Лабораторная работа №9

Понятие подпрограммы

Светцова Анна Дмитриевна

Содержание

# 1 Цель работы

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

# 2 Задание

1. 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. Задание для самостоятельной работы

# 3 Теоретическое введение

Отладка — это процесс поиска и исправления ошибок в программе. 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.

GDB (GNU Debugger — отладчик проекта GNU) 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.

Команда run (сокращённо r) — запускает отлаживаемую программу в оболочке GDB.

Команда kill (сокращённо k) прекращает отладку программы, после чего следует вопрос о прекращении процесса отладки. 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).

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.

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».

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.

Обратно точка останова активируется командой enable.

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

Для продолжения остановленной программы используется команда continue (c). 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.

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.

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.

# 4 Выполнение лабораторной работы

## 4.1 Реализация подпрограмм в NASM

Создаю каталог для выполнения работы №9 (рис. 1).

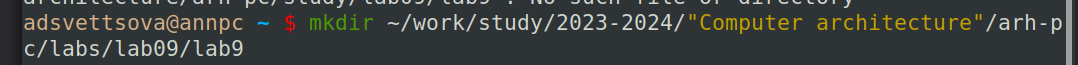


рис.1 Создание каталога

Перехожу в созданную директорию (рис. 2).

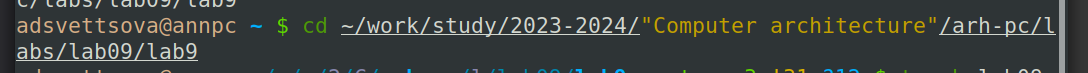


рис.2 Перемещение по директории

Создаю файл lab09-1.asm в новом каталоге (рис. 3).

рис.3 Создание файла

рис.3 Создание файла

Открываю файл и переписываю код программы из листинга 9.1 (рис. 4).

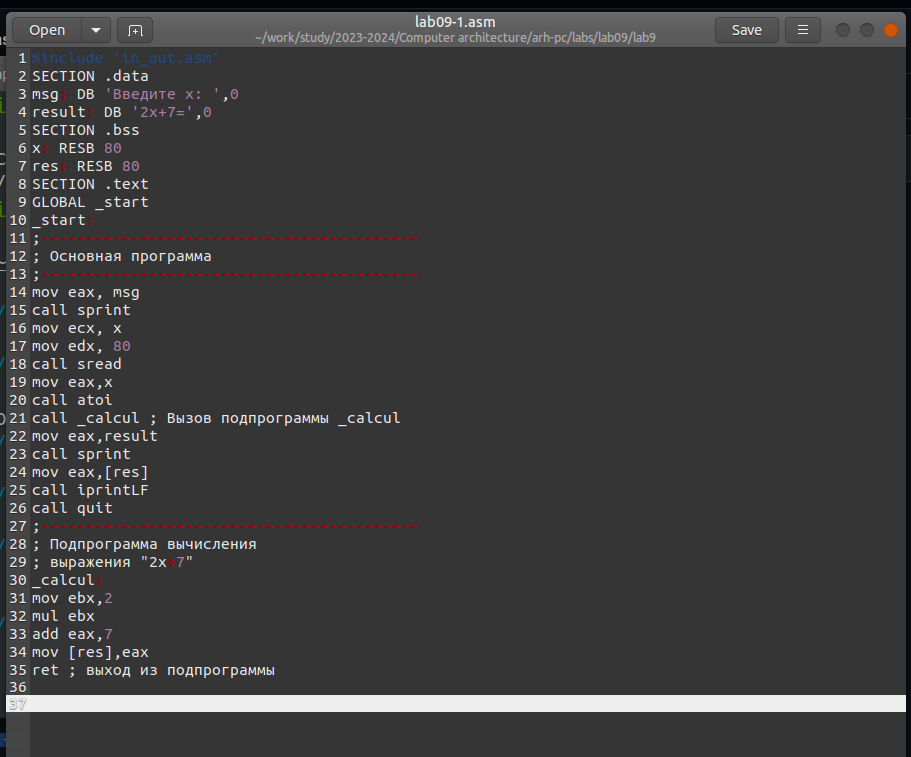


рис.4 Редактирование файла

Создаю объектный файл программы и после компановки запускаю его (рис. 5). Код с подпрограммой работает успешно.

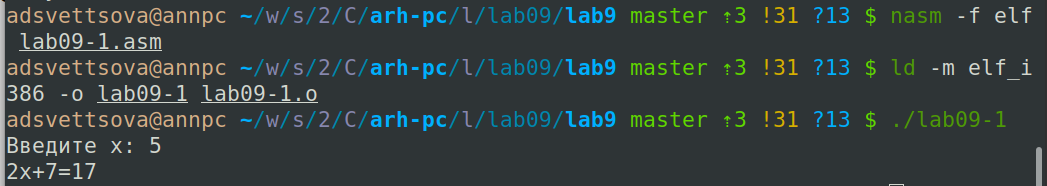


рис.5 Запуск программы

Изменяю текст файла,добавив подпрограмму sub\_calcul в подпрограмму \_calcul (рис. 6).

