

РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ № 7

дисциплина: Архитектура компьютера

Студент: Грицко Сергей

Группа:НКАбд-02-25

МОСКВА

2025 г.

Оглавление

Цель работы.....	3
Теоретическое введение	4
2.1. Безусловный переход.....	4
2.2. Условные переходы.....	4
2.3. Команда сравнения CMP	5
2.4. Таблица команд условного перехода.....	5
Выполнение лабораторной работы	6
3.2. Программа с условными переходами.....	6
3.3. Изучение файла листинга.....	7
Задания для самостоятельной работы.....	8
Задание 4.1. Нахождение наименьшего из трех чисел	8
Задание 4.2. Вычисление значения функции.....	8
Выводы	10
Список литературы	11

Цель работы

Целью данной лабораторной работы является изучение команд условного и безусловного переходов в языке ассемблера NASM. Необходимо приобрести практические навыки написания программ с использованием механизмов ветвления и циклов, что позволяет реализовывать сложные нелинейные алгоритмы. Также важной частью работы является знакомство с назначением и структурой файла листинга (.lst), который необходим для отладки и анализа низкоуровневого кода.

Теоретическое введение

Процессор по умолчанию выполняет инструкции последовательно, одну за другой, в том порядке, в котором они записаны в памяти. Однако для реализации алгоритмов, содержащих условия (конструкции if-else) или циклы (конструкции for, while), необходимо нарушать этот порядок. Для этого в ассемблере используются команды передачи управления. Можно выделить два основных типа переходов:

2.1. Безусловный переход

Безусловный переход выполняется инструкцией `jmp` (от англ. *jump* — прыжок). Она передает управление по указанному адресу без проверки каких-либо условий.

Синтаксис:

```
jmp <адрес_перехода>
```

Адрес перехода может быть задан меткой, адресом в области памяти или именем регистра. В последнем случае переход происходит по адресу, хранящемуся в регистре. Это аналог оператора `goto` в языках высокого уровня.

2.2. Условные переходы

Условные переходы осуществляют передачу управления только в том случае, если выполняется определенное условие. Если условие не выполняется, процессор просто переходит к следующей инструкции.

Условия базируются на состоянии **регистра флагов (EFLAGS)**. Флаги — это биты, которые меняют свое значение (0 или 1) в зависимости от результата предыдущей арифметической или логической операции.

Основные флаги, используемые для переходов:

1. **ZF (Zero Flag)** — Флаг нуля. Устанавливается в 1, если результат операции равен нулю.
2. **SF (Sign Flag)** — Флаг знака. Устанавливается в 1, если результат операции отрицательный (старший бит равен 1).
3. **CF (Carry Flag)** — Флаг переноса. Устанавливается в 1, если при арифметической операции произошло переполнение беззнакового числа (перенос из старшего разряда).

4. **OF (Overflow Flag)** — Флаг переполнения. Используется для знаковой арифметики.

2.3. Команда сравнения *CMP*

Для проверки условий чаще всего используется команда *cmp* (от англ. *compare*).

Синтаксис:

cmp <операнд1>, <операнд2>

Механизм работы: команда выполняет вычитание операнд1 - операнд2, но, в отличие от команды *sub*, **не сохраняет результат** вычитания в операнде. Единственным результатом работы команды является изменение флагов в регистре EFLAGS.

2.4. Таблица команд условного перехода

Команды перехода делятся на команды для знаковых и беззнаковых чисел.

Мнемоника	Описание условия	Анализируемые флаги
JE (Jump Equal)	Переход, если равно	ZF = 1
JNE (Jump Not Equal)	Переход, если не равно	ZF = 0
JG (Jump Greater)	Переход, если больше (со знаком)	ZF = 0 и SF = OF
JL (Jump Less)	Переход, если меньше (со знаком)	SF ≠ OF
JGE (Jump Greater or Equal)	Переход, если больше или равно	SF = OF
JLE (Jump Less or Equal)	Переход, если меньше или равно	ZF = 1 или SF ≠ OF
JA (Jump Above)	Переход, если выше (без знака)	CF = 0 и ZF = 0
JB (Jump Below)	Переход, если ниже (без знака)	CF = 1

Таблица 1. Команды условного перехода

Выполнение лабораторной работы

Для начала работы я создал каталог lab07 в своей рабочей директории и перешел в него. Затем я создал файл lab7-1.asm с помощью текстового редактора.

