Лабораторная работа №9

Понятие подпрограммы. Отладчик GDB.

Хорошева Алёна Евгеньевна

Содержание

# 1 Цель работы

Освоить работы с отладчиком GDB для исправления ошибок в программе. Приобрести практические навыки написания и использования подпрограмм внутри программ.

# 2 Задание

\*\*Задание для самостоятельной работы\*\*

1. Преобразуйте программу из лабораторной работы №8 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции f(x) как подпрограмму.
2. В листинге 9.3 приведена программа вычисления выражения (3 + 2) ∗ 4 + 5. При запуске данная программа дает неверный результат. Проверьте это. С помощью отладчика GDB, анализируя изменения значений регистров, определите ошибку и исправьте ее.

# 3 Теоретическое введение

1. **Понятие об отладке**  
   *Отладка* — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа:

* обнаружение ошибки;
* поиск её местонахождения;
* определение причины ошибки;
* исправление ошибки.
* Можно выделить следующие типы ошибок:
* *синтаксические ошибки* — обнаруживаются во время трансляции исходного кода и вызваны нарушением ожидаемой формы или структуры языка;
* *семантические ошибки* — являются логическими и приводят к тому, что программа запускается, отрабатывает, но не даёт желаемого результата;
* *ошибки в процессе выполнения* — не обнаруживаются при трансляции и вызывают прерывание выполнения программы (например, это ошибки, связанные с переполнением или делением на ноль).
* Второй этап — *поиск местонахождения ошибки*. Некоторые ошибки обнаружить довольно трудно. Лучший способ найти место в программе, где находится ошибка, это разбить программу на части и произвести их отладку отдельно друг от друга.
* Третий этап — *выяснение причины ошибки*. После определения местонахождения ошибки обычно проще определить причину неправильной работы программы.
* Последний этап — *исправление ошибки*. После этого при повторном запуске программы, может обнаружиться следующая ошибка, и процесс отладки начнётся заново.

1. **Методы отладки**  
   Наиболее часто применяют следующие методы отладки:  
   • создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных значений на экран — так называемые *диагностические сообщения*);

• использование специальных программ-отладчиков.

Отладчики позволяют управлять ходом выполнения программы, контролировать и изменять данные. Это помогает быстрее найти место ошибки в программе и ускорить её исправление. Наиболее популярные способы работы с отладчиком — это использование точек останова и выполнение программы по шагам.   
\*Пошаговое выполнение\* — это выполнение программы с остановкой после каждой строчки, чтобы программист мог проверить значения переменных и выполнить другие действия.   
\*Точки останова\* — это специально отмеченные места в программе, в которых 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.

1. **Основные возможности отладчика GDB**  
   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.  
   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.

1. **Запуск отладчика GDB; выполнение программы; выход**  
   Синтаксис команды для запуска отладчика имеет следующий вид:  
   gdb [опции] [имя\_файла | ID процесса]  
   После запуска gdb выводит текстовое сообщение — так называемое «nice GDB logo». В следующей строке появляется приглашение (gdb) для ввода команд.  
   Далее приведён список некоторых команд GDB.  
   Команда run (сокращённо r) — запускает отлаживаемую программу в оболочке GDB. Если точки останова не были установлены, то программа выполняется и выводятся сообщения:  
   (gdb) run  
   Starting program: test  
   Program exited normally.  
   (gdb)  
   Если точки останова были заданы, то отладчик останавливается на соответствующей команде и выдаёт номер точки останова, адрес и дополнительную информацию — текущую строку, имя процедуры, и др.  
   Команда kill (сокращённо k) прекращает отладку программы, после чего следует вопрос о прекращении процесса отладки:  
   Kill the program being debugged? (y or n) y

