2
    add eax, 7              ; Добавляем 7
    mov [res], eax          ; Сохраняем результат в res
    ret                    ; Возврат в основную программу

;-----
; Подпрограмма вычисления g(x)
;  $g(x) = 3x - 1$ 
;-----
_subcalcul:
    push eax                ; Сохраняем x на стеке
    mov ebx, 3
    mul ebx                 ; Умножаем x на 3

```

```

sub eax, 1          ; Вычисляем  $g(x) = 3x - 1$ 
pop ebx            ; Восстанавливаем x (если нужно)
ret                ; Возврат в _calcul

```

4.2 9.4.2. Отладка программ с помощью GDB

Создаю 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
```

Провожу трансляцию файла с ключом ‘-g’. Так как для работы с GDB в исполняемый файл необходимо добавить отладочную информацию (рис. 4.4).

```
aymustafina@vbox:~/work/arch-pc/lab09$ nasm -f elf -g -l lab09-2.lst lab09-2.asm
aymustafina@vbox:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-2 lab09-2.o
aymustafina@vbox:~/work/arch-pc/lab09$ gdb lab09-2
GNU gdb (Fedora Linux) 14.2-1.fc40
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-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 lab09-2...
(gdb) run
Starting program: /home/aymustafina/work/arch-pc/lab09/lab09-2
```

Рис. 4.4: Отладка файла

```
nasm -f elf -g -l lab09-2.lst lab09-2.asm
```

```
ld -m elf_i386 -o lab09-2 lab09-2.o
```

Загружаю исполняемый файл в отладчик gdb: `gdb lab09-2` (рис. 4.5).

```
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.
Downloading separate debug info for system-supplied DSO at 0xf7ffc000
Hello, world!
[Inferior 1 (process 10424) exited normally]
(gdb)
```

Рис. 4.5: Загрузка файла в отладчик

Проверяю работу программы, запустив ее в оболочке GDB с помощью команды `run` (сокращённо `r`) (рис. 4.6).

```
Hello, world!
[Inferior 1 (process 10424) exited normally]
```

Рис. 4.6: Проверка работы программы

```
(gdb) run
Starting program: ~/work/arch-pc/lab09/lab09-2
Hello, world!
[Inferior 1 (process 10220) exited normally]
(gdb)
```

Для более подробного анализа программы устанавливаю брейкпоинт на метку `_start`, с которой начинается выполнение любой ассемблерной программы, и запускаю её(рис. 4.7).

```
Hello, world!
[Inferior 1 (process 10424) exited normally]
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 11.
(gdb) run
Starting program: /home/aymustafina/work/arch-pc/lab09/lab09-2

Breakpoint 1, _start () at lab09-2.asm:11
11      mov eax, 4
(gdb)
```

Рис. 4.7: Брейкпоинт

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 12.
(gdb) run
Starting program: ~/work/arch-pc/lab09/lab09-2
Breakpoint 1, _start () at lab09-2.asm:12
12 mov eax, 4
```

Изучаю дисассимилированный код программы с помощью команды `disassemble` начиная с метки `_start` (рис. 4.8).

```
(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. 4.8: Дисассимилированный код

```
(gdb) disassemble _start
```

Переключаюсь на отображение команд с Intel'овским синтаксисом, введя команду `set disassembly-flavor intel` (рис. 4.9).

