Российский Университет Дружбы Народов

Факультет физико-математических и естественных наук

Отчет по лабораторной работе №8

“Программирование цикла.

Обработка аргументов командной строки.”

Студентка: Богомолова Полина Петровна

Группа: НКАбд-01-25

Москва, 2025

Оглавление

Цель

Приобретение навыков написания программ с использованием циклов и обработкой аргументов командной строки.

Теоретическое введение

Организация стека

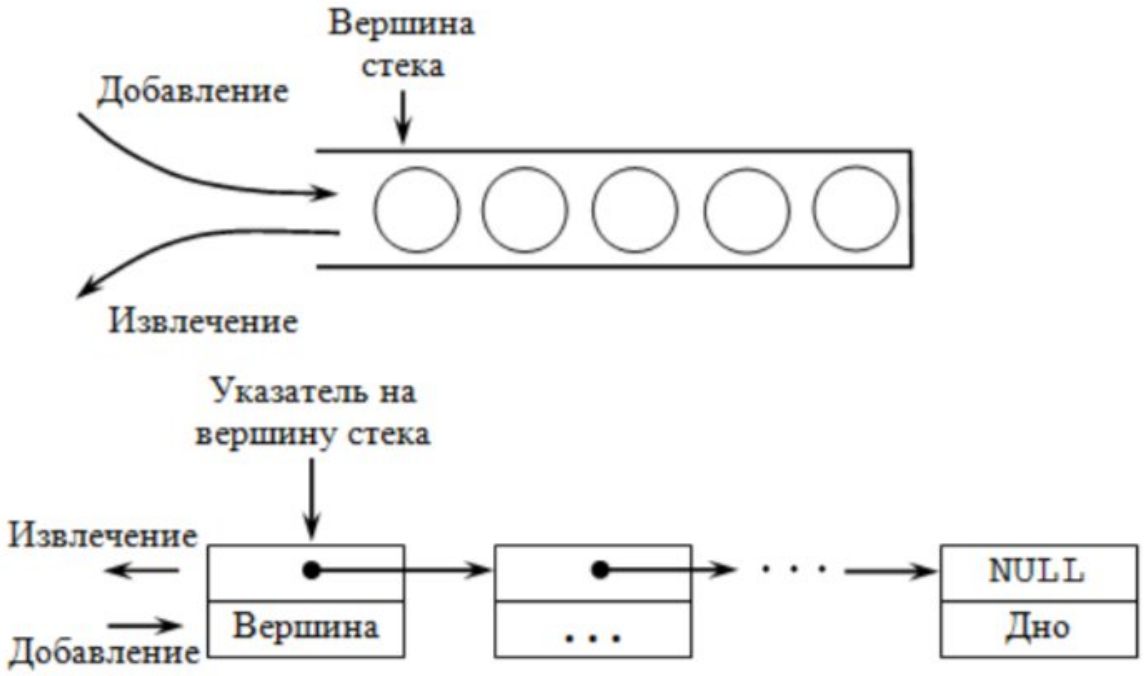
Стек — это структура данных, организованная по принципу LIFO («Last In — First Out» или «последним пришёл — первым ушёл»). Стек является частью архитектуры процессора и реализован на аппаратном уровне. Для работы со стеком в процессоре есть специальные регистры (ss, bp, sp) и команды. Основной функцией стека является функция сохранения адресов возврата и передачи аргументов при вызове процедур. Кроме того, в нём выделяется память для локальных переменных и могут временно храниться значения регистров. На рис. 8.1 показана схема организации стека в процессоре. Стек имеет вершину, адрес последнего добавленного элемента, который хранится в регистре esp (указатель стека). Противоположный конец стека называется дном. Значение, помещённое в стек последним, извлекается первым. При помещении значения в стек указатель стека уменьшается, а при извлечении — увеличивается. Для стека существует две основные операции:

• добавление элемента в вершину стека (push);

• извлечение элемента из вершины стека (pop).

Команда push размещает значение в стеке, т.е. помещает значение в ячейку памяти, на которую указывает регистр esp, после этого значение регистра esp увеличивается на 4. Данная команда имеет один операнд — значение, которое необходимо поместить в стек

Рис. 8.1



push -10 ; Поместить -10 в стек

push ebx ; Поместить значение регистра ebx в стек

push [buf] ; Поместить значение переменной buf в стек

push word [ax] ; Поместить в стек слово по адресу в ax

Существует ещё две команды для добавления значений в стек. Это команда pusha, которая помещает в стек содержимое всех регистров общего назначения в следующем порядке: ax, cx, dx, bx, sp, bp, si, di. А также команда pushf, которая служит для перемещения в стек содержимого регистра флагов. Обе эти команды не имеют операндов.

Команда pop извлекает значение из стека, т.е. извлекает значение из ячейки памяти, на которую указывает регистр esp, после этого уменьшает значение регистра esp на 4. У этой команды также один операнд, который может быть регистром или переменной в памяти. Нужно помнить, что извлечённый из стека элемент не стирается из памяти и остаётся как “мусор”, который будет перезаписан при записи нового значения в стек. Примеры:

pop eax ; Поместить значение из стека в регистр

```
eax pop [buf] ; Поместить значение из стека в buf

pop word[si] ; Поместить значение из стека в слово по адресу в si
```

Аналогично команде записи в стек существует команда popa, которая восстанавливает из стека все регистры общего назначения, и команда popf для перемещения значений из вершины стека в регистр флагов.

Инструкции организации циклов

Для организации циклов существуют специальные инструкции. Для всех инструкций максимальное количество проходов задаётся в регистре ecx. Наиболее простой является инструкция loop. Она позволяет организовать безусловный цикл, типичная структура которого имеет следующий вид:

```
mov ecx, 100 ; Количество проходов

NextStep:

...

 ... ; тело цикла

...

loop NextStep ; Повторить `ecx` раз от метки NextStep
```

Иструкция loop выполняется в два этапа. Сначала из регистра ecx вычитается единица и его значение сравнивается с нулём. Если регистр не равен нулю, то выполняется переход к указанной метке. Иначе переход не выполняется и управление передаётся команде, которая следует сразу после команды loop.