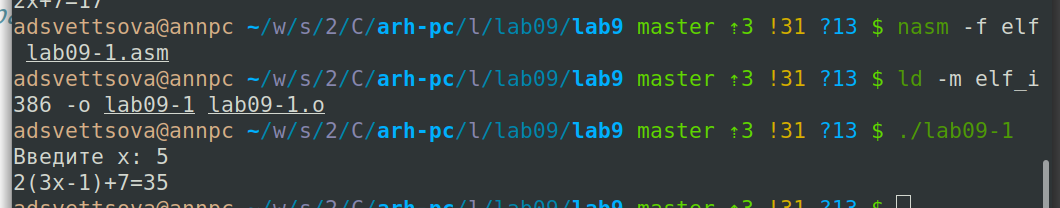


рис.6 Редактирование файла

%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:  
mov eax, msg  
call sprint  
mov ecx, x  
mov edx, 80  
call sread  
mov eax,x  
call atoi  
call \_calcul ; Вызов подпрограммы \_calcul  
mov eax,result  
call sprint  
mov eax,[res]  
call iprintLF  
call quit  
\_calcul:  
call \_subcalcul  
mov ebx,2  
mul ebx  
add eax,7  
mov [res],eax  
ret ; выход из подпрограммы  
\_subcalcul:  
mov ebx,3  
mul ebx  
sub eax,1  
ret

Запускаю исполняемый файл (рис. 7).Программа работает верно.

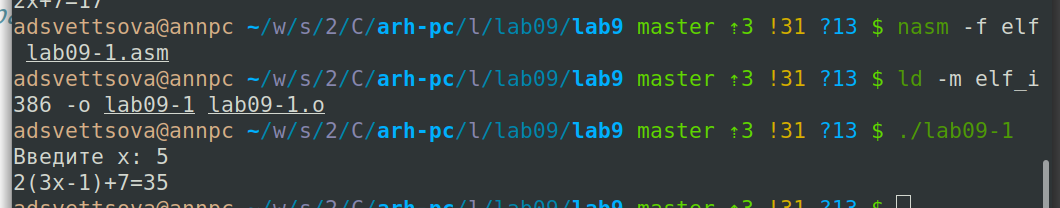


рис.7 Запуск программы

## 4.2 Отладка программ с помощью GDB

Создаю файл lab09-2.asm, используя команду touch (рис. 8).

рис.8 Создание файла

рис.8 Создание файла

Записываю код программы из листинга 9.2,который выводит сообщение Hello world (рис. 9).

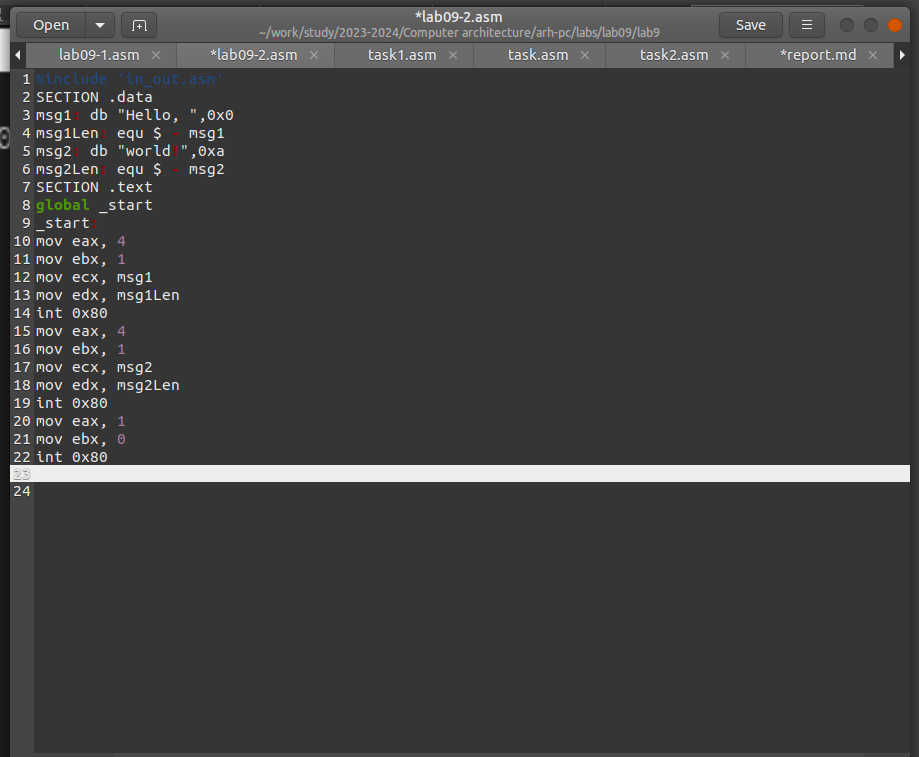


РИС.9 Редактирование файла

Получаю исполняемый файл. для работы с GDB провожу трансляцию программ с ключом “-g” и загружаю исполняемый файл в отладчик (рис. 10).

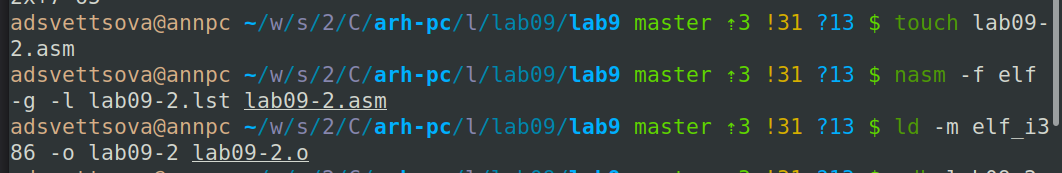


рис.10 Запуск исполняемого файла

Проверяю работу программы в оболочке GDB с помощью команды run (рис. 11).

