Лабораторная работа №8

Программирование цикла. Обработка аргументов командной строки.

Петрова Алевтина Александровна

Содержание

# 1 Цель работы

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

# 2 Задание

1. Реализация циклов в NASM
2. Обработка аргументов командной строки
3. Задание для самостоятельной работы

# 3 Теоретическое введение

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.

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 указа- тель стека уменьшается, а при извлечении — увеличивается. Для стека существует две основные операции: • добавление элемента в вершину стека (push); • извлечение элемента из вершины стека (pop).

# 4 Выполнение лабораторной работы

## 4.1 Реализация циклов в NASM

Создаю директорию,в которой буду выполнять лабораторную работу (рис. [1](#fig:001)).

Figure 1: Создание каталога

Figure 1: Создание каталога

Перехожу в созданный каталог (рис. [2](#fig:002)).

Figure 2: Перемещение по директории

Figure 2: Перемещение по директории

Создаю файл lab8.asm (рис. [3](#fig:003)). В нём буду делать первое задание.

Figure 3: Создание файла

Figure 3: Создание файла

Также копирую в каталог файл in\_out.asm (рис. [4](#fig:004)). Он понадобится для написания будущих программ.

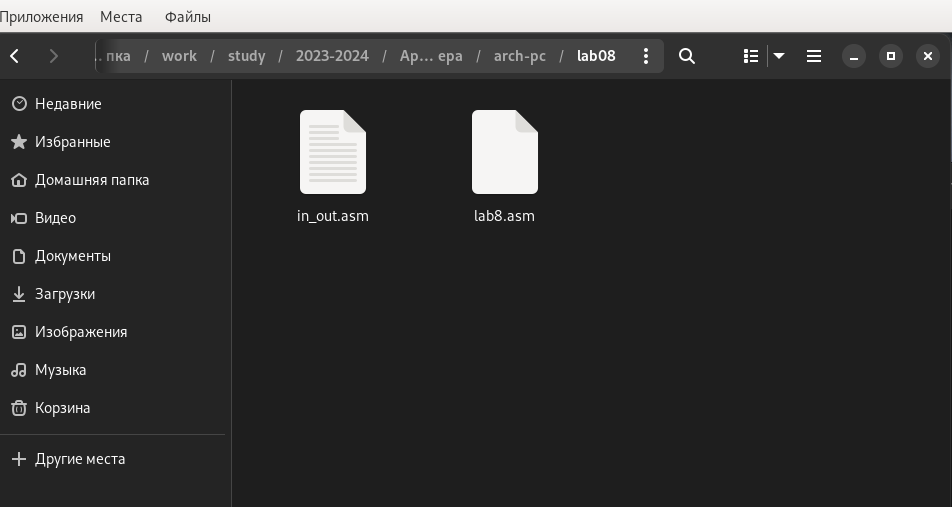


Figure 4: Копирование файла in\_out.asm

Записываю текст кода из листинга 8.1 (рис. [5](#fig:005)). Эта программа запрашивает число N, и выдает все числа перед N вместе с ним до 0.

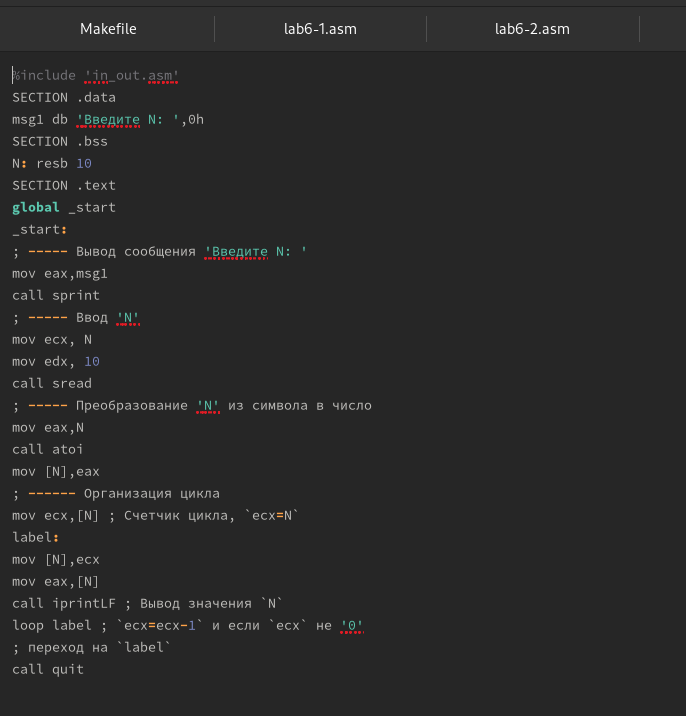


Figure 5: Редактирование программы

Создаю исполняемый код (рис. [6](#fig:006)).После его запуска убеждаюсь,что программа работает успешно.