Выполнение лабораторной работы

1. Создадим каталог для программ лабораторной работы № 8, перейдем в него и создадим файл lab8-1.asm:

```
mkdir ~/work/arch-pc/lab08
cd ~/work/arch-pc/lab08
touch lab8-1.asm
```

Создадим копию файла in_out.asm в каталоге лабораторной работы номер 8 с помощью команды cp, перемещаемся между каталогами с помощью команды cd. С помощью команды ls убедимся в том, что файл in_out.asm был скопирован в каталог лабораторной работы номер 8.

В ходе выполнения лабораторной будем использовать Midnight Commander для работы с файлами, будем запускать его с помощью команды mc.

Результат представлен на рисунках 1-2.

Рис. 1

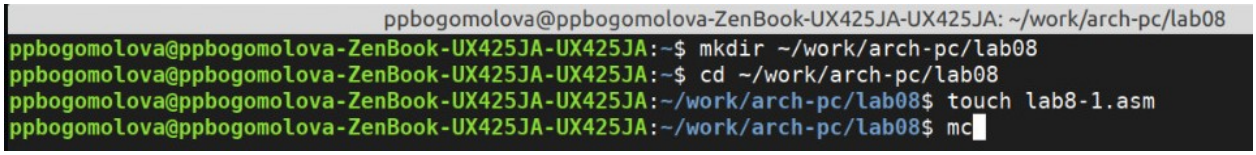
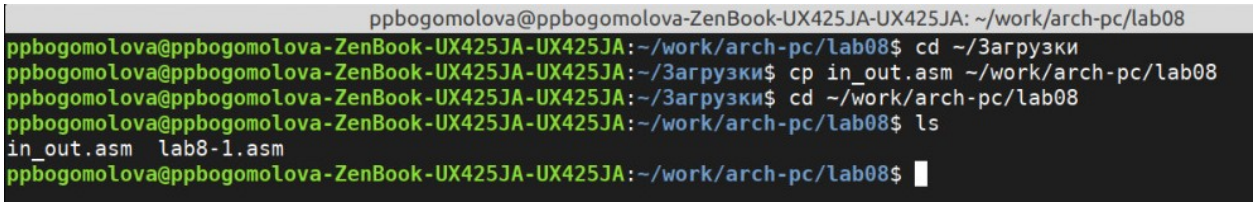


Рис. 2



При реализации циклов в NASM с использованием инструкции loop необходимо помнить о том, что эта инструкция использует регистр ecx в качестве счетчика и на каждом шаге уменьшает его значение на единицу. В качестве примера рассмотрим программу, которая выводит значение регистра ecx.

2. Введем в файл lab8-1.asm текст программы из листинга 8.1. Создадим исполняемый файл и проверим его работу. Для этого будем использовать

команды nasm –f elf, ld –m elf_i386, ./ . Данная программа запрашивает у пользователя число N, преобразовывает введенный символ в число и выводит числа от 1 до N построчно. Результат представлен на рисунках 3-4.