Задача состояла в том, чтобы написать программу, использующую инструкцию jmp для изменения порядка вывода сообщений на экран.

Текст программы lab7-1.asm: (Смотри файл lab7-1.asm, приложенный к отчету)

В данной программе я использовал три метки: _label1, _label2, _label3 и метку завершения _end. Логика работы программы следующая:

1. Сразу после старта (_start) выполняется безусловный переход jmp _label3.
2. Программа переходит к метке _label3, где выводится "Сообщение № 3", после чего выполняется jmp _label2.
3. Управление передается на метку _label2, выводится "Сообщение № 2", затем идет переход jmp _label1.
4. На метке _label1 выводится "Сообщение № 1" и выполняется финальный переход jmp _end.
5. На метке _end программа корректно завершает работу.

Компиляция и запуск: Я использовал следующие команды для сборки:

```
sgritsko@sgritsko:~/works/arch-pc/lab7$ nasm -f elf lab7-1.asm
sgritsko@sgritsko:~/works/arch-pc/lab7$ ld -m elf_i386 -o lab7-1 lab7-1.o
sgritsko@sgritsko:~/works/arch-pc/lab7$ ./lab7-1
```

Рисунок 1. Команды для сборки

Результат работы программы:

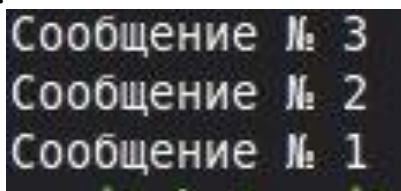


Рисунок 2. Вывод сообщений в обратном порядке

Как видно на скриншоте, несмотря на то, что в коде сообщения записаны последовательно, инструкция jmp позволила изменить порядок их вывода на обратный.

3.2. Программа с условными переходами

Во втором задании требовалось написать программу, которая определяет наибольшее из трех чисел: переменные A и C заданы в коде, а B вводится с клавиатуры.

Я создал файл lab7-2.asm. Для реализации ввода и сравнения чисел я использовал функции из подключаемого файла in_out.asm. Особое внимание я уделил тому, что ввод с клавиатуры дает символьную строку, которую необходимо преобразовать в целое число функцией atoi перед выполнением арифметического сравнения.

Текст программы lab7-2.asm: (Смотри файл lab7-2.asm, приложенный к отчету)

Алгоритм работы программы:

1. Программа выводит приглашение "Введите В".
2. Считывает строку с консоли.
3. Преобразует введенную строку в число (функция atoi).
4. Сравнивает жестко заданное число А с числом С. Если $A > C$, то программа запоминает А как текущий максимум, иначе — С.
5. Затем текущий максимум сравнивается с введенным числом В. Если В больше, то оно становится новым максимумом.
6. Результат выводится на экран.

```
sgritsko@sgritsko:~/works/arch-pc/lab7$ nasm -f elf lab7-2.asm
sgritsko@sgritsko:~/works/arch-pc/lab7$ ld -m elf_i386 -o lab7-2 lab7-2.o
sgritsko@sgritsko:~/works/arch-pc/lab7$ ./lab7-2
Введите В: 5
Наибольшее число: 50
```

Рисунок 3. Запуск программы с вводом числа

Программа корректно определяет максимальное число.