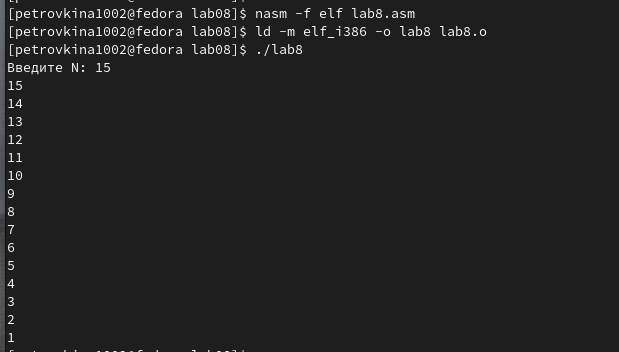


Figure 6: Запуск программы

Теперь я редактирую код,добавив изменение значение регистра ecx в цикле (рис. [7](#fig:007)).

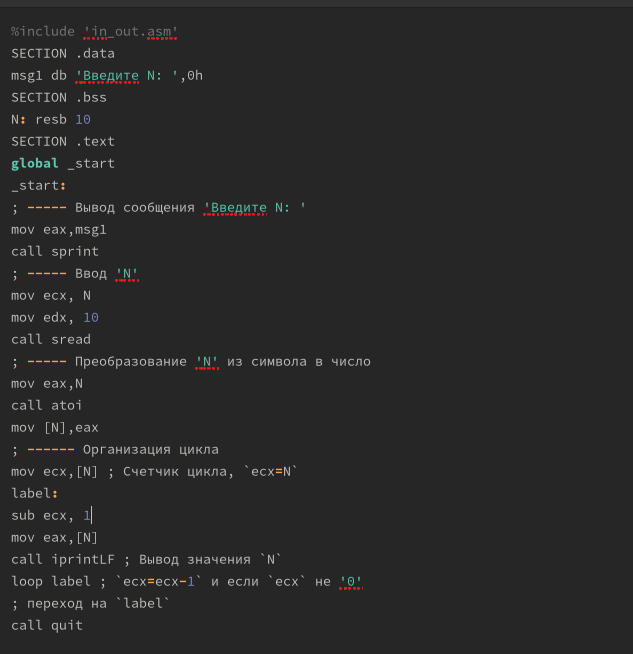


Figure 7: Редактирование программы

Запускаю программу. Теперь код зацикливается и начинает бесконечно передавать последовательные значения (рис. [8](#fig:008)).

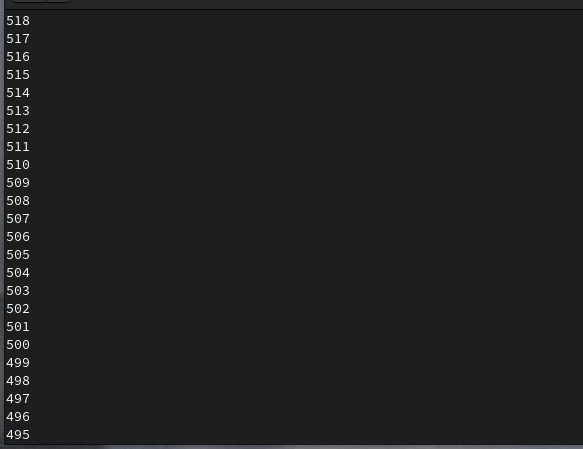


Figure 8: Запуск программы

Еще раз редактирую код программы,добавив команды push и pop (добавления в стек и извлечения из стека) для сохранения значения счетчика цикла loop (рис. [9](#fig:009)).