Рис. 3

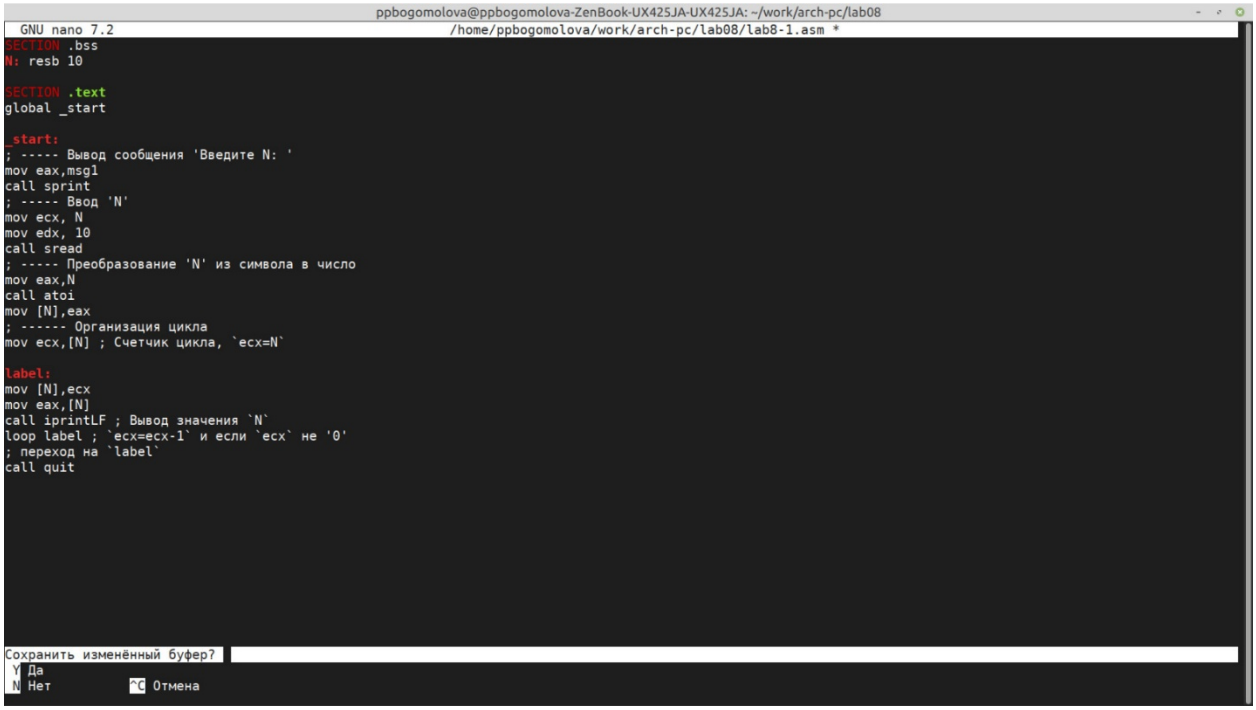
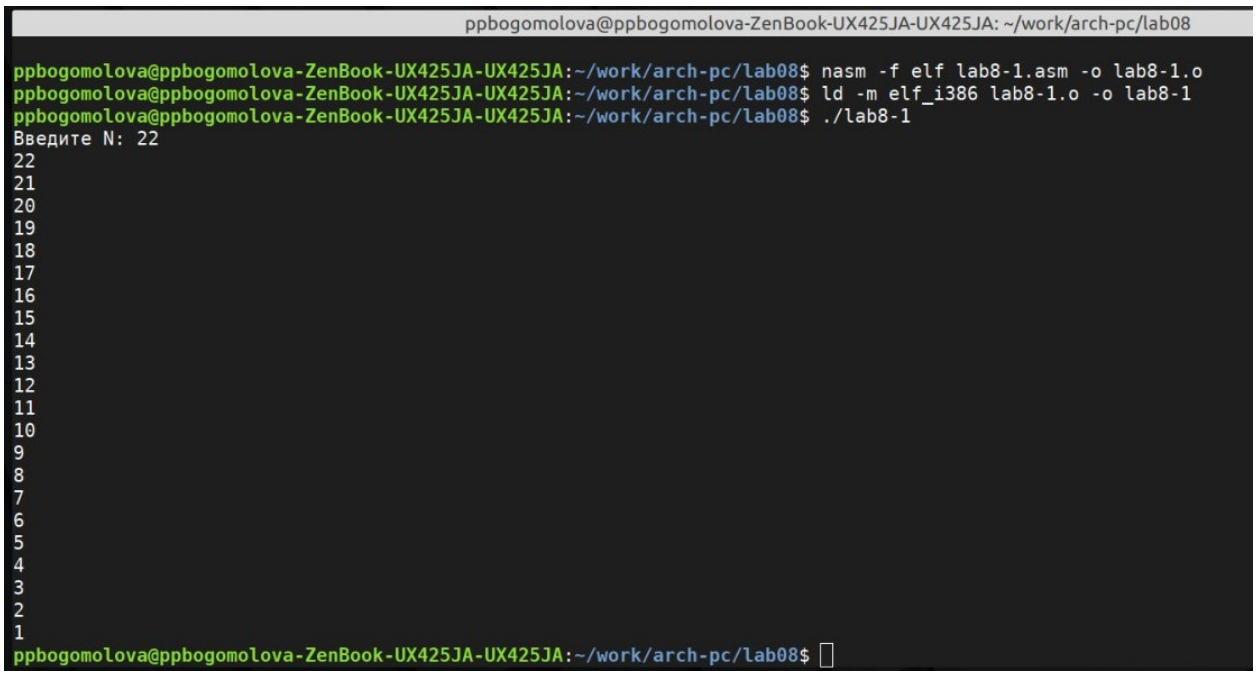


Рис. 4



3. Данный пример показывает, что использование регистра ecx в теле цилка loop может привести к некорректной работе программы. Изменим текст программы, добавив изменение значения регистра ecx в цикле:

```
label:
sub ecx,1 ; `ecx=ecx-1`
mov [N],ecx
mov eax,[N]
call iprintLF
loop label
```

Создадим исполняемый файл и проверим его работу. Для этого будем использовать команды nasm –f elf, ld –m elf_i386, ./ . Данная программа запрашивает N у пользователя, преобразовывает введенное значение в число, в каждой итерации цикла число (ecx) сначала уменьшается с помощью sub ecx, 1 , потом loop уменьшает число еще раз на 1. Результат представлен на рисунках 5-6.

- Какие значения принимает регистр ecx в цикле?

  В начале каждой итерации ecx содержит текущее значение счётчика цикла. В теле цикла ecx уменьшается дважды: сначала вручную командой sub ecx,1, а затем автоматически командой loop. Если N=5, регистр ecx будет последовательно принимать следующие значения перед sub ecx,1: 5, 3, 1, после sub ecx,1: 4, 2, 0, после loop (на следующей итерации): 3, 1, -1

- Соответствует ли число проходов цикла значению *N* введенному с клавиатуры?

  Нет, цикл выполняется меньше чем N раз, потому что ecx уменьшается дважды за проход цикла. Чтобы число проходов цикла соответствовало введенному N, можно убрать sub ecx,1.

