

Отчет по лабораторной работе №7

Дисциплина архитектура компьютера

Царёв Максим Александрович

Содержание

1	Цель работы.....	1
2	Задание	2
3	Теоретическое введение	2
4	Выполнение лабораторной работы.....	7
4.1	Реализация переходов в NASM	7
4.2	Программа с использованием инструкции jmp.....	8
4.3	Изучение структуры файлы листинга	11
5	Выполнение заданий для самостоятельной работы.....	12
6	Выводы	13

Список иллюстраций

Рис. 1: Использование инструкции jmp	3
Рис. 2: Регистр флагов.....	3
Рис. 3: Регистр флагов.....	4
Рис. 4: Инструкции условной передачи управления по результатам арифметического сравнения стр a,b	5
Рис. 5: Инструкции условной передачи управления по результатам арифметического сравнения стр a,b	6
Рис. 6: Инструкции условной передачи управления.....	6
Рис. 7: Фрагмент файла листинга	7
Рис. 8: Структура листинга	7

Список таблиц

No table of figures entries found.

1 Цель работы

Изучение команд условного и безусловного переходов. Приобретение навыков написания программ с использованием переходов. Знакомство с назначением и структурой файла листинга.

2 Задание

1. Программа с использованием инструкции jmp(листинг 1)
2. Программа с использованием инструкции jmp(листинг 2)
3. Изучение структуры файла листинга
4. 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.

Безусловный переход выполняется инструкцией jmp (от англ. jump – прыжок), которая включает в себя адрес перехода, куда следует передать управление: jmp Адрес перехода может быть либо меткой, либо адресом области памяти, в которую предварительно помещен указатель перехода. Кроме того, в качестве операнда можно использовать имя регистра, в таком случае переход будет осуществляться по адресу, хранящемуся в этом регистре.

Тип опера нда	Описание
jmp label	Переход на метку label
jmp [label]	Переход по адресу в памяти, помеченному меткой label
jmp eax	Переход по адресу из регистра eax

В следующем примере рассмотрим использование инструкции jmp:

```

label:
    ...      ;
    ...      ; команды
    ...      ;
    jmp label

```

Рис. 1: Использование инструкции `jmp`

Команды условного перехода__

Как отмечалось выше, для условного перехода необходима проверка какого-либо условия. В ассемблере команды условного перехода вычисляют условие перехода анализируя флаги из регистра флагов.

Регистр флагов

Флаг – это бит, принимающий значение 1 («флаг установлен»), если выполнено некоторое условие, и значение 0 («флаг сброшен») в противном случае. Флаги работают независимо друг от друга, и лишь для удобства они помещены в единый регистр — регистр флагов, отражающий текущее состояние процессора. В следующей таблице указано положение битовых флагов в регистре флагов:

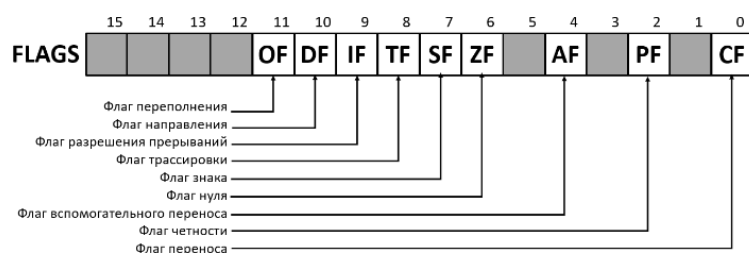


Рис. 2: Регистр флагов

Флаги состояния (биты 0, 2, 4, 6, 7 и 11) отражают результат выполнения арифметических инструкций, таких как ADD, SUB, MUL, DIV.