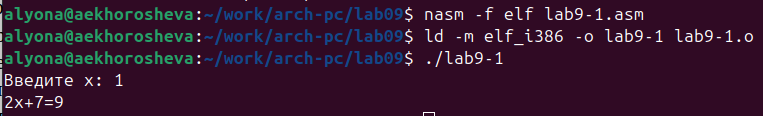
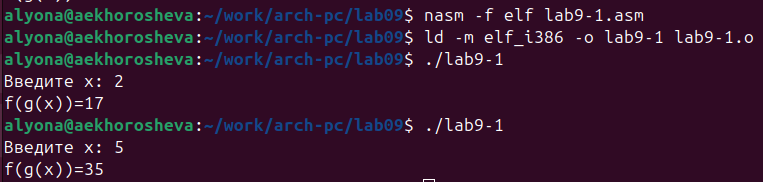
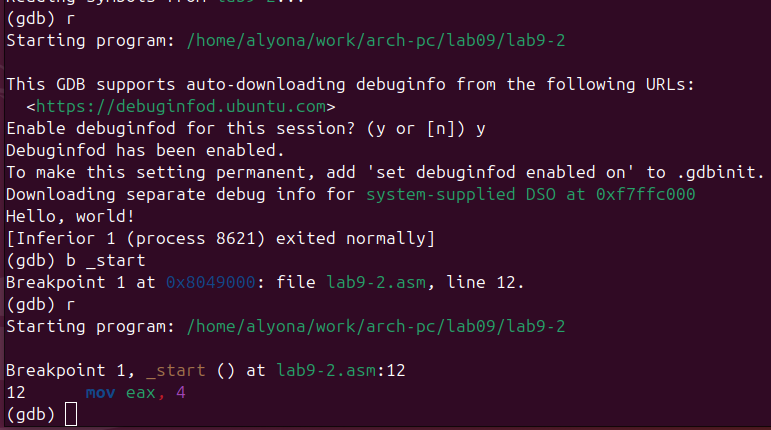
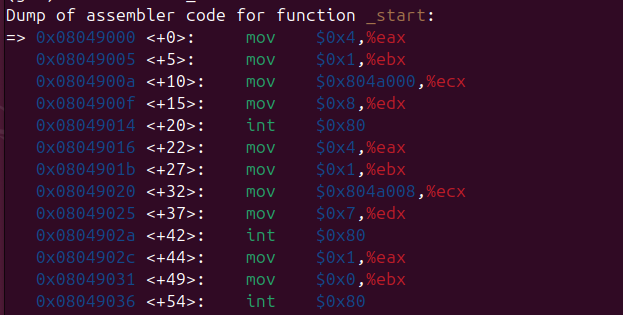
* Если в ответ введено y (то есть «да»), отладка программы прекращается. Командой run её можно начать заново, при этом все точки останова (breakpoints), точки просмотра (watchpoints) и точки отлова (catchpoints) сохраняются.  
  Для выхода из отладчика используется команда quit (или сокращённо q):  
  (gdb) q

1. **Дизассемблирование программы**  
   Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, то программу можно отлаживать, работая в отладчике непосредственно с её исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом -g.  
   Посмотреть дизассемблированный код программы можно с помощью команды disassemble :  
   (gdb) disassemble \_start  
   Существует два режима отображения синтаксиса машинных команд: режим Intel, используемый в том числе в NASM, и режим ATT (значительно отличающийся внешне). По умолчанию в дизассемблере GDB принят режим ATT. Переключиться на отображение команд с привычным Intel’овским синтаксисом можно, введя команду set disassembly-flavor intel.
2. **Точки останова**  
   Установить точку останова можно командой break (кратко b). Типичный аргумент этой команды — место установки. Его можно задать как имя метки или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка»:  
   (gdb) break \*  
   (gdb) b   
   Информацию о всех установленных точках останова можно вывести командой info (кратко i):  
   (gdb) info breakpoints  
   (gdb) i b  
   Для того чтобы сделать неактивной какую-нибудь ненужную точку останова, можно воспользоваться командой disable:  
   disable breakpoint   
   Обратно точка останова активируется командой enable:  
   enable breakpoint   
   Если же точка останова в дальнейшем больше не нужна, она может быть удалена с помощью команды delete:  
   (gdb) delete breakpoint   
   Ввод этой команды без аргумента удалит все точки останова.  
   Информацию о командах этого раздела можно получить, введя help breakpoints.
3. **Пошаговая отладка**  
   Для продолжения остановленной программы используется команда continue (c) (gdb) с [аргумент]. Выполнение программы будет происходить до следующей точки останова.  
   В качестве аргумента может использоваться целое число 𝑁, которое указывает отладчику проигнорировать 𝑁 − 1 точку останова (выполнение остановится на 𝑁-й точке).  
   Команда stepi (кратко sI) позволяет выполнять программу по шагам, т.е. данная команда выполняет ровно одну инструкцию:  
   (gdb) si [аргумент]  
   При указании в качестве аргумента целого числа 𝑁 отладчик выполнит команду step 𝑁 раз при условии, что не будет точек останова или выполнение программы не прервётся по другим причинам.  
   Команда nexti (или ni) аналогична stepi, но вызов процедуры (функции) трактуется отладчиком как одна инструкция:  
   (gdb) ni [аргумент]  
   Информацию о командах этого раздела можно получить, введя (gdb) help running.
4. **Работа с данными программы в GDB**  
   Как уже упоминалось, отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных.  
   Посмотреть содержимое регистров можно с помощью команды info registers (или ir):  
   (gdb) info registers  
   Для отображения содержимого памяти можно использовать команду x/NFU , выдаёт содержимое ячейки памяти по указанному адресу. NFU задает формат, в котором выводятся данные.  
   Например, x/4uh 0x63450 — это запрос на вывод четырёх полуслов (h) из памяти в формате беззнаковых десятичных целых (u), начиная с адреса 0x63450.  
   Чтобы посмотреть значения регистров используется команда print /F (сокращенно p). Перед именем регистра обязательно ставится префикс $. Например, команда p/x $ecx выводит значение регистра в шестнадцатеричном формате. Изменить значение для регистра или ячейки памяти можно с помощью команды set, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс $, а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си).  
   Справку о любой команде gdb можно получить, введя  
   (gdb) help [имя\_команды]
5. **Понятие подпрограммы**  
   *Подпрограмма* — это, как правило, функционально законченный участок кода, который можно многократно вызывать из разных мест программы. В отличие от простых переходов из подпрограмм существует возврат на команду, следующую за вызовом.

