

Отчёт по лабораторной работе №9

Дисциплина: Архитектура компьютера

София Андреевна Кудякова

Содержание

1	Цель работы	4
2	Задания	5
3	Теоретическое введение	6
4	Выполнение лабораторной работы	10
4.1	Реализация подпрограмм в NASM	10
4.2	Отладка программ с помощью GDB	14
4.2.1	Добавление точек останова	17
4.2.2	Работа с данными программы в GDB	18
4.2.3	Обработка аргументов командной строки в GDB	21
4.3	Выполнение заданий для самостоятельной работы	22
5	Выводы	28
	Список литературы	29

Список иллюстраций

4.1	Создание директории и файла	10
4.2	Редактирование файла	11
4.3	Запуск программы файла	12
4.4	Редактирование файла	13
4.5	Запуск программы файла	13
4.6	Создание файла	14
4.7	Редактирование файла	14
4.8	Получение исполняемого файла и загрузка файла в окладчик . . .	15
4.9	Проверка работы программы	15
4.10	Установка брейкпоинта и запуск программы	15
4.11	Команды <code>disassemble</code> и <code>set disassembly-flavor intel</code>	16
4.12	Точки останова	17
4.13	Выполнение инструкции <code>si</code>	18
4.14	Выполнение 5 инструкций <code>si</code>	18
4.15	Просмотр значений	19
4.16	Команда <code>set</code>	19
4.17	Команда <code>print</code> и команда <code>set</code>	20
4.18	Выход из GDB	21
4.19	Копирование файла	21
4.20	Создание исполняемого файла и его загрузка в отладчик	22
4.21	Установка точки останова и запуск программы	22
4.22	Просмотр значений	22
4.23	Редактирование файла	23
4.24	Запуск программы	24
4.25	Редактирование файла	25
4.26	Запуск программы	26
4.27	Исправление ошибки	26
4.28	Запуск программы	27

1 Цель работы

Цель данной лабораторной работы - научиться писать программы с использованием подпрограмм, а также ознакомиться с методами отладки при помощи GDB и его основными возможностями.

2 Задания

1. Реализация подпрограмм в NASM
2. Отладка программ с помощью GDB
3. Добавление точек останова
4. Работа с данными программы в GDB
5. Обработка аргументов командной строки в GDB
6. Выполнение заданий для самостоятельной работы

3 Теоретическое введение

Отладка — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа: обнаружение ошибки; поиск её местонахождения; определение причины ошибки; исправление ошибки. Можно выделить следующие типы ошибок: синтаксические ошибки — обнаруживаются во время трансляции исходного кода и вызваны нарушением ожидаемой формы или структуры языка; семантические ошибки — являются логическими и приводят к тому, что программа запускается, 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. 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 строки, чтобы программист мог проверить значения переменных и выполнить другие действия. Точки останова — это специально отмеченные места в программе, в которых программа-отладчик приостанавливает выполнение программы и ждёт команд. Наиболее популярные виды точек останова: Breakpoint — точка останова (остановка происходит, когда выполнение доходит до определённой строки, адреса или процедуры, отмеченной программистом); Watchpoint — точка просмотра (выполнение программы приостанавливается, если программа обратилась к определённой переменной: либо считала её значение, либо изменила его). Точки останова устанавливаются в отладчике на время сеанса работы с кодом программы, т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы. GDB (GNU Debugger — отладчик проекта GNU) [1] работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования. GDB предлагает обширные средства для слежения и контроля за выполнением компьютерных программ. Отладчик не содержит собственного графического пользовательского интерфейса и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторонних графических надстроек, а кроме того, некоторые интегрированные среды разработки используют его в качестве базовой подсистемы отладки. Отладчик GDB (как и любой другой отладчик) позволяет увидеть, что происходит «внутри» программы в момент её выполнения или что делает программа в момент сбоя. GDB может выполнять следующие действия: начать выполнение программы, задав всё, что может повлиять на её поведение; остановить программу при указанных условиях; исследовать, что случилось, когда программа остановилась; изменить программу так, чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других. 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`) Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать как имя метки или как адрес. Информацию о всех установленных точках останова можно вывести командой `info` (кратко `i`). Для того чтобы сделать неактивной какую-нибудь ненужную точку останова, можно воспользоваться командой `disable`. Обратная точка останова активируется командой `enable`. Если же точка останова в дальнейшем больше не нужна, она может быть удалена с помощью команды `delete`. Ввод этой команды без аргумента удалит все точки останова. Информацию о командах этого раздела можно получить, введя

```
help breakpoints
```


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. Команда `stepi` (`si`) (`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. Информацию о командах этого раздела можно получить, введя

```
(gdb) help running
```

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`). 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.