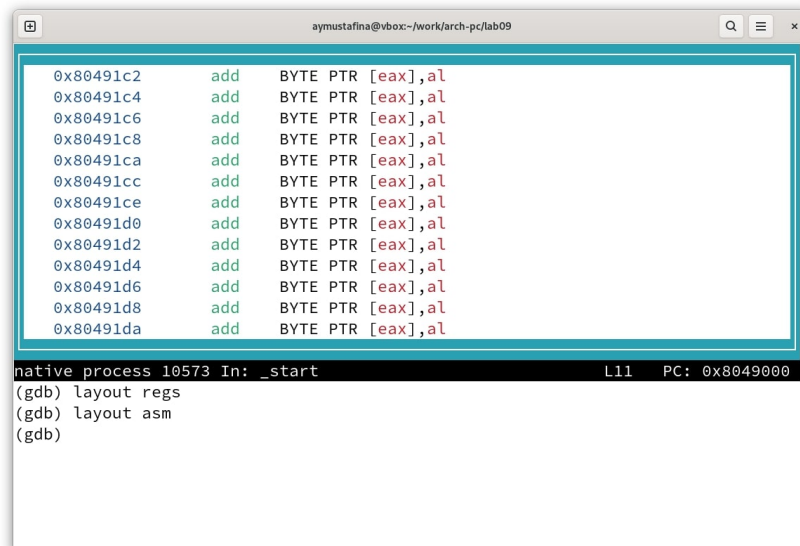
```
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:      mov     eax,0x4
    0x08049005 <+5>:      mov     ebx,0x1
    0x0804900a <+10>:     mov     ecx,0x804a000
    0x0804900f <+15>:     mov     edx,0x8
    0x08049014 <+20>:     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. 4.9: Отображение команд

```
(gdb) set disassembly-flavor intel
```


(gdb) disassemble _start

Различия отображения синтаксиса машинных команд в режимах АТТ и Intel. В режиме АТТ регистры пишутся с “%” и они расположены после указания их размера. Включаю режим псевдографики для более удобного анализа программы (рис. 4.10).



```
0x80491c2    add    BYTE PTR [eax], al
0x80491c4    add    BYTE PTR [eax], al
0x80491c6    add    BYTE PTR [eax], al
0x80491c8    add    BYTE PTR [eax], al
0x80491ca    add    BYTE PTR [eax], al
0x80491cc    add    BYTE PTR [eax], al
0x80491ce    add    BYTE PTR [eax], al
0x80491d0    add    BYTE PTR [eax], al
0x80491d2    add    BYTE PTR [eax], al
0x80491d4    add    BYTE PTR [eax], al
0x80491d6    add    BYTE PTR [eax], al
0x80491d8    add    BYTE PTR [eax], al
0x80491da    add    BYTE PTR [eax], al