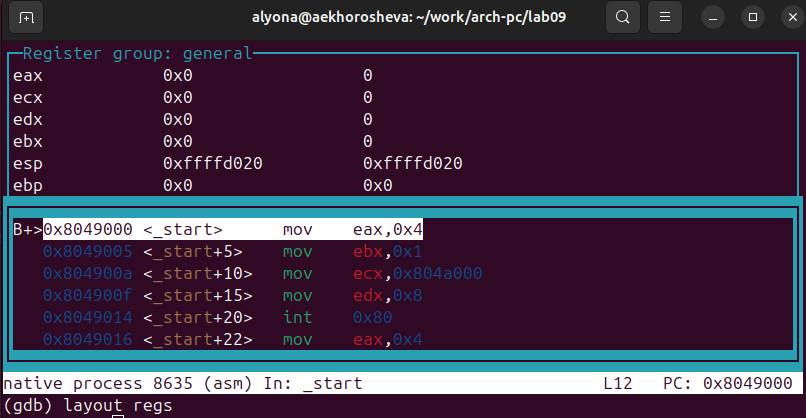
* Если в программе встречается одинаковый участок кода, его можно оформить в виде подпрограммы, а во всех нужных местах поставить её вызов. При этом подпрограмма будет содержаться в коде в одном экземпляре, что позволит уменьшить размер кода всей программы.

1. **Инструкция call и инструкция ret**  
   Для вызова подпрограммы из основной программы используется инструкция call, которая заносит адрес следующей инструкции в стек и загружает в регистр eip адрес соответствующей подпрограммы, осуществляя таким образом переход. Затем начинается выполнение подпрограммы, которая, в свою очередь, также может содержать подпрограммы.  
   Подпрограмма завершается инструкцией ret, которая извлекает из стека адрес, занесённый туда соответствующей инструкцией call, и заносит его в eip. После этого выполнение основной программы возобновится с инструкции, следующей за инструкцией call.  
   Подпрограмма может вызываться как из внешнего файла, так и быть частью основной программы.  
   Важно помнить, что если в подпрограмме занести что-то в стек и не извлечь, то на вершине стека окажется не адрес возврата и это приведёт к ошибке выхода из подпрограммы. Кроме того, надо помнить, что подпрограмма без команды возврата не вернётся в точку вызова, а будет выполнять следующий за подпрограммой код, как будто он является её продолжением.

