**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра математического обеспечения и применения ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №3**

**по дисциплине «Организация ЭВМ и систем»**

# **Тема:** **Изучение организации ветвлений в программах на языке Ассемблера.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Трунов Б.Г. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2024

**Цель работы**

Изучить организацию ветвлений и написать собственную программу на языке Ассемблера процессора Intel X86.

**Задание**

Разработать на языке Ассемблер iX86 программу, которая по заданным целым значениям a, b, i, k, размером 1 слово, вычисляет:

a) значения i1 = fn1(a, b, i) и i2 = fn2(a, b, i);

b) значения res= fn3(i1, i2, k),

где вид функций fn1, fn2 определяется из табл.1, а функции fn3 - из табл.2 по цифрам шифра индивидуального задания (n1.n2.n3). Значения a, b, i, k являются исходными данными, которые должны выбираться студентом самостоятельно и задаваться в процессе исполнения программы в режиме отладки. При этом следует рассмотреть все возможные комбинации параметров a, b и k, позволяющие проверить различные маршруты выполнения программы.

Замечания (требования):

1) При разработке программы не использовать фрагменты, представленные на ЯВУ, в частности, для ввода-вывода данных. Исходные данные должны вводиться, а результаты контролироваться в режиме отладки.

2) При вычислении функций fn1 и fn2 вместо операции умножения следует использовать арифметический сдвиг и, возможно, сложение.

3) Не использовать процедуры (в том числе при вычислении функций fn1 и fn2).

4) Случаи, когда число-результат выходит за пределы одного слова, учитывать не требуется.

5) При разработке программы следует минимизировать длину кода; для этого могут потребоваться преобразования формул и введение промежуточных результатов.

6) В коде должна быть выделена главная вычислительная часть, именно её нужно минимизировать. В главную часть не входят команды для инициализации сегментного регистра и для корректного завершения программы.

Для выполнения задания:

1. Преобразовать формулы, по которым будут выполняться вычисления, согласно замечанию 5.

2. В табличном процессоре (например, LibreOffice Calc) создать документ, в котором в 4 ячейки вводятся входные данные (a, b, i, k), в 3 ячейках появляются результаты i1, i2, res, вычисленные по исходным формулам, и ещё в одной ячейке — res, вычисленный по преобразованной формуле. Проверить, что результаты совпадают для разных входных данных.

3. Разработать программу; в коде должны быть выделены части, посвящённые вычислению итоговых значений (3 штуки) и ключевых промежуточных значений, если таковые были определены при преобразовании формул. Код должен быть с подробными комментариями: после каждой вычислительной команды должно быть указано полученное к текущему моменту значение. Подсчитать количество команд в главной вычислительной части и указать в комментарии.

4. Протестировать программу, оформив таблицу с входными данными и результатами. Тесты должны проверять все пути выполнения вычислений, и их точно не менее 4.

Вариант 23 (Шифр: 5.6.4):

/ 20-4\*i, при a > b

f1 = <

\ -(6\*i-6), при a <= b

/ 2\*(i+1)-4, при a > b

f2 = <

\ 5-3\*(i+1), при a <= b

/ min(| i1 - i2 |, 2), при k < 0

f3 = <

\ max(-6, -i2), при k >= 0

**ОСНОВНЫЕ ТЕОР. ПОЛОЖЕНИЯ**

**label (метка)** – это символическое имя, присваиваемое определенному месту в коде. Метки используются как цели для переходов и как ссылки на адреса памяти.

**Безусловный переход (JMP)**

Инструкция JMP безусловно передает управление на указанную метку или адрес памяти. Она не зависит ни от какого условия и всегда выполняет переход.

**Условные переходы**

Условные переходы передают управление на указанную метку или адрес памяти в зависимости от результата предыдущего сравнения (например, CMP).

**Примеры инструкций условного перехода**

JE (Jump if Equal): Выполняет переход, если предыдущее сравнение привело к равенству.

JNE (Jump if Not Equal): Переход, если предыдущее сравнение привело к неравенству.

JG (Jump if Greater): Переход, если в результате предыдущего сравнения первый операнд оказался больше второго операнда.

JL (Jump if Less): Переход, если в результате предыдущего сравнения первый операнд оказался меньше второго операнда.

JGE (Jump if Greater or Equal): Переход, если в результате предыдущего сравнения первый операнд оказался больше или равен второму операнду.

JLE (Jump if Less or Equal): Выполняет переход, если в результате предыдущего сравнения первый операнд меньше или равен второму операнду.