Бит	Обозначение	Название	Описание
0	CF	Carry Flag - Флаг переноса	Устанавливается в 1, если при выполнении предыдущей операции произошёл перенос из старшего бита или если требуется заём (при вычитании). Иначе установлен в 0.
2	PF	Parity Flag - Флаг чётности	Устанавливается в 1, если младший байт результата предыдущей операции содержит чётное количество битов, равных 1.
4	AF	Auxiliary Carry Flag - Вспомогательный флаг переноса	Устанавливается в 1, если в результате предыдущей операции произошёл перенос (или заём) из третьего бита в четвёртый.
6	ZF	Zero Flag - Флаг нуля	Устанавливается 1, если результат предыдущей команды равен 0.
7	SF	Sign Flag - Флаг знака	Равен значению старшего значащего бита результата, который является знаковым битом в знаковой арифметике.
11	SF	Overflow Flag - Флаг переполнения	Устанавливается в 1, если целочисленный результат слишком длинный для размещения в целевом операнде (регистре или ячейке памяти).

Рис. 3: Регистр флагов

Описание инструкции cmp

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

Инструкция cmp является командой сравнения двух операндов и имеет такой же формат, как и команда вычитания: `cmp ,` Команда `cmp`, так же как и команда вычитания, выполняет вычитание `-`, но результат вычитания никуда не записывается и единственным результатом команды сравнения является формирование флагов. Примеры:

Бит	Обозначение	Название	Описание
0	CF	Carry Flag - Флаг переноса	Устанавливается в 1, если при выполнении предыдущей операции произошёл перенос из старшего бита или если требуется заём (при вычитании). Иначе установлен в 0.
2	PF	Parity Flag - Флаг чётности	Устанавливается в 1, если младший байт результата предыдущей операции содержит чётное количество битов, равных 1.
4	AF	Auxiliary Carry Flag - Вспомогательный флаг переноса	Устанавливается в 1, если в результате предыдущей операции произошёл перенос (или заём) из третьего бита в четвёртый.
6	ZF	Zero Flag - Флаг нуля	Устанавливается 1, если результат предыдущей команды равен 0.
7	SF	Sign Flag - Флаг знака	Равен значению старшего значащего бита результата, который является знаковым битом в знаковой арифметике.
11	SF	Overflow Flag - Флаг переполнения	Устанавливается в 1, если целочисленный результат слишком длинный для размещения в целевом операнде (регистре или ячейке памяти).

Описание команд условного перехода

Команда условного перехода имеет вид `j label`. Мнемоника перехода связана со значением анализируемых флагов или со способом формирования этих флагов. В табл. 8.3. представлены команды условного перехода, которые обычно ставятся после команды сравнения `стр`. В их мнемокодах указывается тот результат сравнения, при котором надо делать переход. Мнемоники, идентичные по своему действию, написаны в таблице через дробь (например, `ja` и `jpbe`). Программист выбирает, какую из них применить, чтобы получить более простой для понимания текст программы.

Инструкции условной передачи управления по результатам арифметического сравнения `стр a, b`

Типы операндов	Мнемокод	Критерий условного перехода $a \vee b$	Значения флагов	Комментарий
Любые	<code>JE</code>	$a = b$	$ZF = 1$	Переход если равно
Любые	<code>JNE</code>	$a \neq b$	$ZF = 0$	Переход если не равно
Со знаком	<code>JL/JNGE</code>	$a < b$	$SF \neq OF$	Переход если меньше
Со знаком	<code>JLE/JNG</code>	$a \leq b$	$SF \neq OF$ или $ZF = 1$	Переход если меньше или равно
Со знаком	<code>JG/JNLE</code>	$a > b$	$SF = OF$ и $ZF = 0$	Переход если больше
Со знаком	<code>JGE/JNL</code>	$a \geq b$	$SF = OF$	Переход если больше или равно
Без знака	<code>JB/JNAE</code>	$a < b$	$CF = 1$	Переход если ниже

Рис. 4: Инструкции условной передачи управления по результатам арифметического сравнения `стр a, b`