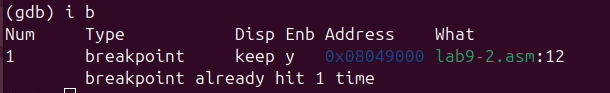
# 4 Выполнение лабораторной работы

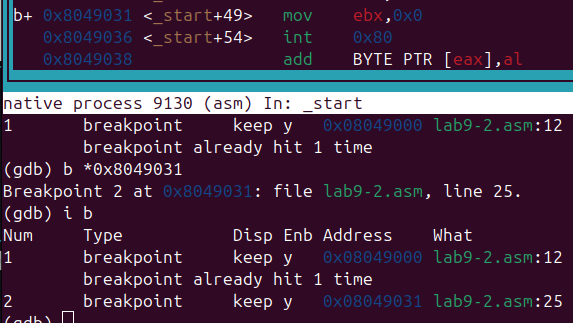
1. Создаём файл lab9-1.asm и записываем туда программу из листинга 9.1 для вычисления выражения f(x) = 2x+7 с помощью подпрограммы \_calcul.  
   Первые строки программы отвечают за вывод сообщения на экран (call sprint), чтение данных введенных с клавиатуры (call sread) и преобразования введенных данных из символьного вида в численный (call atoi).  
   После следующей инструкции call \_calcul, которая передает управление подпрограмме \_calcul, будут выполнены инструкции подпрограммы:  
   mov ebx,2  
   mul ebx  
   add eax,7  
   mov [res],eax  
   ret  
   Инструкция ret является последней в подпрограмме и ее исполнение приводит к возвращению в основную программу к инструкции, следующей за инструкцией call, которая вызвала данную подпрограмму.  
   Последние строки программы реализую вывод сообщения (call sprint), результата вычисления (call iprintLF) и завершение программы (call quit).  
   Проверяем работу программы:  
   
2. Теперь изменим текст программы, добавив в подпрограмму \_calcul вызов другой подпрограммы \_subcalcul, которая будет вычислять значение g(x) = 3x-1. Таким образом, нам нужно получить программу, которая вычислит значение f(g(x)) для x, введенного с клавиатуры.  
   ; Подпрограмма вычисления выражения g(x)=“3x-1”  
   \_subcalcul:  
   mov ebx,3  
   mul ebx  
   dec eax  
   ret  
   Вызов данной подпрограммы мы производим в самом начале \_calcul:  
   \_calcul:  
   call \_subcalcul  
   mov ebx,2  
   …  
   Запускаем исполняемый файл:  
   
3. Создадим новый файл lab09-2.asm с текстом из листинга 9.2 - это программа печати сообщения “Hello, world!”. Будем проверять работы этой программы с помощью отладчика *GDB*. Для этого при создании исполняемого файла и предшествующих этому инструкций, нужно использовать ключ “-g”.  
   nasm -f elf -g -l lab09-2.lst lab09-2.asm  
   ld -m elf\_i386 -o lab09-2 lab09-2.o  
   gdb lab09-2  
   Теперь можно запустить программу в gdb для проверки её работы командой run. Для более же подробного анализа работы программы поставим брейкпоинт на метку \_start, т.е. на начало выполнения программы. Запускаем всё той же командой run или r.  
   
4. Посмотрим *дизассемблерированный код* программы(т.е. *отображение инструкций, которые создал компилятор*) с помощью команды disassemble \_start(начинаем с метки \_start).  
     
   Теперь изменим отображение команд на синтаксис Intel с помощью команды set disassembly-flavor intel.  
   

Рассмотрим основные *различия отображения* синтаксиса машинных команд в режимах *ATT* и *Intel*:  
- расположение значения регистра и его названия: в режиме ATT сначала идёт значение, потом название регистра, а в Intel - наоборот;  
- в синтаксисе Intel нет префиксов перед регистром и его значением, а в ATT они есть: “$” перед значением, “%” перед названием регистра.

