

Отчёт по лабораторной работе №9

Дисциплина: Архитектура компьютера

Луангсуваннавонг Сайпхачан

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
4	Выполнение лабораторной работы	10
4.1	Реализация подпрограмм в NASM	10
4.2	Отладка программ с помощью GDB	15
4.3	Выполнение заданий для самостоятельной работы	30
5	Выводы	40
6	Ответы на вопросы для самопроверки	41
7	Список литературы	43

Список иллюстраций

4.1	Создание файла и директории	10
4.2	Копирование файла	10
4.3	Редактирование файла	11
4.4	Запуск файла	12
4.5	Редактирование файла	12
4.6	Запуск файла	13
4.7	Создание файла	15
4.8	Редактирование файла	16
4.9	Запуск файла	16
4.10	Загрузка файла в отладчик	17
4.11	Запуск программы	17
4.12	Добавление точки останова	17
4.13	Дизассемблирование программного кода	18
4.14	Настройка синтаксиса дизассемблирования	19
4.15	Настройка синтаксиса дизассемблирования	20
4.16	Включение псевдографического режима	21
4.17	Просмотр информации о точках останова	22
4.18	Просмотр информации о точках останова	23
4.19	Ручное выполнение одной строки кода	24
4.20	Просмотр содержимого регистров	24
4.21	Отображение содержимого переменных	25
4.22	Отображение содержимого переменных	25
4.23	Изменение значений регистров	26
4.24	Изменение значений регистров	26
4.25	Просмотр содержимого реестра	27
4.26	Окно регистров и его текущие значения	27
4.27	Настройка и просмотр содержимого реестра	27
4.28	Выход из отладчика	28
4.29	Копирование файла	28
4.30	Загрузка файла в отладчик	28
4.31	Добавление точки останова	29
4.32	Просмотр содержимого регистров	29
4.33	Просмотр содержимого регистров	29
4.34	Копирование и переименование файла	30
4.35	Редактирование файла	31
4.36	Запуск файла	32
4.37	Создание файла	33

4.38 Редактирование файла	34
4.39 Запуск файла	35
4.40 Открытие программного файла в отладчике	35
4.41 Проверка процесса работы программы	36
4.42 Редактирование файла	37
4.43 Запуск файла	38

1 Цель работы

Целью данной лабораторной работы является приобретение навыков написания программ с использованием подпрограмм. Знакомство с методами отладки при помощи GDB и его основными возможностями.

2 Задание

1. Реализация подпрограмм в NASM
2. Отладка программ с помощью GDB
3. Выполнение заданий для самостоятельной работы

3 Теоретическое введение

Отладка — это процесс поиска и исправления ошибок в программе. Общем случае его можно разделить на четыре этапа:

- обнаружение ошибки;
- поиск её местонахождения;
- определение причины ошибки;
- исправление ошибки.

Второй этап — Поиск местонахождения ошибки

Некоторые ошибки трудно обнаружить, но лучший способ найти их местоположение — это разделить программу на части и проверять их поочередно. Это позволяет сузить область поиска и быстрее обнаружить проблемный участок.

Третий этап— Выяснение причины ошибки

После того как ошибка была локализована, обычно проще определить ее причину. Это может включать анализ данных, условий выполнения и логики программы.

Последний этап — Исправление ошибки

После того как ошибка найдена и причины ее возникновения поняты, можно приступить к исправлению. После этого программа запускается снова, и, возможно, будет обнаружена другая ошибка, что заставит начать процесс отладки заново.

Отладчики — это инструменты, которые помогают контролировать выполнение программы, изменять данные и управлять процессом отладки. Они ускоряют поиск ошибок и их исправление. Наиболее распространенные методы работы с

отладчиком включают использование точек останова и пошагового выполнения.

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.

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).

GDB (GNU Debugger — отладчик проекта GNU)— это отладчик для UNIX-подобных систем, который поддерживает отладку программ на многих языках программирования. GDB предоставляет обширные средства для контроля за выполнением программ и анализа их состояния. Он не имеет графического интерфейса, используя текстовый интерфейс командной строки, но существует ряд графических оболочек, использующих GDB в качестве основного инструмента для отладки.

С помощью GDB можно:

- Начать выполнение программы, настроив все параметры, влияющие на её поведение.
- Остановить выполнение программы при определенных условиях.
- Исследовать состояние программы, если она была остановлена.

- Изменить программу, чтобы протестировать изменения и устранить ошибки.

Подпрограмма — это функционально завершённый участок кода, который можно многократно вызывать из разных частей программы. Это позволяет избежать повторений кода, упрощая его поддержку и сокращая размер программы.

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.

В отличие от простых переходов, подпрограммы содержат инструкцию возврата (return), которая позволяет вернуться в точку вызова. В языке ассемблера для вызова подпрограммы используется инструкция call, которая помещает адрес следующей инструкции в стек и передаёт управление подпрограмме. Когда подпрограмма завершает выполнение, она использует инструкцию ret, чтобы извлечь адрес из стека и вернуть управление в программу.