Типы операндов	Мнемокод	Критерий условного перехода $a \vee b$	Значения флагов	Комментарий
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	JAE/JNB	$a \geq b$	CF = 0	Переход если выше или равно

Рис. 5: Инструкции условной передачи управления по результатам арифметического сравнения стр a, b

Примечание: термины «выше» («a» от англ. «above») и «ниже» («b» от англ. «below») применимы для сравнения беззнаковых величин (адресов), а термины «больше» («g» от англ. «greater») и «меньше» («l» от англ. «lower») используются при учёте знака числа. Таким образом, мнемонику инструкции JA/JNBE можно расшифровать как «jump if above (переход если выше) / jump if not below equal (переход если не меньше или равно)». Помимо перечисленных команд условного перехода существуют те, которые можно использовать после любых команд, меняющих значения флагов.

Мнемокод	Значение флага для осуществления перехода	Мнемокод	Значение флага для осуществления перехода
JZ	ZF = 1	JNZ	ZF = 0
JS	SF = 1	JNS	SF = 0
JC	CF = 1	JNC	CF = 0
JO	OF = 1	JNO	OF = 0
JP	PF = 1	JNP	PF = 0

Рис. 6: Инструкции условной передачи управления

В качестве примера рассмотрим фрагмент программы, которая выполняет умножение переменных a и b и если произведение превосходит размер байта, передает управление на метку Error. `mov al, a mov bl, b mul bl jc Error`

Файл листинга и его структура

Листинг (в рамках понятийного аппарата NASM) — 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. Ниже приведён 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

```

Рис. 7: Фрагмент файла листинга

Строки в первой части листинга имеют следующую структуру



Рис. 8: Структура листинга

Все ошибки и предупреждения, обнаруженные при ассемблировании, транслятор выводит на экран, и файл листинга не создаётся. Итак, структура листинга:

- номер строки — это номер строки файла листинга (нужно помнить, что номер строки в файле листинга может не соответствовать номеру строки в файле с исходным текстом программы);
- адрес — это смещение машинного кода от начала текущего сегмента;
- машинный код представляет собой ассемблированную исходную строку в виде шестнадцатеричной последовательности. (например, инструкция `int 80h` начинается по смещению `00000020` в сегменте кода; далее идёт машинный код, в который ассемблируется инструкция, то есть инструкция `int 80h` ассемблируется в `CD80` (в шестнадцатеричном представлении); `CD80` — это инструкция на машинном языке, вызывающая прерывание ядра);

4 Выполнение лабораторной работы

4.1 Реализация переходов в NASM

создал каталог `lab07` для программ лабораторной работы, перешёл в него и создал файл `lab7-1.asm`

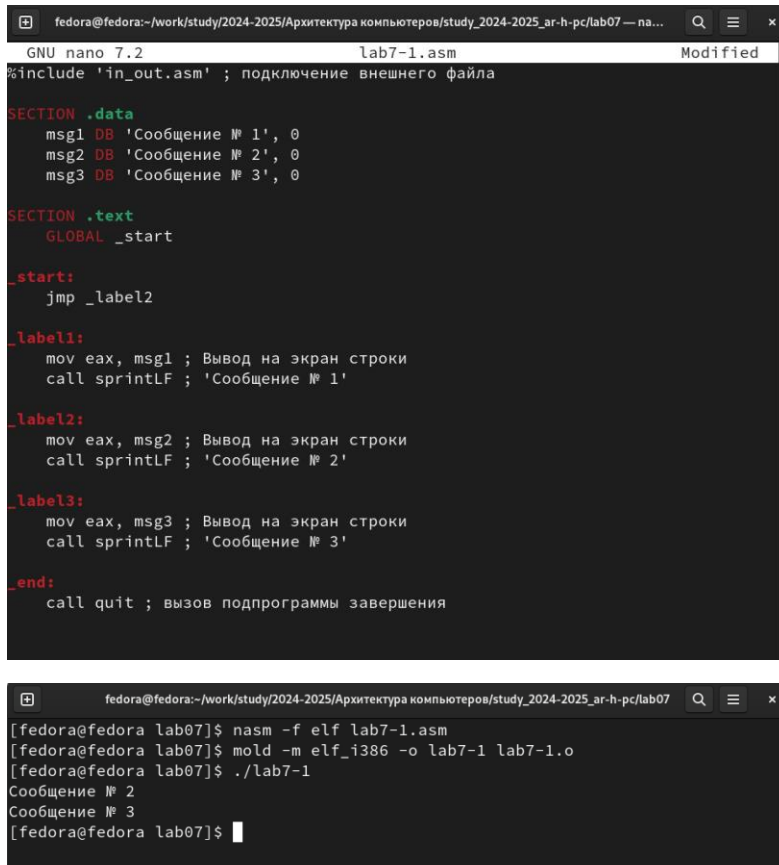
```

