РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук Кафедра прикладной информатики и теории вероятностей

Архипов Олег Константинович

Содержание

# 1 Цель работы

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

# 2 Выполнение лабораторной работы

## 2.1 Реализация подпрограмм в NASM

Создаю каталог для ЛР9, а также файл lab09-1.asm (рис. [1](#fig:001)).

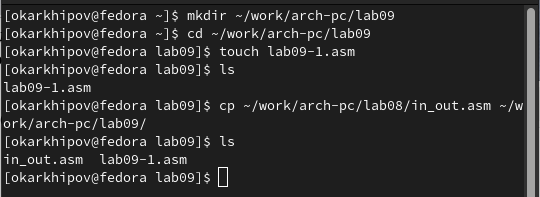


Figure 1: Папка новой ЛР и файл lab09-1.asm

Ввожу программу ычисления арифметического выражения с помощью подпрограммы \_calcul (рис. [2](#fig:002)).

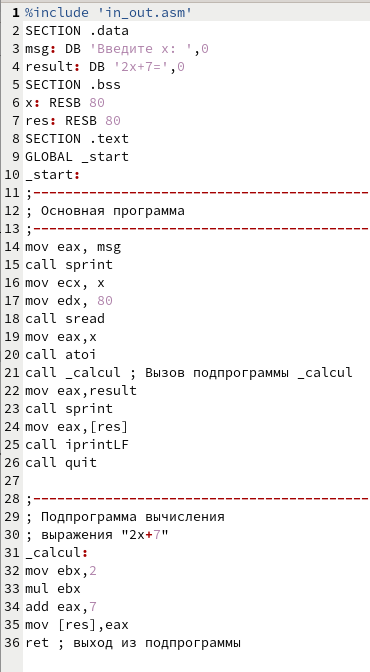


Figure 2: Код lab09-1.asm

Создаю исполняемый файл и проверяю работу программы со значением х=5 (рис. [3](#fig:003)).

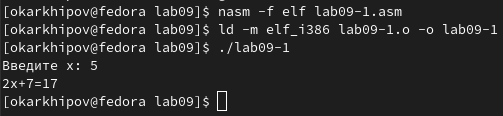


Figure 3: Работа программы lab09-1

Изменяю текст программы, добавляя подпрограмму \_subcalcul в подпрограмму \_calcul, для вычисления выражения , где вводится с клавиатуры, , . Т.е. передается в подпрограмму \_calcul из нее в подпрограмму \_subcalcul, где вычисляется выражение , результат возвращается в \_calcul и вычисляется выражение . Результат возвращается в основную программу для вывода результата на экран. Помимо этого добавляю сообщение, напоминающее о возникновении сложной функции (см. листинг ниже, ср. с рис. [2](#fig:002)).

**Листинг программы вычисления значения сложной функции**

%include 'in\_out.asm'  
SECTION .data  
msg: DB 'Введите x: ',0  
result: DB 'Результат: 2\*g(x)+7=',0  
ad: DB 'где g(x)=3x-1',0 ; новое выражение является сложной функцией  
SECTION .bss  
x: RESB 80  
res: RESB 80  
SECTION .text  
GLOBAL \_start  
\_start:  
;------------------------------------------  
; Основная программа  
;------------------------------------------  
mov eax, msg  
call sprint  
mov ecx, x  
mov edx, 80  
call sread  
mov eax,x  
call atoi  
call \_calcul ; Вызов подпрограммы \_calcul  
mov eax,result  
call sprint  
mov eax,[res]  
call iprintLF  
mov eax,ad  
call sprintLF  
call quit  
  
;------------------------------------------  
; Подпрограмма вычисления  
; выражения "2x+7"  
\_calcul:  
call \_subcalcul ; Вызов подпрограммы \_subcalcul  
mov ebx,2  
mul ebx  
add eax,7  
mov [res],eax  
ret ; выход из подпрограммы  
  
\_subcalcul: ; подпрограмма подпрограммы  
mov ebx,3 ; ebx=3  
mul ebx ; eax=eax\*ebx  
sub eax,1 ; eax=eax-1  
ret ; завершение \_subcalcul

Проверяю результат (рис. [4](#fig:004)).

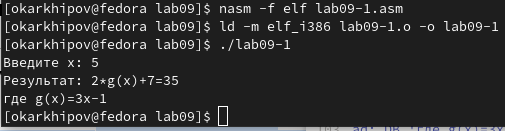


Figure 4: Работа измененной программы lab09-1

## 2.2 Отладка программам с помощью GDB

Создаю файл lab09-2.asm (рис. [5](#fig:005)).

Figure 5: Файл lab09-2.asm

Figure 5: Файл lab09-2.asm

Ввожу код для вывода сообщения ‘Hello world!’ (рис. [6](#fig:006)).

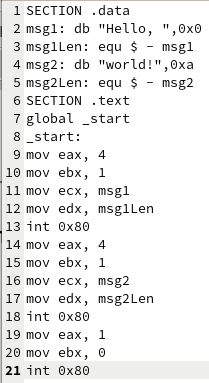


Figure 6: Код lab09-2.asm

Создаю исполняемый файл lab09-2 (рис. [7](#fig:007)).

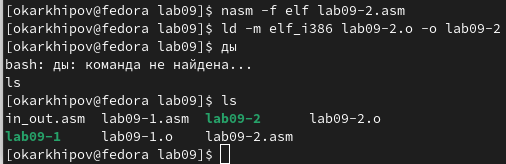


Figure 7: Код lab09-2.asm

Провожу трансляцию с ключом ‘-g’ , чтобы программу можно было отлаживать на уровне строк исходного кода (рис. [8](#fig:008)).