Рис. 5

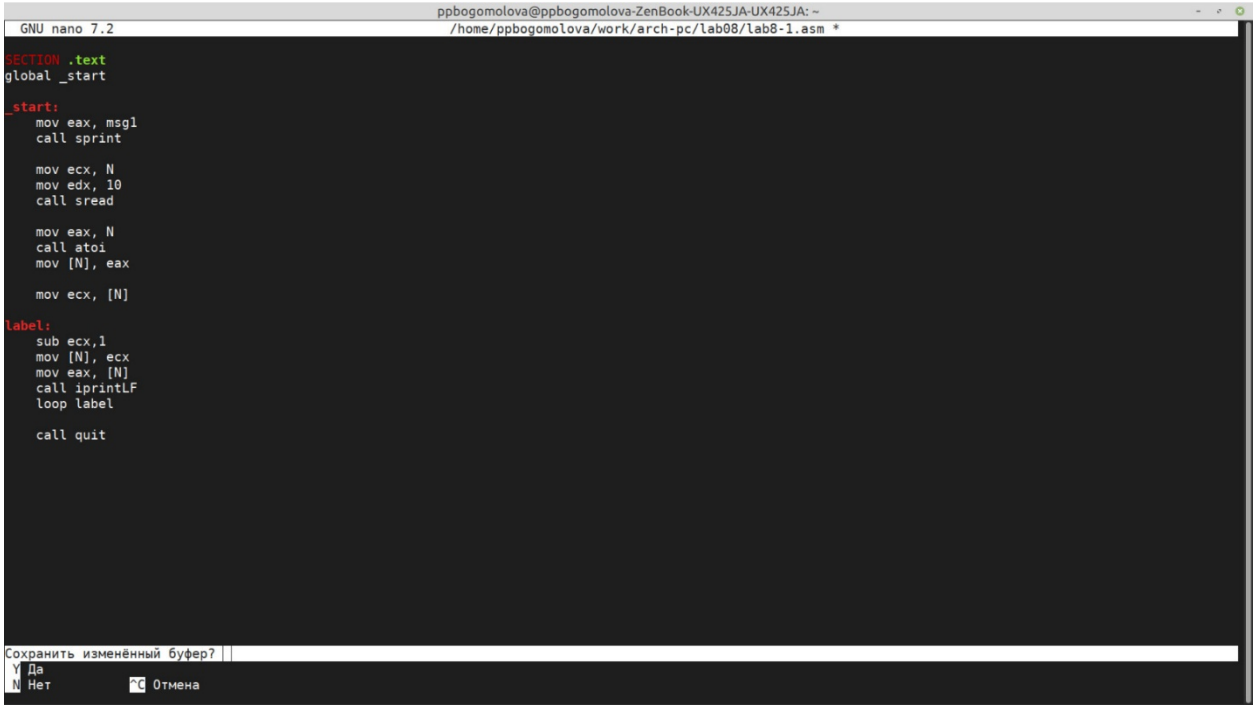
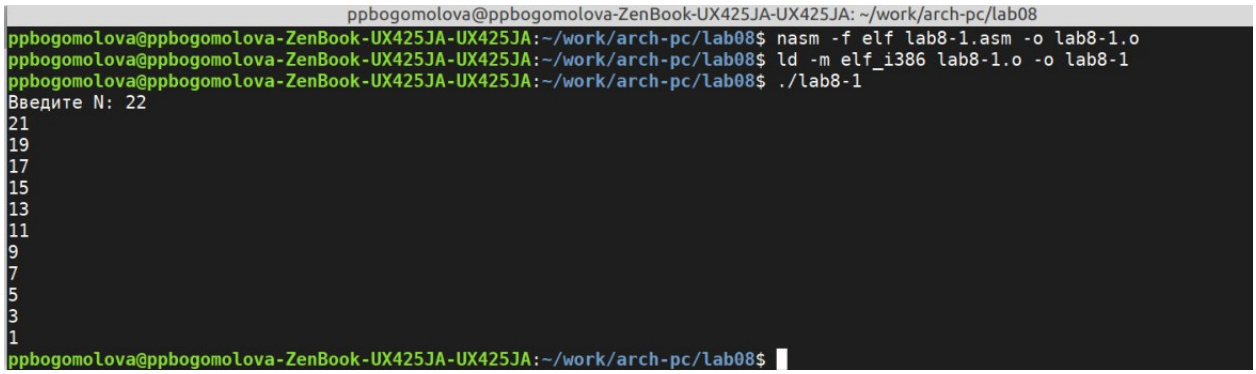


Рис. 6



4. Для использования регистра ecx в цикле и сохранения корректности работы программы можно использовать стек. Внесем изменения в текст программы, добавив команды push и pop (добавления в стек и извлечения из стека) для сохранения значения счетчика цикла loop:

```
label:
push ecx ; добавление значения ecx в стек
sub ecx,1
mov [N],ecx
mov eax,[N]
call iprintLF
```

pop ecx ; извлечение значения ecx из стека

loop label

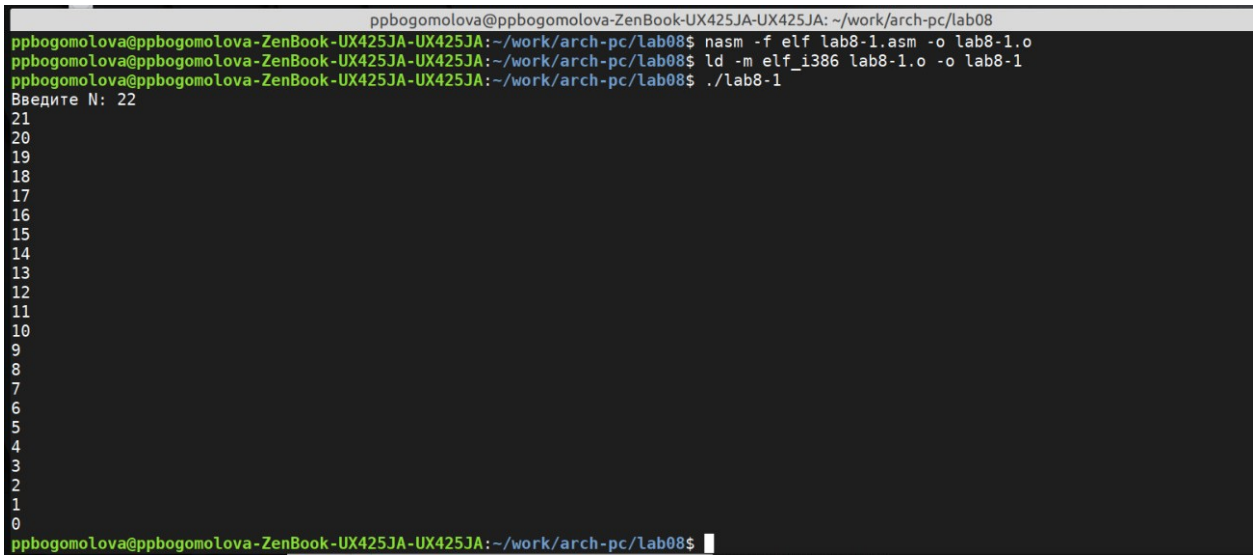
Создадим исполняемый файл и проверим его работу. Для этого будем использовать команды nasm –f elf, ld –m elf_i386, ./ . В данной программе ecx не уменьшается за проход, потому что мы восстановили его из стека перед loop, ecx уменьшается лишь единожды. На экран будут выводиться значения от N-1 до 0. Результат представлен на рисунках 7-8.