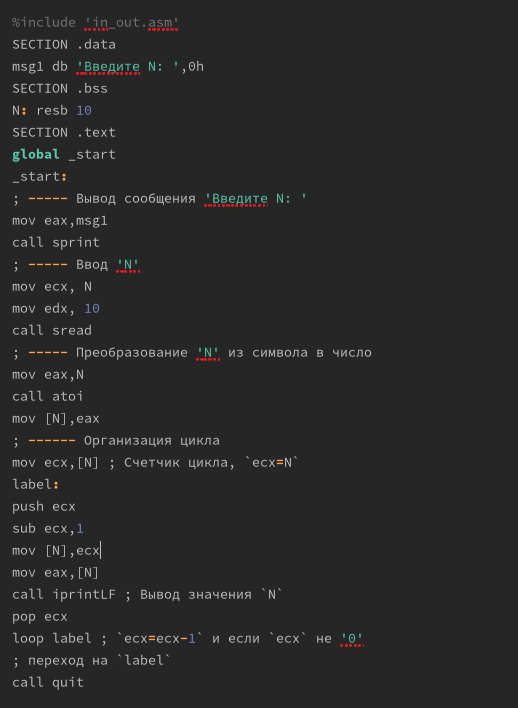


Figure 9: Редактирование программы

Создаю и запускаю исполняемый файл (рис. [10](#fig:010)). Теперь программа показывает все предыдущие числа до 0,не включая заданное N

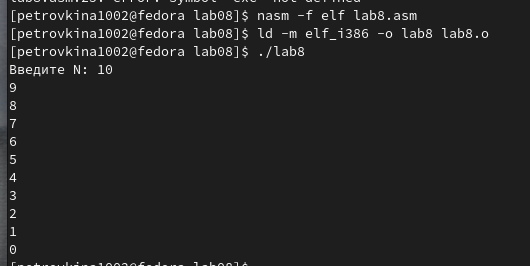


Figure 10: Запуск программы

## 4.2 Обработка аргументов командной строки

Создаю новый файл lab8-2.asm, используя команду touch (рис. [11](#fig:011)).

Figure 11: Создание файла

Figure 11: Создание файла

Открываю файл в GNU nano и записываю код из листинга 8.2 (рис. [12](#fig:012)). Данная программа позволяет выводить на экран аргументы командной строки.

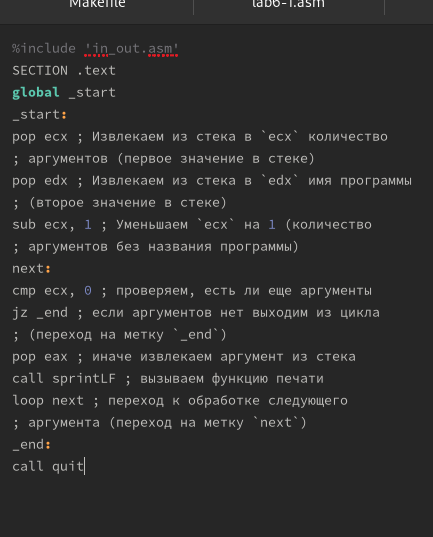


Figure 12: Редактирование программы

Запускаю исполняемый файл вместе с аргументами (аргумент1, аргумент2, ‘аргумент3’) (рис. [13](#fig:013)).

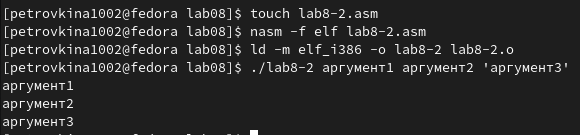


Figure 13: Запуск программы

Создаю новый файл lab8-3.asm, используя команду touch (рис. [14](#fig:014)).

Figure 14: Создание файла

Figure 14: Создание файла

Открываю файл в GNU nano и записываю код из листинга 8.3 (рис. [15](#fig:015)). Данная программа позволяет выводить на экран сумму аргументов командной строки.

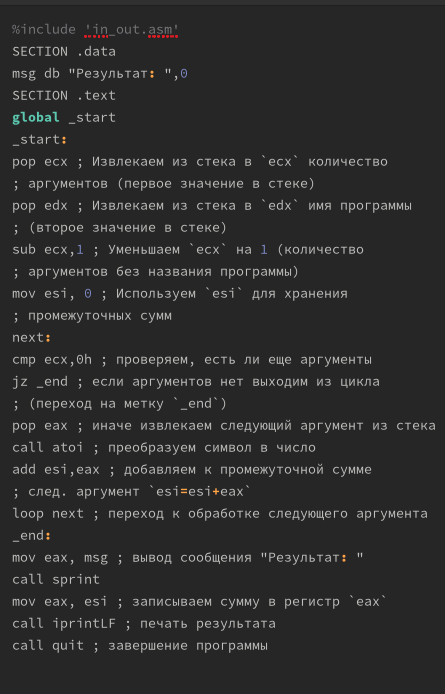


Figure 15: Редактирование программы

Запускаю исполняемый файл вместе с аргументами (12,13,7,10,5) (рис. [16](#fig:016)). Программа действительно выдаёт сумму всех аргументов.