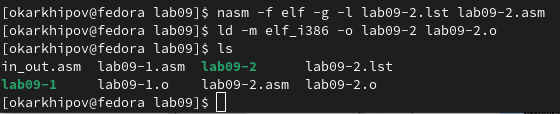


Figure 8: Трансляция и компановка lab09-2.asm

Загружаю исполняемый файл в отладчик. Выводится общая информация об отладчике и полезные ссылки (рис. [9](#fig:009)).

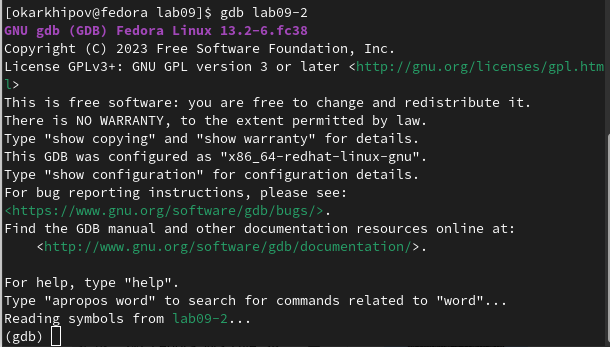


Figure 9: Загрузка lab09-2 в отладчик

Проверяю работу программы, запустив ее в оболочке GDB с помощью команды r , подтверждаю начало сессии, нажав ‘y’->‘ENTER’ (рис. [10](#fig:010)).

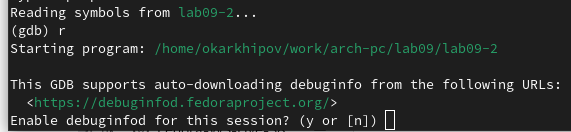


Figure 10: Запуск lab09-2 в оболочке GDB

На экран выводится результат работы программы (рис. [11](#fig:011)).

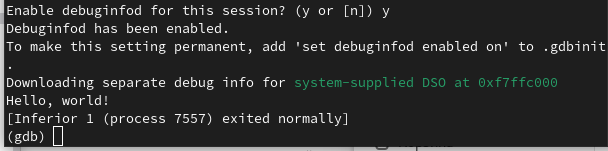


Figure 11: Работа lab09-2 в оболочке GDB

Устанавливаю брейкпоинт на метку \_start, с которой начинается выполнение любой ассемблерной программы, после чего запускаю программу. Появляется краткая информация о точке останова и строке, следующей за \_start (рис. [12](#fig:012)).

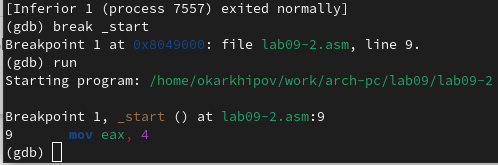


Figure 12: Брейкпоинт на метке \_start

Ввожу команду disassemble \_start , чтобы посмотреть дизассемблированный код программы (рис. [13](#fig:013)).



Figure 13: Дизассемблированный код программы lab09-2

Переключаюсь на отображение команд с Intel’овским синтаксисом:

(gdb) set disassembly-flavor intel  
(gdb) disassemble \_start

Можно сравнить рис. [13](#fig:013) и рис. [14](#fig:014). В последней колонке режима ATT сначала указан адрес второго операнда, а затем со знаком ‘%’ - первый. Синтаксис Intel больше похож на исходный код, как он вводится в текстовый редактор: сохранен порядок адресов операндов и нет ‘%’.

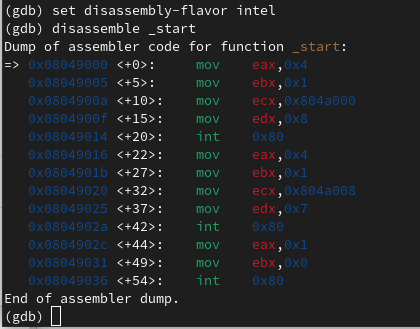


Figure 14: Дизассемблированный код с синтаксисом Intel

Ввожу команды для перехода в режим псевдографики. Как видим, значения регистров недоступны (рис. [15](#fig:015)), так что снова ввожу команду run , после чего значения регистров появляются в верхней секции (рис. [16](#fig:016)).