- Соответствует ли в данном случае число проходов цикла значению *N* введенному с клавиатуры?

Число проходов цикла в данном случае равно N.

Рис. 7

Рис. 8



5. При разработке программ иногда встает необходимость указывать аргументы, которые будут использоваться в программе, непосредственно из командной строки при запуске программы. При запуске программы в NASM аргументы командной строки загружаются в стек в обратном порядке, кроме того в стек записывается имя программы и общее количество аргументов. Последние два элемента стека для программы, скомпилированной NASM, – это всегда имя программы и количество переданных аргументов. Таким образом, для того чтобы использовать аргументы в программе, их просто нужно извлечь из стека. Обработку аргументов нужно проводить в цикле. Т.е. сначала нужно извлечь из стека количество аргументов, а затем циклично для каждого аргумента выполнить логику программы. В качестве примера рассмотрим программу, которая выводит на экран аргументы командной строки.

Создадим файл lab8-2.asm в каталоге ~/work/arch-pc/lab08 и введем в него текст программы из листинга 8.2. Для этого будем использовать команду touch. Создадим исполняемый файл и запустим его, указав аргументы:./lab8-2 аргумент1 аргумент 2 'аргумент 3'. Также будем использовать команды nasm –f elf, ld –m elf_i386. Результат представлен на рисунках 9-11.

- Сколько аргументов было обработано программой?

  Цикл был выполнен 3 раза, каждый из 3 аргументов был обработан и выведен на экран.

Рис. 9

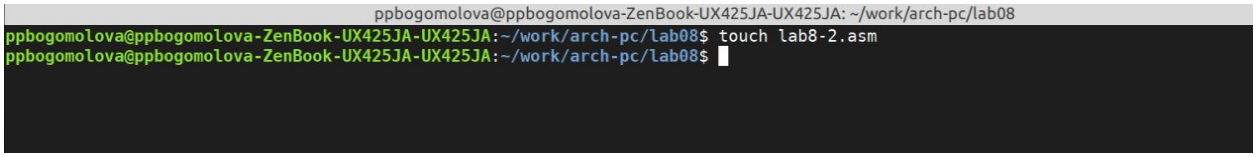


Рис. 10

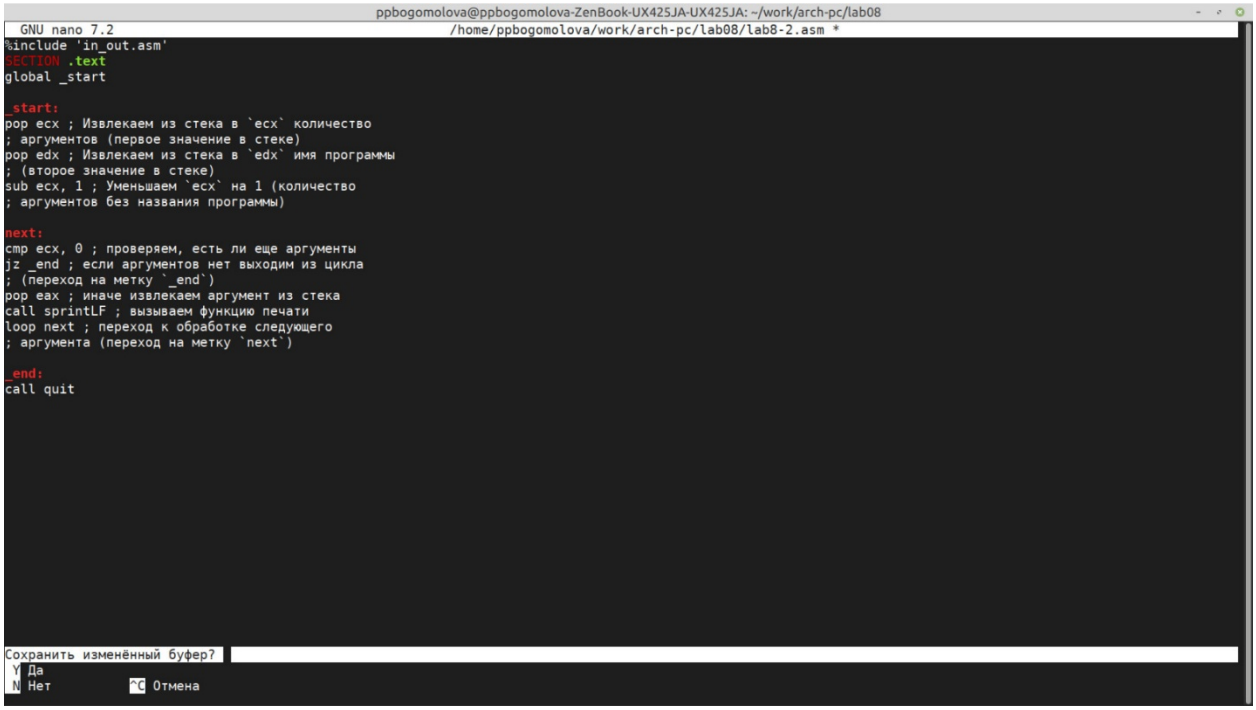
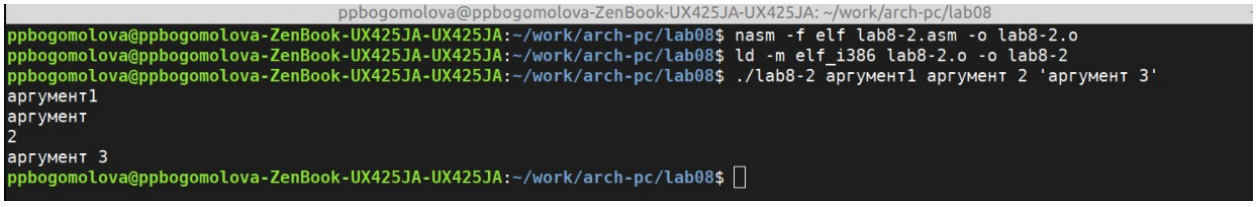


Рис. 11



6. Рассмотрим еще один пример программы которая выводит сумму чисел, которые передаются в программу как аргументы. Создадим файл lab8-3.asm в каталоге ~/work/archpc/lab08 и введем в него текст программы из листинга 8.3. Данная программа считывает и преобразовывает все аргументы командной строки в числа, суммирует все числа и выводит сообщение и сумму аргументов.