Подпрограммы могут быть как частью основной программы, так и находиться в отдельных внешних файлах.

4 Выполнение лабораторной работы

4.1 Реализация подпрограмм в NASM

Я создаю новую директорию, в которой буду создавать файлы с программами для лабораторной работы № 9, используя команду `mkdir`. Затем я перехожу в созданный каталог и создаю файл `lab9-1.asm`, используя команду `touch`. (Рис .4.1)

```
sayprachanh@desktop:~$ mkdir work/study/2024-2025/Архитектура\ компьютера/arch-pc/lab09
sayprachanh@desktop:~$ cd work/study/2024-2025/Архитектура\ компьютера/arch-pc/lab09
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ touch lab9-1.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ls
lab9-1.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$
```

Рис. 4.1: Создание файла и директории

Я копирую файл `in_out.asm` из последней лабораторной работы, потому что он будет использоваться в других программах(Рис .4.2)

```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ cp ~/work/study/2024-2025/Архитектура\ компьютера/arch-pc/lab08/in_out.asm .
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ls
in_out.asm  lab9-1.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$
```

Рис. 4.2: Копирование файла

Я открываю созданный файл `lab9-1.asm`, затем вставляю программу, которая будет вычислять арифметическое выражение $f(x) = 2x + 7$ с помощью подпрограммы `_calcul`(Рис .4.3)

```

1  %include 'in_out.asm'
2
3  SECTION .data
4  msg:
5  DB 'Введите x: ',0
6  result: DB '2x+7=',0
7
8  SECTION .bss
9  x: RESB 80
10 res: RESB 80
11
12 SECTION .text
13 GLOBAL _start
14 _start:
15 ;-----
16 ; Основная программа
17 ;-----
18
19 mov eax, msg      ; вызов подпрограммы печати сообщения
20 call sprint       ; 'Введите x: '
21
22 mov ecx, x
23 mov edx, 80
24 call sread       ; вызов подпрограммы ввода сообщения
25
26 mov eax, x        ; вызов подпрограммы преобразования
27 call atoi        ; ASCII кода в число, 'eax=x'
28
29 call _calcul      ; Вызов подпрограммы _calcul
30
31 mov eax, result
32 call sprint
33 mov eax, [res]
34 call iprintLF
35
36 call quit
37
38 ;-----
39 ; Подпрограмма вычисления
40 ; выражения "2x+7"
41
42 _calcul:
43 mov ebx, 2
44 mul ebx
45 add eax, 7
46 mov [res], eax
47
48 ret              ; выход из подпрограммы

```

Рис. 4.3: Редактирование файла

Я создаю исполняемый файл и запускаю его. Я ввожу значение для расчета,

после чего программа отображает результат.(Рис .4.4)

```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ nasm -f elf lab9-1.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ./lab9-1
Введите x: 9
2x+7=25
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$
```

Рис. 4.4: Запуск файла

Я снова открываю файл lab9-1.asm для редактирования, чтобы вычислить выражение $f(g(x))$, (Рис .4.5) я ввожу другую подпрограмму (`_subcalcul`) для вычисления выражения $g(x) = 3x - 1$, и я использую инструкцию 'call' в подпрограмме `_calcul` для того, чтобы пусть подпрограмма `_calcul` использует подпрограмму `_subcalcul` для вычисления результата выражения $f(g(x))$.

```
38 ;-----
39 ; Подпрограмма вычисления
40 ; выражения "2x+7"
41
42 _calcul:
43 call _subcalcul
44 mov ebx,2
45 mul ebx
46 add eax,7
47 mov [res], eax
48
49 ret ; выход из подпрограммы
50
51 ; выражения "3x-1"
52
53 _subcalcul:
54 mov ebx, 3
55 mul ebx
56 sub eax, 1
57
58 ret
~
```

Рис. 4.5: Редактирование файла

Я создаю исполняемый файл и запускаю его, ввожу значение для расчета, и программа выдает результат(Рис .4.6). Я проверяю результат самостоятельно. Программа работает правильно.

```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ nasm -f elf lab9-1.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ./lab9-1
Введите x: 3
2(3x-1)+7=23
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ./lab9-1
Введите x: 1
2(3x-1)+7=11
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$
```

Рис. 4.6: Запуск файла

Программа для вычисления выражения $f(g(x))$

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

```
SECTION .data
```

```
msg:
```

```
DB 'Введите x: ',0
```

```
result: DB '2(3x-1)+7=',0
```

```
SECTION .bss
```

```
x: RESB 80
```

```
res: RESB 80
```

```
SECTION .text
```

```
GLOBAL _start
```

```
_start:
```

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

```
; Основная программа
```

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

```
mov eax, msg ; вызов подпрограммы печати сообщения
```

```

call sprint      ; 'Введите x: '

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`

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
```

```
; выражения "3x-1"
```

```
_subcalcul:
```

```
mov ebx, 3
```

