

# **Отчёт по лабораторной работе №7**

**Дисциплина: Архитектура Компьютера**

Азарцова Вероника Валерьевна

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>5</b>
<b>2</b>	<b>Задание</b>	<b>6</b>
<b>3</b>	<b>Теоретическое введение</b>	<b>7</b>
3.1	Команды безусловного перехода . . . . .	7
3.2	Команды условного перехода . . . . .	8
3.2.1	Регистр флагов . . . . .	8
3.2.2	Описание инструкции <code>cmpr</code> . . . . .	9
3.2.3	Описание команд условного перехода . . . . .	9
3.3	Файл листинга и его структура . . . . .	10
<b>4</b>	<b>Выполнение лабораторной работы</b>	<b>12</b>
4.1	Реализация переходов в NASM . . . . .	12
4.2	Изучение структуры файлы листинга . . . . .	17
<b>5</b>	<b>Задание для самостоятельной работы</b>	<b>22</b>
<b>6</b>	<b>Выводы</b>	<b>30</b>
	<b>Список литературы</b>	<b>31</b>

# Список иллюстраций

4.1	Создания каталога лабораторной работы и lab7-1.asm . . . . .	12
4.2	Текст программы с использованием jmp в lab7-1.asm . . . . .	13
4.3	Запуск lab7-1 . . . . .	13
4.4	Измененный текст программы lab7-1.asm . . . . .	14
4.5	Запуск измененного lab7-1 . . . . .	14
4.6	Текст снова измененного lab7-1 . . . . .	15
4.7	Запуск снова измененного lab7-1 . . . . .	15
4.8	Создание lab7-2.asm . . . . .	15
4.9	Текст программы в lab7-2.asm . . . . .	16
4.10	Запуск lab7-2 . . . . .	16
4.11	Создание lab7-2.lst . . . . .	17
4.12	lab2-2.lst в mcedit . . . . .	18
4.13	Измененный текст программы в lab2-1.asm . . . . .	20
4.14	Трансляция с получением файла листинга . . . . .	20
4.15	Ошибка в файле листинга . . . . .	21
5.1	Создание lab7-3.asm . . . . .	22
5.2	Текст программы в lab7-3.asm . . . . .	23
5.3	Запуск lab7-3 . . . . .	25
5.4	Создание lab7-3.asm . . . . .	26
5.5	Создание lab7-3.asm . . . . .	26
5.6	Создание lab7-3.asm . . . . .	29

# Список таблиц

3.1	Флаги состояния	8
3.2	Флаги состояния	10

# 1 Цель работы

Цель данной лабораторной работы - изучение команд условного и безусловного переходов, приобретение навыков написания программ с использованием переходов и знакомство с назначением и структурой файла листинга.

## **2 Задание**

1. Ознакомление с теоретическим введением
2. Выполнение лабораторной работы
3. Выполнение заданий для самостоятельной работы

## 3 Теоретическое введение

Для реализации ветвлений в ассемблере используются так называемые команды передачи управления или команды перехода. Можно выделить 2 типа переходов: \* условный переход – выполнение или не выполнение перехода в определенную точку программы в зависимости от проверки условия. \* безусловный переход – выполнение передачи управления в определенную точку программы без каких-либо условий.

### 3.1 Команды безусловного перехода

Безусловный переход выполняется инструкцией `jmp (jump)`, которая включает в себя адрес перехода, куда следует передать управление:

```
jmp <адрес_перехода>
```

Например:

```
jmp label
```

```
jmp [label]
```

```
jmp eax
```

(Переход на метку `label`, переход по адресу в памяти, помеченному меткой `label`, переход по адресу из регистра `eax`)

## 3.2 Команды условного перехода

Для условного перехода необходима проверка какого-либо условия. В ассемблере команды условного перехода анализируют флаги из регистра флагов.

### 3.2.1 Регистр флагов

Флаг – это бит, принимающий значение 1 («флаг установлен»), если выполнено некоторое условие, и значение 0 («флаг сброшен») в противном случае. Флаги лишь для удобства помещены в единый регистр – регистр флагов, отражающий текущее состояние процессора. Флаги состояния отражают результат выполнения арифметических инструкций (табл. 3.1).

Таблица 3.1: Флаги состояния

Имя		
каталога	Название	Описание каталога
CA	Carry Flag	Устанавливается в 1, если при выполнении предыдущей операции произошёл перенос из старшего бита или если требуется заём (при вычитании). Иначе установлен в 0.
PF	Parity Flag	Устанавливается в 1, если младший байт результата предыдущей операции содержит чётное количество битов, равных 1.
Af	Auxiliary Carry Flag	Устанавливается в 1, если в результате предыдущей операции произошёл перенос (или заём) из третьего бита в четвёртый.
ZF	Zero Flag	Устанавливается 1, если результат предыдущей команды равен 0.



Имя каталога	Название	Описание каталога
SF	Sign Flag	Равен значению старшего значащего бита результата, который является знаковым битом в знаковой арифметике.
OF	Overflow Flag	Устанавливается в 1, если целочисленный результат слишком длинный для размещения в целевом операнде (регистре или ячейке памяти).

### 3.2.2 Описание инструкции `cmp`

Инструкция `cmp` является одной из инструкций, которая позволяет сравнить операнды и выставляет флаги в зависимости от результата сравнения:

`cmp <операнд_1>, <операнд_2>`

Единственным результатом команды сравнения является формирование флагов.

### 3.2.3 Описание команд условного перехода

Команда условного перехода имеет вид

`j<мнемоника перехода> label`

Мнемоника перехода связана со значением анализируемых флагов или со способом формирования этих флагов. В табл. 3.2 представлены команды условного перехода, которые обычно ставятся после команды сравнения `cmp`.

