

Отчёт по лабораторной работе №9

***Дисциплина: Архитектура компьютера**

Долгаев Евгений Сергеевич

Содержание

1 Цель работы	6
2 Задание	7
3 Теоретическое введение	8
3.1 Понятие об отладке	8
3.2 Методы отладки	9
3.3 Основные возможности отладчика GDB	10
3.4 Запуск отладчика GDB; выполнение программы; выход	11
3.5 Дизассемблирование программы	12
3.6 Точки останова	12
3.7 Пошаговая отладка	13
3.8 Работа с данными программы в GDB	14
3.9 Понятие подпрограммы	16
3.9.1 Инструкция call и инструкция ret	16
4 Выполнение лабораторной работы	18
5 Задание для самостоятельной работы	28
5.1 Задание 1	28
5.2 Задание 2	29
6 Выводы	31
Список литературы	32

Список иллюстраций

3.1 Основные моменты выполнения подпрограммы	17
4.1 Подготовка рабочего пространства	18
4.2 Текст программы	18
4.3 Создание и работа исполняемого файла	19
4.4 Текст программы	19
4.5 Создание и работа исполняемого файла	19
4.6 Создание файла	20
4.7 Текст программы	20
4.8 Создание файла	20
4.9 Загрузка в отладчик	20
4.10 Запуск программы	21
4.11 Установка брейкпойнта	21
4.12 Дисассимилированный код	21
4.13 Дисассимилированный код Intel	22
4.14 Режим псевдографики	22
4.15 Установка брейкпоинта	23
4.16 Изменение регистров	23
4.17 Изменение регистров	24
4.18 Содержение переменной msg1	24
4.19 Содержение переменной msg2	24
4.20 Работа команды set	24
4.21 Работа команды set	25
4.22 Вывод значения регистра	25
4.23 Работа команды set	25
4.24 Работа команды set	25
4.25 Копирование файла	26
4.26 Создание исполняемого файла	26
4.27 Установка брейкпоинта	26
4.28 Регистр esp	26
4.29 Остальные позиции стека	27
5.1 Текст программы	28
5.2 Создание и работа исполняемого файла	28
5.3 Текст программы	29
5.4 Создание и работа исполняемого файла	29
5.5 Текст программы	30

5.6 Создание и работа исполняемого файла	30
--	----

Список таблиц

3.1 Формат отображения данных команды x	14
---	----

1 Цель работы

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

2 Задание

- 1) Выполнение лабораторной работы
 - 1) Реализация подпрограмм в NASM
 - 2) Отладка программам с помощью GDB
- 2) Задание для самостоятельной работы

3 Теоретическое введение

3.1 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, 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.

3.2 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 программаотладчик 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 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 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 у (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 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 :

```
(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

3.6 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 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 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 данные (см. таблицу 3.1).

Таблица 3.1: Формат отображения данных команды `x`

Значение	Описание
N	Десятичное число Счётчик повторений. Определяет, сколько ячеек памяти отобразить (считая в единицах), по умолчанию 1.
F	Формат отображения

Значение	Описание
s	строка оканчивающаяся нулём
x	машинная инструкция
i	шестнадцатеричное число
a	адрес
U	Размер отображаемых ячеек памяти
b	байт
h	полуслово, 2 байта
w	машинное слово, 4 байта(значение по умолчанию)
g	длинное слово, 8 байт

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 (сокращенно p). Перед именем регистра обязательно ставится префикс \$. Например, команда p/x \$есх 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 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 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 `eip` 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 `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. 3.1.



Рис. 3.1: Основные моменты выполнения подпрограммы

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

Создим каталог для выполнения лабораторной работы № 9, перейдём в него и со-здадим файл lab09-1.asm(рис. 4.1)

```
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ cd  
esdolgaev@esdolgaev-VirtualBox: $ mkdir ~/work/arch-pc/Lab09  
esdolgaev@esdolgaev-VirtualBox: $ cd ~/work/arch-pc/lab09  
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ touch lab9-1.asm  
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ls  
lab9-1.asm  
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ cp ~/work/arch-pc/lab08/in_out.asm ~/work/arch-pc/lab09/  
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ls  
in_out.asm lab9-1.asm  
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$
```

Рис. 4.1: Подготовка рабочего пространства

В качестве примера рассмотрим программу вычисления арифметического выражения $f(x) = 2x + 7$ с помощью подпрограммы _calcul. В данном примере x вводится с клавиатуры, а само выражение вычисляется в подпрограмме.

Введём в файл lab09-1.asm текст программы. Создим исполняемый файл и проверим его работу(рис. 4.2, 4.3).