native process 10573 In: _start      L11    PC: 0x8049000
(gdb) layout regs
(gdb) layout asm
(gdb)
```

Рис. 4.10: Режим псевдографики.1

(рис. 4.11)

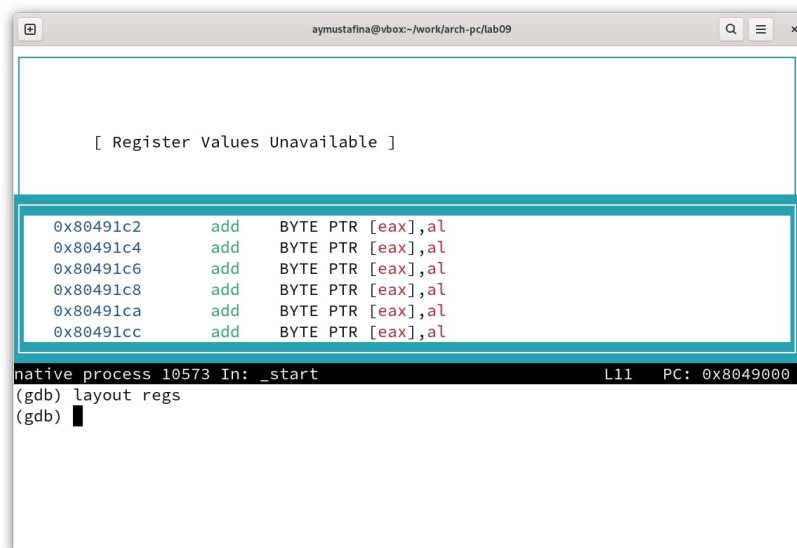


Рис. 4.11: Режим псевдографики.2

(gdb) layout asm

(gdb) layout regs

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.

Однако, к сожалению, у меня не отображаются регистры.

4.2.1 9.4.2.1. 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`). Проверяю это с помощью команды `info breakpoints` (кратко `i b`) (рис. 4.12).

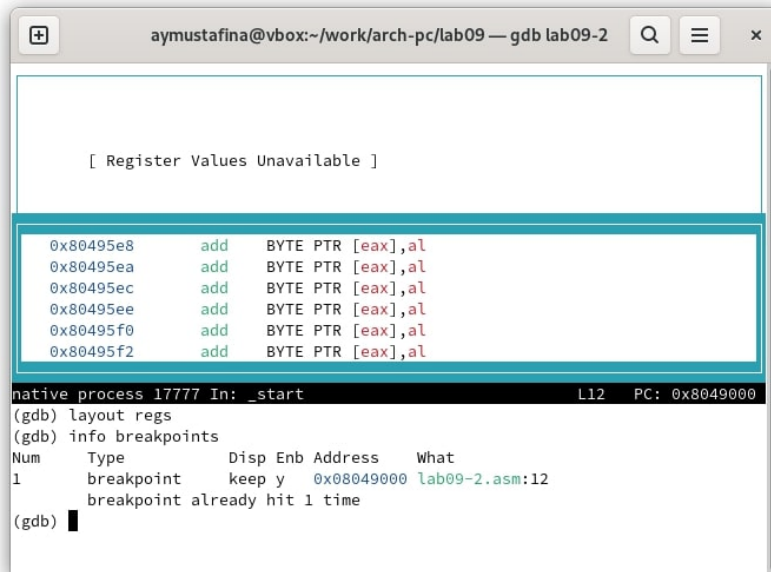


Рис. 4.12: Проверка точек останова

(gdb) info breakpoints

Устанавливаю еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции. Определяю адрес предпоследней инструкции (mov ebx,0x0) и устанавливаю точку останова

(gdb) break *<адрес>

Посматриваю информацию о всех установленных точках останова (рис. 4.13).

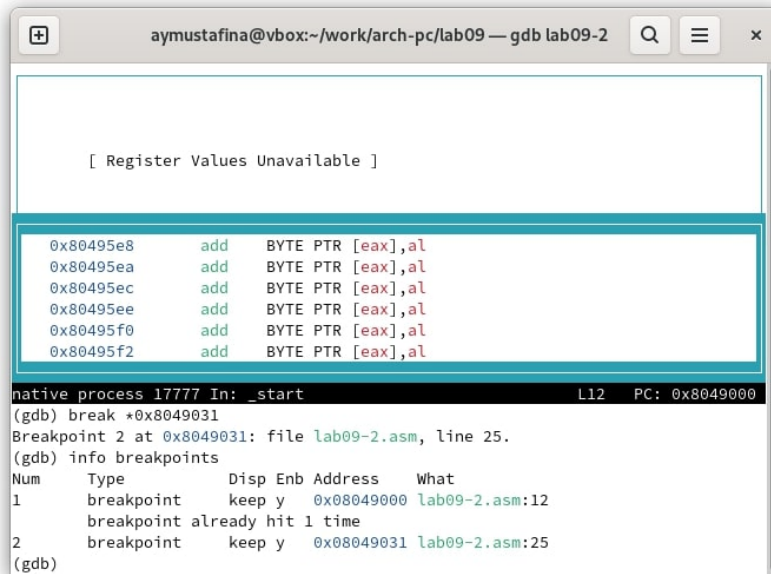


Рис. 4.13: Точки останова

(gdb) i b

4.2.2 9.4.2.2. Работа с данными программы в GDB

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных.

Пытаюсь вновь просмотреть содержимое регистров с помощью команды `info registers` (или `i r`)(рис. 4.14).



Рис. 4.14: Содержимое регистров

(gdb) 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. Просмотрю значение переменной `msg1` по имени (рис. 4.15).