Таблица 3.2: Флаги состояния

Типы операндов	Мнемокод	Критерий условного перехода а			Значение флагов	Комментарий
		v b				
Любые	JE	$a = b$			ZF=1	Переход если равно
Любые	JNE	$a \neq b$			ZF=0	Переход если не равно
Со знаком	JL/JNGE	$a < b$			SF≠OF	Переход если меньше
Со знаком	JLE/JNG	$a \leq b$			SF≠OF или ZF=1	Переход если меньше или равно
Со знаком	JG/JNLE	$a > b$			SF=OF и ZF=0	Переход если больше
Со знаком	JGE/JNL	$a \geq b$			SF=OF	Переход если больше или равно
Без знака	JB/JNAE	$a < b$			CF=1	Переход если ниже
Без знака	JBE/JNA	$a \leq b$			CF=1 или ZF=1	Переход если ниже или равно
Без знака	JA/JNBE	$a > b$			CF=0 и ZF=0	Переход если выше
Без знака	JAЕ/JNB	$a \geq b$			CF = 0	Переход если выше или равно

### 3.3 Файл листинга и его структура

Листинг — это один из выходных файлов, создаваемых транслятором. Он имеет текстовый вид и нужен при отладке программы, так как кроме строк самой программы он содержит дополнительную информацию.

Фрагмент файла листинга:

```
10 00000000 B804000000 mov eax,4
11 00000005 BB01000000 mov ebx,1
12 0000000A B9[00000000] mov ecx,hello
13 0000000F BA0D000000 mov edx,helloLen
14
15 00000014 CD80 int 80h
```

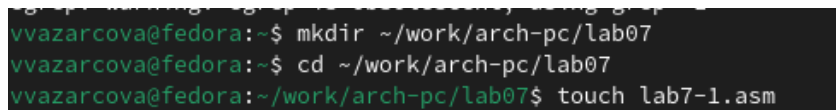
Строки первой части листинга имеют следующую структуру:

1. Номер строки — это номер строки файла листинга (не обязательно соответствует строкам в тексте программы);
2. Адрес — это смещение машинного кода от начала текущего сегмента;
3. Машинный код представляет собой ассемблированную исходную строку в виде шестнадцатеричной последовательности.
4. Исходный текст программы — строка исходной программы вместе с комментариями.

## 4 Выполнение лабораторной работы

### 4.1 Реализация переходов в NASM

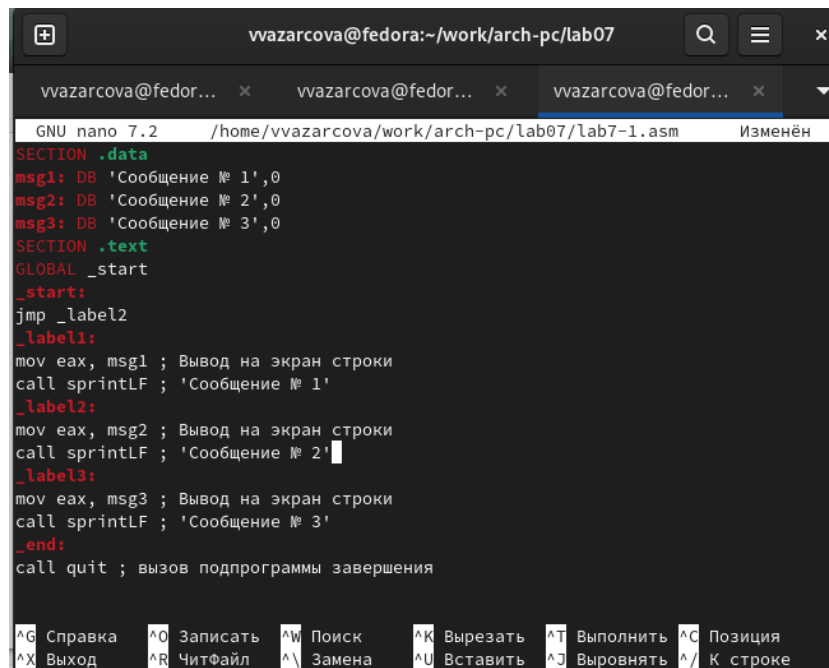
1. Создаю каталог для программ лабораторной работы № 7, перехожу в него и создаю файл lab7-1.asm (рис. 4.1).



```
vvazarcova@fedora:~$ mkdir ~/work/arch-pc/lab07  
vvazarcova@fedora:~$ cd ~/work/arch-pc/lab07  
vvazarcova@fedora:~/work/arch-pc/lab07$ touch lab7-1.asm
```

Рис. 4.1: Создания каталога лабораторной работы и lab7-1.asm

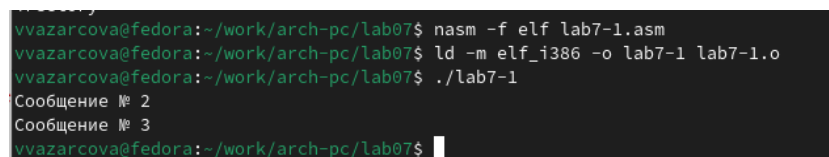
2. Открываю файл lab7-1.asm с помощью NASM и ввожу текст программы с использованием инструкции jmp (рис. 4.2).



```
GNU nano 7.2 /home/vvazarcova/work/arch-pc/lab07/lab7-1.asm
SECTION .data
msg1: DB 'Сообщение № 1',0
msg2: DB 'Сообщение № 2',0
msg3: DB 'Сообщение № 3',0
SECTION .text
GLOBAL _start
_start:
jmp _label2
_label1:
mov eax, msg1 ; Вывод на экран строки
call sprintf ; 'Сообщение № 1'
_label2:
mov eax, msg2 ; Вывод на экран строки
call sprintf ; 'Сообщение № 2'
_label3:
mov eax, msg3 ; Вывод на экран строки
call sprintf ; 'Сообщение № 3'
_end:
call quit ; вызов подпрограммы завершения
```