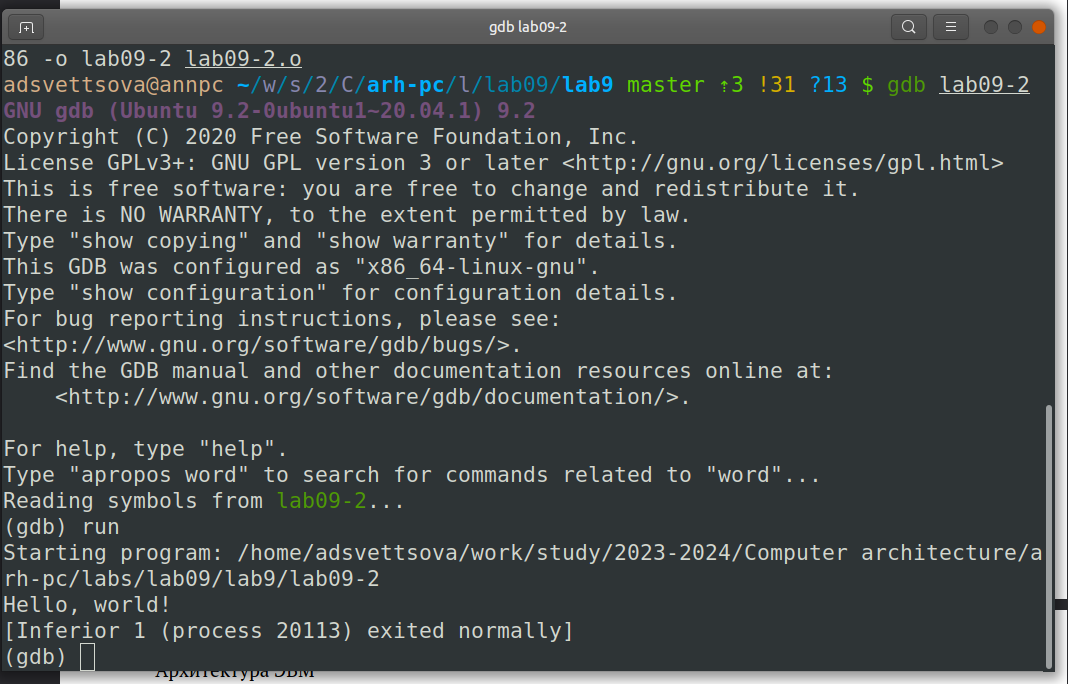


рис.11 Запуск программы в отладчике

Для более подробного анализа устанавливаю брейкпоинт на метку \_start,с которой начинается выполнение ассемблерной программы (рис. 12).

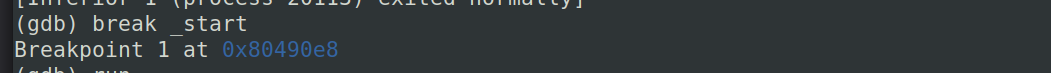


рис.12 Установка брейкпоинта

Запускаю её (рис. 13).

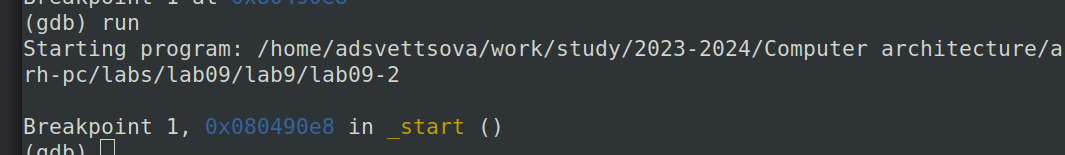


рис.13 Запуск

С помощью команды “disassemble \_start” просматриваю дисассимилированный код программы (рис. 14).

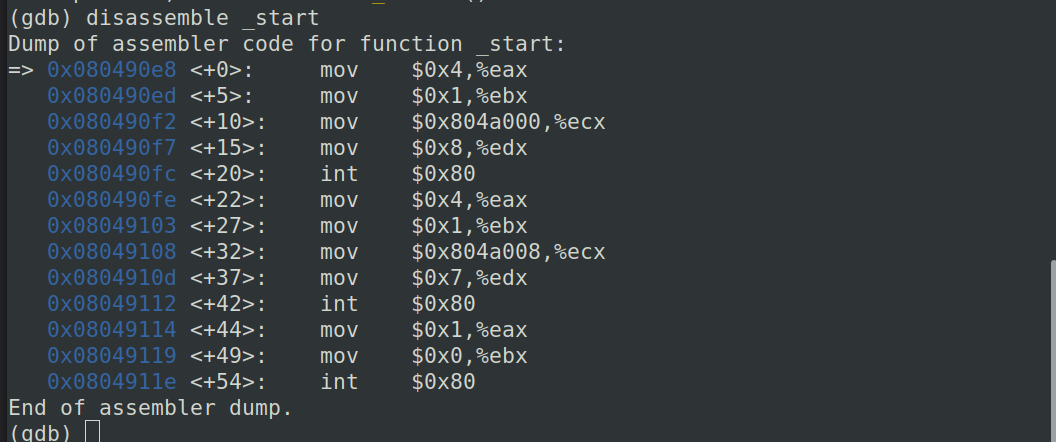


рис.14 Диссассимилированный код программы

Переключаюсь на отображение команд с Intel’овским синтаксисом, введя команду “set disassembly-flavor intel” (рис. 15).