```
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 iprintf  
call quit  
;  
;-----  
; Подпрограмма вычисления  
; выражения "2x+7"  
.calcul:  
mov ebx,2  
mul ebx  
add eax,7  
mov [res],eax  
ret ; выход из подпрограммы
```

1 Помощь 2 Сохранить 3 Блок 4 Замена 5 Копия 6 Перем-тить 7 Поиск 8 Удалить 9 меню ИС 10 Выход

Рис. 4.2: Текст программы

```

esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ nasm -f elf lab9-1.asm
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ld -m elf_i386 -o lab9-1 lab9-1.o
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ./lab9-1
Введите x: 4
2x+7=15
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ./lab9-1
Введите x: 5
2x+7=17
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ 

```

Рис. 4.3: Создание и работа исполняемого файла

Изменим 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. 4.4, 4.5).

```

; Основная программа
-----
mov eax, msg
call sprint
mov ecx, x
mov edx, 0
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
mov [res],eax
ret ; выход из подпрограммы

_subcalcul:
imul eax,3
sub eax,1
ret

```

Рис. 4.4: Текст программы

```

esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 4
f(g(x))=29
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ 

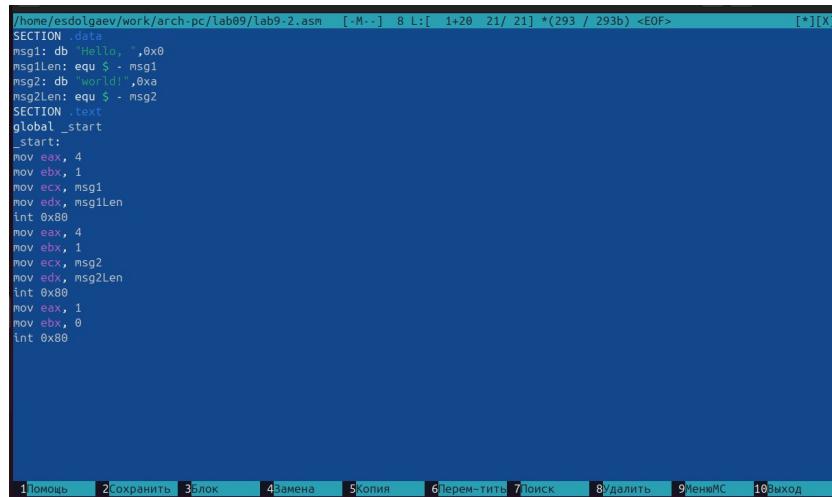
```

Рис. 4.5: Создание и работа исполняемого файла

Создим файл lab09-2.asm с текстом программы. (Программа печати сообщения Hello world!)(рис. 4.6, 4.7):

```
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab09$ touch lab9-2.asm
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab09$ ls
in_out.asm lab9-1.o lab9-2.o
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/Lab09$
```

Рис. 4.6: Создание файла



```
/home/esdolgaev/work/arch-pc/lab09/lab9-2.asm [-M--] 8 L:[ 1+20 21/ 21 ] *(293 / 293b) <EOF>
[*][X]
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, 9
    int 0x80

1 Помощь 2 Сохранить 3 Блок 4 Замена 5 Копия 6 Перем-тить 7 Поиск 8 Удалить 9 ПечатиC 10 Выход
```

Рис. 4.7: Текст программы

Получим исполняемый файл. Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию программ необходимо проводить с ключом ‘-g’(рис. 4.8).

```
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab09$ nasm -f elf -g -l lab9-2.lst lab9-2.asm
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab09$ ld -m elf_i386 -o lab9-2 lab9-2.o
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab09$ ls
in_out.asm lab9-1.o lab9-1.asm lab9-2.o lab9-2.asm lab9-2.lst lab9-2.o
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/Lab09$
```

Рис. 4.8: Создание файла

Загрузим исполняемый файл в отладчик gdb(рис. 4.9):



```
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab09$ gdb lab9-2
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
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-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 lab9-2...
(gdb)
```

Рис. 4.9: Загрузка в отладчик

Проверим работу программы, запустив ее в оболочке GDB с помощью команды run (сокращённо r)(рис. 4.10):

```
(gdb) run
Starting program: /home/esdolgaev/work/arch-pc/lab09/lab9-2

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
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 0xf7fc000
Hello, world!
[Inferior 1 (process 8200) exited normally]
(gdb) █
```

Рис. 4.10: Запуск программы

Для более подробного анализа программы установим брейкпойнт на метку _start, с которой начинается выполнение любой ассемблерной программы, и запустим её(рис. 4.11).

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab9-2.asm, line 9.
(gdb) run
Starting program: /home/esdolgaev/work/arch-pc/lab09/lab9-2

Breakpoint 1, _start () at lab9-2.asm:9
9  mov    eax, 4
(gdb) █
```