fedora@fedora:~/work/study/2024-2025/Архитектура компьютеров/study_2024-2025_ar-h-pc/lab07$ 
[~]$ cd work/study/2024-2025/Архитектура\ компьютеров/study_2024-2025_ar-h-pc/
[study_2024-2025_ar-h-pc]$ mkdir lab07
[study_2024-2025_ar-h-pc]$ cd lab07
[lab07]$ touch lab7-1.asm
[lab07]$ ls
lab7-1.asm
[lab07]$

```

4.2 Программа с использованием инструкции jmp

я скопировал внешний файл в созданный каталог, ввёл текст программы с использованием инструкции jmp в текстовый файл lab7-1.asm, создал объектный файл и проверил работу программы



```
GNU nano 7.2 lab7-1.asm Modified
%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'

_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

[fedora@fedora lab07]$ nasm -f elf lab7-1.asm
[fedora@fedora lab07]$ mold -m elf_i386 -o lab7-1 lab7-1.o
[fedora@fedora lab07]$ ./lab7-1
Сообщение № 2
Сообщение № 3
[fedora@fedora lab07]$
```

изменил текст программы и проверил её работу


```
fedora@fedora:~/work/study/2024-2025/Архитектура компьютеров/study_2024-2025_ar-h-pc/lab07 — na...
GNU nano 7.2 lab7-1.asm Modified
#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
```

```
fedora@fedora:~/work/study/2024-2025/Архитектура компьютеров/study_2024-2025_ar-h-pc/lab07
[fedora@fedora lab07]$ nasm -f elf lab7-1.asm
[fedora@fedora lab07]$ mold -m elf_i386 -o lab7-1 lab7-1.o
[fedora@fedora lab07]$ ./lab7-1
Сообщение № 3
Сообщение № 2
Сообщение № 1
[fedora@fedora lab07]$
```

Программа, которая определяет и выводит на экран наибольшую из 3 целочисленных переменных: А,В и С. Создаю файл с названием lab7-2.asm и ввожу текст программы.

```
fedora@fedora:~/work/study/2024-2025/Архитектура компьютеров/study_2024-2025_ar-h-pc/lab07 — na...
GNU nano 7.2 lab7-2.asm
#include 'in_out.asm' ; подключение внешнего файла

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 eax, [A]
    mov [max], eax

    ; ----- Сравниваем 'A' и 'B' -----
    mov eax, [B]
    cmp eax, [max]
    jg update_max

    ; ----- Сравниваем 'max' и 'C' -----

^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute
^X Exit      ^R Read File ^_ Replace   ^U Paste     ^J Justify
```

При введении числа до 50, программа выводит наибольшее число 50, при введении числа больше 50, программа выводит введенное нами число. Программа сравнивает число A (значение 20) и C (значение 50) и инициализирует переменную max значением большего из них. Сравнивает текущее значение max с введённым числом B и обновляет max, если B больше. Выводит сообщение “Наибольшее число:” и затем значение переменной max, которая содержит наибольшее из трёх чисел: A, B и C