```

0x8049a98      add     BYTE PTR [eax],al
0x8049a9a      add     BYTE PTR [eax],al
0x8049a9c      add     BYTE PTR [eax],al
0x8049a9e      add     BYTE PTR [eax],al
0x8049aa0      add     BYTE PTR [eax],al
0x8049aa2      add     BYTE PTR [eax],al
0x8049aa4      add     BYTE PTR [eax],al
0x8049aa6      add     BYTE PTR [eax],al
0x8049aa8      add     BYTE PTR [eax],al
0x8049aaa      add     BYTE PTR [eax],al
native process 17777 In: _start                               L12  PC: 0x8049000
eax            0x0            0
ecx            0x0            0
edx            0x0            0
ebx            0x0            0
esp            0xffffd0b0     0xffffd0b0
ebp            0x0            0x0
esi            0x0            0
--Type <RET> for more, q to quit, c to continue without paging--
Quit
(gdb) layout regs
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb)

```

Рис. 4.15: Значение переменной msg1

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

```
0x804a000 <msg1>: "Hello, "
```

Посмотрю значение переменной msg2 по адресу. Адрес переменной можно определить по дизассемблированной инструкции. Посмотрю инструкцию mov esx,msg2 которая запи- сывает в регистр esx адрес переменной msg2 (рис. 4.16).

```

0x8049ae0      add     BYTE PTR [eax],al
0x8049ae2      add     BYTE PTR [eax],al
0x8049ae4      add     BYTE PTR [eax],al
0x8049ae6      add     BYTE PTR [eax],al
0x8049ae8      add     BYTE PTR [eax],al
0x8049aea      add     BYTE PTR [eax],al
0x8049aec      add     BYTE PTR [eax],al
0x8049aee      add     BYTE PTR [eax],al
0x8049af0      add     BYTE PTR [eax],al
0x8049af2      add     BYTE PTR [eax],al
native process 17777 In: _start          L12  PC: 0x8049000
edx          0x0          0
ebx          0x0          0
esp          0xffffd0b0   0xffffd0b0
ebp          0x0          0x0
esi          0x0          0
--Type <RET> for more, q to quit, c to continue without paging--q
Quit
(gdb) layout regs
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n\034"
(gdb)

```

Рис. 4.16: 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). Меняю первый символ переменной `msg1` (рис. 4.17).

```

(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n\034"
(gdb) set {char}&msg1='h'
(gdb) set {char}0x804a001='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hhllo, "

```

Рис. 4.17: Символ переменной msg1

```

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

```

(gdb)

Меняю символ во второй переменной msg2 (рис. 4.18).

```
(gdb) set {char}0x804a008='L'
(gdb) set {char}0x804a00b=' '
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hhllo, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "Lor d!\n\034"
(gdb)
```

Рис. 4.18: Символ переменная msg2

Чтобы посмотреть значения регистров используется команда print /F (перед именем регистра обязательно ставится префикс \$)

p/F \$<регистр>

С помощью команды set меняю значение регистра ebx. Вывод команд отличается, так как в первом случае мы записывали символ, а во втором случае уже само число (рис. 4.19).

```
(gdb) p/s $ebx
$3 = 0
(gdb) set $ebx='2'
(gdb) p/s $ebx
$4 = 50
(gdb) p/F $edx
No symbol "F" in current context.
(gdb) set $ebx=2
(gdb) p/s $ebx
$5 = 2
(gdb)
```

Рис. 4.19: Изменение значения регистра ebx

(gdb) set \$ebx='2'

(gdb) p/s \$ebx

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

4.2.3 9.4.2.3. 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`:

```
cp ~/work/arch-pc/lab08/lab8-2.asm ~/work/arch-pc/lab09/lab09-3.asm
```

Создаю исполняемый файл.