4 Выполнение лабораторной работы

4.1 Реализация подпрограмм в NASM

Ввожу команду `mkdir`, с помощью которой создаю директорию, в которой буду создавать файлы. Перехожу в нее. С помощью команды `touch` создаю файл `lab09-1.asm`. (рис. 4.1).

```
sakudyakova@dk4n68 ~/work/study/2023-2024/Архитектура компьютера/arch-pc $ mkdir lab09
sakudyakova@dk4n68 ~/work/study/2023-2024/Архитектура компьютера/arch-pc $ cd lab09
sakudyakova@dk4n68 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ touch lab09-1.asm
sakudyakova@dk4n68 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ls
in_out.asm  lab09-1.asm
sakudyakova@dk4n68 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ █
```

Рис. 4.1: Создание директории и файла

Открываю созданный файл в редакторе и вставляю в него программу с использованием вызова подпрограммы. (рис. 4.2).

```

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

```

Рис. 4.2: Редактирование файла

Создаю исполняемый файл и запускаю его. (рис. 4.3).

```
sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ nasm -f elf lab09-1.asm
sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ld -m elf_i386 -o lab09-1 lab09-1.o
sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ./lab09-1
Введите x: 5
2x+7=17
```

Рис. 4.3: Запуск программы файла

Изменяю 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).

```

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

```

Рис. 4.4: Редактирование файла

Создаю исполняемый файл и запускаю его. (рис. 4.5).

```

sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ nasm -f elf lab09-1.asm
sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ld -m elf_i386 -o lab09-1 lab09-1.o
sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ./lab09-1
Введите x: 5
2x+7=35

```

Рис. 4.5: Запуск программы файла

4.2 Отладка программ с помощью GDB

С помощью команды `touch` создаю файл `lab09-2.asm`. (рис. 4.6).

```
sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ touch lab09-2.asm
sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $
```

Рис. 4.6: Создание файла

Ввожу в файл текст программы вывода сообщения `Hello world!`. (рис. 4.7).

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

Рис. 4.7: Редактирование файла

Получаю 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).

```

sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ nasm -f elf -g -l lab09-2.lst lab09-2.asm
sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ld -m elf_i386 -o lab09-2 lab09-2.o
sakudyakova@dk2n24 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ gdb lab09-2
GNU gdb (Gentoo 13.2 vanilla) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab09-2...
(gdb) █

```

Рис. 4.8: Получение исполняемого файла и загрузка файла в окладчик

Проверяю работу программы, запустив ее в оболочке GDB с помощью команды `run`. (рис. 4.9).

```

(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/s/a/sakudyakova/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09/lab09-2
Hello, world!
[Inferior 1 (process 6460) exited normally]
(gdb) █

```

Рис. 4.9: Проверка работы программы

Для более подробного анализа программы устанавливаю брейкпоинт на метку `_start`, с которой начинается выполнение любой ассемблерной программы, и запускаю её. (рис. 4.10).

```

(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/s/a/sakudyakova/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09/lab09-2
Breakpoint 1, _start () at lab09-2.asm:9
9      mov eax, 4
(gdb) █