Рис. 4.11: Установка брейкпомта

Посмотрим дисассимилированный код программы с помощью команды disassemble начиная с метки _start(рис. 4.12).

```
(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    $0x9,%edx
 0x08049014 <+20>: int    $0x80
 0x08049016 <+22>: mov    $0x1,%eax
 0x0804901b <+27>: mov    $0x1,%ebx
 0x08049020 <+32>: mov    $0x804a000,%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. 4.12: Дисассимилированный код

Переключимся на отображение команд с Intel'овским синтаксисом, введя команду set disassembly-flavor intel(рис. 4.13).

```
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x00049000 <_start>:    mov    eax,0x4
  0x00049005 <+5>:    mov    ebx,0x1
  0x0004900a <+10>:   mov    ecx,0x804a000
  0x0004900f <+15>:   mov    edx,0x8
  0x00049014 <+20>:   int    0x80
  0x00049016 <+22>:   mov    eax,0x4
  0x0004901b <+27>:   mov    ebx,0x1
  0x00049020 <+32>:   mov    ecx,0x804a000
  0x00049025 <+37>:   mov    edx,0x7
  0x0004902a <+42>:   int    0x80
  0x0004902c <+44>:   mov    eax,0x1
  0x00049031 <+49>:   mov    ebx,0x0
  0x00049036 <+54>:   int    0x80
End of assembler dump.
(gdb)
```

Рис. 4.13: Дисассимилированный код Intel

Имена регистров и их значения написаны в обратном порядке и без спецсимволов Включим режим псевдографики для более удобного анализа программы. На предыдущих шагах была установлена точка останова по имени метки (_start). Продвигните это с помощью команды info breakpoints (кратко i b)(рис. 4.14).

```
Register group: general
eax      0x0          0           ecx      0x0          0
edx      0x0          0           ebx      0x0          0
esp     0xfffffd020  0xfffffd020  ebp      0x0          0x0
est      0x0          0           edi      0x0          0
eip     0x00049000  0x00049000 <_start>  eflags   0x202        [  IF  ]
cs       0x23         35          ss       0x2b          43
ds       0x2b         43          es       0x2b          43
fs       0x0          0           gs       0x0          0

B+>0x00049000 < start>:    mov    eax,0x4
0x00049001 < start+5>:    mov    ebx,0x1
0x0004900a < start+10>:   mov    ecx,0x804a000
0x0004900f < start+15>:   mov    edx,0x8
0x00049014 < start+20>:   int    0x80
0x00049016 < start+22>:   mov    eax,0x4
0x0004901b < start+27>:   mov    ebx,0x1
0x00049020 < start+32>:   mov    ecx,0x804a000
0x00049025 < start+37>:   mov    edx,0x7

native process 8295 (asm) In: _start
L9      PC: 0x00049000
(gdb) layout regs
(gdb) i b
Num  Type      Disp Enb Address  What
1    breakpoint  keep y  0x00049000 lab9-2.asm:9
      breakpoint already hit 1 time
(gdb) ■
```

Рис. 4.14: Режим псевдографики

Установим еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции(рис. 4.15).

```

Register group: general-
eax    0x0          0           ecx    0x0          0
edx    0x0          0           ebx    0x0          0
esp    0xfffffd020  0xfffffd020 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
fs     0x0          0           gs     0x0          0

0x8049016 <_start+22>  mov    eax,0x4
0x804901b <_start+27>  mov    ebx,0x1
0x8049028 <_start+32>  mov    ecx,0x804a088
0x8049033 <_start+37>  mov    edx,0x7
0x8049032 <_start+42>  int    0x80
0x8049030 <_start+44>  mov    eax,0x1
b+ 0x8049031 <_start+49>  mov    ebx,0x8
0x8049036 <_start+54>  int    0x80
0x8049038 add    BYTE PTR [eax],al

native process 8295 (asm) In: _start
Make breakpoint pending on future shared library load? (y or [n]) [n]
Please answer y or [n].
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) b *0x8049031
Breakpoint 2 at 0x8049031: file lab9-2.asm, line 20.
(gdb) i b
Num  Type        Disp Enb Address  What
1   breakpoint  keep y  0x08049000 lab9-2.asm:9
2   breakpoint  keep y  0x08049031 lab9-2.asm:20
(gdb) 

```

Рис. 4.15: Установка брейкпойнта

Выполните 5 инструкций с помощью команды stepi (или si)(рис. 4.16).