```
nasm -f elf -g -l lab09-3.lst lab09-3.asm
ld -m elf_i386 -o lab09-3 lab09-3.o
```

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

```
gdb --args lab09-3 аргумент1 аргумент 2 'аргумент 3'
```

Как отмечалось в предыдущей лабораторной работе, при запуске программы аргументы командной строки загружаются в стек. Исследую расположение аргументов командной строки в стеке после запуска программы с помощью `gdb` (рис. 4.20).

```

aymustafina@vbox:~/work/arch-pc/lab09$ gdb --args lab09-3 1 3 '5'
GNU gdb (Fedora Linux) 14.2-1.fc40
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-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 11.
(gdb) run
Starting program: /home/aymustafina/work/arch-pc/lab09/lab09-3 1 3 5

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.

```

Рис. 4.20: Запуск программы

Для начала устанавливаю точку останова перед первой инструкцией в программе и запускаю ее (рис. 4.21).

```

aymustafina@vbox:~/work/arch-pc/lab09$ gdb --args lab09-3 1 3 '5'
GNU gdb (Fedora Linux) 14.2-1.fc40
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-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 11.
(gdb) run
Starting program: /home/aymustafina/work/arch-pc/lab09/lab09-3 1 3 5

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.

```

Рис. 4.21: запуск

(рис. 4.22).

```

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:11
11      pop ecx                ; Извлекаем из стека в `ecx` количество
(gdb)

```

Рис. 4.22: run

```
(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)(рис. 4.23).

```

(gdb) x/x $esp
0xffffd080:      0x00000004
(gdb)

```

Рис. 4.23: Адрес вершины стека

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

```
0xffffd200: 0x05
```

Как видно, число аргументов равно 4 – это имя программы lab09-3 и непосредственно аргументы: аргумент1, аргумент, 2 и ‘аргумент 3’.

Просматриваю 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. (рис. ??).

```

(gdb) x/x $esp
0xffffd080: 0x00000004
(gdb) x/s *(void**)( $esp + 4)
0xffffd24d: "/home/aymustafina/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)( $esp + 8)
0xffffd27a: "1"
(gdb) x/s *(void**)( $esp + 12)
0xffffd27c: "3"
(gdb) x/s *(void**)( $esp + 16)
0xffffd27e: "5"
(gdb) x/s *(void**)( $esp + 20)
0x0: <error: Cannot access memory at address 0x0>
(gdb) x/s *(void**)( $esp + 24)
0xffffd280: "SHELL=/bin/bash"
(gdb)

```

```

(gdb) x/s *(void**)( $esp + 4)
0xffffd358: "~/lab09-3"
(gdb) x/s *(void**)( $esp + 8)
0xffffd3bc: "аргумент1"
(gdb) x/s *(void**)( $esp + 12)
0xffffd3ce: "аргумент"
(gdb) x/s *(void**)( $esp + 16)
0xffffd3df: "2"
(gdb) x/s *(void**)( $esp + 20)
0xffffd3e1: "аргумент 3"
(gdb) x/s *(void**)( $esp + 24)
0x0: <error: Cannot access memory at address 0x0>
(gdb)

```

Шаг изменения адреса равен 4 ([esp+4], [esp+8], [esp+12] и т.д.). Стек в x86 организован таким образом, что каждый элемент стека (например, адреса или указатели) занимает 4 байта. Когда вы обращаетесь к элементам стека, вы фактически работаете с указателями на данные, и каждый указатель занимает 4 байта.

4.3 9.5. Задание для самостоятельной работы

1. Преобразую программу из лабораторной работы №8 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции $f(x)$ как подпрограмму (рис. 4.24).

```
aymustafina@vbox:~/work/arch-pc/lab09$ touch lab9-5.asm
aymustafina@vbox:~/work/arch-pc/lab09$ mc

aymustafina@vbox:~/work/arch-pc/lab09$ nasm -f elf -g -l lab9-5.lst lab9-5.asm
aymustafina@vbox:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-5 lab9-5.o
aymustafina@vbox:~/work/arch-pc/lab09$ gdb --args lab9-5 5
GNU gdb (Fedora Linux) 14.2-1.fc40
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-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 lab9-5...
(gdb) run
Starting program: /home/aymustafina/work/arch-pc/lab09/lab9-5 5

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.
Результат: 45
[Inferior 1 (process 34961) exited normally]
(gdb)
```