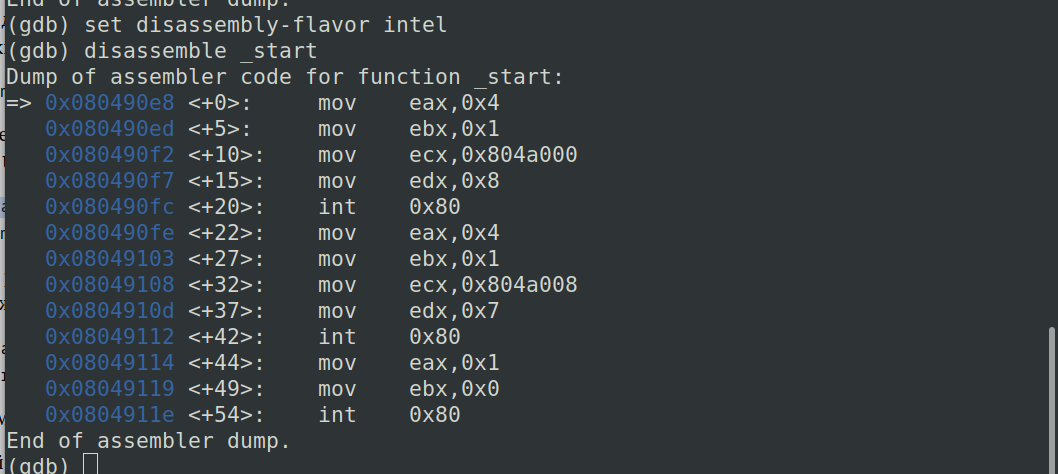


рис.15 Отображение с Intel’овским синтаксисом

Основное различие заключается в том,что в режиме Intel пишется сначала сама команда,а потом её машинный код, в то время как в режиме ATT идет сначала машинный код,а только потом сама команда.

## 4.3 Добавление точек останова

Проверяю наличие точки останова с помощью команды info breakpoints (i b) (рис. 16).

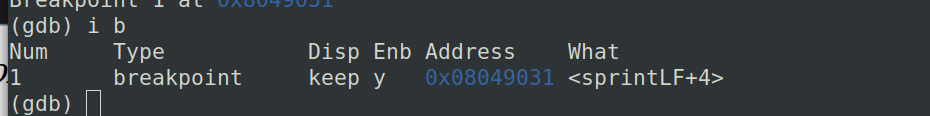


рис.16 Точка останова

Устанавливаю ещё одну точку останова по адресу инструкции,которую можно найти в средней части в левом столбце соответствующей инструкции (рис. 17).

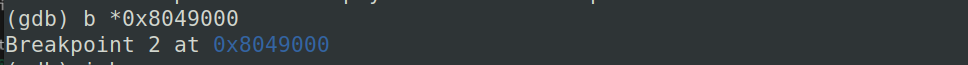


рис.17 Установка точки останова

Просматриваю информацию о точках останова (рис. 18).

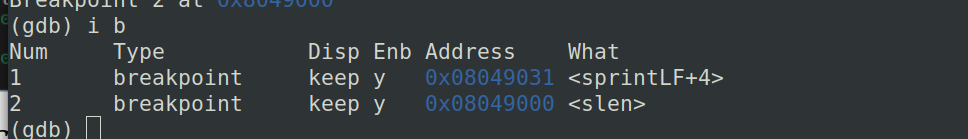


рис.18 Точки останова

## 4.4 Работа с данными программы в GDB

Просматриваю содержимое регистров с помощью команды info register (i r) (рис. 19).

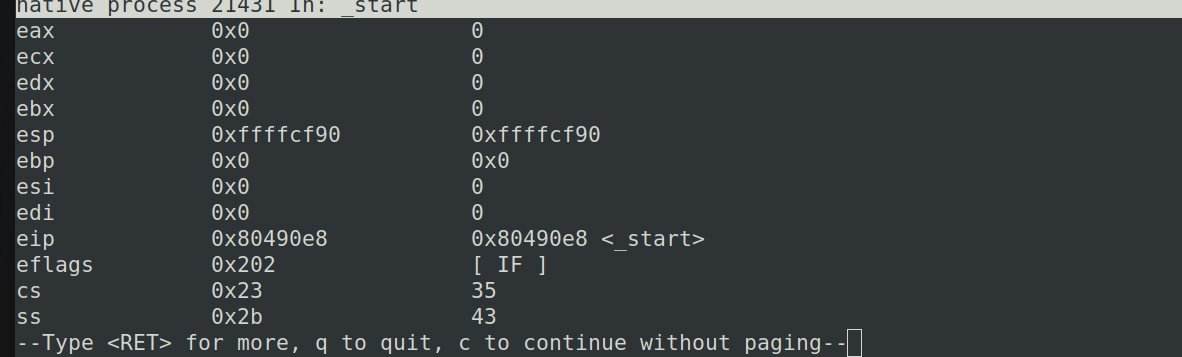


рис.19 info register

Узнаю значение переменной msg1 по имени (рис. 20).

рис.20 Значение переменной по имени

рис.20 Значение переменной по имени

Меняю первый символ переменной msg1 (рис. 21).