```

eax    0x4          4           ecx    0x804a080  134526832
eax    0x1          1           ecx    0x804a088  134526840
edx    0x7          7           ebp    0x0          0x0
esi    0x0          0           edi    0x0          0
eip    0x804901b   0x804901b < start+27> eflags 0x202      [ IF ]
cs     0x23         31          ss     0x2b         43
ds     0x2b         43          es     0x2b         43
fs     0x0          0           gs     0x0          0

b+ 0x8049000 <_start>  mov    eax,0x4
0x804901b <_start+27>  mov    ebx,0x1
0x8049028 <_start+32>  mov    ecx,0x804a088
0x8049033 <_start+37>  mov    edx,0x7
0x8049032 <_start+42>  int    0x80
0x8049030 <_start+44>  mov    eax,0x1
B+0x8049031 <_start+49>  mov    ebx,0x0
0x8049036 <_start+54>  int    0x80  04a088
0x8049038 add    BYTE PTR [eax],al
d+ 0x8049039 <_start+55> add    BYTE PTR [eax],al

native process 8295 (asm) In: _start
L15 PC: 0x804901b
cs    0x23         35          28          31
fs    0x0          0           0
gs    0x0          0           0

(gdb) si 1
(gdb) si 2
(gdb) si 3
(gdb) si 4
world!
(gdb) si 5

Breakpoint 2, _start () at lab9-2.asm:20
(gdb) 

```

Рис. 4.16: Изменение регистров

Посмотрим содержимое регистров с помощью команды info registers(или ir)(рис. 4.17).

```
native process 8295 (asm) In: _start
cs          0x23      35
eax         0x1       1
ebx         0x1       1
esp         0xfffffd020 0xfffffd020
ebp         0x0       0
esi         0x0       0
edi         0x0       0
eip         0x8049031 0x8049031 <_start+49>
eflags      0x20      [ IF ]
--Type <RET> for more, q to quit, c to continue without paging--q
Quit
(gdb) ■
```

Рис. 4.17: Изменение регистров

С помощью команды `x &` также можно посмотреть содержимое переменной.

Посмотрите значение переменной `msg1` по имени(рис. 4.18).

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

Рис. 4.18: Содержание переменной `msg1`

Посмотрим значение переменной `msg2` по адресу. Адрес переменной можно определить по дизассемблированной инструкции(рис. 4.19).

```
(gdb) x/1sb 0x804a008      "world!\n\034"
0x804a008 <msg2>:      "world!\n\034"
(gdb) ■
```

Рис. 4.19: Содержание переменной `msg2`

Изменить значение для регистра или ячейки памяти можно с помощью команды `set`, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс `$`, а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си).

Изменим первый символ переменной `msg1`(рис. 4.20).

```
(gdb) set {char}&msg1='h'
(gdb) x/sb1 &msg1
'msg1' has unknown type; cast it to its declared type
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hello, "
(gdb) ■
```

Рис. 4.20: Работа команды `set`

Заменим символ во второй переменной `msg2`(рис. 4.21).

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

Рис. 4.21: Работа команды set

Выведем в различных форматах значение регистра edx(рис. 4.22).

```
(gdb) p/s $edx  
$5 = 7  
(gdb) p/f $edx  
$6 = 9.80908925e-45  
(gdb) p/x $edx  
$7 = 0x7  
(gdb) p/t $edx  
$8 = 111  
(gdb) █
```

Рис. 4.22: Вывод значения регистра

С помощью команды set изменим значение регистра ebx(рис. 4.23, 4.24):

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

Рис. 4.23: Работа команды set

```
(gdb) set $ebx=2  
(gdb) p/s $ebx  
$10 = 2  
(gdb) █
```

Рис. 4.24: Работа команды set

Скопируем файл lab8-2.asm, созданный при выполнении лабораторной работы №8, с программой выводящей на экран аргументы командной строки в файл с именем lab09-3.asm(рис. 4.25):

```
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ cp ~/work/arch-pc/lab08/lab8-2.asm ~/work/arch-pc/lab09/lab09-3.asm
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ls
in_out.asm  lab9-1.o  lab9-2.asm  lab9-2.o
lab09-3.asm  lab9-1.asm  lab9-2.o  lab9-2.lst
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$
```

Рис. 4.25: Копирование файла

Создайте исполняемый файл. Для загрузки в gdb программы с аргументами необходимо использовать ключ –args. Загрузите исполняемый файл в отладчик, указав аргументы(рис. 4.26).