Рис. 4.24: Вычисление значения функции $f(x)$

- Листинг для задания 1 из самостоятельной работы *

;----- Задание для самостоятельной работы -----

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

```
SECTION .data
```

```
msg db "Результат: ",0
```

```
SECTION .text
```

```
global _start
```

```
_start:
```

```
    pop ecx          ; Получаем количество аргументов
    pop edx          ; Получаем адрес первого аргумента
    sub ecx, 1        ; Уменьшаем на 1, так как первый аргумент - это имя программы
    mov esi, 0        ; Инициализируем сумму
```

```
next:
```

```
    cmp ecx, 0h      ; Проверка количества аргументов
    jz _end          ; Если нет аргументов, переходим к завершению
```

```
    pop eax          ; Получаем следующий аргумент
    call atoi         ; Преобразуем строку в число
    call f            ; Вызываем функцию f(x)
    add esi, eax       ; Суммируем значения
    loop next         ; Переходим к следующему аргументу
```

```
_end:
```

```
    mov eax, msg      ; Подготовка сообщения "Результат: "
    call sprintf       ; Выводим сообщение
    mov eax, esi       ; Загружаем полученное значение
    call iprintLF      ; Выводим результат
    call quit          ; Завершение программы
```

```
; Подпрограмма для вычисления f(x)
```

```
;  $f(x) = (x + 10) * 3$ 
```

```
f:
```

```
    add eax, 10        ; Прибавляем 10
```

```
imul eax, eax, 3; Умножаем на 3
ret          ; Возвращаемся из подпрограммы
```

2. В листинге 9.3 приведена программа вычисления выражения $(3 + 2) \cdot 4 + 5$. При запуске данная программа дает неверный результат 10. С помощью отладчика GDB, анализируя изменения значений регистров, исправляю ошибку (рис. 4.25).

```
aymustafina@vbox:~/work/arch-pc/lab09$ touch lab9-6.asm
aymustafina@vbox:~/work/arch-pc/lab09$ mc

aymustafina@vbox:~/work/arch-pc/lab09$ nasm -f elf -g -l lab9-6.lst lab9-6.asm
aymustafina@vbox:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-6 lab9-6.o
aymustafina@vbox:~/work/arch-pc/lab09$ gdb --args lab9-6
GNU gdb (Fedora Linux) 14.2-1.fc40
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-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 lab9-6...
(gdb) run
Starting program: /home/aymustafina/work/arch-pc/lab09/lab9-6

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.
Результат: 10
[Inferior 1 (process 35637) exited normally]
(gdb) █
```

Рис. 4.25: Неверный результат

- Листинг 9.3. Программа вычисления выражения $(3 + 2) \cdot 4 + 5$

```
%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

```

Проанализируем значения регистров (рис. 4.26).


```

(gdb) run
Starting program: /home/aymustafina/work/arch-pc/lab09/lab9-6

Breakpoint 1, _start () at lab9-6.asm:11
11      mov ebx,3
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x080490e8 <+0>:    mov     $0x3,%ebx
0x080490ed <+5>:    mov     $0x2,%eax
0x080490f2 <+10>:   add     %eax,%ebx
0x080490f4 <+12>:   mov     $0x4,%ecx
0x080490f9 <+17>:   mul     %ecx
0x080490fb <+19>:   add     $0x5,%ebx
0x080490fe <+22>:   mov     %ebx,%edi
0x08049100 <+24>:   mov     $0x804a000,%eax
0x08049105 <+29>:   call    0x804900f <sprint>
0x0804910a <+34>:   mov     %edi,%eax
0x0804910c <+36>:   call    0x8049086 <iprintf>
0x08049111 <+41>:   call    0x80490db <quit>
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x080490e8 <+0>:    mov     ebx,0x3
0x080490ed <+5>:    mov     eax,0x2
0x080490f2 <+10>:   add     ebx,eax
0x080490f4 <+12>:   mov     ecx,0x4
0x080490f9 <+17>:   mul     ecx
0x080490fb <+19>:   add     ebx,0x5
0x080490fe <+22>:   mov     edi,ebx
0x08049100 <+24>:   mov     eax,0x804a000
0x08049105 <+29>:   call    0x804900f <sprint>
0x0804910a <+34>:   mov     eax,edi
0x0804910c <+36>:   call    0x8049086 <iprintf>
0x08049111 <+41>:   call    0x80490db <quit>
End of assembler dump.
(gdb) █