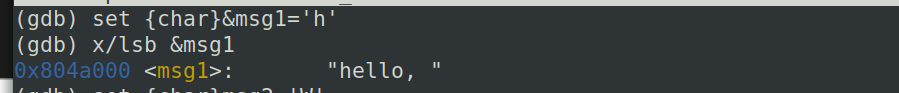


рис.21 Изменение переменной

Также меняю первый символ переменной msg2 (рис. 22).

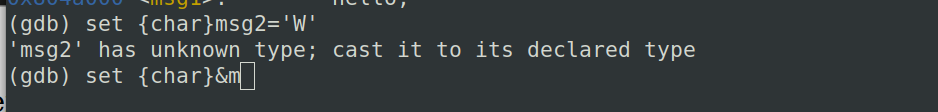


рис.22 Изменение второй переменной

Вывожу значение регистра edx в различных форматах (в шестнадцатеричном,двоичном и символьном форматах) (рис. 23).

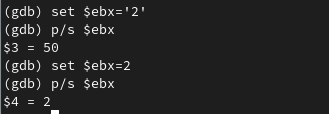


рис.23 Изменение значений в разные форматы

С помощью команды set изменяю значение регистра ebx (рис. 24).

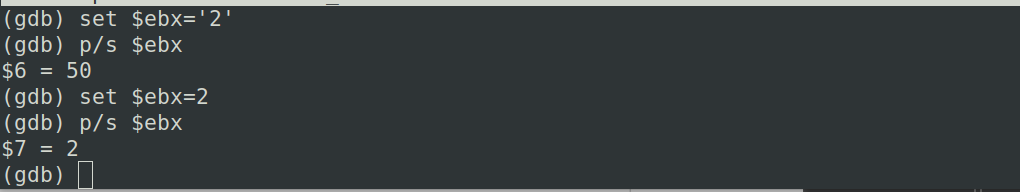


рис.24 Изменение значений ebx

Значение регистра отличаются,так как в первом случае мы выводим код символа 2,который в десятичной системе счисления равен 50,а во втором случае выводится число 2,представленное в этой же системе.

## 4.5 Обработка аргументов командной строки в GDB

Копирую файл lab8-2.asm,созданный при выполнении лабораторной работы №8,который выводит на экран аргументы, в файл с именем lab09-3.asm (рис. 25).

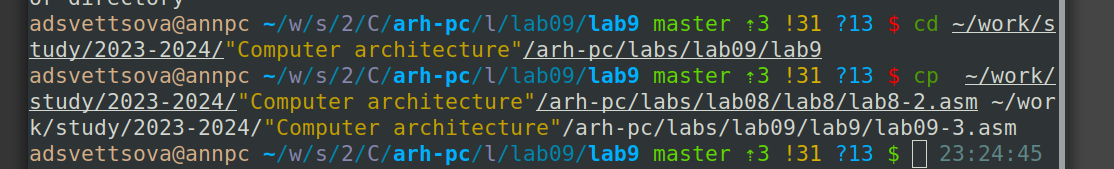


рис.25 Копирование файла

Создаю исполняемый файл,использую ключ –args для загрузки программы в GDB. Загружаю исполняемый файл,указав аргументы (рис. 26).

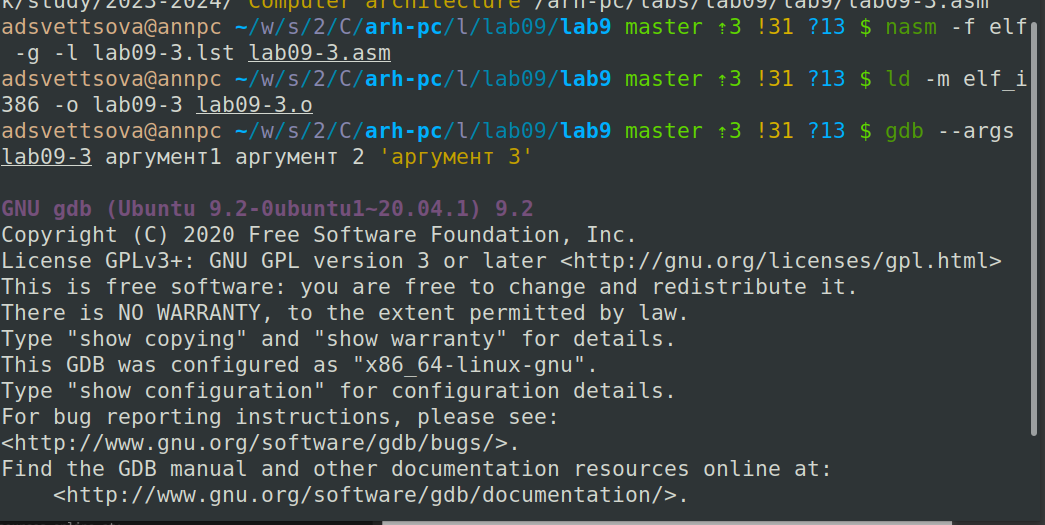


рис.26 Создание файла

Устанавливаю точку останова перед первой инструкцией в программе и запускаю её (рис. 27).

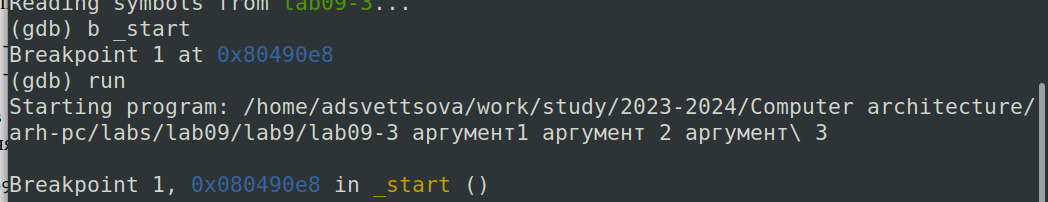


рис.27 Запуск программы с точкой останова

Просматриваю адрес вершины стека,который хранится в регистре esp (рис. 28).