3.3. Изучение файла листинга

Файл листинга — это текстовый файл, который генерируется ассемблером и показывает взаимосвязь между исходным кодом и машинными инструкциями. Я создал файл листинга командой:

```
sgritsko@sgritsko:~/works/arch-pc/lab7$ nasm -f elf -l lab7-2.lst lab7-2.asm
sgritsko@sgritsko:~/works/arch-pc/lab7$
```

Рисунок 4. Выполнение команды

Затем я открыл полученный файл lab7-2.lst в текстовом редакторе. Вот фрагмент содержимого файла:

```
35 0000011C 3B0D[39000000]      cmp esx, [C]      ; Сравниваем А и С (как символы, согласно методичке)
36 00000122 7F0C              jg check_B       ; Если А > С, то идем проверять В
```

Рисунок 5. Просмотр файла листинга

Разберем одну из строк листинга подробно:

1. **35, 36** — это номера строк в исходном коде .asm.
2. **00000045, 0000004B** — это смещения (адреса) команд от начала сегмента кода в шестнадцатеричном формате.
3. **3B0D[16000000]** — машинный код команды `cmp esx, [C]`.
4. **7F0C** — машинный код команды `jg check_B`. Здесь 7F — это код операции (опкод) перехода, а 0C — это операнд, указывающий, на сколько байт нужно прыгнуть вперед.

Эксперимент с ошибкой: Я попробовал удалить один из операндов в инструкции `cmp` и снова запустить сборку с ключом `-l`. Ассемблер выдал ошибку `invalid combination of opcode and operands` в терминал. При этом **файл листинга не был создан**. Это доказывает, что листинг формируется только при успешной трансляции кода.

Задания для самостоятельной работы

Вариант задания: 14

Задание 4.1. Нахождение наименьшего из трех чисел

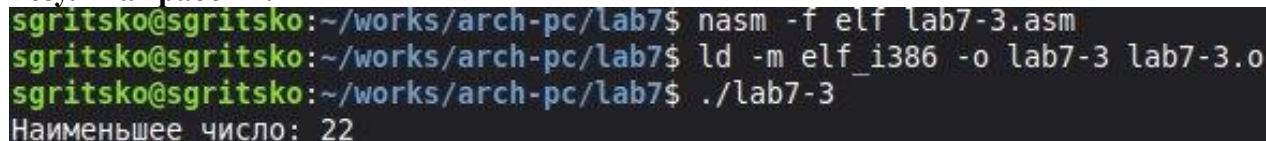
Условие: Даны три числа: $a = 81$, $b = 22$, $c = 72$. Необходимо написать программу на ассемблере, которая находит наименьшее из них.

Я создал файл lab7-3.asm. **Алгоритм решения:**

1. Принимаем за условный минимум (min) первое число a (81). Для этого помещаем его в регистр EAX.
2. Сравниваем текущий min (в EAX) со вторым числом b (22).
3. Используем команду `cmp eax, [b]` и переход `jl (jump less)`.
4. Если значение в EAX меньше b , пропускаем шаг обновления. Но так как $81 > 22$, условие "меньше" не выполняется. Мы обновляем min: `mov eax, [b]`. Теперь в EAX число 22.
5. Сравниваем текущий min (22) с третьим числом c (72).
6. Так как $22 < 72$, обновление не требуется.
7. Выводим значение регистра EAX.

Листинг программы lab7-3.asm: (Полный код представлен в приложенном файле lab7-3.asm)

Результат работы:



```
sgritsko@sgritsko:~/works/arch-pc/lab7$ nasm -f elf lab7-3.asm
sgritsko@sgritsko:~/works/arch-pc/lab7$ ld -m elf_i386 -o lab7-3 lab7-3.o
sgritsko@sgritsko:~/works/arch-pc/lab7$ ./lab7-3
Наименьшее число: 22
```

Рисунок 6. Результат работы программы поиска минимума

Программа выдала число 22, что является верным ответом, так как $22 < 72$ и $22 < 81$.

Задание 4.2. Вычисление значения функции

Условие: Напишите программу, которая для введенных с клавиатуры значений x и a вычисляет значение заданной функции $f(x)$ и выводит результат.