```

Рис. 4.10: Установка брейкпоинта и запуск программы

Смотрю дисассимилированный код программы с помощью команды `disassemble` начиная с метки `_start`. Переключаюсь на отображение команд с Intel'овским синтаксисом, введя команду `set disassembly-flavor intel`. (рис. 4.11).

```

(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) 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. 4.11: Команды disassemble и set disassembly-flavor intel

Различия отображения синтаксиса машинных команд в режимах АТТ и Intel: в режиме АТТ имена регистров начинаются с символа %, а имена операндов с \$, в

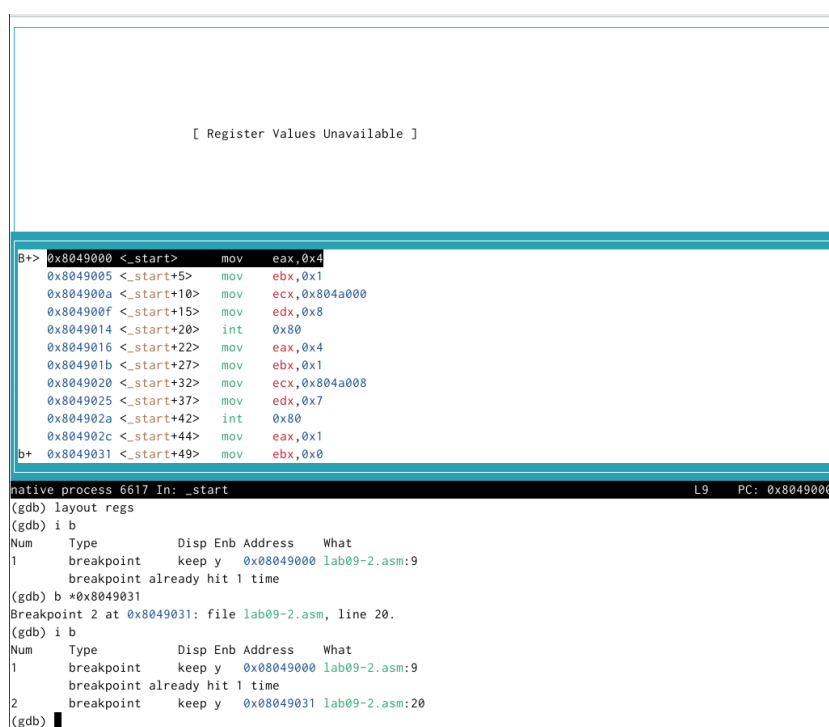
то время как в Intel используется привычный синтаксис.

Включаю режим псевдографики для более удобного анализа программы. (рис. ??).

[Включение режима псевдографики] (image/lab09_12.png){#fig:12 width=70%}

4.2.1 Добавление точек останова

На предыдущих шагах была установлена точка останова по имени метки (`_start`). Проверяю это с помощью команды `info breakpoints` (кратко `i b`). Устанавливаю еще одну точку останова по адресу инструкции. Затем смотрю информацию о всех установленных точках останова. (рис. 4.12).



The screenshot shows a GDB terminal window. At the top, a message says "[Register Values Unavailable]". Below that, a list of assembly instructions is displayed, starting with `B+> 0x8049000 <_start> mov eax,0x4`. The instructions are: `0x8049005 <_start+5> mov ebx,0x1`, `0x804900a <_start+10> mov ecx,0x804a000`, `0x804900f <_start+15> mov edx,0x8`, `0x8049014 <_start+20> int 0x80`, `0x8049016 <_start+22> mov eax,0x4`, `0x804901b <_start+27> mov ebx,0x1`, `0x8049020 <_start+32> mov ecx,0x804a008`, `0x8049025 <_start+37> mov edx,0x7`, `0x804902a <_start+42> int 0x80`, `0x804902c <_start+44> mov eax,0x1`, and `b+ 0x8049031 <_start+49> mov ebx,0x0`. Below the instructions, the GDB prompt shows `native process 6617 In: _start` and `L9 PC: 0x8049000`. The user enters `(gdb) layout regs`, `(gdb) i b`, and `(gdb) b *0x8049031`. The output shows two breakpoints: `1 breakpoint keep y 0x8049000 lab09-2.asm:9` and `2 breakpoint keep y 0x8049031 lab09-2.asm:20`. The first breakpoint is already hit 1 time.

Рис. 4.12: Точки останова

4.2.2 Работа с данными программы в GDB

Выполняю 5 инструкций с помощью команды `stepi` (или `si`) и смотрю за изменением значений регистров. (рис. 4.13).

```
Register group: general
eax      0x4      4      ecx      0x0      0
edx      0x0      0      ebx      0x0      0
esp      0xffffc2b0 0xffffc2b0  ebp      0x0      0x0
esi      0x0      0      edi      0x0      0
eip      0x8049005 0x8049005 <_start+5>  eflags  0x202    [ IF ]
cs       0x23     35     ss       0x2b     43
ds       0x2b     43     es       0x2b     43
fs       0x0      0      gs       0x0      0

B+ 0x8049000 <_start> mov eax,0x4
> 0x8049005 <_start+5> mov ebx,0x1
0x804900a <_start+10> mov ecx,0x804a000
0x804900f <_start+15> mov edx,0x8
0x8049014 <_start+20> int 0x80
0x8049016 <_start+22> mov eax,0x4
0x804901b <_start+27> mov ebx,0x1
0x8049020 <_start+32> mov ecx,0x804a000
0x8049025 <_start+37> mov edx,0x7
0x804902a <_start+42> int 0x80

Native process 3789 In: _start L10 PC: 0x8049005
Num Type Disp Enb Address What
1 breakpoint keep y 0x8049000 lab09-2.asm:9
breakpoint already hit 1 time
(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 0x8049000 lab09-2.asm:9
breakpoint already hit 1 time
2 breakpoint keep y 0x8049031 lab09-2.asm:20
(gdb) si
(gdb)
```

Рис. 4.13: Выполнение инструкции `si`

Выполняю 4 остальные инструкции `si`. (рис. 4.14).