(gdb) layout asm  
(gdb) layout regs

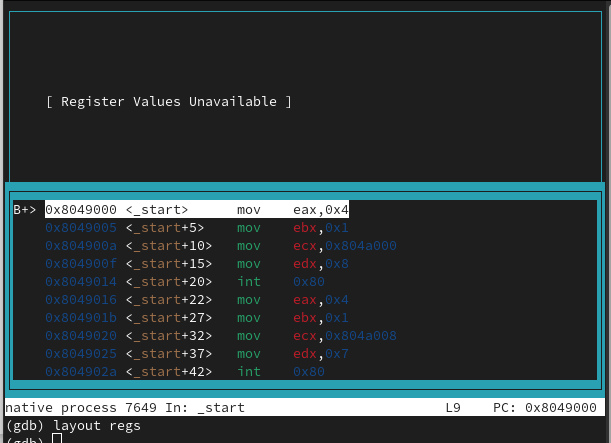


Figure 15: Режим псевдографики gdb 1

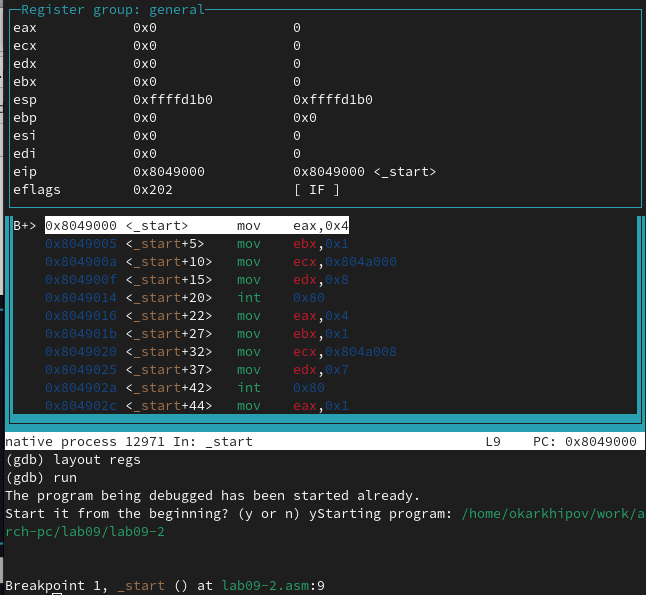


Figure 16: Режим псевдографики gdb 2

Проверяю наличие введенной ранее точки останова (рис. [17](#fig:017)).

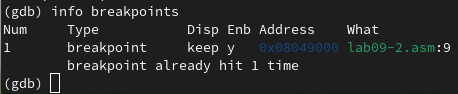


Figure 17: Точка останова break \_start

Создаю новую точку останова по адресу предпоследней инструкции (mov ebx,0x0) и проверяю, какие точки останова созданы (рис. [18](#fig:018)).

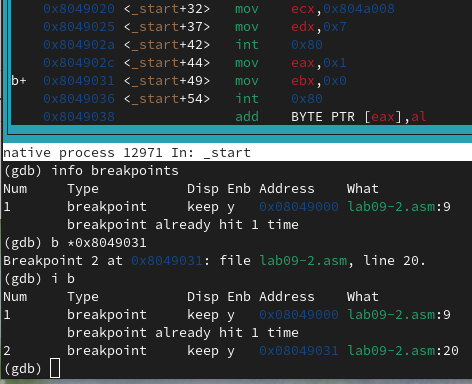


Figure 18: Точка останова по адресу инструкции

C помощью команды stepi (или si) выполняю 5 инструкций (рис. [19](#fig:019)-[24](#fig:024)). Как видим, на каждом шаге измененные регистры выделены белым: 2 раза eax, ebx, 3 раза eip , ecx , edx .