Создадим исполняемый файл и запустим его, указав аргументы, а также будем использовать команды nasm –f elf, ld –m elf_i386. Пример результата работы программы:

```
user@dk4n31:~$ ./main 12 13 7 10 5
Результат: 47
user@dk4n31:~$
```

Результат представлен на рисунках 12-14.

Рис. 12

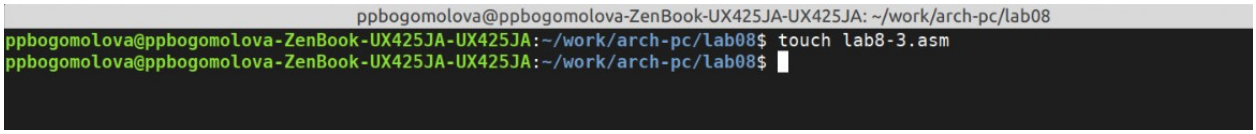


Рис. 13

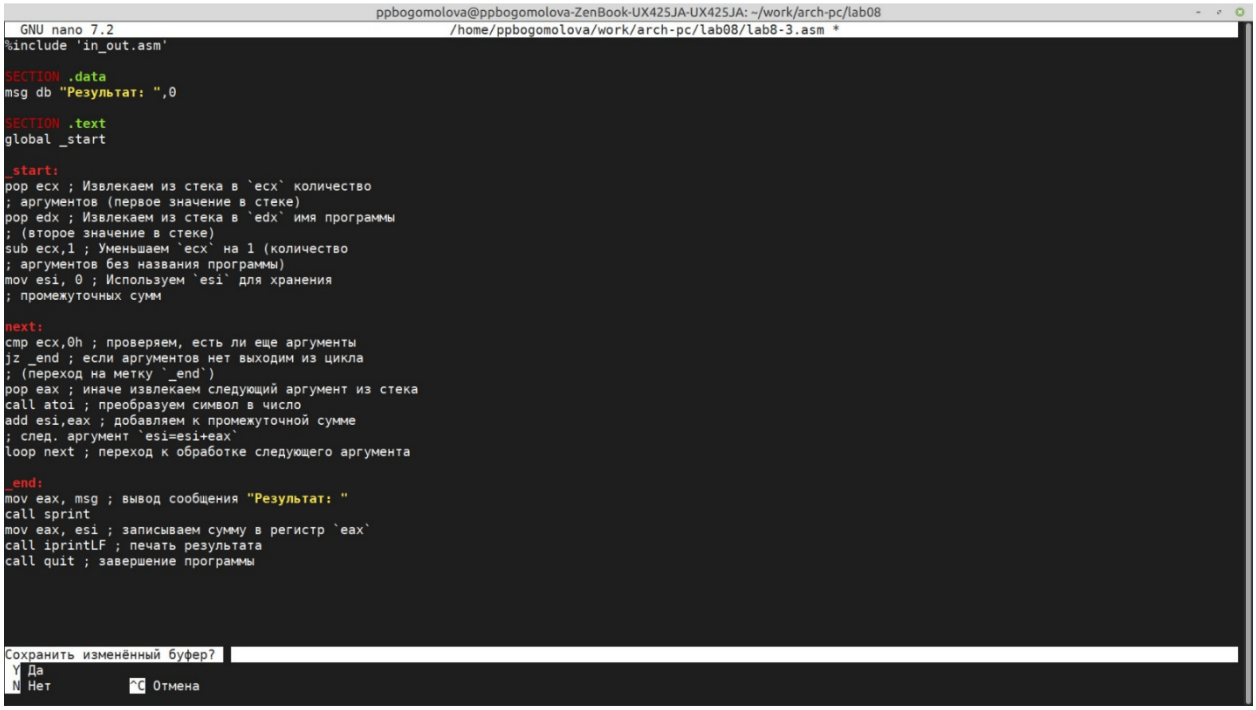
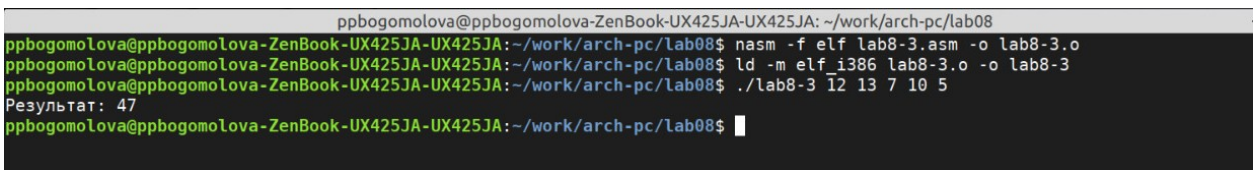


Рис. 14

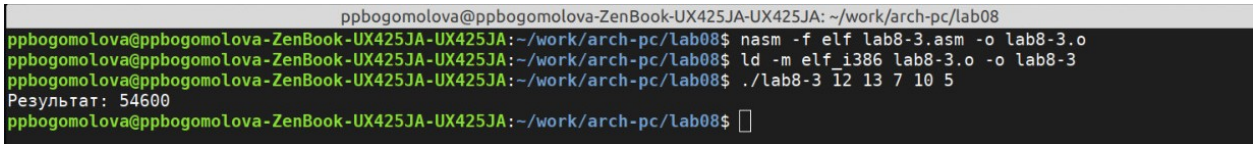


7. Изменим текст программы из листинга 8.3 для вычисления произведения аргументов. Данная программа считывает аргументы командной строки и