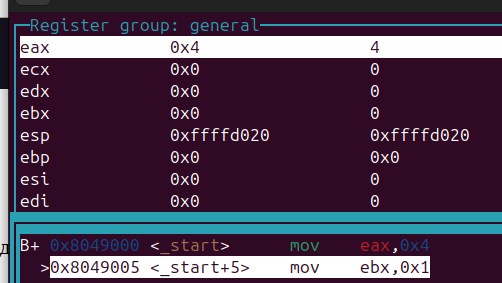
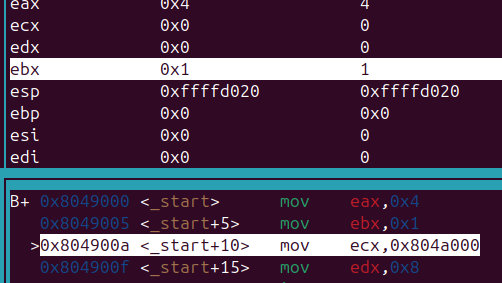
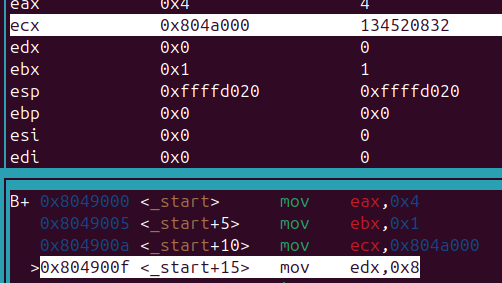
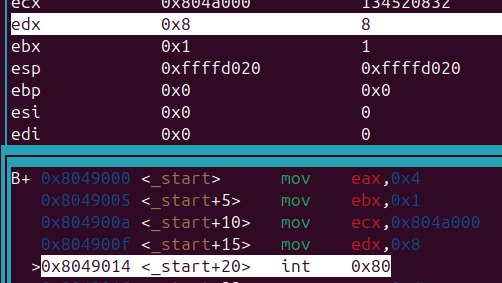
1. Далее включаем режим псевдографики - это позволит удобнее анализировать изменения в процессе работы программы. Для этого используем две программы:layout asm и layout regs.  
     
   Как видно на скриншоте, в режиме есть три окна(сверху вниз):

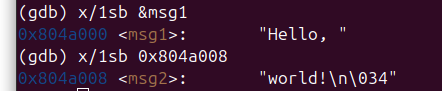
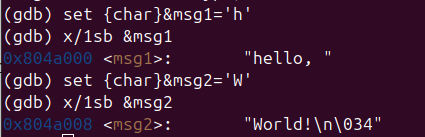
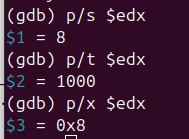
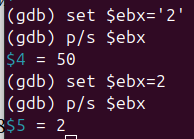
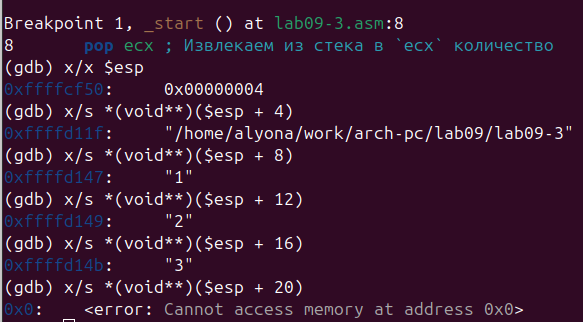
* названия регистров с их текущими значениями;
* результат дизассемблирования кода;
* поле для ввода команд.

1. Продолжим работу с lab9-2.asm. Командой i b (info breakpoints) посмотрим, где установлены точки останова:  
     
   Установим ещё одну такую точку командой break \*

* . Адрес инструкции можно узнать в средней части экрана в левом столбце. После установки новой точки останова проверяем информацию о всех точках командой i b.  
  

1. Выполним 5 инструкций программы, чтобы посмотреть на изменение значений регистров. Для этого используем команду stepi(si).