```
Register group: general
eax      0x8      8      ecx      0x804a000 134520832
edx      0x8      8      ebx      0x1      1
esp      0xffffc2b0 0xffffc2b0  ebp      0x0      0x0
esi      0x0      0      edi      0x0      0
eip      0x8049016 0x8049016 <_start+22>  eflags  0x202    [ IF ]
cs       0x23     35     ss       0x2b     43
ds       0x2b     43     es       0x2b     43
fs       0x0      0      gs       0x0      0

B+ 0x8049000 <_start> mov eax,0x4
0x8049005 <_start+5> mov ebx,0x1
0x804900a <_start+10> mov ecx,0x804a000
0x804900f <_start+15> mov edx,0x8
0x8049014 <_start+20> int 0x80
> 0x8049016 <_start+22> mov eax,0x4
0x804901b <_start+27> mov ebx,0x1
0x8049020 <_start+32> mov ecx,0x804a000
0x8049025 <_start+37> mov edx,0x7
0x804902a <_start+42> int 0x80
```

Рис. 4.14: Выполнение 5 инструкций `si`

Изменились значения регистров `eax`, `ebx`, `ecx`, `edx`.

Посматриваю значение переменной msg1 по имени и значение msg2 по ее адресу.(рис. 4.15).

```
--Register group: general
eax 0x8 8 ecx 0x804a000 134520832
edx 0x8 8 ebx 0x1 1
esp 0xffffc2b0 0xffffc2b0 ebp 0x0 0x0
esi 0x0 0 edi 0x0 0
eip 0x8049016 0x8049016 <_start+22> eflags 0x202 [ IF ]
cs 0x23 35 ss 0x2b 43
ds 0x2b 43 es 0x2b 43
fs 0x0 0 gs 0x0 0

0+ 0x8049000 <_start> mov eax,0x4
0x8049005 <_start+5> mov ebx,0x1
0x804900a <_start+10> mov ecx,0x804a000
0x804900f <_start+15> mov edx,0x8
0x8049014 <_start+20> int 0x80
> 0x8049016 <_start+22> mov eax,0x4
0x804901b <_start+27> mov ebx,0x1
0x8049020 <_start+32> mov ecx,0x804a008
0x8049025 <_start+37> mov edx,0x7
0x804902a <_start+42> int 0x80

native process 3789 In: _start L14 PC: 0x8049016
(gdb) x/1sb &msg1
0x804a000 <msg1>: "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>: "world!\n\034"
(gdb)
```

Рис. 4.15: Просмотр значений

Изменяю первый символ переменной msg1 с помощью команды set и заменяю первый символ переменной msg2 .(рис. 4.16).

```
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>: "hello, "
(gdb) set {char}&msg2='b'
(gdb) x/1sb &msg2
0x804a008 <msg2>: "borld!\n\034"
(gdb)
```

Рис. 4.16: Команда set

Вывожу в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра edx с помощью команды print p/s \$. Далее изменяю значение регистра ebx с помощью команды set. (рис. 4.17).

```

sakudyakova@dk1n22:~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09
Register group: general
eax      0x8      8      ecx      0x804a000  134520832
edx      0x8      8      ebx      0x2      2
esp      0xffffc2b0 0xffffc2b0  ebp      0x0      0x0
esi      0x0      0      edi      0x0      0
eip      0x8049016 0x8049016 <_start+22>  eflags   0x202    [ IF ]
cs       0x23     35     ss       0x2b     43
ds       0x2b     43     es       0x2b     43
fs       0x0      0      gs       0x0      0