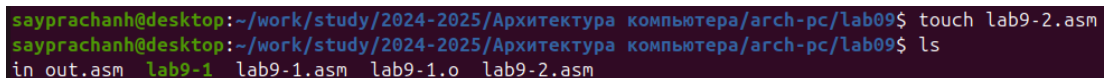
```
mul ebx
```

```
sub eax, 1
```

```
ret
```

4.2 Отладка программ с помощью GDB

Я создаю новый файл lab9-2.asm, используя команду touch(Рис .4.7)

A screenshot of a terminal window with a dark background. The prompt is 'sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09\$'. The first command is 'touch lab9-2.asm' and the second is 'ls'. The output of 'ls' shows 'in_out.asm', 'lab9-1', 'lab9-1.asm', 'lab9-1.o', and 'lab9-2.asm'.

```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ touch lab9-2.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ls
in_out.asm  lab9-1  lab9-1.asm  lab9-1.o  lab9-2.asm
```

Рис. 4.7: Создание файла

Я открываю созданный файл lab9-2.asm и вставляю программу, которая будет выводить текст: “Hello, world!”(Рис .4.8)

```

1  SECTION .data
2      msg1:    db "Hello, ",0x0
3      msg1Len: equ $- msg1
4
5      msg2:    db "world!",0xa
6      msg2Len: equ $- msg2
7
8  SECTION .text
9  global _start
10
11 _start:
12  mov eax, 4
13  mov ebx, 1
14  mov ecx, msg1
15  mov edx, msg1Len
16  int 0x80
17
18  mov eax, 4
19  mov ebx, 1
20  mov ecx, msg2
21  mov edx, msg2Len
22  int 0x80
23
24  mov eax, 1
25  mov ebx, 0
26  int 0x80

```

Рис. 4.8: Редактирование файла

Я создаю исполняемый файл, на этот раз я добавляю ключ отладочной информации “-g”, чтобы работать с исполняемым файлом GDB.(Рис .4.9)

```

sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ nasm -f elf -g -l lab9-2.lst lab9-2.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ld -m elf_i386 -o lab9-2 lab9-2.o
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$

```

Рис. 4.9: Запуск файла

Я загружаю исполняемый файл lab9.2 в отладчике gdb(Рис .4.10)


```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/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. 4.10: Загрузка файла в отладчик

Используя команду “run” в отладчике GDB, я проверяю работу программы(Рис .4.11). Программа работает нормально

```
(gdb) run
Starting program: /home/sayprachanh/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09/lab9-2
Hello, world!
[Inferior 1 (process 14770) exited normally]
(gdb) █
```

Рис. 4.11: Запуск программы

Я устанавливаю точку останова с меткой _start, которая запускает выполнение программы сборки, чтобы получить более подробный анализ программы.(Рис .4.12)

Я запускаю программу снова, на этот раз она отображает точку останова, которую я создал.

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab9-2.asm, line 12.
(gdb) run
Starting program: /home/sayprachanh/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09/lab9-2

Breakpoint 1, _start () at lab9-2.asm:12
12      mov eax, 4
(gdb) █
```

Рис. 4.12: Добавление точки останова

Используя команду “disassemble _start”, я просматриваю дизассемблированный программный код в метке _start.(Рис .4.13)

```
(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. 4.13: Дизассемблирование программного кода

С помощью команды “set disassembly -flavor intel” я переключаюсь на отображение команд с синтаксисом Intel. Мы видим, что после выполнения команды “disassemble _start” она отображается с другим синтаксисом.(Рис .4.14)

```

(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. 4.14: Настройка синтаксиса дизассемблирования

Другой синтаксис - это синтаксис AT&T. Основное различие между синтаксисами AT&T и Intel в том, что в AT&T исходный операнд идет перед целевым, регистры имеют префикс “%”, а немедленные значения — префикс “\$”, в то время как в Intel целевой операнд идет первым, регистры не имеют префикса, а немедленные значения не имеют префикса. (Рис .4.15)

```

(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) set disassembly-flavor att
(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. 4.15: Настройка синтаксиса дизассемблирования

Для более удобного анализа программы я включаю псевдографический режим, используя команды `layout asm` и `layout regs`. В первом окне отображаются названия и текущие значения регистров, в середине отображается результат усвоения программы, а нижняя часть предназначена для ввода команд (Рис .4.16)

```

Register group: general
eax      0x0      0      ecx      0x0      0      edx      0x0      0
ebx      0x0      0      esp      0xffffcf80  0xffffcf80  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

0x8049000 <_start> mov     eax,0x4
0x8049005 <_start+5> mov     ecx,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     ecx,0x7
0x804902a <_start+42> int     0x80
0x804902c <_start+44> mov     eax,0x1

native process 14777 (asm) In: _start
(gdb) layout regs
(gdb)

```

Рис. 4.16: Включение псевдографического режима

Чтобы проверить, что я уже установил точку останова на метке `_start`, я использую команду “`info breakpoints`” для просмотра информации обо всех точках останова (Рис .4.17)