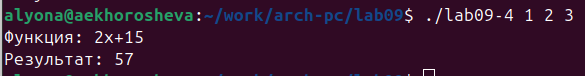
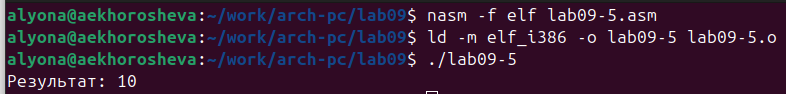
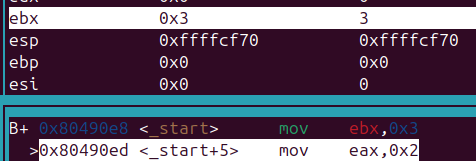
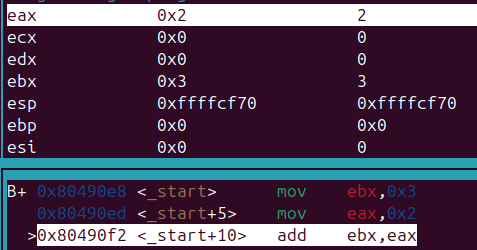
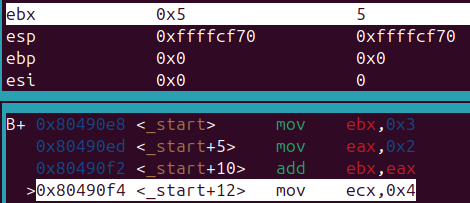
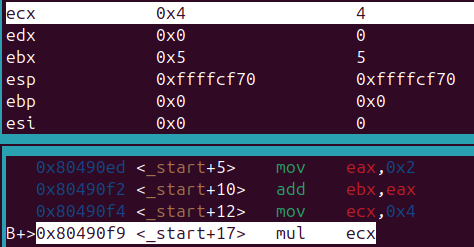
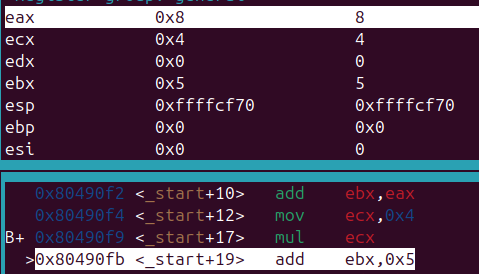
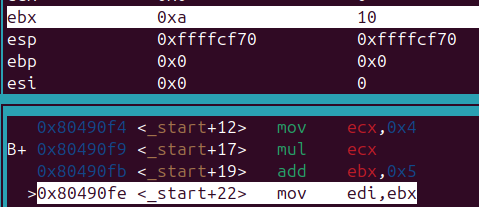
* В регистре eax находится значение 4. Тем временем в средней части экрана мы видим, что выполняется уже следующая инструкция, которая помещает значение 1 в регистр ebx:  
  
* Теперь мы наблюдаем результат предыдущей инструкции - в регистре ebx значение 1. Переход к следующей инструкции - перемещение значения с определённого адреса в регистр ecx:  
  
* Регистр ecx получил нужное значение с указанным ранее адресом. Следующая инструкция помещает 8 в регистр edx:  
  
* В edx уже находится 8, также начинается выполнение инструкции int:  
  
* Последнее выполнение - в eax значение 8, а следующая инструкция задаст знчаение 4 этому регистру:  
  

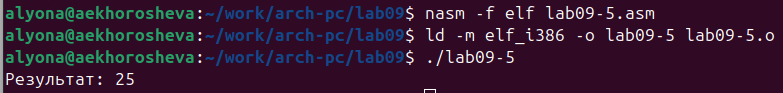
1. Теперь нужно отобразить содержимое памяти - для этого нужна команда x или x/NFU (чтобы задать формат вывода данных). Смотреть значение переменных можно как по имени, так и по адресу.  
   Для начала узнаем адрес переменной msg2. Используя дизассемблированный код находим инструкцию mov ecx,msg2 и определяем адрес:  
   “Адрес переменной msg2”  
   Можно тестировать команды: по имени переменной посмотрим значение msg1, а по адресу - msg2:  
   
2. Командой set можно изменять значения для ячейки памяти. Используем префикс & перед именем переменной, а в фигурных скобках указываем тип данных из языка Си:  
     
   Таким образом, мы изменили надпись “Hello, world!” на “hello, world!”.
3. Чтобы посмотреть значения регистров используется команда print /F $. Выведем в различных форматах значение edx.  
     
   Мы видим, что p/s - это вывод в символьном виде, а p/t и p/x - в двоичной и шестнадцатеричной системах счисления соответственно.
4. Поменяем значение ebx на ‘2’ и 2 по порядку. Посмотрим, как будет меняться вывод команды p/s(т.е. символьный вид вывода).  
     
   Наглядно заметно, что вывод в этих двух случаях отличается. Сначала мы ввели “2” в кавычках, поэтому компьютер воспринимает это как символ, а не само число 2. Символу “2” соотвествует число 50 в десятичной системе счисления согласно таблице символов ASCII (Приложение 2.Лабораторная работа №6).  
   Во втором случае мы ввели 2 именно как число, поэтому такой же результат и выводится командой p/s.
5. Копируем файл из лабораторной работы №8 с программой вывода на экран аргументов командной строки(листинг 8.2). Копию помещяем в файл lab09-3.asm и создаём исполняемый файл.  
   Затем загружаем полученный файл в отладчик с указанием аргументов. Я указала три аргумента: 1, 2, “3”. Устанавливаем точку останова - b \_start. Запускаем командой run.  
   Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы):  
     
   Вывод первой команды x/x - число 4. Это 3 аргумента + 1 имя программы.  
   Далее идут 5 команд x/s: из них выполняются первые 4 команды(расположение файла с программой и введённые ранее с клавиатуры аргументы), а последняя выводит ошибку, так как аргументы в регистре esp закончились.