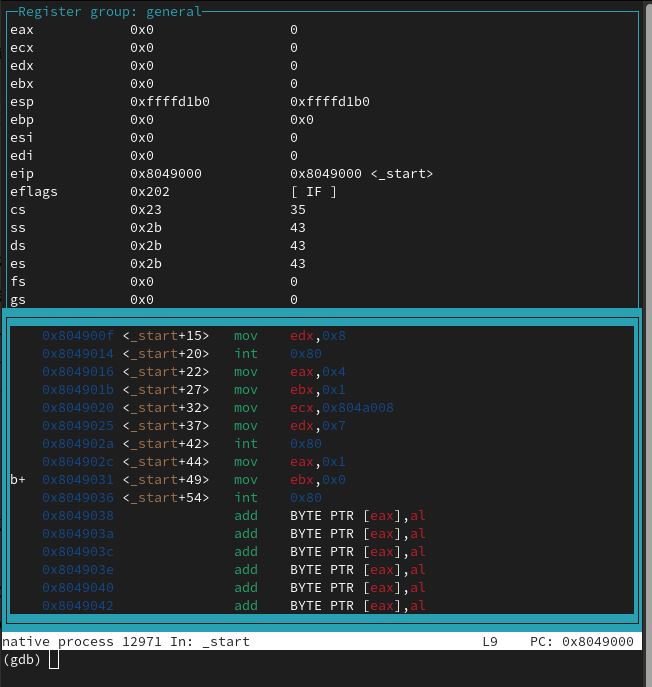


Figure 19: Начало

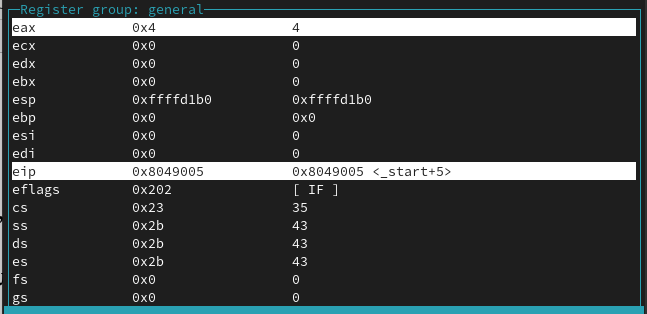


Figure 20: si 1

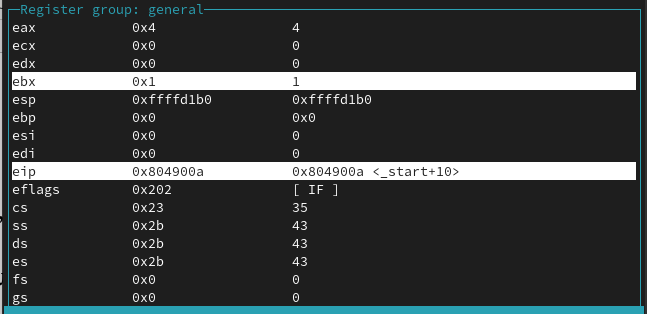


Figure 21: si 2

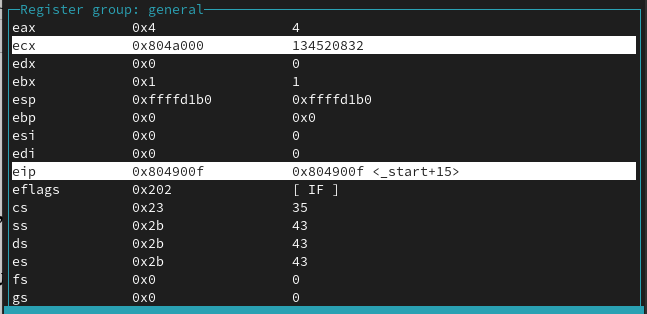


Figure 22: si 3

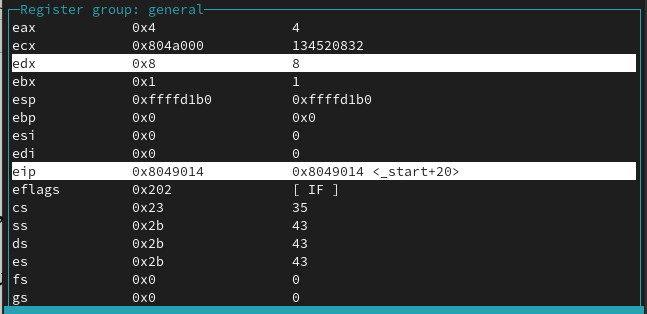


Figure 23: si 4

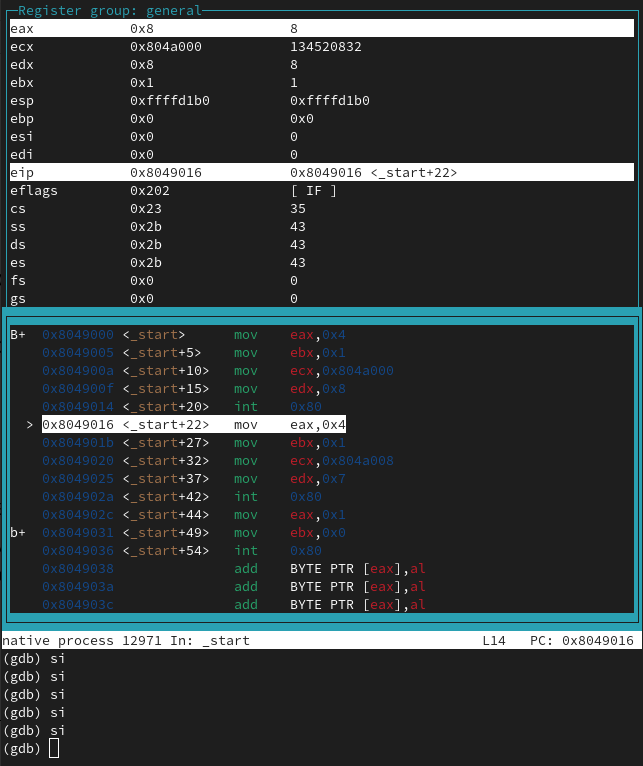


Figure 24: si 5

Далее ввожу команду i r , чтобы просмотреть содержимое регистров другим способом (рис. [25](#fig:025)).

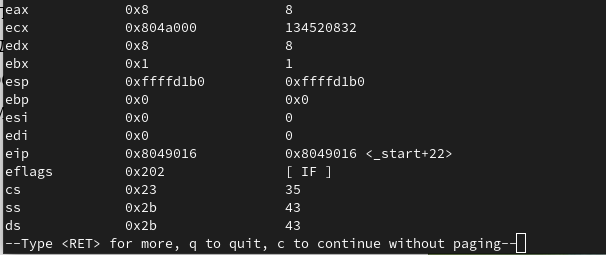


Figure 25: Работа команды info registers

Смотрю значение переменной msg1 по имени (рис. [26](#fig:026)).

Figure 26: Значение переменной msg1 по имени

Figure 26: Значение переменной msg1 по имени

Смотрю значение переменной msg2 по адресу, определяя его по дизассемблированной инструкции mov ecx,msg2 (рис. [27](#fig:027)).

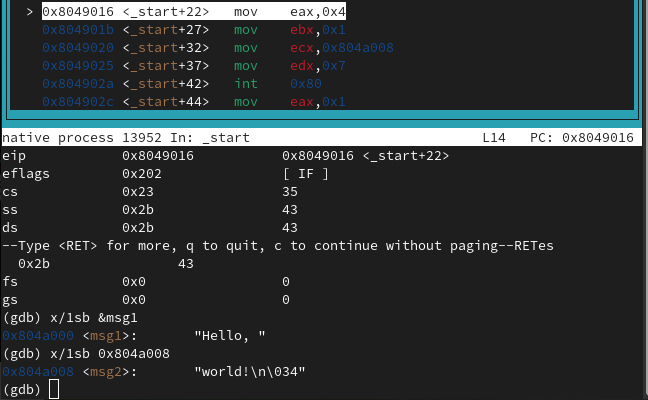


Figure 27: Значение переменной msg2 по адресу

Заменяю первый и второй символы переменной msg1 (рис. [28](#fig:028)).

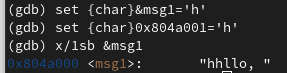


Figure 28: Замена символов переменной msg1

Заменяю третий символ переменной msg2 (‘r’ на ‘w’) (рис. [29](#fig:029)).

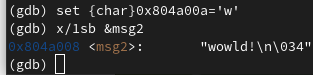


Figure 29: Замена символов переменной msg2

Вывожу в различных форматах значение регистра edx (p/x и p/a - символьный, p/t - двоичный, p/s - шестнадцатиричный) (рис. [30](#fig:030)-[31](#fig:031)).

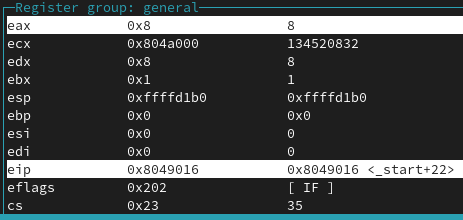


Figure 30: Секция значений регистров

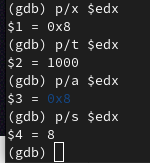


Figure 31: Различные форматы edx

С помощью команды set изменяю значение регистра ebx (рис. [32](#fig:032)). Результаты различны из-за того, что в первом случае значение ebx было заменено на символ ‘2’, код которого в ASCII равен 50, во втором случае значение ebx заменяется на код.