```
B+>0x8049000 <_start>    mov    eax,0x4
      0x8049005 <_start+5>    mov    ebx,0x1
      0x804900a <_start+10>   mov    ecx,0x804a000
      0x804900f <_start+15>   mov    edx,0x8
      0x8049014 <_start+20>   int     0x80
      0x8049016 <_start+22>   mov    eax,0x4
      0x804901b <_start+27>   mov    ebx,0x1
      0x8049020 <_start+32>   mov    ecx,0x804a008
      0x8049025 <_start+37>   mov    edx,0x7
      0x804902a <_start+42>   int     0x80
      0x804902c <_start+44>   mov    eax,0x1

native process 14777 (asm) In: _start
(gdb) layout regs
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint       keep y  0x08049000 lab9-2.asm:12
breakpoint already hit 1 time
(gdb) █
```

Рис. 4.17: Просмотр информации о точках останова

Я определяю адрес предпоследней инструкции (mov ebx, 0x0) и устанавливаю другую точку останова, затем я использую команду “info breakpoints” или “i b” для просмотра информации обо всех точках останова. (Рис. 4.18)

```
b+ 0x8049031 <_start+49>  mov    ebx,0x0
0x8049036 <_start+54>  int     0x80
0x8049038              add     BYTE PTR [eax],al

native process 14777 (asm) In: _start
(gdb) layout regs
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint       keep y  0x08049000 lab9-2.asm:12
        breakpoint already hit 1 time
(gdb) b *0x8049031
Breakpoint 2 at 0x8049031: file lab9-2.asm, line 25.
(gdb) i b
Num      Type             Disp Enb Address      What
1        breakpoint       keep y  0x08049000 lab9-2.asm:12
        breakpoint already hit 1 time
2        breakpoint       keep y  0x08049031 lab9-2.asm:25
(gdb) █
```

Рис. 4.18: Просмотр информации о точках останова

Используя команду `stepi (si)`, я вручную просматриваю изменение значения регистров. Я выполнил команду 5 раз, в первом окне мы видим, что значение регистра `eax` изменено на 8, а регистр `ebx` изменен с 0 (по умолчанию) на 1. (Рис .4.19)

```

eax      0x8      8      ecx      0x804a000      134520832      edx      0x8      8
ebx      0x1      1      esp      0xffffcf80      0xffffcf80      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,0x804a000
0x8049025 <_start+37>     mov     edx,0x7
0x804902a <_start+42>     int     0x80
0x804902c <_start+44>     mov     eax,0x1

native process 16853 (asm) In: _start
(gdb) si
(gdb) si
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sayprachanh/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09/lab9-2
Breakpoint 1, _start () at lab9-2.asm:12
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si

```

Рис. 4.19: Ручное выполнение одной строки кода

Я просматриваю содержимое регистров с помощью команды info registers или (i r)(Рис .4.20)

```

native process 16861 (asm) In: _start
esp      0xffffcf80      0xffffcf80
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
--Type <RET> for more, q to quit, c to continue without paging--
es       0x2b      43
fs       0x0      0
gs       0x0      0
(gdb)

```

Рис. 4.20: Просмотр содержимого регистров

затем я отображаю содержимое переменной msg1 с помощью команды 'x/1sb &msg1'. Я задаю имя указанной переменной, которое я хочу использовать для просмотра содержимого, в данном случае это переменная msg1(Рис .4.21)


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

Рис. 4.21: Отображение содержимого переменных

Затем я просматриваю содержимое msg2, но на этот раз я использую адрес переменной, который может быть определен с помощью разобранной инструкции, в этом случае адрес msg2 можно увидеть в mov ecx, 0x804a008, адрес которого равен 0x804a008(Рис .4.22)

```
B+ 0x8049000 <_start>      mov     eax,0x4
    0x8049005 <_start+5>   mov     ebx,0x1
    0x804900a <_start+10>  mov     ecx,0x804a000
    0x804900f <_start+15>  mov     edx,0x8
    0x8049014 <_start+20>  int     0x80
> 0x8049016 <_start+22>  mov     eax,0x4
    0x804901b <_start+27>  mov     ebx,0x1
    0x8049020 <_start+32>  mov     ecx,0x804a008
    0x8049025 <_start+37>  mov     edx,0x7
    0x804902a <_start+42>  int     0x80
    0x804902c <_start+44>  mov     eax,0x1

native process 16861 (asm) In: _start
eip          0x8049016          0x8049016 <_start+22>
eflags       0x202             [ IF ]
cs           0x23              35
ss           0x2b              43
ds           0x2b              43
--Type <RET> for more, q to quit, c to continue without paging--
es           0x2b              43
fs           0x0               0
gs           0x0               0
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n\034"
(gdb)
```

Рис. 4.22: Отображение содержимого переменных

Используя команду “set”, я могу изменить значение регистра или памяти, указав имя регистра или адрес в качестве аргумента, а также указать тип данных в фигурных скобках. Я меняю первый и второй символы переменной msg1.(Рис .4.23)

```
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hello, "
(gdb) set {char}0x804a001='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hhllo, "
(gdb) █
```

Рис. 4.23: Изменение значений регистров

Затем я также меняю первый и четвертый символы переменных msg2(Рис .4.24)