```

Рис. 4.26: Значения регистров

Я заметила, что есть ошибка, связанная с использованием инструкции mul (рис. 4.27).

```

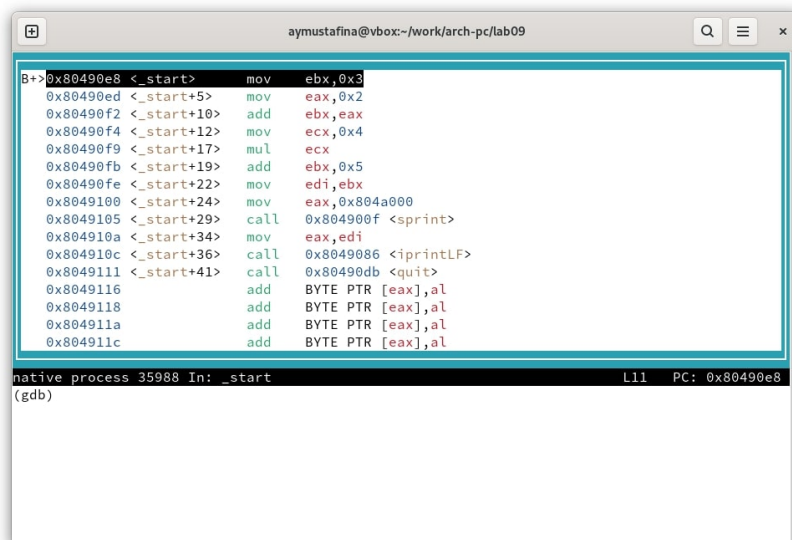
(gdb) break _start
Breakpoint 1 at 0x80490e8: file lab9-6.asm, line 11.
(gdb) run
Starting program: /home/aymustafina/work/arch-pc/lab09/lab9-6

Breakpoint 1, _start () at lab9-6.asm:11
11      mov ebx,3
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x080490e8 <+0>:      mov     $0x3,%ebx
      0x080490ed <+5>:      mov     $0x2,%eax
      0x080490f2 <+10>:     add     %eax,%ebx
      0x080490f4 <+12>:     mov     $0x4,%ecx
      0x080490f9 <+17>:     mul     %ecx
      0x080490fb <+19>:     add     $0x5,%ebx
      0x080490fe <+22>:     mov     %ebx,%edi
      0x08049100 <+24>:     mov     $0x804a000,%eax
      0x08049105 <+29>:     call    0x804900f <sprint>
      0x0804910a <+34>:     mov     %edi,%eax
      0x0804910c <+36>:     call    0x8049086 <iprintLF>
      0x08049111 <+41>:     call    0x80490db <quit>
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x080490e8 <+0>:      mov     ebx,0x3
      0x080490ed <+5>:      mov     eax,0x2
      0x080490f2 <+10>:     add     ebx,eax
      0x080490f4 <+12>:     mov     ecx,0x4
      0x080490f9 <+17>:     mul     ecx
      0x080490fb <+19>:     add     ebx,0x5
      0x080490fe <+22>:     mov     edi,ebx
      0x08049100 <+24>:     mov     eax,0x804a000
      0x08049105 <+29>:     call    0x804900f <sprint>
      0x0804910a <+34>:     mov     eax,edi
      0x0804910c <+36>:     call    0x8049086 <iprintLF>
      0x08049111 <+41>:     call    0x80490db <quit>
End of assembler dump.
(gdb)