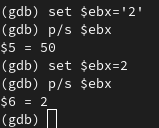


Figure 32: Замена значения регистра ebx

Завершаю выполнение программы при помощи continue и выхожу при помощи q (рис. [33](#fig:033)).

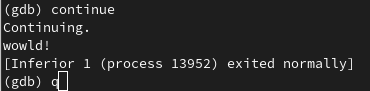


Figure 33: Завершение программы и выход из отладчика

## 2.3 Обработка аргументов командной строки в GDB

Копирую файл lab8-2.asm в lab09 с именем lab09-3.asm , создаю исполняемый файл (рис. [34](#fig:034)).

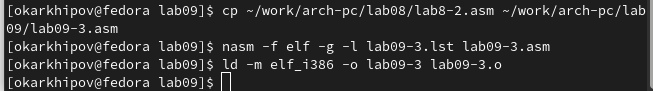


Figure 34: Файл lab09-3.asm

Используя ключ –args , загружаю программу в gdb с аргументами (рис. [35](#fig:035)).

Figure 35: Загрузка lab09-3 в gdb с аргументами

Figure 35: Загрузка lab09-3 в gdb с аргументами

Для исследования расположения аргументов командной строки в стеке после запуска программы с помощью gdb устанавливаю точку останова перед первой инструкцией в программе и запускаю ее (рис. [36](#fig:036)).

(gdb) b \_start  
(gdb) run

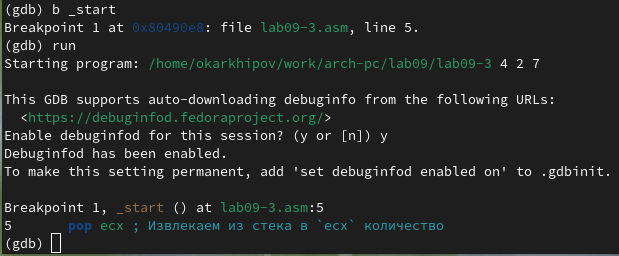


Figure 36: Установка точки останова и запуск lab09-3

Вывожу адрес вершины стека, где располагается число равное количеству аргументов командной строки (включая имя программы), их 4 (рис. [37](#fig:037)).

Figure 37: Количество аргументов

Figure 37: Количество аргументов

Просматриваю остальные позиции стека – по адресу [esp+4] располагается адрес в памяти, где находится имя программы, по адресу [esp+8] хранится адрес первого аргумента, по адресу [esp+12] – второго, по адресу [esp+16] - третьего (рис. [38](#fig:038)). Шаг изменения равен 4, т.к. данные аргументы смещены относительно вершины стека на 4, 8 и т.д. байта.

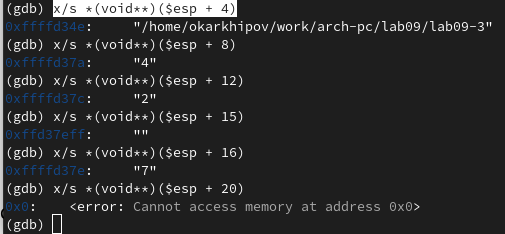


Figure 38: Остальные позиции стека

# 3 Самостоятельная работа

## 3.1 Преобразуйте программу из лабораторной работы №8

Копирую файл самостоятельной работы из ЛР8 с новым именем (рис. [39](#fig:039)).

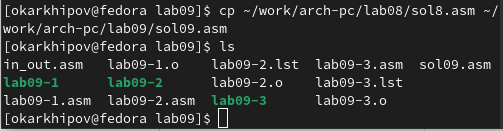


Figure 39: Файл sol09.asm

Редактирую код так, чтобы вычисление значений стало подпрограммой (рис. [40](#fig:040)).

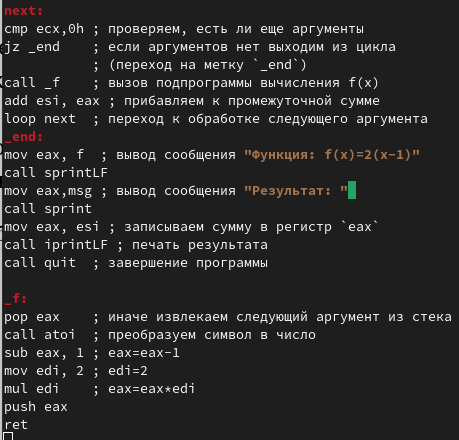


Figure 40: Часть кода с изменениями

Проверка корректности работы измененного кода (рис. [41](#fig:041)).

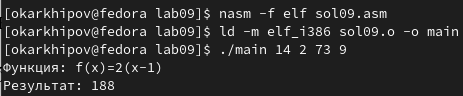


Figure 41: Проверка работы программы

## 3.2 Проверка программы при помощи отладчика

Создаю файл sol09-2.asm и ввожу код из листинга для вычисления выражения (рис. [42](#fig:042)-[43](#fig:043)).

Figure 42: Файл для кода листинга

Figure 42: Файл для кода листинга

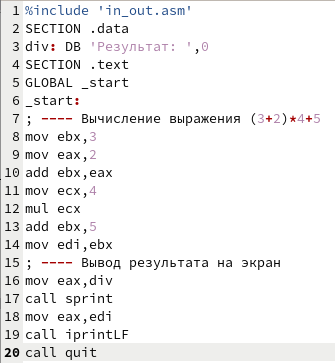


Figure 43: Код листинга

Проверка показывает ошибочный ответ (рис. [44](#fig:044)).