```
(gdb) set {char}0x804a008='W'
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "World!\n\034"
(gdb) set {char}0x804a00b = ' '
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "Wor d!\n\034"
(gdb) █
```

Рис. 4.24: Изменение значений регистров

Используя команду print, я печатаю значение регистра edx в различных форматах, таких как /x для шестнадцатеричного, /t для двоичного и /c для символьного. (Рис .4.25) Некоторые значения можно просмотреть в окне, которое отображает название и текущее значение регистров.(Рис .4.26)

```

(gdb) p/x $edx
$1 = 0x8
(gdb) p/t $edx
$2 = 1000
(gdb) p/c $edx
$3 = 8 '\b'
(gdb) █

```

Рис. 4.25: Просмотр содержимого реестра

Register group: general											
eax	0x0	8	ecx	0x804a000	134520832	edx	0x8	8			
ebx	0x1	1	esp	0xffffcf80	0xffffcf80	ebp	0x0	0x0			
esi	0x0	0	edi	0x0	0	ebp	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									

Рис. 4.26: Окно регистров и его текущие значения

Мы можем видеть разницу вывода команд `p/s $ebx`. Разница в том, что `set $ebx='2'` присваивает строку '2', и команда `p/s $ebx` выводит её ASCII значение (50), а `set $ebx=2` присваивает число 2, и команда `p/s $ebx` выводит именно его. (Рис. 4.27)

```

(gdb) set $ebx='2'
(gdb) p/s $ebx
$4 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$5 = 2
(gdb) █

```

Рис. 4.27: Настройка и просмотр содержимого реестра

с помощью команды 'continue' (c) я завершаю работу программы и выхожу из GDB, используя команду quit (q). (Рис. 4.28)

```
(gdb) c
Continuing.
Wor d!

Breakpoint 2, _start () at lab9-2.asm:25
(gdb) q
```

Рис. 4.28: Выход из отладчика

Я копирую файл lab 8-2.asm, созданный во время лабораторной работы № 8, и называю его lab9-3.asm(Рис .4.29)

```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ cp ~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab08/lab8-2.asm ~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09/lab9-3.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ls
in_out.asm lab9-1.asm lab9-1.o lab9-2 lab9-2.asm lab9-2.lst lab9-2.o lab9-3.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$
```

Рис. 4.29: Копирование файла

Затем я создаю исполняемый файл Используя key –args, я загружаю программу с аргументами в gdb.(Рис .4.30)

```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ nasm -f elf -g -l lab9-3.lst lab9-3.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ld -m elf_i386 -o lab9-3 lab9-3.o
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ gdb --args lab9-3 аргумент1 аргумент 2 'аргумент 3'
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 lab9-3...
(gdb)
```

Рис. 4.30: Загрузка файла в отладчик

Я устанавливаю точку останова, используя команду break (b), и запускаю программу(Рис .4.31)

```
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab9-3.asm, line 7.
(gdb) run
Starting program: /home/sayprachanh/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09/lab9-3 аргумент1 аргумент 2 аргумент\ 3
Breakpoint 1, _start () at lab9-3.asm:7
7      pop ecx ; Извлекаем из стека в 'ecx' количество
(gdb)
```

Рис. 4.31: Добавление точки останова

Введя команду `x/x $esp`, я могу увидеть количество аргументов, которые передаются программе (Рис. 4.32). Поскольку адрес вершины стека хранится в регистре `esp`, и это число равно количеству аргументов командной строки (включая название программы). В этом случае аргументами являются: `./lab9-3`, `аргумент1`, `аргумент2` и `‘аргумент 3’`. Что равно 5 аргументам

```
(gdb) x/x $esp
0xffffcf40:      0x00000005
(gdb)
```

Рис. 4.32: Просмотр содержимого регистров

Я просматриваю остальную часть стека. (Рис. 4.33)

в `[esp + 4]` адрес в памяти, по которому находится название программы.

в `[esp + 8]`, где хранится адрес первого аргумента, в `[esp + 12]` хранится адрес второго аргумента, в `[esp + 16]` - третьего и `[esp + 20]` четвертый.

в `[esp + 24]` ошибка появляется, когда GDB пытается получить доступ к NULL-указателю, что происходит, если передано меньше аргументов, чем ожидалось. В данном случае указывает на NULL, потому что аргументов меньше 6.

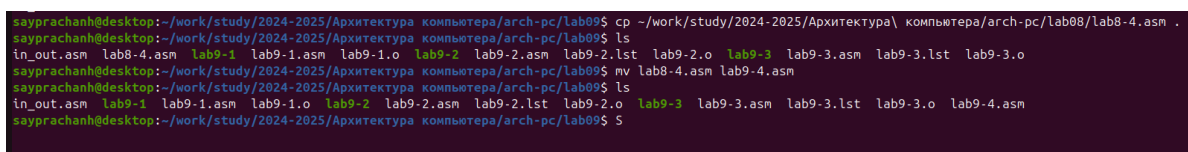
```
(gdb) x/s *(void**)(esp + 4)
0xffffd103:      "/home/sayprachanh/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09/lab9-3"
(gdb) x/s *(void**)(esp + 8)
0xffffd16b:      "аргумент1"
(gdb) x/s *(void**)(esp + 12)
0xffffd17d:      "аргумент"
(gdb) x/s *(void**)(esp + 16)
0xffffd18e:      "2"
(gdb) x/s *(void**)(esp + 20)
0xffffd190:      "аргумент 3"
(gdb) x/s *(void**)(esp + 24)
0x0:      <error: Cannot access memory at address 0x0>
(gdb)
```

Рис. 4.33: Просмотр содержимого регистров

Адреса увеличиваются на 4 байта ([esp+4], [esp+8] и т.д.), потому что в архитектуре x86 аргументы функции записываются в стек как 4-байтовые указатели. Каждый указатель занимает 4 байта, поэтому указатель стека (esp) изменяется на 4 байта для каждого аргумента.

4.3 Выполнение заданий для самостоятельной работы

Я копирую файл самостоятельного задания для последней лабораторной работы (лабораторная работа № 8) и меняю его название на lab9-4.asm (Рис .4.34)