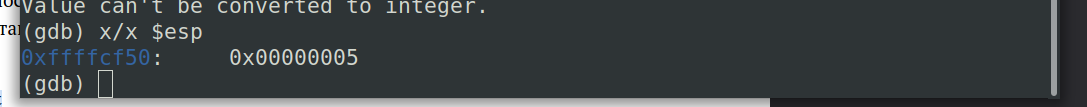


рис.28 Регистр esp

Ввожу другие позиции стека- в отличие от адресов,располагается адрес в памяти: имя,первый аргумент,второй и т.д (рис. 29).

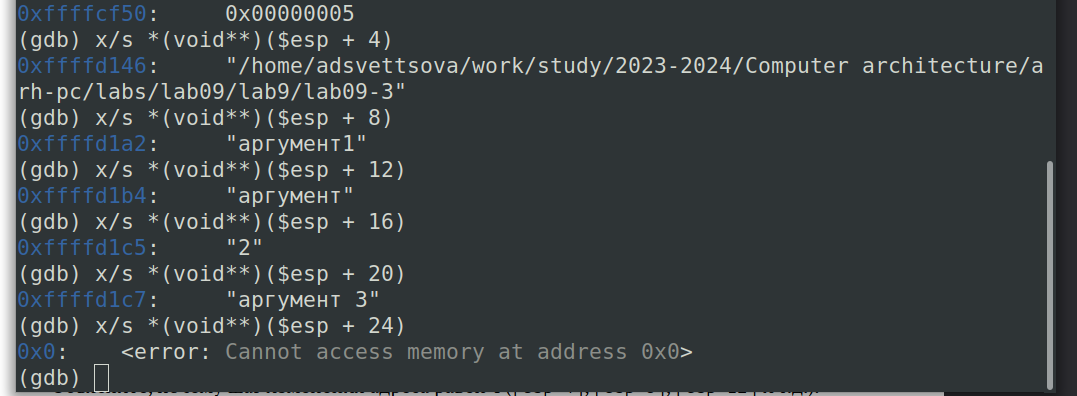


рис.29 Позиции стека

Количество аргументов командной строки 4,следовательно и шаг равен четырем.

## 4.6 Задание для самостоятельной работы

Создаю файл для первого самостоятельного задания,который будет называться task1.asm. Редактирую код программы lab8-4.asm,добавив подпрограмму,которая вычисляет значения функции f(x) (рис. 30).

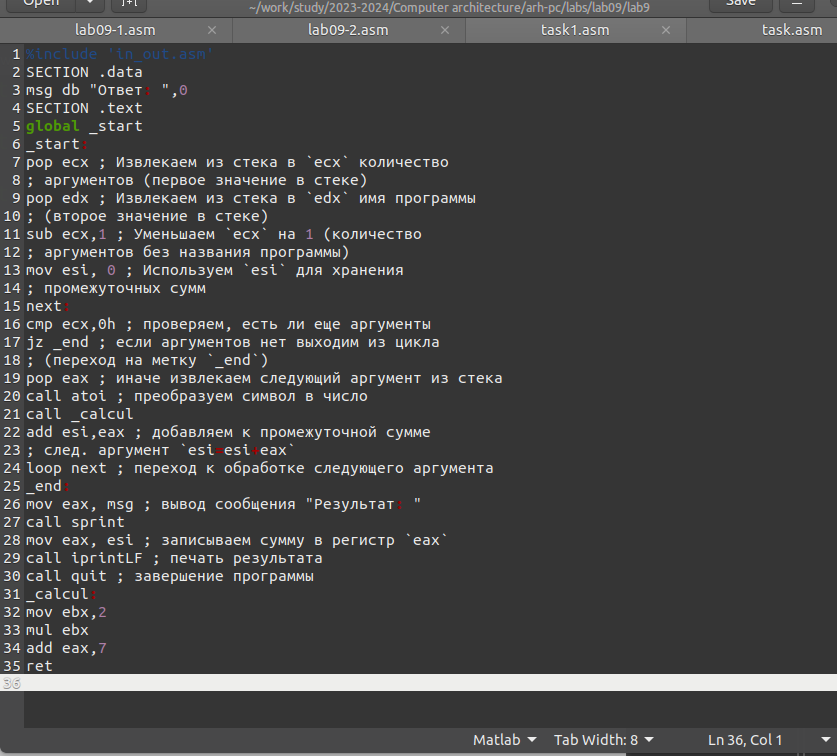


рис.30 Редактирование файла

%include 'in\_out.asm'  
SECTION .data  
msg db "Ответ: ",0  
SECTION .text  
global \_start  
\_start:  
pop ecx ; Извлекаем из стека в `ecx` количество  
; аргументов (первое значение в стеке)  
pop edx ; Извлекаем из стека в `edx` имя программы  
; (второе значение в стеке)  
sub ecx,1 ; Уменьшаем `ecx` на 1 (количество  
; аргументов без названия программы)  
mov esi, 0 ; Используем `esi` для хранения  
; промежуточных сумм  
next:  
cmp ecx,0h ; проверяем, есть ли еще аргументы  
jz \_end ; если аргументов нет выходим из цикла  
; (переход на метку `\_end`)  
pop eax ; иначе извлекаем следующий аргумент из стека  
call atoi ; преобразуем символ в число  
call \_calcul  
add esi,eax ; добавляем к промежуточной сумме  
; след. аргумент `esi=esi+eax`  
loop next ; переход к обработке следующего аргумента  
\_end:  
mov eax, msg ; вывод сообщения "Результат: "  
call sprint  
mov eax, esi ; записываем сумму в регистр `eax`  
call iprintLF ; печать результата  
call quit ; завершение программы  
\_calcul:  
mov ebx,2  
mul ebx  
add eax,7  
ret