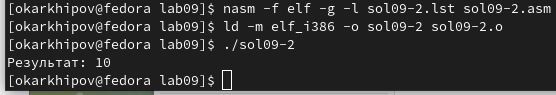


Figure 44: Проверка работы программы sol09-2

Запускаю sol09-2 в gdb , ставлю точку останова на \_start , т.к. ошибка, очевидно, в теле программы и дизассемблирую ее при помощи синтаксиса Intel (рис. [45](#fig:045)).

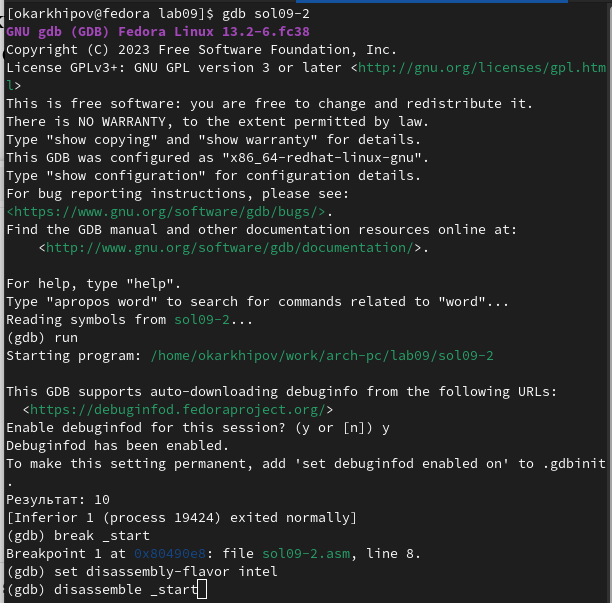


Figure 45: Запуск sol09-2 в gdb , точка останова и дизассемблирование

Включаю режим псевдографики (рис. [46](#fig:046)).

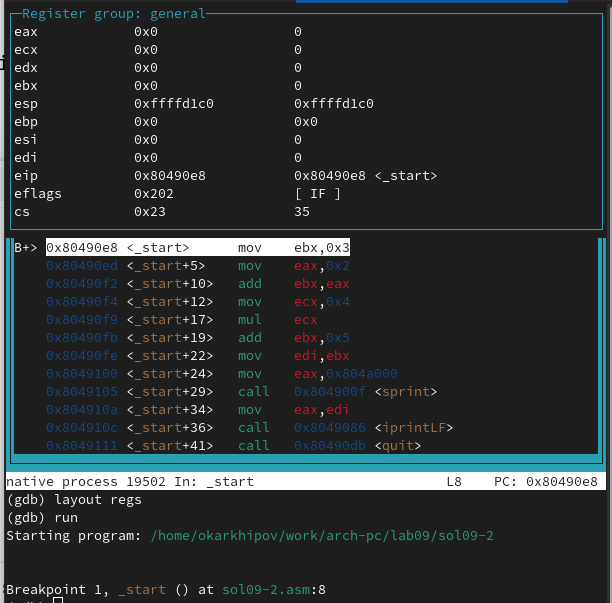


Figure 46: Режим псевдографики

Установлю точки останова после каждого действия (рис. [47](#fig:047)). Буду проверять значения регистров eax , ebx , ecx на каждом шаге, т.к. они участвуют в вычислении значения выражения.

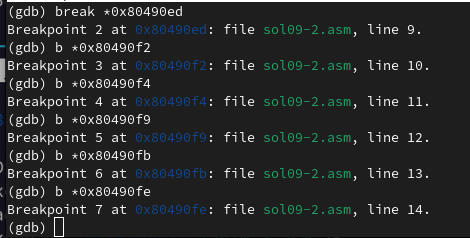


Figure 47: Точки останова

Запуcтив программу, и введя команды

(gdp) p/d $eax  
$1=0  
(gdp) p/d $ebx  
$2=0  
(gdp) p/d $ecx  
$3=0

где d отвечает за отображение результатов в десятичном формате, получаю пока значения во всех 3-х регистрах, равные нулю (рис. [48](#fig:048)).

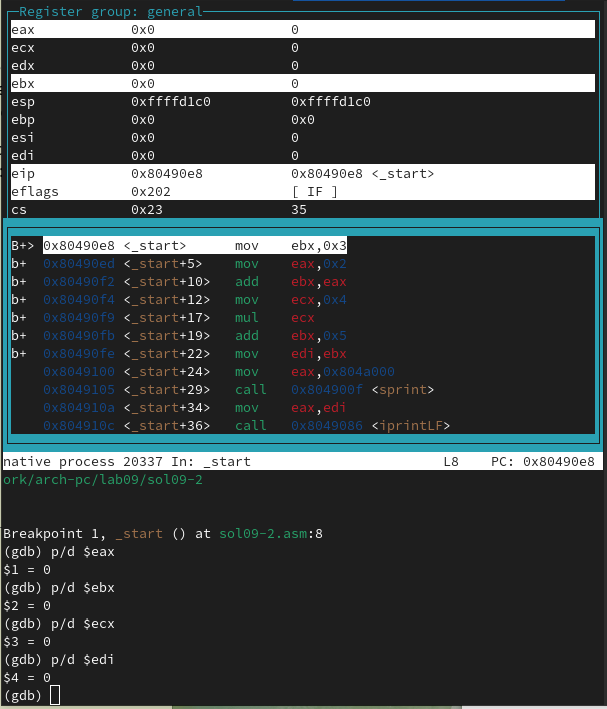


Figure 48: Шаг 0

Далее проматываю программу до следующей точки останова (на 1 строку) при помощи c . В верхней секции вижу, что из интересующих меня изменился только регистр ebx , он стал равен 3 (рис. [49](#fig:049)).