Рис. 4.2: Текст программы с использованием jmp в lab7-1.asm

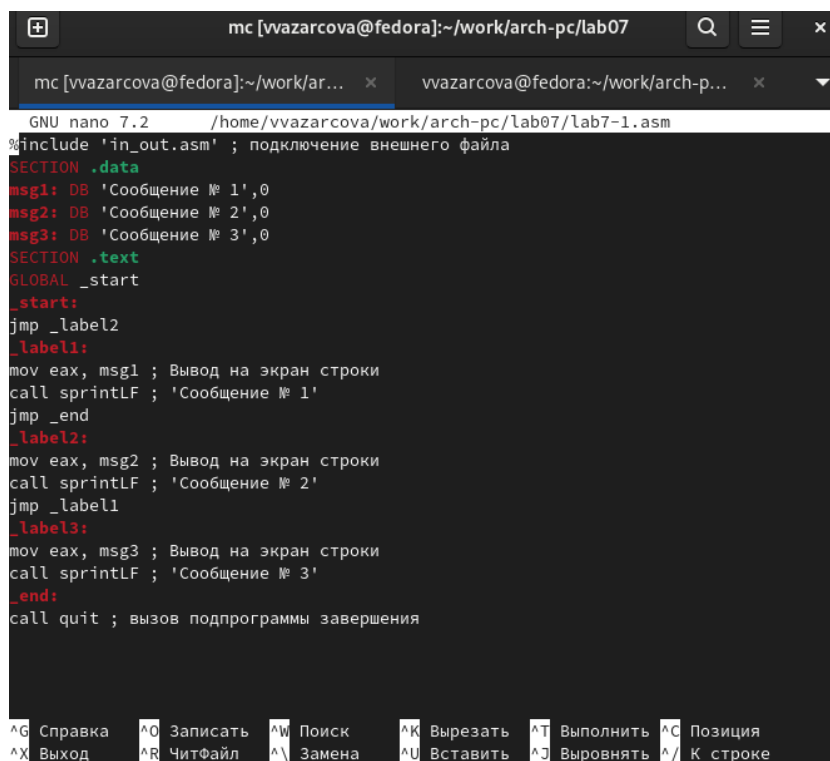
Создаю исполняемый файл и запускаю его (рис. 4.3).



```
vvazarcova@fedora:~/work/arch-pc/lab07$ nasm -f elf lab7-1.asm
vvazarcova@fedora:~/work/arch-pc/lab07$ ld -m elf_i386 -o lab7-1 lab7-1.o
vvazarcova@fedora:~/work/arch-pc/lab07$ ./lab7-1
Сообщение № 2
Сообщение № 3
vvazarcova@fedora:~/work/arch-pc/lab07$
```

Рис. 4.3: Запуск lab7-1

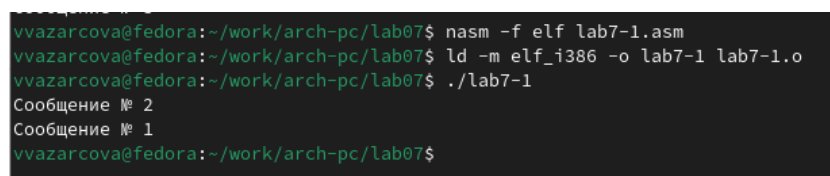
Результат выполнения программы соответствует ожиданиям. Использование инструкции jmp меняет порядок исполнения инструкций, следовательно, программа сразу переходит к выводу сообщения два и три, не выводя первое. Изменяю программу таким образом, чтобы она выводила сначала ‘Сообщение №2’, потом ‘Сообщение №1’ и завершала работу (рис. 4.4).



```
mc [vvazarcova@fedora]:~/work/arch-pc/lab07
GNU nano 7.2 /home/vvazarcova/work/arch-pc/lab07/lab7-1.asm
%include 'in_out.asm' ; подключение внешнего файла
SECTION .data
msg1: DB 'Сообщение № 1',0
msg2: DB 'Сообщение № 2',0
msg3: DB 'Сообщение № 3',0
SECTION .text
GLOBAL _start
_start:
jmp _label2
_label1:
mov eax, msg1 ; Вывод на экран строки
call sprintf ; 'Сообщение № 1'
jmp _end
_label2:
mov eax, msg2 ; Вывод на экран строки
call sprintf ; 'Сообщение № 2'
jmp _label1
_label3:
mov eax, msg3 ; Вывод на экран строки
call sprintf ; 'Сообщение № 3'
_end:
call quit ; вызов подпрограммы завершения
```

Рис. 4.4: Измененный текст программы lab7-1.asm

Создаю исполняемый файл и запускаю его (рис. 4.5).

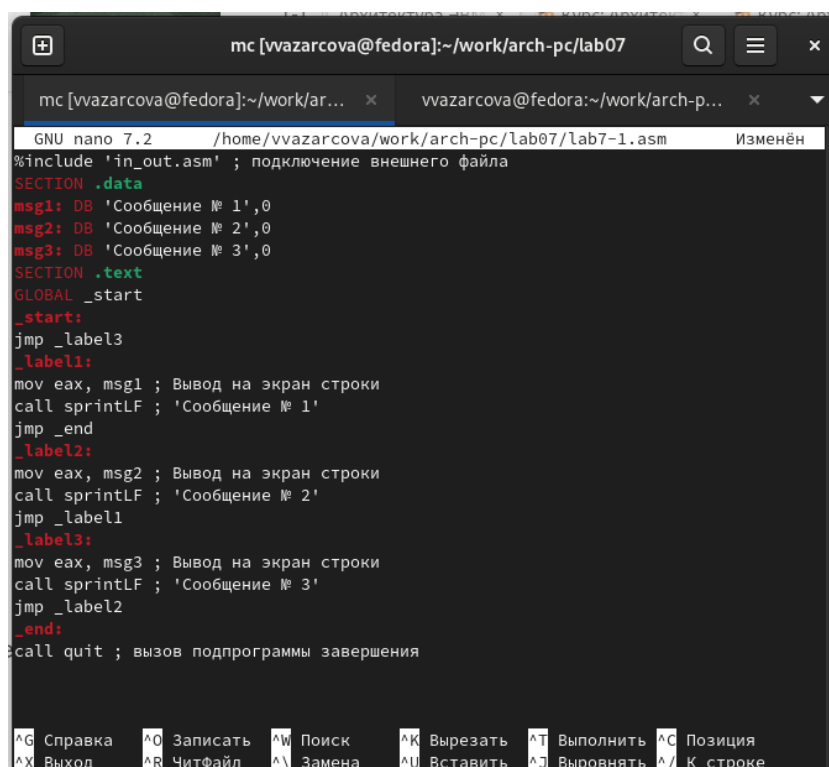


```
vvazarcova@fedora:~/work/arch-pc/lab07$ nasm -f elf lab7-1.asm
vvazarcova@fedora:~/work/arch-pc/lab07$ ld -m elf_i386 -o lab7-1 lab7-1.o
vvazarcova@fedora:~/work/arch-pc/lab07$ ./lab7-1
Сообщение № 2
Сообщение № 1
vvazarcova@fedora:~/work/arch-pc/lab07$
```