0x8049000 <_start>      mov     eax,0x4
0x8049005 <_start+5>    mov     ebx,0x1
0x804900a <_start+10>   mov     ecx,0x804a000
0x804900f <_start+15>   mov     edx,0x8
0x8049014 <_start+20>   int     0x80
> 0x8049016 <_start+22> mov     eax,0x4
0x804901b <_start+27>   mov     ebx,0x1
0x8049020 <_start+32>   mov     ecx,0x804a000
0x8049025 <_start+37>   mov     edx,0x7
0x804902a <_start+42>   int     0x80

native process 3789 In: _start      L14  PC: 0x8049016
$1 = 0x8
(gdb) p/t $edx
$2 = 1000
(gdb) p/c $edx
$3 = 8 '\b'
(gdb) set $ebx='2'
(gdb) p/s $ebx
$4 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$5 = 2
(gdb)

```

Рис. 4.17: Команда print и команда set

Разница вывода команд `p/s $ebx` отличается тем, что в первом случае мы переводим символ в его строковый вид, а во втором случае число в строковом виде не изменяется.

Завершаю выполнение программы с помощью команды `continue` (сокращенно `c`) и выхожу из GDB с помощью команды `quit` (сокращенно `q`). (рис. 4.18).

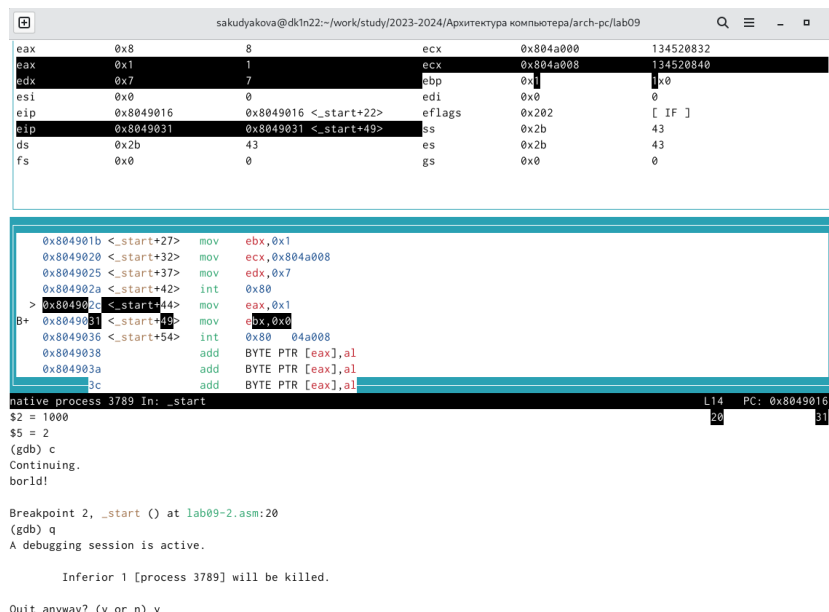


Рис. 4.18: Выход из GDB

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 в файл с именем lab09-3.asm.(рис. 4.19).

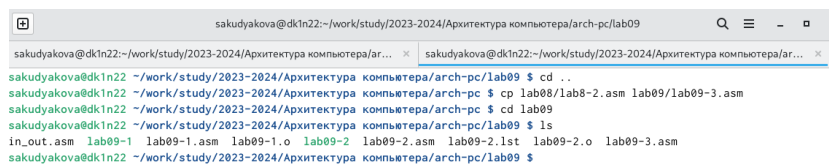


Рис. 4.19: Копирование файла

Создаю исполняемый файл и загружаю его в отладчик, указав аргументы. (рис. 4.20).

```

sakudyakova@dkln22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ nasm -f elf -g -l lab09-3.lst lab09-3.asm
sakudyakova@dkln22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ld -m elf_i386 -o lab09-3 lab09-3.o
sakudyakova@dkln22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ gdb --args lab09-3 аргумент1 аргумент 2 'аргумент 3'
GNU gdb (Gentoo 13.2 vanilla) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab09-3...
(gdb)

```

Рис. 4.20: Создание исполняемого файла и его загрузка в отладчик

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

```

(gdb) b _start
Breakpoint 1 at 0x0490e8: file lab09-3.asm, line 5.
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/s/a/sakudyakova/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09/lab09-3 аргумент1 аргумент 2 аргумент\ 3

Breakpoint 1, _start () at lab09-3.asm:5
5      pop ecx ; Разблокируем стек в 'ecx' количество

```

Рис. 4.21: Установление точки останова и запуск программы

Просматриваю вершину стека и позиции стека по их адресам. (рис. 4.22).