```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ cp ~/work/study/2024-2025/Архитектура\ компьютера/arch-pc/lab08/lab8-4.asm .
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ls
in_out.asm lab8-4.asm lab9-1 lab9-1.asm lab9-1.o lab9-2 lab9-2.asm lab9-2.lst lab9-2.o lab9-3 lab9-3.asm lab9-3.lst lab9-3.o
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ mv lab8-4.asm lab9-4.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ls
in_out.asm lab9-1 lab9-1.asm lab9-1.o lab9-2 lab9-2.asm lab9-2.lst lab9-2.o lab9-3 lab9-3.asm lab9-3.lst lab9-3.o lab9-4.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$
```

Рис. 4.34: Копирование и переименование файла

Я открываю файл lab9-4.asm для редактирования. Я реализую подпрограмму, которая вычисляет значение функции $f(x)$, ($f(x) = 12x - 7$), а также добавляет инструкцию цикла для выполнения цикла программы, чтобы получить тот же результат. (Рис .4.35)

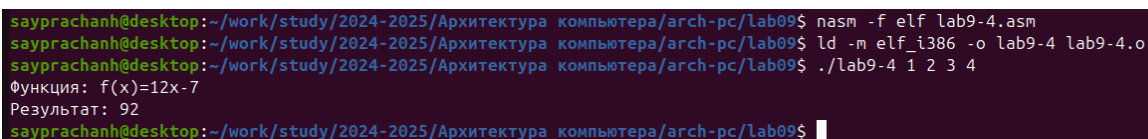
```

7  section .text
8  global _start
9  _start:
10 pop ecx
11 pop edx
12 sub ecx, 1
13 mov esi, 0 ; esi = 0
14
15 _process:
16 cmp ecx, 0
17 jz _end
18
19 pop eax
20 call atoi
21 call _subcalcul ; вызов подпрограммы _subcalcul
22 loop _process
23
24 _end:
25 mov eax, msg1
26 call sprintf ;Вывод сообщения 'Функция: f(x)=12x-7 '
27 mov eax, result
28 call sprintf ;Вывод сообщения 'Результат: '
29 mov eax, esi
30 call sprintf ;Результат
31
32 call quit
33
34 ; -- подпрограмма _subcalcul
35 _subcalcul:
36 mov ebx, 12
37 mul ebx ; eax = eax * ebx(12)
38
39 mov ebx, 7
40 sub eax, ebx ; eax = eax - ebx(7)
41
42 add esi, eax ; esi = esi + eax
43
44 ret

```

Рис. 4.35: Редактирование файла

Я создаю исполняемый файл и запускаю его. Я ввожу аргументы для вычисления значения функции, и программа выдает результат, который является тем же результатом, что и в исходной программе, у которой нет подпрограммы. (Рис .4.36)



```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ nasm -f elf lab9-4.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ld -m elf_i386 -o lab9-4 lab9-4.o
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ./lab9-4 1 2 3 4
Функция: f(x)=12x-7
Результат: 92
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$
```

Рис. 4.36: Запуск файла

Программа для выполнения задания 1

```
%include 'in_out.asm' ; подключение внешнего файла
```

```
section .data
```

```
msg1 db "Функция: f(x)=12x-7 ",0
```

```
result db "Результат: ",0
```

```
section .text
```

```
global _start
```

```
_start:
```

```
pop ecx
```

```
pop edx
```

```
sub ecx, 1
```

```
mov esi, 0 ; esi = 0
```

```
_process:
```

```
cmp ecx, 0
```

```
jz _end
```

```
pop eax
```



```

call atoi
call _subcalcul ; вызов подпрограммы _subcalcul
loop _process

_end:
mov eax, msg1
call sprintf ;Вывод сообщения 'Функция: f(x)=12x-7 '
mov eax, result
call sprintf ;Вывод сообщения 'Результат: '
mov eax, esi
call sprintf ;Результат

call quit

; -- подпрограмма _subcalcul
_subcalcul:
mov ebx, 12
mul ebx      ; eax = eax * ebx(12)