Рис. 4.5: Запуск измененного lab7-1

Результат выполнения программы соответствует ожиданиям: программа сначала выводит второе сообщение, затем первое, после чего завершает работу. Изменяю текст программы так, чтобы программа выводила сообщения в обратном порядке, т.е. сначала выводила третье сообщение, затем второе, затем первое. Для этого поменяю `jmp _label2` в самом начале программы на `jmp _label3` и добавлю `jmp _label2` после вывода третьего сообщения. Т.к. после второго сообщения и так уже выводится первое, а после первого уже и так вызывается завершение программы, программа будет выводить третье, второе, а затем первое сообщение

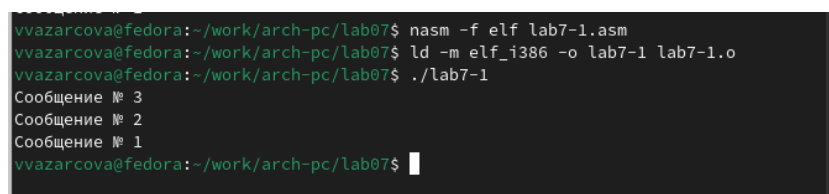
(рис. 4.6).



```
GNU nano 7.2 /home/vvazarcova/work/arch-pc/lab07/lab7-1.asm
%include 'in_out.asm' ; подключение внешнего файла
SECTION .data
msg1: DB 'Сообщение № 1',0
msg2: DB 'Сообщение № 2',0
msg3: DB 'Сообщение № 3',0
SECTION .text
GLOBAL _start
_start:
jmp _label3
_label1:
mov eax, msg1 ; Вывод на экран строки
call sprintf ; 'Сообщение № 1'
jmp _end
_label2:
mov eax, msg2 ; Вывод на экран строки
call sprintf ; 'Сообщение № 2'
jmp _label1
_label3:
mov eax, msg3 ; Вывод на экран строки
call sprintf ; 'Сообщение № 3'
jmp _label2
_end:
call quit ; вызов подпрограммы завершения
```

Рис. 4.6: Текст снова измененного lab7-1

Создаю исполняемый файл и проверяю работу программы (рис. 4.7).

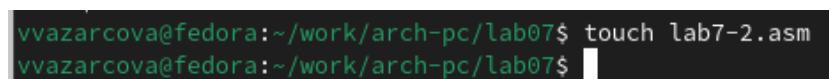


```
vvazarcova@fedora:~/work/arch-pc/lab07$ nasm -f elf lab7-1.asm
vvazarcova@fedora:~/work/arch-pc/lab07$ ld -m elf_i386 -o lab7-1 lab7-1.o
vvazarcova@fedora:~/work/arch-pc/lab07$ ./lab7-1
Сообщение № 3
Сообщение № 2
Сообщение № 1
vvazarcova@fedora:~/work/arch-pc/lab07$
```

Рис. 4.7: Запуск снова измененного lab7-1

Результат выполнения программы соответствует ожиданиям.

3. Создаю файл lab7-2.asm (рис. 4.8).



```
vvazarcova@fedora:~/work/arch-pc/lab07$ touch lab7-2.asm
vvazarcova@fedora:~/work/arch-pc/lab07$
```

Рис. 4.8: Создание lab7-2.asm

Ввожу в lab7-2.asm текст программы, использующей условные переходы для того, чтобы вывести наибольшую из трех переменных, две из которых задаются в тексте программы, и одна вводится с клавиатуры (рис. 4.9).

```

GNU nano 7.2 /home/vvazarcova/work/arch-pc/lab07/lab7-2.asm
#include 'in_out.asm'
section .data
msg1 db 'Введите B: ',0h
msg2 db "Наибольшее число: ",0h
A dd '20'
C dd '50'
section .bss
max resb 10
B resb 10
section .text
global _start
_start:
; ----- Вывод сообщения 'Введите B: '
mov eax,msg1
call sprint
; ----- Ввод 'B'
mov ecx,B
mov edx,10
call sread
; ----- Преобразование 'B' из символа в число
mov eax,B
call atoi ; Вызов подпрограммы перевода символа в число
mov [B],eax ; запись преобразованного числа в 'B'
; ----- Записываем 'A' в переменную 'max'
mov ecx,[A] ; 'ecx = A'
mov [max],ecx ; 'max = A'
; ----- Сравниваем 'A' и 'C' (как символы)
cmp ecx,[C] ; Сравниваем 'A' и 'C'
jg check_B ; если 'A>C', то переход на метку 'check_B',
mov ecx,[C] ; иначе 'ecx = C'
mov [max],ecx ; 'max = C'
; ----- Преобразование 'max(A,C)' из символа в число
check_B:
mov eax,max
call atoi ; Вызов подпрограммы перевода символа в число
mov [max],eax ; запись преобразованного числа в 'max'
; ----- Сравниваем 'max(A,C)' и 'B' (как числа)
mov ecx,[max]
cmp ecx,[B] ; Сравниваем 'max(A,C)' и 'B'
jg fin ; если 'max(A,C)>B', то переход на 'fin',
mov ecx,[B] ; иначе 'ecx = B'
mov [max],ecx

```

Рис. 4.9: Текст программы в lab7-2.asm

Создаю исполняемый файл и запускаю его (рис. 4.10).

```

vvazarcova@fedora:~/work/arch-pc/lab07$ nasm -f elf lab7-2.asm
vvazarcova@fedora:~/work/arch-pc/lab07$ ld -m elf_i386 -o lab7-2 lab7-2.o
vvazarcova@fedora:~/work/arch-pc/lab07$ ./lab7-2
Введите B: 100
Наибольшее число: 100
vvazarcova@fedora:~/work/arch-pc/lab07$ ./lab7-2
Введите B: 5
Наибольшее число: 50
vvazarcova@fedora:~/work/arch-pc/lab07$

```

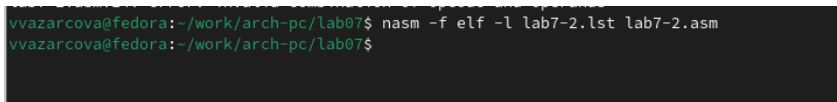
Рис. 4.10: Запуск lab7-2



В этой программе, переменные A и C сравниваются как символы, а переменная B и максимум из A и C как числа. Функция работает соответствует ожиданиям: при вводе 100, функция выводит сообщение о том, что наибольшее число 100, т.к.  $100 > 50 > 20$ . При вводе 5, функция выводит, что наибольшее число 50, т.к.  $50 > 20 > 5$ .