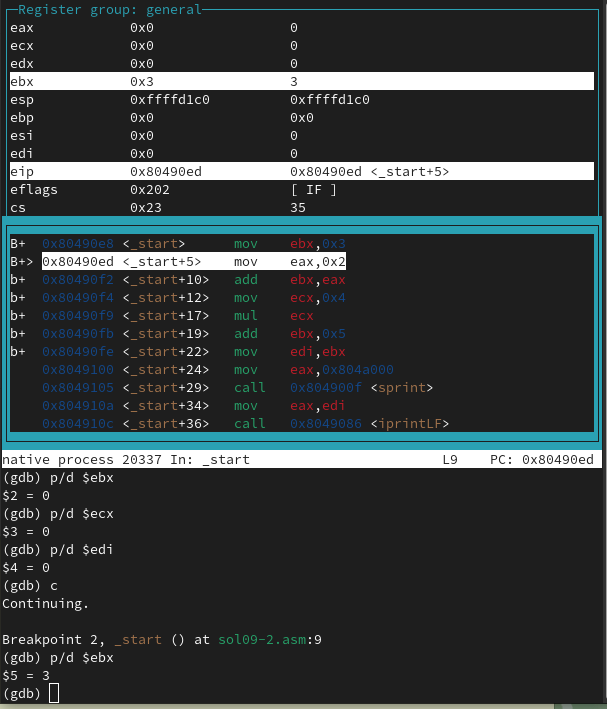


Figure 49: Шаг 1

Так же проматываю далее. Изменился eax , ему присвоено значение 2 (рис. [50](#fig:050)).

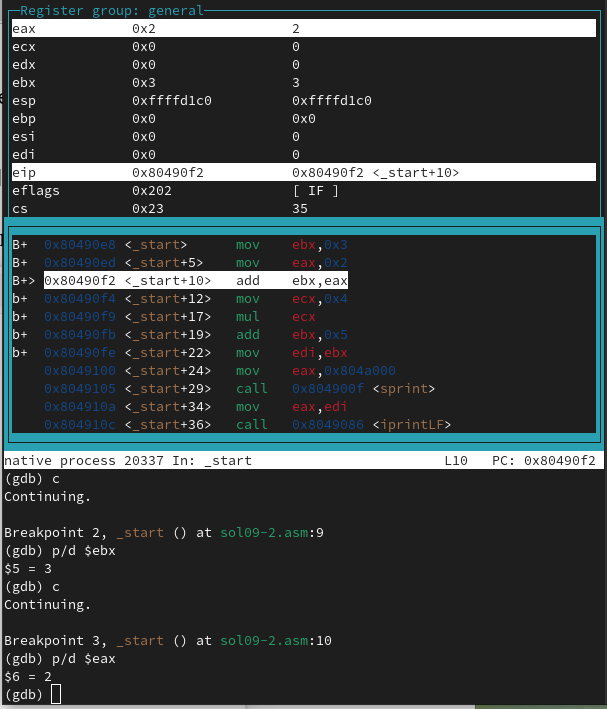


Figure 50: Шаг 2

Теперь эти два числа нужно сложить, за что отвечает следующая строка. Операция выполнена верно (получено число 5), и результат записан в регистр ebx (рис. [51](#fig:051)).

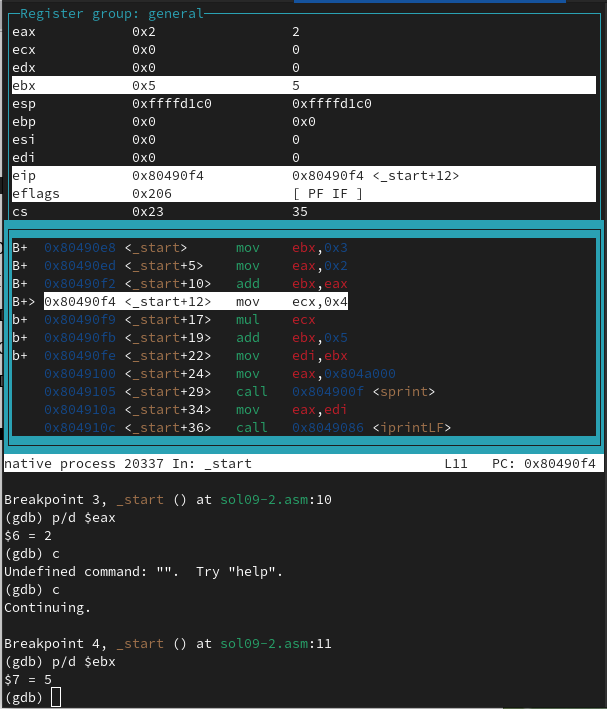


Figure 51: Шаг 3

Далее в регистр ecx было записано число 4 (рис. [52](#fig:052)).