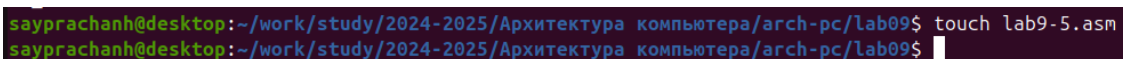
mov ebx, 7
sub eax, ebx ; eax = eax - ebx(7)

add esi, eax ; esi = esi + eax

ret

```

Используя команду “touch”, я создаю новый файл lab9-5.asm(Рис .4.37)



```

sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ touch lab9-5.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$

```

Рис. 4.37: Создание файла

Я открываю созданный файл lab9-5.asm, вставляю программу(Рис .4.38), которая выведет результат вычисления $(3+2)*4+5$

```
1  %include 'in_out.asm'
2
3  SECTION .data
4  div: DB 'Результат: ',0
5
6  SECTION .text
7  GLOBAL _start
8  _start:
9
10 ;---- Вычисление выражения (3+2)*4+5
11 mov ebx,3
12 mov eax,2
13 add ebx,eax
14 mov ecx,4
15 mul ecx
16 add ebx,5
17 mov edi,ebx
18
19 ;---- Вывод результата на экран
20 mov eax,div
21 call sprint
22 mov eax,edi
23 call iprintLF
24
25 call quit
```

Рис. 4.38: Редактирование файла

Я создаю исполняемый файл и запускаю его. Как мы видим, программа работает нормально, но выдает неверный результат. Правильным результатом должно быть 25, но программа выдает 10, что неверно(Рис .4.39)

```

sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ nasm -f elf -g -l lab9-5.lst lab9-5.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ld -m elf_i386 -o lab9-5 lab9-5.o
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ./lab9-5
Результат: 10
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$

```

Рис. 4.39: Запуск файла

Я открываю отладчик (gdb) Затем, используя команды `layout regs` и `layout asm`, я открываю окна, в одном из которых отображаются названия регистров и их текущие значения, а в другом - результат усвоения программы. Я добавляю точку останова в программу в `_start`, а затем запускаю программу, чтобы посмотреть, как работает эта программа.(Рис .4.40)

The screenshot shows the GDB interface with two main windows. The top window, titled 'Register group: general', displays the current values of various registers. The bottom window shows the assembly code being executed, with a breakpoint set at the start of the program.

Register	Value	Register	Value	Register	Value
eax	0	ecx	0x0	edx	0x0
ebx	0	esp	0xffffcf80	ebp	0x0
esi	0	edi	0x0	eip	0x80490e8
eflags	0x202	cs	0x23	ss	0x2b
ds	0x2b	es	0x2b	fs	0x0
gs	0x0				

The assembly window shows the following code:

```

0x80490e8 <_start> mov $0x3,%ebx
0x80490ee <_start+5> mov $0x7,%ecx
0x80490f2 <_start+10> add %eax,%ebx
0x80490f4 <_start+12> mov $0x4,%ecx
0x80490f9 <_start+17> mul %ecx
0x80490fb <_start+19> add $0x5,%ebx
0x80490fe <_start+22> mov %ebx,%edi
0x8049100 <_start+24> mov $0x804a000,%eax
0x8049105 <_start+29> call 0x804900f <sprint>
0x804910a <_start+34> mov %edi,%eax
0x804910c <_start+36> call 0x8049086 <iprintf>
0x8049111 <_start+41> call 0x80490db <quit>

```

The bottom status bar indicates the native process is 4875 (asm) in: `_start`. The GDB layout shows the registers window, the assembly window, and the status bar.

Рис. 4.40: Открытие программного файла в отладчике

Используя инструкцию `stepi (si)`, я вручную изменяю значения регистров, чтобы увидеть процесс работы программы.(Рис .4.41) Затем я вижу, что в части инструкции `mul` регистр `ecx` (`ecx` равен 4) не умножается на значение регистра `ebx`, а вместо этого умножается на `eax`, который равен 2. Таким образом, в первом окне (в котором отображаются названия регистров и их текущие значения) мы видим, что значение регистра `eax` равно 8, поскольку $eax(2) * ecx(4) = 8$

```

Register group: general
eax      0x8      8      ecx      0x4      4
ebx      0xa      10     esp      0xffffcf80  0xffffcf80
esi      0x0      0      edi      0x0      0
eflags   0x10206  [ PF IF RF ] cs      0x23      35
ds       0x2b      43     es       0x2b      43
gs       0x0      0

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

native process 4875 (asm) In: _start
(gdb) layout regs
(gdb) break _start
Breakpoint 1 at 0x80490e8: file lab9-5.asm, line 11.
(gdb) run
Starting program: /home/sayprachanh/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09/lab9-5