```

Рис. 4.27: Инструкция

Инструкция `mul` умножает значение в регистре `eax` на значение, находящееся в другом регистре (в данном случае, `ecx`). Перед вызовом `mul` необходимо убедиться, что в `eax` находится правильное значение, которое нужно умножить. В вашем коде `eax` не инициализируется перед вызовом `mul`, что приводит к неопределенному поведению (рис. 4.28).



```
aymustafina@vbox:~/work/arch-pc/lab09
B+> 0x80490e8 <_start> mov ebx,0x3
0x80490ed <_start+5> mov eax,0x2
0x80490f2 <_start+10> add ebx,eax
0x80490f4 <_start+12> mov ecx,0x4
0x80490f9 <_start+17> mul ecx
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 <iprintLF>
0x8049111 <_start+41> call 0x80490db <quit>
0x8049116 add BYTE PTR [eax],al
0x8049118 add BYTE PTR [eax],al
0x804911a add BYTE PTR [eax],al
0x804911c add BYTE PTR [eax],al
native process 35988 In: _start L11 PC: 0x80490e8
(gdb)
```

Рис. 4.28: Ошибка

Поэтому перед выполнением инструкции `mul` мы загружаем значение из `ebx` (которое равно 5) в `eax`. Это необходимо, потому что `mul` использует значение из `eax` как один из множителей. После выполнения `mul` мы добавляем 5 к значению в `eax`, и затем сохраняем результат в `edi`. (рис. 4.29).

```

aymustafina@vbox:~/work/arch-pc/lab09$ touch lab9-6-1.asm
aymustafina@vbox:~/work/arch-pc/lab09$ mc

aymustafina@vbox:~/work/arch-pc/lab09$ nasm -f elf -g -l lab9-6-1.lst lab9-6-1.asm
aymustafina@vbox:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-6-1 lab9-6-1.o
aymustafina@vbox:~/work/arch-pc/lab09$ gdb --args lab9-6-1
GNU gdb (Fedora Linux) 14.2-1.fc40
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-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 lab9-6-1...
(gdb) run
Starting program: /home/aymustafina/work/arch-pc/lab09/lab9-6-1

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.
Результат: 25
[Inferior 1 (process 36613) exited normally]
(gdb) █

```

Рис. 4.29: Исправленный вариант

- Исправленный листинг 9.3 *

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

```
SECTION .data
```

```
div: DB 'Результат: ',0
```

```
SECTION .text
```

```
GLOBAL _start
```

```
_start:
```

```
    ; ---- Вычисление выражения (3 + 2) * 4 + 5 ----
```

```
    mov ebx, 3      ; Загружаем 3 в ebx
```

```
    add ebx, 2      ; Прибавляем 2 (ebx = 3 + 2 = 5)
```

```

mov eax, ebx    ; Загружаем результат (5) в eax
mov ecx, 4      ; Загружаем 4 в ecx
mul ecx         ; Умножаем eax на ecx (eax = 5 * 4 = 20)

add eax, 5      ; Прибавляем 5 (eax = 20 + 5 = 25)

mov edi, eax    ; Сохраняем результат в edi

; ---- Вывод результата на экран ----
mov eax, div     ; Загружаем адрес строки результата
call sprint      ; Выводим строку "Результат: "

mov eax, edi     ; Загружаем результат в eax для вывода
call iprintLF    ; Выводим результат с переводом строки

call quit       ; Завершение программы

```

5 Выводы

Я приобрела навыки написания программ с использованием подпрограммы.
Узнала о методах отладки при помощи GDB и его основными возможностями.

6 Список литературы

Лабораторная работа №9