преобразовывает их в числа, вычисляет произведение чисел и выводит сообщение и итоговое число- результат произведения. Создадим исполняемый файл и запустим его, указав аргументы, а также будем использовать команды nasm –f elf, ld –m elf_i386. Результат представлен на рисунках 15-16

Рис. 15

Рис. 16



Задание для самостоятельной работы

1. Напишите программу, которая находит сумму значений функции $f(x)$ для $x = x1 , x2 , ..., xn$, т.е. программа должна выводить значение $f(x1) + f(x2) + ... + f(xn)$. Значения xi передаются как аргументы. Вид функции $f(x)$ выбрать из таблицы 8.1 вариантов заданий в соответствии с вариантом, полученным при выполнении лабораторной работы № 7. Создайте исполняемый файл и проверьте его работу на нескольких наборах $x = x1 , x2 , ..., xn$.

   Пример работы программы для функции $f(x) = x + 2$ и набора $x1 = 1$, $x2 = 2$, $x3 = 3$, $x4 = 4$:

```
user@dk4n31:~$ ./main 1 2 3 4
Функция: f(x)=x+2
Результат: 18
user@dk4n31:~$
```

Вид функции выбираем из таблицы в соответствии с вариантом, полученным при выполнении лабораторной работы номер 6. Данная программа считывает аргументы командной строки и преобразовывает их в числа, затем вычисляет значение функции $f(x)$=10*x-5 для каждого числа, а далее суммирует все результаты и выводит сообщение и итоговую сумму. Создадим файл lab8-4.asm с помощью команды touch. Создадим исполняемый файл и запустим его, указав аргументы, а также будем использовать команды nasm –f elf, ld –m elf_i386. Результат представлен на рисунках 17-19.

Таблица 8.1

Номер варианта	$f(x)$	Номер варианта	$f(x)$
1	$2x + 15$	**11**	$15x + 2$
2	$3x - 1$	**12**	$15x - 9$
3	$10x - 5$	**13**	$12x - 7$
4	$2(x - 1)$	**14**	$7(x + 1)$
5	$4x + 3$	**15**	$6x + 13$
6	$4x - 3$	**16**	$30x - 11$
7	$3(x + 2)$	**17**	$10(x - 1)$
8	$7 + 2x$	**18**	$17 + 5x$
9	$10x - 4$	**19**	$8x - 3$
10	$5(2 + x)$	**20**	$3(10 + x)$

Рис. 17

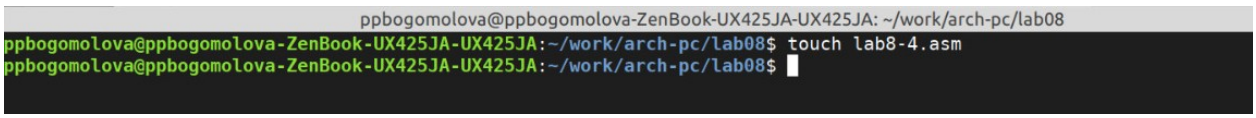


Рис. 18

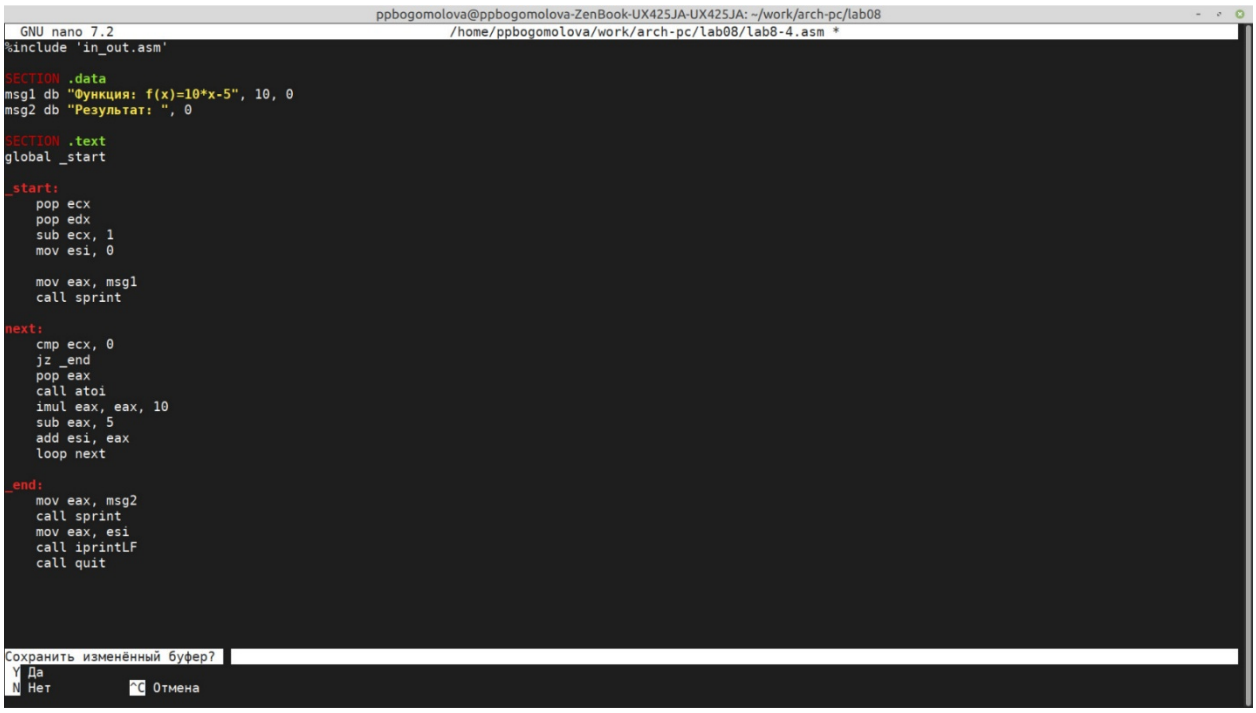
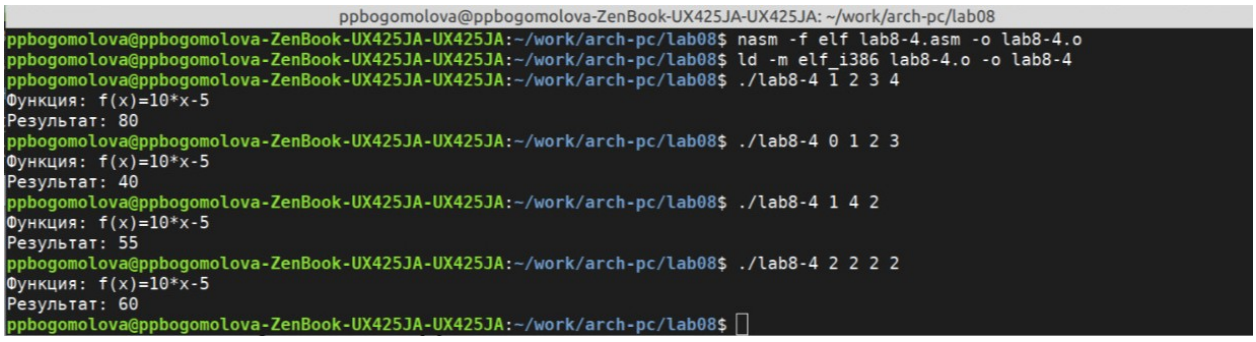