Breakpoint 1, _start () at lab9-5.asm:11
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb)

```

Рис. 4.41: Проверка процесса работы программы

Увидев неверную часть, я выхожу из отладчика и снова открываю lab9-5.asm, чтобы исправить программу. Я изменил программу, так что теперь регистр eax будет содержать основное значение для расчета, а также для отображения результата.(Рис .4.42)

Объяснение изменений:

- Из add ebx, eax в add eax, ebx. Чтобы установить регистр eax в качестве основного значения.
- Из add ebx, 5 в add eax, 5. чтобы добавить 5 в регистр eax

- Из `mov edi, ebx` в `mov edi, eax`. чтобы перенести значение `eax` в регистр `edi`

```
1  %include 'in_out.asm'
2
3  SECTION .data
4  div: DB 'Результат: ',0
5
6  SECTION .text
7  GLOBAL _start
8  _start:
9
10 ;---- Вычисление выражения (3+2)*4+5
11 mov ebx,3
12 mov eax,2
13 add eax,ebx ; от (add ebx, eax)
14 mov ecx,4
15 mul ecx
16 add eax,5 ; от (add ebx, 5)
17 mov edi,eax ; от (mov edi, ebx)
18
19 ;---- Вывод результата на экран
20 mov eax,div
21 call sprint
22 mov eax,edi
23 call iprintLF
24
25 call quit
~
```

Рис. 4.42: Редактирование файла

Я создаю исполняемый файл и запускаю его. на этот раз программа выдает 25 в качестве результата, что является правильным результатом.(Рис .4.43) Это означает, что теперь программа работает правильно.

```
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ nasm -f elf -g -l lab9-5.lst lab9-5.asm
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ld -m elf_i386 -o lab9-5 lab9-5.o
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$ ./lab9-5
Результат: 25
sayprachanh@desktop:~/work/study/2024-2025/Архитектура компьютера/arch-pc/lab09$
```

Рис. 4.43: Запуск файла

Программа для выполнения задания 2

```
%include 'in_out.asm'

SECTION .data
div: DB 'Результат: ',0

SECTION .text
GLOBAL _start
_start:

;---- Вычисление выражения (3+2)*4+5
mov ebx,3
mov eax,2
add eax,ebx ; от (add ebx, eax)
mov ecx,4
mul ecx
add eax,5 ; от (add ebx, 5)
mov edi,eax ; от (mov edi, ebx)

;---- Вывод результата на экран
mov eax,div
call sprint
mov eax,edi
call iprintLF
```

call quit

5 Выводы

При выполнении данной лабораторной работы, Я приобрел навыки написания программ с использованием подпрограмм, разобрался в методах отладки с использованием GDB и ее основных функциях.

6 Ответы на вопросы для самопроверки

1. Какие языковые средства используются в ассемблере для оформления и активизации подпрограмм?

В ассемблере используются инструкции `call` для вызова подпрограммы и `ret` для возврата из подпрограммы.

2. Объясните механизм вызова подпрограмм.

При вызове подпрограммы инструкция `call` сохраняет адрес следующей инструкции в стек и передает управление подпрограмме. По завершении подпрограммы инструкция `ret` извлекает адрес из стека и передает управление обратно в вызывающую программу.

3. Как используется стек для обеспечения взаимодействия между вызывающей и вызываемой процедурами?

Стек сохраняет адрес возврата (инструкцию, следующую после `call`), параметры и локальные переменные. Это обеспечивает правильное возвращение в вызывающую программу и передачу данных между процедурами.

4. Каково назначение операнда в команде `ret`?

Операнд команды `ret` указывает, сколько данных следует удалить из стека перед возвратом. Обычно это количество параметров, переданных в подпрограмму.

5. Для чего нужен отладчик?

Отладчик позволяет анализировать выполнение программы, выявлять ошибки, контролировать данные, изменять состояние программы и пошагово её вы-

полнять для проверки логики.

6. Объясните назначение отладочной информации и как нужно компилировать программу, чтобы в ней присутствовала отладочная информация.

Отладочная информация помогает отслеживать исходный код при отладке. Для её включения программу нужно компилировать с флагом, например, -g в GCC, чтобы включить символы отладки и маппинг исходного кода.

7. Расшифруйте и объясните следующие термины: breakpoint, watchpoint, checkpoint, catchpoint и call stack.

Breakpoint: Точка останова — место, где выполнение программы приостанавливается.

Watchpoint: Точка просмотра — программа приостанавливается, если изменяется или считывается указанная переменная.

Checkpoint: Место в программе, где состояние программы сохраняется для последующего восстановления.

Catchpoint: Точка, в которой отладчик перехватывает исключение или событие.

Call stack: Стек вызовов — структура данных, которая хранит последовательность вызовов функций.

8. Назовите основные команды отладчика gdb и как они могут быть использованы для отладки программ.

run: Запуск программы.

break : Установка точки останова.

next: Пошаговое выполнение, переход к следующей строке.

step: Пошаговое выполнение, заход в подпрограмму.

continue: Продолжение выполнения после точки останова.

print : Вывод значения переменной.

backtrace: Печать стека вызовов.

quit: Выход из отладчика.

7 Список литературы

Архитектура ЭВМ