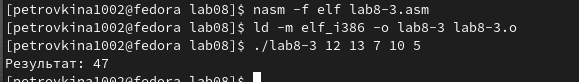


Figure 16: Запуск программы

Теперь редактирую код программы так,чтобы она выводила произведение всех аргументов (рис. [17](#fig:017)).

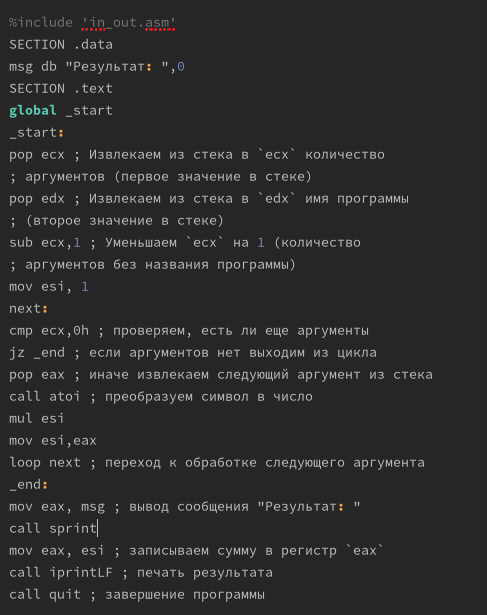


Figure 17: Редактирование программы

%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, 1   
next:  
cmp ecx,0h ; проверяем, есть ли еще аргументы  
jz \_end ; если аргументов нет выходим из цикла  
pop eax ; иначе извлекаем следующий аргумент из стека  
call atoi ; преобразуем символ в числ  
mul esi  
mov 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

Запускаю исполняемый файл вместе с аргументами (1,3,4,7) (рис. [18](#fig:018)). Программа выдаёт произведение всех аргументов.

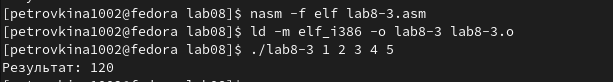


Figure 18: Запуск программы

## 4.3 Задание для самостоятельной работы

Создаю файл lab8-4.asm в котором буду писать код для последней задачи (рис. [19](#fig:019)).

Figure 19: Создание файла

Figure 19: Создание файла

Пишу код программы,который позволяет вывести сумму всех преобразованных аргументов. Преобразования я беру из варианта задания №8 (7+2x) (рис. [20](#fig:020)).

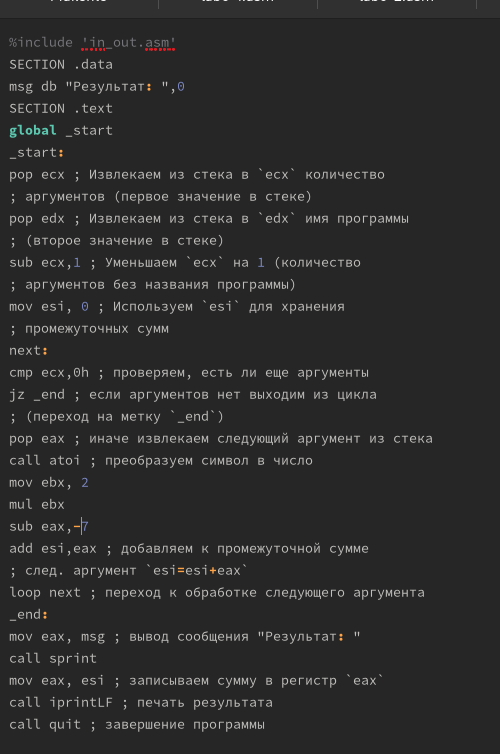


Figure 20: Редактирование программы

%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  
mov ebx,2  
mul ebx  
sub eax,-7  
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

Запускаю исполняемый файл вместе с аргументами (1,2,3,4) (рис. [21](#fig:021)). Программа выдаёт верную сумму всех преобразованных аргументов.

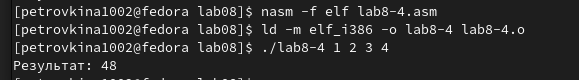


Figure 21: Запуск программы

Повторно запускаю программу,чтобы убедиться, что всё работает верно (рис. [22](#fig:022)). Программа выдает верный ответ.

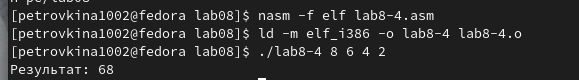


Figure 22: Повторный запуск программы

# 5 Выводы

В данной лабораторной работать я научился работать с циклами, выводом аргументов и функций.

# Список литературы

1. Лабораторная работа №8