## 4.2 Изучение структуры файлы листинга

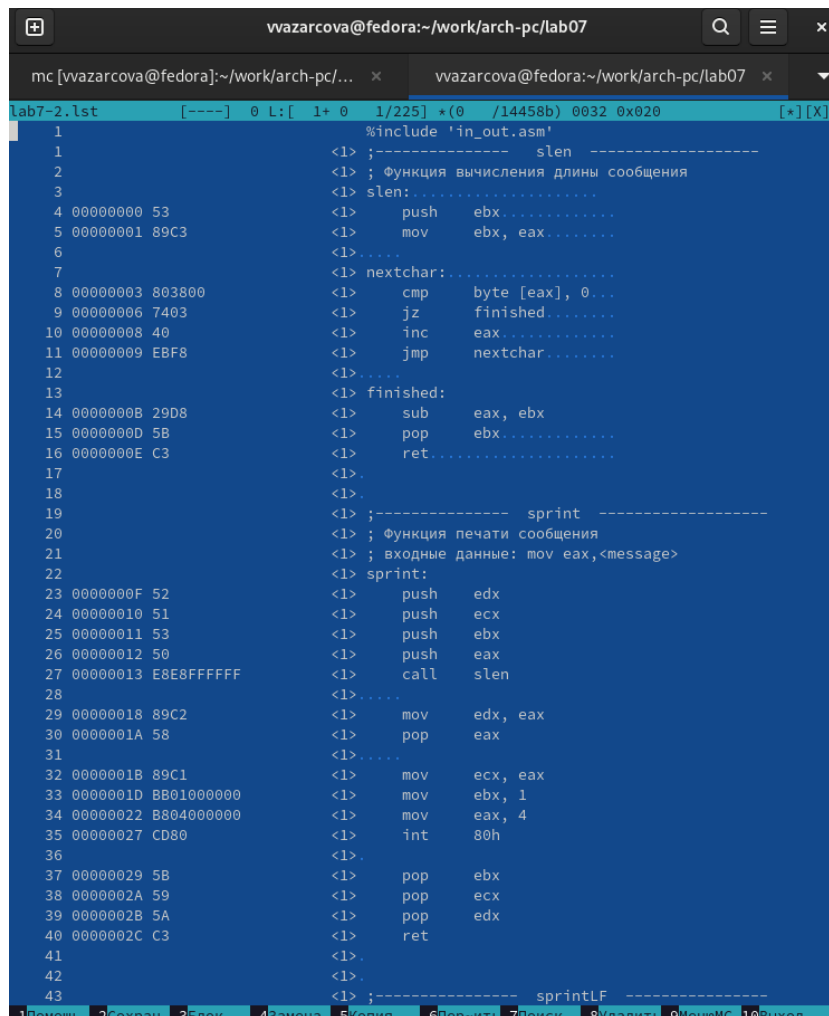
4. Создаю файл листинга для программы из файла lab7-2.asm (рис. 4.11).



```
v vazarcova@fedora:~/work/arch-pc/lab07$ nasm -f elf -l lab7-2.lst lab7-2.asm
v vazarcova@fedora:~/work/arch-pc/lab07$
```

Рис. 4.11: Создание lab7-2.lst

Открываю файл листинга с помощью mcedit - командой “mcedit lab7-2.lst” (рис. 4.12).



```
1 %include 'in_out.asm'
1 <1> ;----- slen -----
2 <1> ; Функция вычисления длины сообщения
3 <1> slen:
4 00000000 53 <1> push ebx
5 00000001 89C3 <1> mov ebx, eax
6 <1>
7 <1> nextchar:
8 00000003 803800 <1> cmp byte [eax], 0
9 00000006 7403 <1> jz finished
10 00000008 40 <1> inc eax
11 00000009 EBF8 <1> jmp nextchar
12 <1>
13 <1> finished:
14 0000000B 29D8 <1> sub eax, ebx
15 0000000D 5B <1> pop ebx
16 0000000E C3 <1> ret
17 <1>
18 <1>
19 <1> ;----- sprint -----
20 <1> ; Функция печати сообщения
21 <1> ; входные данные: mov eax, <message>
22 <1> sprint:
23 0000000F 52 <1> push edx
24 00000010 51 <1> push ecx
25 00000011 53 <1> push ebx
26 00000012 50 <1> push eax
27 00000013 E8E8FFFFFF <1> call slen
28 <1>
29 00000018 89C2 <1> mov edx, eax
30 0000001A 58 <1> pop eax
31 <1>
32 0000001B 89C1 <1> mov ecx, eax
33 0000001D B801000000 <1> mov ebx, 1
34 00000022 B804000000 <1> mov eax, 4
35 00000027 CD80 <1> int 80h
36 <1>
37 00000029 5B <1> pop ebx
38 0000002A 59 <1> pop ecx
39 0000002B 5A <1> pop edx
40 0000002C C3 <1> ret
41 <1>
42 <1>
43 <1> ;----- sprintLF -----
```

Рис. 4.12: lab2-2.lst в mcedit

Объясню содержимое первых трёх строк файла листинга.

1. 4 00000000 53 push ebx

- 4 это просто номер строки в листинге, не имеющий отношения к тексту программы. Он нужен для удобства чтения листинга.
- 00000000 это адрес в памяти, по которому находится инструкция. В этом случае, адрес 00000000 — это начало этой инструкции в памяти программы.
- 53 это машинный код инструкции. Когда программа собирается в машинный код, каждая инструкция преобразуется в набор байтов. В этом случае,

53 - шестнадцатиричное представление команды push ebx.

- push ebx - исходная команда на языке ассемблера, она означает что содержимое ebx помещается в стек.

2. 5 00000001 89C3 mov ebx, eax

- 5 - аналогично, номер строки в листинге.
- 00000002 - аналогично, адрес в памяти, по которому хранится инструкция mov ebx, eax.
- 89C3 - аналогично, машинный код для команды mov ebx, eax в виде шестнадцатиричного числа.
- mov ebx, eax - аналогично, исходная команда на языке ассемблера, которая копирует содержимое регистра eax в регистр ebx.