```

(gdb) x/x $esp
0xffffc1e0: 0x05
(gdb) x/s *(void**)(esp + 4)
0xffffc44f: "/afs/.dk.sci.pfu.edu.ru/home/s/a/sakudyakova/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)(esp + 8)
0xffffc4d3: "аргумент1"
(gdb) x/s *(void**)(esp + 12)
0xffffc4e5: "аргумент"
(gdb) x/s *(void**)(esp + 16)
0xffffc4f6: "2"
(gdb) x/s *(void**)(esp + 20)
0xffffc4f8: "аргумент 3"
(gdb) x/s *(void**)(esp + 24)
0x0: <error: Cannot access memory at address 0x0>
(gdb)

```

Рис. 4.22: Просмотр значений

Шаг изменения адреса равен 4, потому что количество аргументов командной строки равно 4.

4.3 Выполнение заданий для самостоятельной работы

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

как подпрограмму. (рис. 4.23).

```
1 %include 'in_out.asm'
2 SECTION .data
3 msg db "Результат: ",0
4 SECTION .text
5 global _start
6 _start:
7 pop ecx
8 pop edx
9 sub ecx,1
10 mov esi,0
11 mov edi,7
12 call .next
13
14 .next:
15 pop eax
16 call atoi
17 add eax,1
18 mul edi
19 add esi,eax
20 cmp ecx,0h
21 jz .done
22 loop .next
23
24 .done:
25 mov eax,msg
26 call sprint
27 mov eax,esi
28 call iprintLF
29 call quit
30 ret
31
```

Рис. 4.23: Редактирование файла

Создаю исполняемый файл и запускаю его. (рис. 4.24).

```
sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ nasm -f elf lab09-4.asm
sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ld -m elf_i386 -o lab09-4 lab09-4.o
sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ./lab09-4 1 2 3
Результат: 42
sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $
```

Рис. 4.24: Запуск программы

Программа отработала корректно.

Листинг 4.3.1. Программа вычисления значения функции как подпрограммы.

```
%include 'in_out.asm'

SECTION .data
msg db "Результат: ",0

SECTION .text
global _start
_start:
pop ecx
pop edx
sub ecx,1
mov esi, 0
mov edi,7
call .next

.next:
pop eax
call atoi
add eax,1
mul edi
add esi,eax
cmp ecx,0h
jz .done
loop .next
```



```

.done:
mov eax, msg
call sprint
mov eax, esi
call iprintLF
call quit
ret

```

2. Ввожу в файл программу для вычисления выражения $(3 + 2) * 4 + 5$. (рис. 4.25).

```

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

```

Рис. 4.25: Редактирование файла

Создаю исполняемый файл и запускаю его. (рис. 4.26).

```

sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ nasm -f elf lab09-5.asm
sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ld -m elf_i386 -o lab09-5 lab09-5.o
sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ./lab09-5
Результат: 10

```

Рис. 4.26: Запуск программы

При запуске данная программа дает неверный результат, так как результат должен быть равен 25. С помощью отладчика GDB анализирую изменения значений регистров и понимаю, что неверный результат получается из-за того, что при выполнении инструкции `mul ecx` происходит умножение `ecx` на `eax`, то есть 4 на 2, вместо умножения 4 на 5 (регистр `ebx`). Происходит это из-за того, что стоящая перед `mov ecx,4` инструкция `add ebx,eax` не связана с `mul ecx`, но связана инструкция `mov eax,2`. Исправляю данную ошибку. (рис. 4.27).

```

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

```

Рис. 4.27: Исправление ошибки

Создаю исполняемый файл и запускаю его. (рис. 4.28).

```
sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ nasm -f elf lab09-5.asm
sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ld -m elf_i386 -o lab09-5 lab09-5.o
sakudyakova@dk1n22 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/lab09 $ ./lab09-5
Результат: 25
```

Рис. 4.28: Запуск программы

Программа отработала верно.

Листинг 4.3.2. Программа для вычисления выражения $(3 + 2) * 4 + 5$.

```
%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 eax,ebx
mov ecx,4
mul ecx
add eax,5
mov edi,eax
; ---- Вывод результата на экран
mov eax,div
call sprint
mov eax,edi
call iprintLF
call quit
```

5 Выводы

В ходе данной лабораторной работы я научилась писать программы с использованием подпрограмм, а также ознакомилась с методами отладки при помощи GDB и его основными возможностями

Список литературы

Архитектура ЭВМ