Алгоритм (блок-схема логики):

1. **Ввод данных:** Считываем строки x и a с клавиатуры, преобразуем их в целые числа (функция `atoi`) и сохраняем в переменные.
2. **Сравнение:** Выполняем команду `cmp [x], [a]`.
3. **Ветвление:**
 - 3.1. Если x и a (используем инструкцию `jge case_greater_equal`), переходим к метке расчета второй формулы.
 - 3.2. Если условие ложно (значит $x < a$), продолжаем выполнение следующей строки кода (первая формула).
4. **Вычисление ветви 1 ($x < a$):**
 - 4.1. Загружаем a в регистр `EAX`.
 - 4.2. Умножаем на 3 (`mov ebx, 3, mul ebx`).
 - 4.3. Прибавляем 1 (`add eax, 1`).
 - 4.4. Безусловно переходим к выводу (`jmp print_res`), чтобы пропустить код второй ветки.
5. **Вычисление ветви 2 (x и a):**
 - 5.1. Метка `case_greater_equal`.
 - 5.2. Загружаем x в регистр `EAX`.
 - 5.3. Умножаем на 3.
 - 5.4. Прибавляем 1.
6. **Вывод:** Печатаем значение из `EAX`.

Листинг программы lab7-4.asm: (Полный код представлен в приложенном файле `lab7-4.asm`)

Тестирование программы: Для проверки правильности алгоритма я провел два теста с разными входными данными.

Тест 1: Ввод: $x = 2$, $a = 3$. Проверка условия: $2 < 3$ (истина). Ожидаемый результат: $f(x) = 3 * 3 + 1 = 10$.

Тест 2: Ввод: $x = 4$, $a = 2$. Проверка условия: 4 и 2 (истина). Ожидаемый результат: $f(x) = 3 * 4 + 1 = 13$.

Скриншот с результатами тестов:



```
sgritsko@sgritsko:~/works/arch-pc/lab7$ ./lab7-4
Введите x: 2
Введите a: 3
Результат: 10
sgritsko@sgritsko:~/works/arch-pc/lab7$ ./lab7-4
Введите x: 4
Введите a: 2
Результат: 13
```

Рисунок 7. Проверка вычисления функции

Программа работает корректно и выдает ожидаемые математические результаты.

Выводы

В ходе выполнения лабораторной работы №7 я изучил механизмы управления потоком выполнения программ в архитектуре i386 с использованием ассемблера NASM.

Мною были выполнены следующие задачи:

1. Изучены теоретические основы ветвлений: роль регистра флагов (EFLAGS) и принцип работы команд переходов.
2. На практике освоена работа с инструкцией безусловного перехода `jmp`. Я убедился, что с её помощью можно произвольно менять порядок выполнения команд, создавая "спагетти-код" или, наоборот, структурированные циклы.
3. Реализованы программы с условными переходами. Я научился комбинировать команду сравнения `cmp` с командами `jl`, `jg`, `jge` для реализации логики `if-else`. Это позволило решить задачи поиска максимума и минимума чисел, а также вычисления кусочно-заданной функции.
4. Проведен анализ файла листинга (`.lst`). Я выяснил, что листинг является мощным инструментом отладки, позволяющим увидеть машинный код и проверить корректность трансляции меток и адресов переходов. Опытным путем подтверждено, что листинг генерируется только при отсутствии синтаксических ошибок в коде.

Таким образом, цель работы достигнута: я приобрел навыки программирования ветвлений на ассемблере, что является базой для реализации любых сложных алгоритмов на низком уровне.

Список литературы

1. GDB: The GNU Project Debugger. — URL: <https://www.gnu.org/software/gdb/>.
2. NASM Assembly Language Tutorials. — 2021. — URL: <https://asmtutor.com/>.
3. Колдаев В. Д., Лупин С. А. Архитектура ЭВМ. — М. : Форум, 2018.
4. Куляс О. Л., Никитин К. А. Курс программирования на ASSEMBLER. — М. : Солон-Пресс, 2017.
5. Столяров А. Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011.
6. Таненбаум Э. Архитектура компьютера. — 6-е изд. — СПб. : Питер, 2013. — 874 с.