3. 8 00000003 803800 cmp byte [eax], 0

- 8 - аналогично, номер строки в листинге.
- 00000003 - аналогично, алрес в памяти, по которому хранится инструкция cmp byte [eax], 0.
- 89C3 - аналогично, машинный код для команды cmp byte [eax], 0 в виде шестнадцатиричного числа.
- cmp byte [eax], 0 - аналогично, исходная команда на языке ассемблера, которая копирует содержимое регистра eax в регистр ebx.

Открываю файл с программой lab2-1.asm и в строчке “mov eax, msg1” удаляю операнд msg1 (рис. 4.13).

```

GNU nano 7.2 /home/vvazarcova/work/arch-pc/lab07/lab7-2.asm
%include 'in_out.asm'
section .data
msg1 db 'Введите B: ',0h
msg2 db "Наибольшее число: ",0h
A dd '20'
C dd '50'
section .bss
max resb 10
B resb 10
section .text
global _start
_start:
; ----- Вывод сообщения 'Введите B: '
mov eax,
call sprint
; ----- Ввод 'B'
mov ecx,B
mov edx,10
call read
; ----- Преобразование 'B' из символа в число
mov eax,B
call atoi ; Вызов подпрограммы перевода символа в число
mov [B],eax ; запись преобразованного числа в 'B'
; ----- Записываем 'A' в переменную 'max'
mov ecx,[A] ; 'ecx = A'
mov [max],ecx ; 'max = A'
; ----- Сравниваем 'A' и 'C' (как символы)
cmp ecx,[C] ; Сравниваем 'A' и 'C'
jg check_B ; если 'A>C', то переход на метку 'check_B',
mov ecx,[C] ; иначе 'ecx = C'
mov [max],ecx ; 'max = C'
; ----- Преобразование 'max(A,C)' из символа в число
check_B:
mov eax,max
call atoi ; Вызов подпрограммы перевода символа в число
mov [max],eax ; запись преобразованного числа в 'max'
; ----- Сравниваем 'max(A,C)' и 'B' (как числа)
mov ecx,[max]
cmp ecx,[B] ; Сравниваем 'max(A,C)' и 'B'
jg fin ; если 'max(A,C)>B', то переход на 'fin',
mov ecx,[B] ; иначе 'ecx = B'
mov [max],ecx

```

Рис. 4.13: Измененный текст программы в lab2-1.asm

Выполняю трансляцию с получением файла листинга (рис. 4.14).

```

vvazarcova@fedora:~/work/arch-pc/lab07$ nasm -f elf -l lab7-2.lst lab7-2.asm
lab7-2.asm:14: error: invalid combination of opcode and operands
vvazarcova@fedora:~/work/arch-pc/lab07$

```

Рис. 4.14: Трансляция с получением файла листинга

При этом, терминал выводит ошибку, и создается только листинг, без создания объектного файла. Проверяю содержимое листинга и вижу, что теперь в нем появилась такая же ошибка, как в терминале (рис. 4.15).

```

vazarcova@fedora:~/work/arch-pc/lab07
mc [vazarcova@fedora:~/work/arch-pc/... x vazarcova@fedora:~/work/arch-pc/lab07 x]
lab7-2.lst [B---] 90 L:[171+19 190/226] *(11682/14544b) 0010 0x00A [*][X]
170 000000E7 C3 <1> ret
2 section .data
3 00000000 D092D0B2D0B5D0B4D0- msg1 db 'Введите B: ',0h
3 00000009 B8D182D0B520423A20-
3 00000012 00.....
4 00000013 D09DD0B0D0B8D0B1D0- msg2 db "Наибольшее число: ",0h
4 0000001C BED0BBD18CD188D0B5-
4 00000025 D0B520D187D0B8D181-
4 0000002E D0BBD0BE3A2000....
5 00000035 32300000 A dd '20'
6 00000039 35300000 C dd '50'
7 section .bss
8 00000000 <res Ah> max resb 10
9 0000000A <res Ah> B resb 10
10 section .text
11 global _start
12 _start:
13 ; ----- Вывод сообщения 'Введите B: '
14 mov eax
14 ***** error: invalid combination of opcode and operands
15 000000E8 E822FFFFFF call sprint
16 ; ----- Ввод 'B'
17 000000ED B9[0A000000] mov ecx,B
18 000000F2 BA0A000000 mov edx,10
19 000000F7 E847FFFFFF call sread
20 ; ----- Преобразование 'B' из символа в число
21 000000FC B8[0A000000] mov eax,B
22 00000101 E896FFFFFF call atoi ; Вызов подпрограммы перевода символа в чи
23 00000106 A3[0A000000] mov [B],eax ; запись преобразованного числа в 'B'
24 ; ----- Записываем 'A' в переменную 'max'
25 00000108 8B0D[35000000] mov ecx,[A] ; 'ecx = A'
26 00000111 890D[00000000] mov [max],ecx ; 'max = A'
27 ; ----- Сравниваем 'A' и 'C' (как символы)
28 00000117 3B0D[39000000] cmp ecx,[C] ; Сравниваем 'A' и 'C'
29 0000011D 7F0C jg check_B ; если 'A>C', то переход на метку 'check_
30 0000011F 8B0D[39000000] mov ecx,[C] ; иначе 'ecx = C'
31 00000125 890D[00000000] mov [max],ecx ; 'max = C'
32 ; ----- Преобразование 'max(A,C)' из символа в
33 check_B:
34 0000012B B8[00000000] mov eax,max
35 00000130 E867FFFFFF call atoi ; Вызов подпрограммы перевода символа в чи
36 00000135 A3[00000000] mov [max],eax ; запись преобразованного числа в 'max
37 ; ----- Сравниваем 'max(A,C)' и 'B' (как числа)
38 0000013A 8B0D[00000000] mov ecx,[max]
1Помощь 2Сохран 3Блок 4Замена 5Копия 6Пер-ить 7Поиск 8Удалить 9МенюМС 10Выход

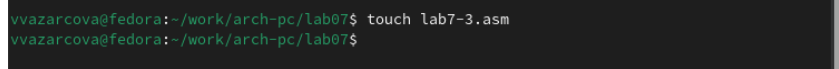
```

Рис. 4.15: Ошибка в файле листинга

## 5 Задание для самостоятельной работы

1. Мне требуется написать программу нахождения наименьшей из 3 целочисленных переменных  $a$ ,  $b$  и  $c$ , создать исполняемый файл и проверить его работу. Т.к. мой вариант - 12, значения переменных будут следующими:  $a=99$ ,  $b=29$ ,  $c=26$ .