Создаю исполняемый файл и ввожу аргументы (рис. 31). Программа работает верно.

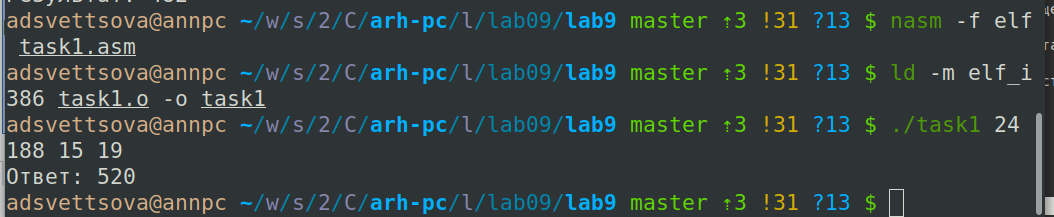


рис.31 Запуск программы

Создаю файл и ввожу код из листинга 9.3 (рис. 32).

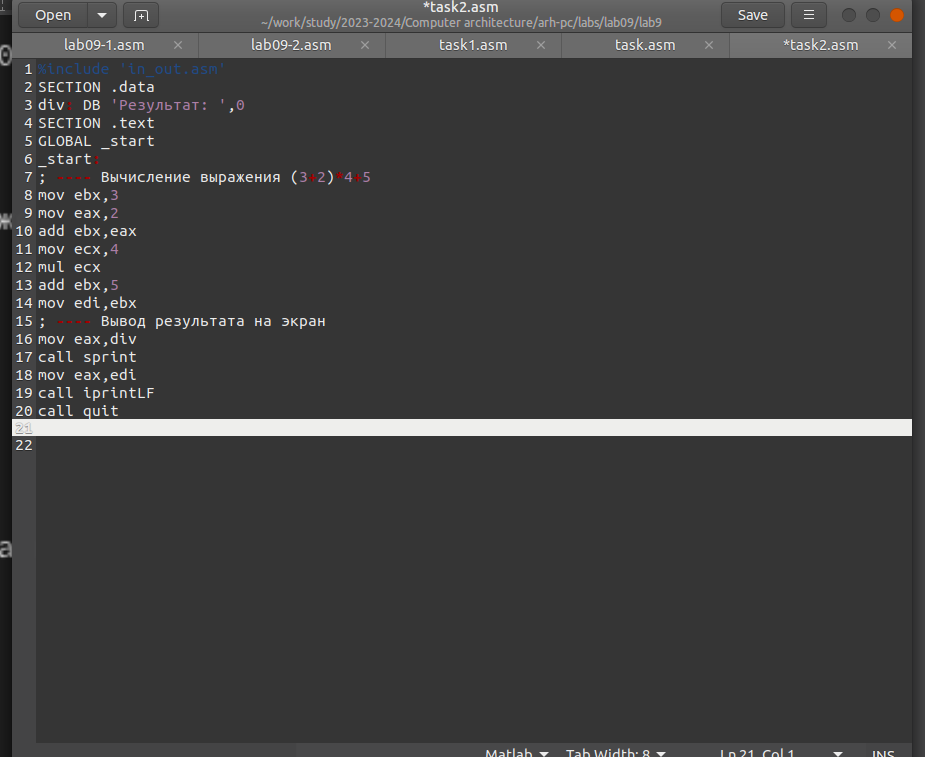


рис.32 Редактирование файла

Открываю файл в отладчике GDB и запускаю программу (рис. 33, рис.34). Программа выдает ответ 10.

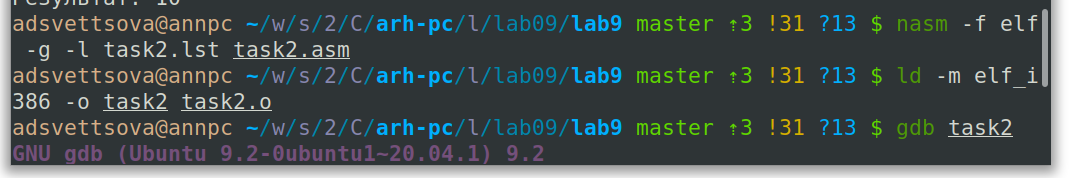


рис.33 Запуск программы в отладчике

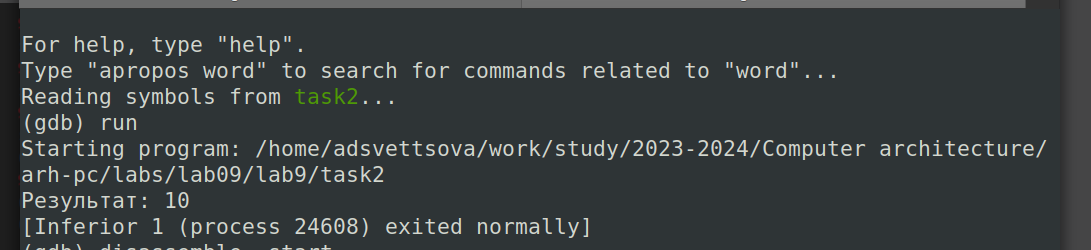


рис.34 Запуск программы в отладчике

Просматриваю дисассимилированный код программы, ставлю точку останова перед прибавлением 5 и открываю значения регистров на данном этапе (рис. 35).