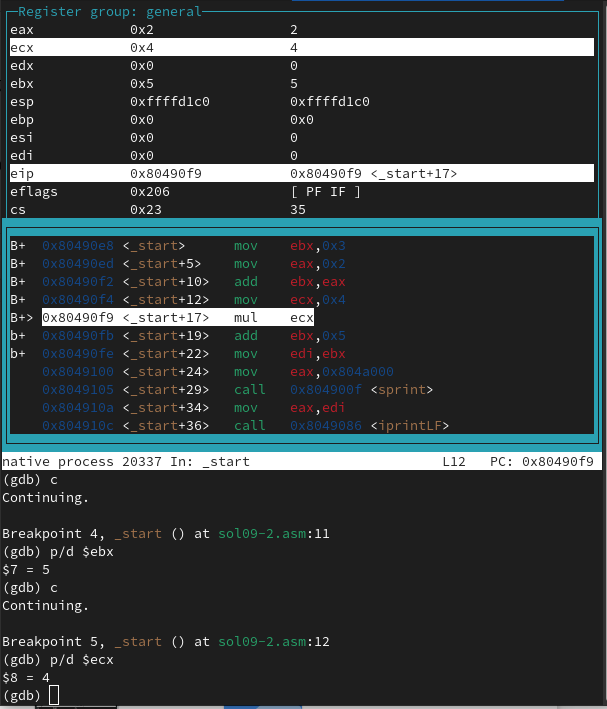


Figure 52: Шаг 4

Перехожу к следующему действию. Теперь в регистре eax стоит число 8 (рис. [53](#fig:053)). Что является результатом операции ‘eax=eax*ecx=2*4’ , это неверно, т.к. необходима операция ‘eax=ebx*ecx=5*4=20’ из-за того что результат первого действия был записан в регистр ebx , однако когда размер операндов 4 байта один из сомножителей всегда берется из регистра eax и записывается в него же. Итак, необходимо записать результат первого сложения в регистр eax .

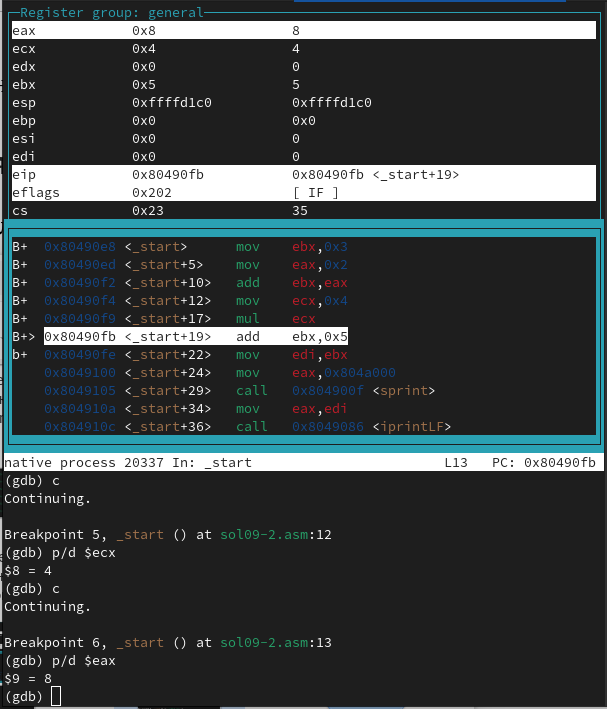


Figure 53: Шаг 5

На всякий случай продолжу проверку. После перехода к следующему шагу получаю ‘ebx=ebx+5=10’. Это итоговый ответ, и он неверный, т.к. в регистре ebx по сказанному выше не учитывается результат умножения (рис. [54](#fig:054)).

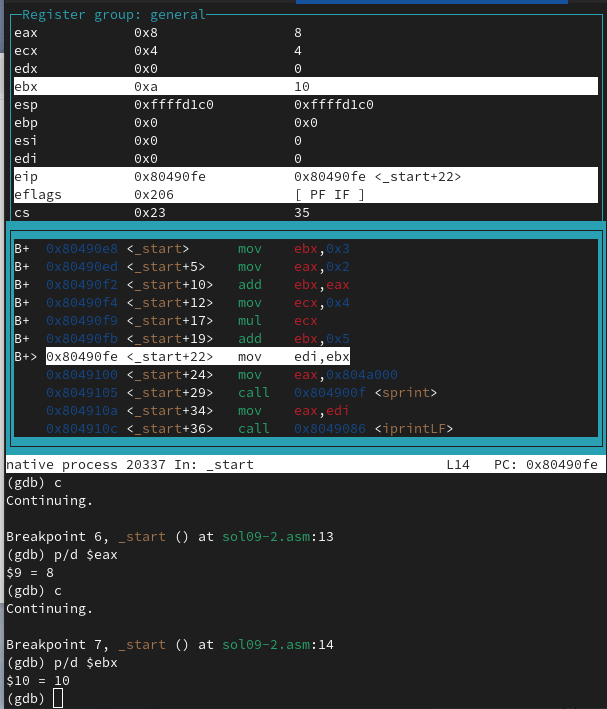


Figure 54: Шаг 6

Итак, чтобы программа работала корректно, нужно записать результат первого и последнего действий в регистр ‘eax’ вместо ‘ebx’ , а также поменять первое слогаемое в последнем действии на eax . Кроме того, т.к. для записи итогового значения используется регистр eax , то в edi нужно поместить значение eax

Выхожу из отладчика и исправляю код (рис. [55](#fig:055)).

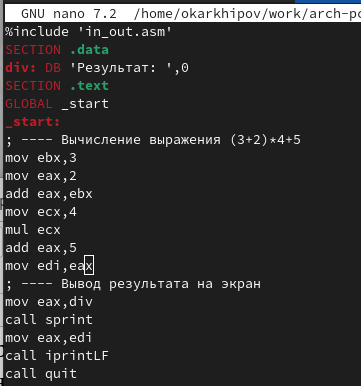


Figure 55: Исправление кода

Сохраняю изменения, выхожу из редактора, создаю исполняемый файл и запускаю его (рис. [56](#fig:056)-[57](#fig:057)).

Figure 56: Создание исполняемого файла

Figure 56: Создание исполняемого файла

Figure 57: Запуск исправленной программы

Figure 57: Запуск исправленной программы

Получаю верно работающую программу. Хууух

# 4 Выводы

Освоены навыки написания программ при помощи подпрограмм и методы отладки при помощи gdb.

# Список литературы