Создам файл `lab7-3.asm` (рис. 5.1).



```
vvazarcova@fedora:~/work/arch-pc/lab07$ touch lab7-3.asm
vvazarcova@fedora:~/work/arch-pc/lab07$
```

Рис. 5.1: Создание `lab7-3.asm`

Напишу требуемую команду в `lab7-3.asm`, основываясь на программе, которую я ввела во время выполнения лабораторной работы (рис. 5.2).

```

GNU nano 7.2 /home/vvazarcova/work/arch-pc/lab07/lab7-3.asm
#include 'in_out.asm'
section .data
msg db "Наибольшее число: ",0h
A dd '99'
B dd '29'
C dd '26'
section .bss
max resb 10
section .text
global _start
_start:
; ----- Преобразование 'A' из символа в число
mov eax,A
call atoi ; Вызов подпрограммы перевода символа в число
mov [A],eax ; запись преобразованного числа в 'A'
; ----- Преобразование 'B' из символа в число
mov eax,B
call atoi ; Вызов подпрограммы перевода символа в число
mov [B],eax ; запись преобразованного числа в 'B'
; ----- Преобразование 'C' из символа в число
mov eax,C
call atoi ; Вызов подпрограммы перевода символа в число
mov [C],eax ; запись преобразованного числа в 'C'
; ----- Записываем 'A' в переменную 'max'
mov ecx,[A] ; 'ecx = A'
mov [max],ecx ; 'max = A'
; ----- Сравниваем 'A' и 'C' (как символы)
cmp ecx,[C] ; Сравниваем 'A' и 'C'
jg check_B ; если 'A>C', то переход на метку 'check_B',
mov ecx,[C] ; иначе 'ecx = C'
mov [max],ecx ; 'max = C'
; ----- Преобразование 'max(A,C)' из символа в число
check_B:
; ----- Сравниваем 'max(A,C)' и 'B' (как числа)
mov ecx,[max]
cmp ecx,[B] ; Сравниваем 'max(A,C)' и 'B'
jg fin ; если 'max(A,C)>B', то переход на 'fin',
mov ecx,[B] ; иначе 'ecx = B'
mov [max],ecx
; ----- Вывод результата
fin:
mov eax, msg

```

Рис. 5.2: Текст программы в lab7-3.asm

Программа аналогична тексту программы из lab7-2.asm, но отличается тем, что B задается в тексте самой программы, и все переменные сразу переводятся в числа. Поэтому, в отличие от lab7-2.asm, я задала B в начале программы, и вызвала команду `atoi` для каждой из трех переменных, только после чего я их сравнила.

### ***Листинг 7.1 - первая программа***

```

#include 'in_out.asm'

section .data

msg db "Наибольшее число: ",0h

A dd '99'

```

```

B dd '29'
C dd '26'
section .bss
max resb 10
section .text
global _start
_start:
; ----- Преобразование 'A' из символа в число
mov eax,A
call atoi ; Вызов подпрограммы перевода символа в число
mov [A],eax ; запись преобразованного числа в 'A'
; ----- Преобразование 'B' из символа в число
mov eax,B
call atoi ; Вызов подпрограммы перевода символа в число
mov [B],eax ; запись преобразованного числа в 'B'
; ----- Преобразование 'C' из символа в число
mov eax,C
call atoi ; Вызов подпрограммы перевода символа в число
mov [C],eax ; запись преобразованного числа в 'C'
; ----- Записываем 'A' в переменную 'max'
mov ecx,[A] ; 'ecx = A'
mov [max],ecx ; 'max = A'
; ----- Сравниваем 'A' и 'C' (как символы)
cmp ecx,[C] ; Сравниваем 'A' и 'C'
jg check_B ; если 'A>C', то переход на метку 'check_B',
mov ecx,[C] ; иначе 'ecx = C'
mov [max],ecx ; 'max = C'
; ----- Преобразование 'max(A,C)' из символа в число
check_B:

```

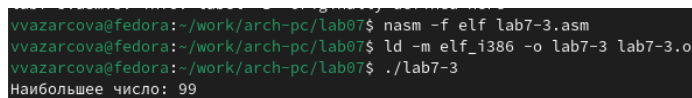


```

; ----- Сравниваем 'max(A,C)' и 'B' (как числа)
mov ecx, [max]
cmp ecx, [B] ; Сравниваем 'max(A,C)' и 'B'
jg fin ; если 'max(A,C)>B', то переход на 'fin',
mov ecx, [B] ; иначе 'ecx = B'
mov [max], ecx
; ----- Вывод результата
fin:
mov eax, msg
call sprint ; Вывод сообщения 'Наибольшее число: '
mov eax, [max]
call iprintLF ; Вывод 'max(A,B,C)'
call quit ; Выход

```

Создаю исполняемый файл и проверяю его работу (рис. 5.3).



```

vvazarcova@fedora: ~/work/arch-pc/lab07$ nasm -f elf lab7-3.asm
vvazarcova@fedora: ~/work/arch-pc/lab07$ ld -m elf_i386 -o lab7-3 lab7-3.o
vvazarcova@fedora: ~/work/arch-pc/lab07$ ./lab7-3
Наибольшее число: 99

```

Рис. 5.3: Запуск lab7-3

Программа выводит сообщение о том, что наибольшее число 99, что верно, т.к.  $99 > 29 > 26$ . Значит, программа работает корректно.

- Мне требуется написать программу, которая для введенных с клавиатуры значений  $x$  и  $a$  вычисляет значение заданной функции  $f(x)$  и выводит результат вычислений, создать исполняемый файл и проверить его работу для значений  $x=3, a=7$  и  $x=6, a=4$ .

Т.к. мой вариант - 12, мне нужно создать эту программу для функции следующего вида:

$$f(n) = \begin{cases} ax & x < 5 \\ x - 5 & x \geq 5 \end{cases}$$

Создаю lab7-4.asm (рис. 5.4).