```
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ gdb --args lab09-3 аргумент1 аргумент2 'аргумент3'
GNU gdb (Ubuntu 15.0.50.20240403-ubuntu1) 15.0.50.20240403-glt
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-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
```

Рис. 4.26: Создание исполняемого файла

Установим точку останова перед первой инструкцией в программе и запустим ее(рис. 4.27).

```
(gdb) b _start
Breakpoint 1 at 0x8049000: file lab09-3.asm, line 5.
(gdb) run
Starting program: /home/esdolgaev/work/arch-pc/lab09/lab09-3 1 2 3

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
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

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

Рис. 4.27: Установка брейкпоинта

Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы)(рис. 4.28):

```
(gdb) x/x $esp
0xfffffcfe:    0x00000005
(gdb)
```

Рис. 4.28: Регистр esp

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(рис. 4.29).

```
(gdb) x/s *(void**)(Sesp + 4)
0xffffd100: "/home/esdolgaev/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)(Sesp + 8)
0xffffd104: "аргумент1"
(gdb) x/s *(void**)(Sesp + 12)
0xffffd108: "аргумент"
(gdb) x/s *(void**)(Sesp + 16)
0xffffd10c: "2"
(gdb) x/s *(void**)(Sesp + 20)
0xffffd110: "аргумент 3"
(gdb) x/s *(void**)(Sesp + 24)
0xffffd114: "аргумент 3"
(gdb) x/s *(void**)(Sesp + 28)
0xffffd118: <error: Cannot access memory at address 0x0>
(gdb)
```

Рис. 4.29: Остальные позиции стека

5 Задание для самостоятельной работы

5.1 Задание 1

Преобразуем программу из лабораторной работы №8 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции как подпрограмму(рис. 5.1, 5.2).

```
/home/esdolgaev/work/arch-pc/lab09/variant8 [---] 0 L:[ 8+ 3 11/ 37] *(164 / 712b) 0059 0x03B [*][X]
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 tprintf
call quit
; -----
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
mov ebx,2
mul ebx
add eax,7
mov [res],eax
ret ; выход из подпрограммы
```

Рис. 5.1: Текст программы

```
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ nasm -f elf variant8.asm
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ld -m elf_i386 -o variant8 variant8.o
Ведите x: 4
f(x)=2x+7=15
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$
```

Рис. 5.2: Создание и работа исполняемого файла

5.2 Задание 2

Введём созданный файл текст программы и убедимся, что она работает неправильно(рис. 5.3, 5.4).

```
%include 'in_out.asm'
SECTION .data
d1v: DB 'Результат: ',0
SECTION .text
GLOBAL _start
_start:
; ----- Вычисление выражения (3+2)*4+5
mov ebx,3
mov eax,2
add edx, eax
mov ecx,4
mul ecx
add ebx,5
mov edi,ebx
; ----- Вывод результата на экран
mov eax,d1v
call sprint
mov eax,edi
call iprintf
call quit
```

Рис. 5.3: Текст программы

```
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab0$ nasm -f elf zadanie2.asm
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab0$ ld -m elf_i386 -o zadanie zadanie2.o
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab0$ ./zadanie2
bash: ./zadanie2: Нет такого файла или каталога
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab0$ ld -m elf_i386 -o zadanie2 zadanie2.o
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab0$ ./zadanie2
Результат: 10
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab0$
```

Рис. 5.4: Создание и работа исполняемого файла

С помощью отладчика GDB можно прийти к выводу: имена регистров в аргументах к командам mov, add, mull и т.д. указаны неправильно.

Исправим код программы создадим исполняемый файл и проверим его работы(рис. 5.5, 5.6).

```
#!/home/esdolgaev/work/arch-pc/lab09/zadanie2.asm [---] 10 L:[ 1+13 14/ 20 ] *(232 / 348b) 0120 0x078 [*][X]
#include "in_out.asm"
SECTION .data
div: DB 'Результат: ',0
SECTION .text
GLOBAL _start
_start:
; ---- Вычисление выражения (3+2)*4+5
mov eax,3
mov ebx,2
add eax,ebx
mov ebx,4
mul ebx
add eax,5
mov edi, eax
; ---- Вывод результата на экран
mov eax,div
call sprint
mov eax,edi
call iprintLF
call quit
```

Рис. 5.5: Текст программы

```
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ^C
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ nasm -f elf zadanie2.asm
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ^
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ld -m elf_i386 -o zadanie2 zadanie2
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$ ./zadanie2
Результат: 25
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab09$
```

Рис. 5.6: Создание и работа исполняемого файла

6 Выводы

В ходе выполнения лабораторной работы я приобрёл навыки написания программ с использованием подпрограмм и познакомился с методами отладки при помощи GDB и его основными возможностями.

Список литературы