Рис. 19



Ссылка на мой репозиторий в GitHub:

https://github.com/bogomolova-pp/study_2025-2026_arh-pc

Листинги

Листинг 8.1

```
%include 'in_out.asm'

SECTION .data
msg1 db 'Введите N: ',0h

SECTION .bss
N: resb 10

SECTION .text
global _start

_start:
; ----- Вывод сообщения 'Введите N: '
mov eax,msg1
call sprint

; ----- Ввод 'N'
mov ecx, N
mov edx, 10
call sread

; ----- Преобразование 'N' из символа в число
mov eax,N
call atoi
mov [N],eax

; ------ Организация цикла
mov ecx,[N] ; Счетчик цикла, `ecx=N`

label:
mov [N],ecx
```

```
mov eax,[N]
call iprintLF ; Вывод значения `N`
loop label ; `ecx=ecx-1` и если `ecx` не '0'
; переход на `label`
call quit
```

Листинг 8.1.2

```
SECTION .bss

N: resb 10

SECTION .text
global _start
_start:

; ----- Вывод сообщения 'Введите N: '
mov eax,msg1
call sprint

; ----- Ввод 'N'

mov ecx, N
mov edx, 10
call sread

; ----- Преобразование 'N' из символа в число
mov eax,N
call atoi
mov [N],eax

; ------ Организация цикла
```

```
mov ecx,[N] ; Счетчик цикла, `ecx=N`
label:
sub ecx,1 ; `ecx=ecx-1`
mov [N],ecx
mov eax,[N]
call iprintLF
loop label
call quit
```

Листинг 8.1.3

```
SECTION .bss
N: resb 10

SECTION .text
global _start
_start:

; ----- Вывод сообщения 'Введите N: '
mov eax,msg1
call sprint

; ----- Ввод 'N'
mov ecx, N
mov edx, 10
call sread

; ----- Преобразование 'N' из символа в число
mov eax,N
call atoi
mov [N],eax

; ------ Организация цикла
mov ecx,[N] ; Счетчик цикла, `ecx=N`
```

```
label:
push ecx ; добавление значения ecx в стек
sub ecx,1
mov [N],ecx
mov eax,[N]
call iprintLF
pop ecx ; извлечение значения ecx из стека
loop label

call quit
```

Листинг 8.2

```
%include 'in_out.asm'

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)
next:
cmp ecx, 0 ; проверяем, есть ли еще аргументы
jz _end ; если аргументов нет выходим из цикла
; (переход на метку `_end`)
pop eax ; иначе извлекаем аргумент из стека
call sprintLF ; вызываем функцию печати
loop next ; переход к обработке следующего
; аргумента (переход на метку `next`)
_end:
call quit
```

Листинг 8.3

```
%include 'in_out.asm'

SECTION .data
msg db "Результат: ",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
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
```

Листинг 8.3.2

```
%include 'in_out.asm'

SECTION .data
msg db "Результат: ",0

SECTION .text
global _start

_start:
    pop ecx
    pop edx
    sub ecx, 1
    mov esi, 1

next:
    cmp ecx, 0
    jz _end
    pop eax
    call atoi
    imul esi, eax
    loop next

_end:
    mov eax, msg
    call sprint
    mov eax, esi
    call iprintLF
    call quit
```

Листинг 8.4

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

```
SECTION .data
msg1 db "Функция: f(x)=10*x-5", 10, 0
msg2 db "Результат: ", 0

SECTION .text
global _start

_start:
    pop ecx
    pop edx
    sub ecx, 1
    mov esi, 0

    mov eax, msg1
    call sprint

next:
    cmp ecx, 0
    jz _end
    pop eax
    call atoi
    imul eax, eax, 10
    sub eax, 5
    add esi, eax
    loop next

_end:
    mov eax, msg2
    call sprint
    mov eax, esi
    call iprintLF
    call quit
```

Вывод

В результате выполнения лабораторной работы номер 8 я приобрела навыки написания программ с использованием циклов и с обработкой аргументов командной строки.

Список литературы

1. Демидова А.В-Лабораторная работа №8. Программирование цикла. Обработка аргументов командной строки.