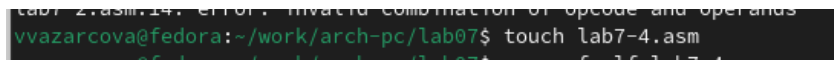


Рис. 5.4: Создание lab7-3.asm

Пишу программу в lab7-4.asm (рис. 5.5).

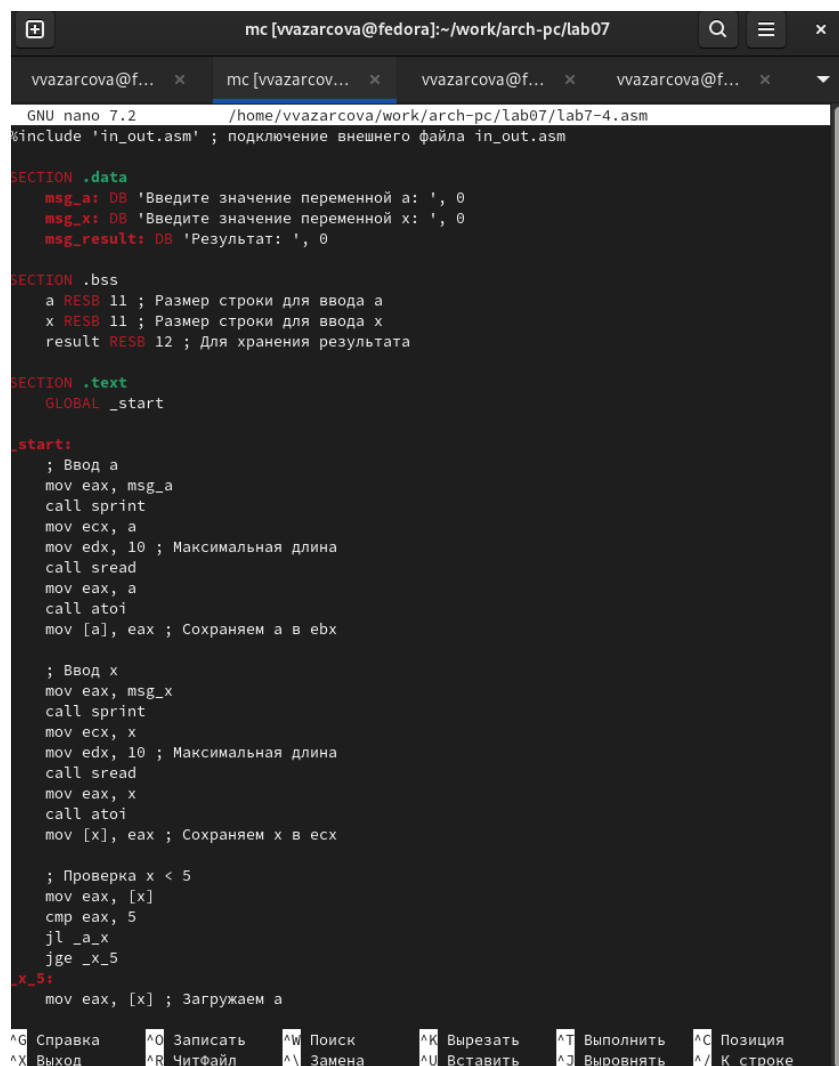


Рис. 5.5: Создание lab7-3.asm

### ***Листинг 7.1 - вторая программа***

```
%include 'in_out.asm' ; подключение внешнего файла in_out.asm
```

#### **SECTION .data**

```
msg_a: DB 'Введите значение переменной a: ', 0
msg_x: DB 'Введите значение переменной x: ', 0
msg_result: DB 'Результат: ', 0
```

#### **SECTION .bss**

```
a RESB 11 ; Размер строки для ввода a
x RESB 11 ; Размер строки для ввода x
result RESB 12 ; Для хранения результата
```

#### **SECTION .text**

```
GLOBAL _start
```

```
_start:
```

```
    ; Ввод a
    mov eax, msg_a
    call sprint
    mov ecx, a
    mov edx, 10 ; Максимальная длина
    call sread
    mov eax, a
    call atoi
    mov [a], eax ; Сохраняем a в ebx
```

```
    ; Ввод x
    mov eax, msg_x
```

```

call sprint
mov ecx, x
mov edx, 10 ; Максимальная длина
call sread
mov eax, x
call atoi
mov [x], eax ; Сохраняем x в ecx

; Проверка x < 5
mov eax, [x]
cmp eax, 5
jl _a_x
jge _x_5
_x_5:
mov eax, [x] ; Загружаем a
sub eax, 5 ; Загружаем 5
mov ecx, eax ; Переносим результат
jmp _res

_a_x:
; Вычисляем a*x
mov eax, [x] ; Загружаем a
mov ebx, [a] ; Загружаем x
mul ebx ; Умножаем a на x
jmp _res

_res:
; Подготовка результата для вывода
mov edi, eax ; Результат находится в eax

```

```

mov eax, msg_result
call sprint ; Вывод строки 'Результат: '
mov eax, edi
call iprintLF ; Вывод результата на новой строке

; Завершение программы
call quit

```

Создаю исполняемый файл и запускаю его, проверяя два разных варианта переменных, соответствующих моему варианту (рис. 5.6).

```

vvazarcova@fedora:~/work/arch-pc/lab07$ nasm -f elf lab7-4.asm
vvazarcova@fedora:~/work/arch-pc/lab07$ ld -m elf_i386 -o lab7-4 lab7-4.o
vvazarcova@fedora:~/work/arch-pc/lab07$ ./lab7-4
Введите значение переменной a: 7
Введите значение переменной x: 3
Результат: 21
vvazarcova@fedora:~/work/arch-pc/lab07$ ./lab7-4
Введите значение переменной a: 4
Введите значение переменной x: 6
Результат: 1
vvazarcova@fedora:~/work/arch-pc/lab07$

```

Рис. 5.6: Создание lab7-3.asm

Программа выводит 21 и 1, что является правильными значениями ( $3 \cdot 7 = 21$ ,  $6 - 5 = 1$ ).

## **6 Выводы**

Подводя итоги проведенной лабораторной работе, я научилась использовать команды условного и безусловного переходов и написала две программы с их применением.

## **Список литературы**