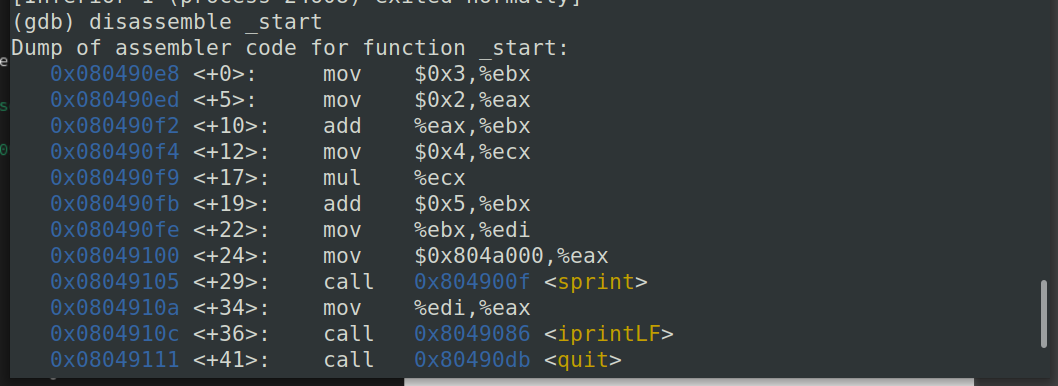


рис.35 Действия в отладчике

Как можно увидеть, регистр ecx со значением 4 умножается не на ebx,сложенным c eax, а только с eax со значением 2. Значит нужно поменять значения регистров(например присвоить eax значение 3 и просто прибавит 2. После изменений программа будет выглядить следующим образом:

%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 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

Пробуем запустить программу (рис. 36. Она работает верно.

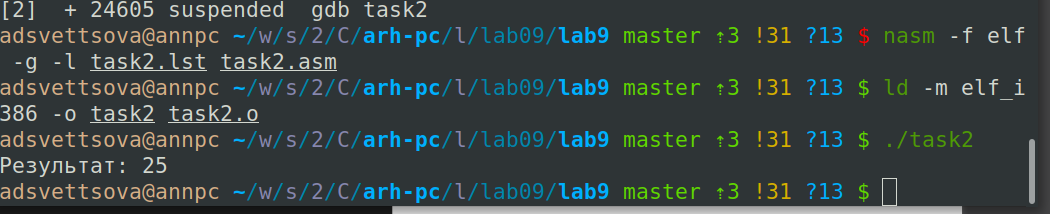


рис.36 Запуск программы

# 5 Выводы

Во время выполнения данной лабораторной работы я приобрела навыки написания программ с использованием подпрограмм и ознакомилась с методами отладки при помощи GDB и его основными возможностями.

# Список литературы

1. GDB: The GNU Project Debugger. — URL: https://www.gnu.org/software/gdb/.
2. GNU Bash Manual. — 2016. — URL: https://www.gnu.org/software/bash/manual/.
3. Midnight Commander Development Center. — 2021. — URL: https://midnight-commander. org/.
4. NASM Assembly Language Tutorials. — 2021. — URL: https://asmtutor.com/.
5. Newham C. Learning the bash Shell: Unix Shell Programming. — O’Reilly Media, 2005 — 354 с. — (In a Nutshell). — ISBN 0596009658. — URL: http://www.amazon.com/Learningbash-Shell-Programming-Nutshell/dp/0596009658.
6. Robbins A. Bash Pocket Reference. — O’Reilly Media, 2016. — 156 с. — ISBN 978-1491941591.
7. The NASM documentation. — 2021. — URL: https://www.nasm.us/docs.php.
8. Zarrelli G. Mastering Bash. — Packt Publishing, 2017. — 502 с. — ISBN 9781784396879.
9. Колдаев В. Д., Лупин С. А. Архитектура ЭВМ. — М. : Форум, 2018.
10. Куляс О. Л., Никитин К. А. Курс программирования на ASSEMBLER. — М. : Солон-Пресс, 2017.
11. Новожилов О. П. Архитектура ЭВМ и систем. — М. : Юрайт, 2016.
12. Расширенный ассемблер: NASM. — 2021. — URL: https://www.opennet.ru/docs/RUS/nasm/.
13. Робачевский А., Немнюгин С., Стесик О. Операционная система UNIX. — 2-е изд. — БХВПетербург, 2010. — 656 с. — ISBN 978-5-94157-538-1.
14. Столяров А. Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011. — URL: http://www.stolyarov.info/books/asm\_unix.
15. Таненбаум Э. Архитектура компьютера. — 6-е изд. — СПб. : Питер, 2013. — 874 с. — (Классика Computer Science).
16. Таненбаум Э., Бос Х. Современные операционные системы. — 4-е изд. — СПб. : Питер,2015. — 1120 с. — (Классика Computer Science).