```
fedora@fedora:~/work/study/2024-2025/Архитектура компьютеров/study_2024-2025_ar-h-pc/lab07
[fedora@fedora lab07]$ nasm -f elf lab7-2.asm
[fedora@fedora lab07]$ mold -m elf_i386 -o lab7-2 lab7-2.o
[fedora@fedora lab07]$ ./lab7-2
Введите B: 2
Наибольшее число: 50
[fedora@fedora lab07]$ ./lab7-2
Введите B: -1
Наибольшее число: 50
[fedora@fedora lab07]$ ./lab7-2
Введите B: 100
Наибольшее число: 100
[fedora@fedora lab07]$ ./lab7-2
Введите B: 250
Наибольшее число: 250
[fedora@fedora lab07]$ ./lab7-2
Введите B: 100000
Наибольшее число: 100000
```

4.3 Изучение структуры файлы листинга

Создаю файл листинга для программы из файла

```
fedora@fedora:~/work/study/2024-2025/Архитектура компьютеров/study_2024-2025_ar-h
[fedora@fedora lab07]$ nasm -f elf -l lab7-2.lst lab7-2.asm
[fedora@fedora lab07]$
```

Открываю его через mcedit

```
mcedit [fedora@fedora:~/work/study/2024-2025/Архитектура компьютеров/study_2024-2025_ar-h-pc/lab07]
lab7-2.lst [----] 0 L: [ 1+ 0 1/236] *(0 /14609b) 0032 0x020[*][X]
1                                     %include 'in_out.asm' ; подключение вне
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 BB01000000                 <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> -----
```

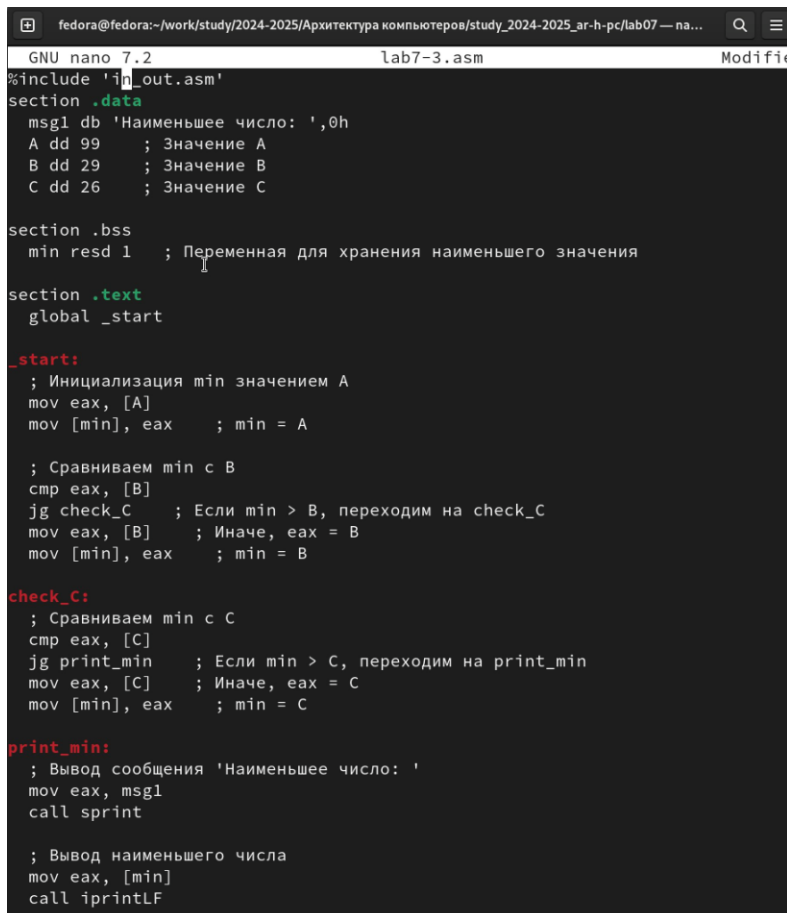
При компиляции и сборке программы на ассемблере создаются следующие файлы: Объектный файл (.o): Это промежуточный файл, содержащий машинный код, но ещё не готовый для выполнения. Исполняемый файл: После связывания объектных файлов с библиотеками (например, с помощью ld), создается исполняемый файл, который можно запустить. Файл листинга (.lst): Это текстовый файл, который включает исходный код программы вместе с адресами и скомпилированным машинным кодом. В этом файле обычно содержатся комментарии и информация о процессе компиляции.

В файл листинга могут быть добавлены следующие элементы: Исходный код: Полный исходный код программы, как он написан в ассемблере. Адреса: Для каждой инструкции будут указаны адреса в памяти, по которым эти инструкции

будут располагаться после компиляции. Машинный код: Бинарный код, соответствующий каждой инструкции, представленный в шестнадцатеричном формате. Комментарии: Комментарии из исходного кода, которые могут помочь понять логику программы. Информация о секциях: Данные о том, как разделены секции кода (.text, .data, .bss и т.д.) и их размеры. Ошибки и предупреждения: Если при компиляции были обнаружены ошибки или предупреждения, они также могут быть записаны в файл листинга.

5 Выполнение заданий для самостоятельной работы

Создаю файл с названием lab7-3.asm, написал программу для нахождения наименьшего из 3 переменных, значения переменных беру исходя из своего варианта, полученного в ходе лабораторной работы номер 6, номер моего варианта 12.