* Шаг изменения адреса равен 4([esp+4], [esp+8], [esp+12] и т.д.), потому что размер операнда равен 4 байтам (целое число - int32).

# 5 Задание для самостоятельной работы

1. Создаём файл lab09-4.asm и копируем текст программы для вычисления значения функции f(x) для введённого с клавиатуры x(из лабораторной работы №8). Необходимо реализовать вычисление функции как подпрограмму:  
   \_calculateTheFunction:  
   mov ebx, 2  
   mul ebx  
   add esi, eax  
   add esi, 15 ; добавляем к промежуточной сумме  
   ret  
   Из изменений этого участка кода: добавление название подпрограммы и команда ret для её окончания. Также изменения коснулись цикла next: теперь мы вызываем функцию для вычисления выражения, а не считаем внутри цикла:  
   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 \_calculateTheFunction\* ; вызов фукнкции для вычисления 2х + 15  
   loop next ; переход к обработке следующего аргумента

Запускаем исполняемый файл для значений аргумента 1,2,3. Результатом должно стать число 57:  
  
Всё работает корректно.  
2. Создаём файл lab09-5.asm и пишем туда текст программы из листинга 9.3. Эта программа должна вычислять значение (3+2)\*4+5. Однако при запуске программа выдаёт неверный результат, а именно 10 вместо 25:  
  
С помощью отладчика gdb необходимо исправить ошибку в данной программе. Для этоого переходим в режим псевдографики для удобства анализа изменения регистров в процессе работы программы.  
Используем команду si(stepi) до тех пор, пока не выявим ошибку:  
- В регистре ebx находится значение 3. Число 2 передаётся в регистр eax.  
  
- Мы видим, что значение в eax теперь стало равно 2. Следующей выполняется инструкция для сложения регистров ebx и eax, при этом их сумма остаётся в регистре ebx.  
  
- В регистре ebx имеем значение 5, т.е. пока что программа работает верно. Следующая инструкция задает значение 4 регистру ecx.  
  
- Регистр ecx теперь равен 4. Выполняется инструкция по умножению регистра ecx. Как нам известно, результат умножения будет в регистре eax в данном случае. Значит и умножаются значения eax и ecx. Однако сумма, которая нам нужно умножить на 4, помещена в регистр ebx, а не eax(см. второе выполнение инструкции).  
  
- В регистре eax значение 8 - это ошибка. Операция умножения производилась лишь на одно из слагаемых, а не на необходимую сумму, поэтому результат получили неверный. Далее инструкция сложения прибавляет к ebx число 5. При этом мы не учитываем результат умножения, который при этом тоже работает некорректно.  
  
- В регистре ebx значение 10. Это не тот конечный результат, который нам нужен. Он помещается в регистр edi для дальнейшего вывода на экран.  


Итак, мы выявили ошибки в коде программы благодаря отладчику. Теперь приступим к изменениям текста программы.  
Во-первых, нужно сохранить сумму (3+2) в регистр eax, чтобы затем умножение происходило именно на неё.  
mov ebx,3  
mov eax,2  
add eax,ebx  
Во-вторых, прибавлять 5 в конце необходимо к регистру eax, так как именно там хранится результат умножения.  
mov ecx,4  
mul ecx  
add eax,5  
mov edi,eax  
Теперь создадим исполняемый файл и проверим работу программы.  
  
Получен корректный результат.  
# Выводы

В результате выполнения лабораторной работы получены теоретические знания о методах отладки, типах ошибок, понятии подпрограмм и их значимости.  
На практике были получены навыки работы с отладчиком GDB, а именно: нахождение ошибок в коде, установка точек останова, дизассемблизация кода для нахождения адресов ячеек памяти, обработка аргументов командной строки и т.д..  
В итоге, написана программа для нахождения значения f(g(x)), исправлена программа вычисления выражения (3+2)\*4+5.

# Список литературы