```
GNU nano 7.2 lab7-3.asm
%include 'in_out.asm'
section .data
    msg1 db 'Наименьшее число: ',0h
    A dd 99 ; Значение A
    B dd 29 ; Значение B
    C dd 26 ; Значение C

section .bss
    min resd 1 ; Переменная для хранения наименьшего значения

section .text
    global _start

_start:
    ; Инициализация min значением A
    mov eax, [A]
    mov [min], eax ; min = A

    ; Сравниваем min с B
    cmp eax, [B]
    jg check_C ; Если min > B, переходим на check_C
    mov eax, [B] ; Иначе, eax = B
    mov [min], eax ; min = B

check_C:
    ; Сравниваем min с C
    cmp eax, [C]
    jg print_min ; Если min > C, переходим на print_min
    mov eax, [C] ; Иначе, eax = C
    mov [min], eax ; min = C

print_min:
    ; Вывод сообщения 'Наименьшее число: '
    mov eax, msg1
    call sprintf

    ; Вывод наименьшего числа
    mov eax, [min]
    call iprintLF
```

Проверяю работу программы, программа работает верно.

```
/lab07$ nasm -f elf lab7-3.asm
/lab07$ ld -m elf_i386 -o lab7-3 lab7-3.o
/lab07$ ./lab7-3
```

Наименьшее число: 26

Создаю файл с названием lab7-4.asm, написал программу для вычисления $f(x)$, пишу программу для функции исходя из своего варианта, полученного в ходе лабораторной работы номер 6, номер моего варианта 12.

```
[fedora@fedora lab07]$ nano lab7-4.asm
[fedora@fedora lab07]$ nasm -f elf lab7-4.asm
[fedora@fedora lab07]$ mold -m elf_i386 -o lab7-4 lab7-4.o
[fedora@fedora lab07]$ ./lab7-4
```

```
Введите x: 3
Введите a: 7
Результат  $f(x)$  = 21
```

Программа работает верно, это я выяснил подставив значения.

6 Выводы

В результате выполнения данной лабораторной работы, я изучил команды условного и безусловного переходов, приобрел навыки написания программ с использованием переходов и познакомился с назначением и структурой файла листинга