**ПРОТОКОЛ**

**Выполнение работы**

1. Согласно замечанию 5 для минимизации длины кода выполнены следующие преобразования формул:

При a > b: i1 = f1 = 20 – 4\*i

i2 = f2 = 2\*i - 2

При a <= b: i1 = f1 = 6 – 6\*i

i2 = f2 = 2 – 3\*i

Формулы для вычисления итогового результата минимизировать не представляется возможным.

При k < 0: res = f3 = min(|i1-i2|, 2)

При k >= 0: res = f3 = max(-6, -i2)

1. Создана таблица в EXEL для проверки соответствия данных, полученных по исходным и преобразованным формулам.
2. Разработана программа lr3.ASM с минимальной длиной кода. Код программы представлен в приложении А.

4)Программа протранслирована с разными значениями a, b, i, k. После каждой трансляции получен загрузочный модуль lr3.EXE.

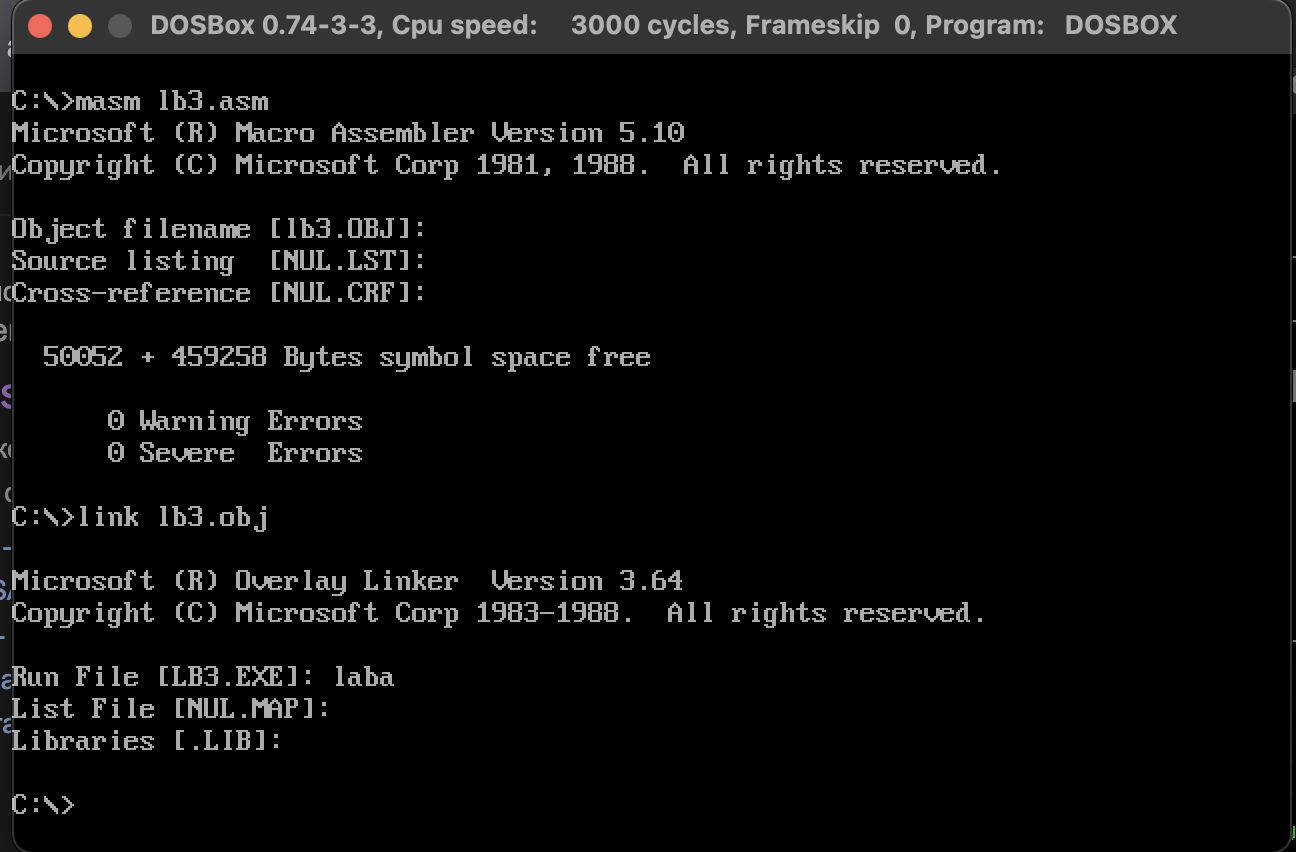
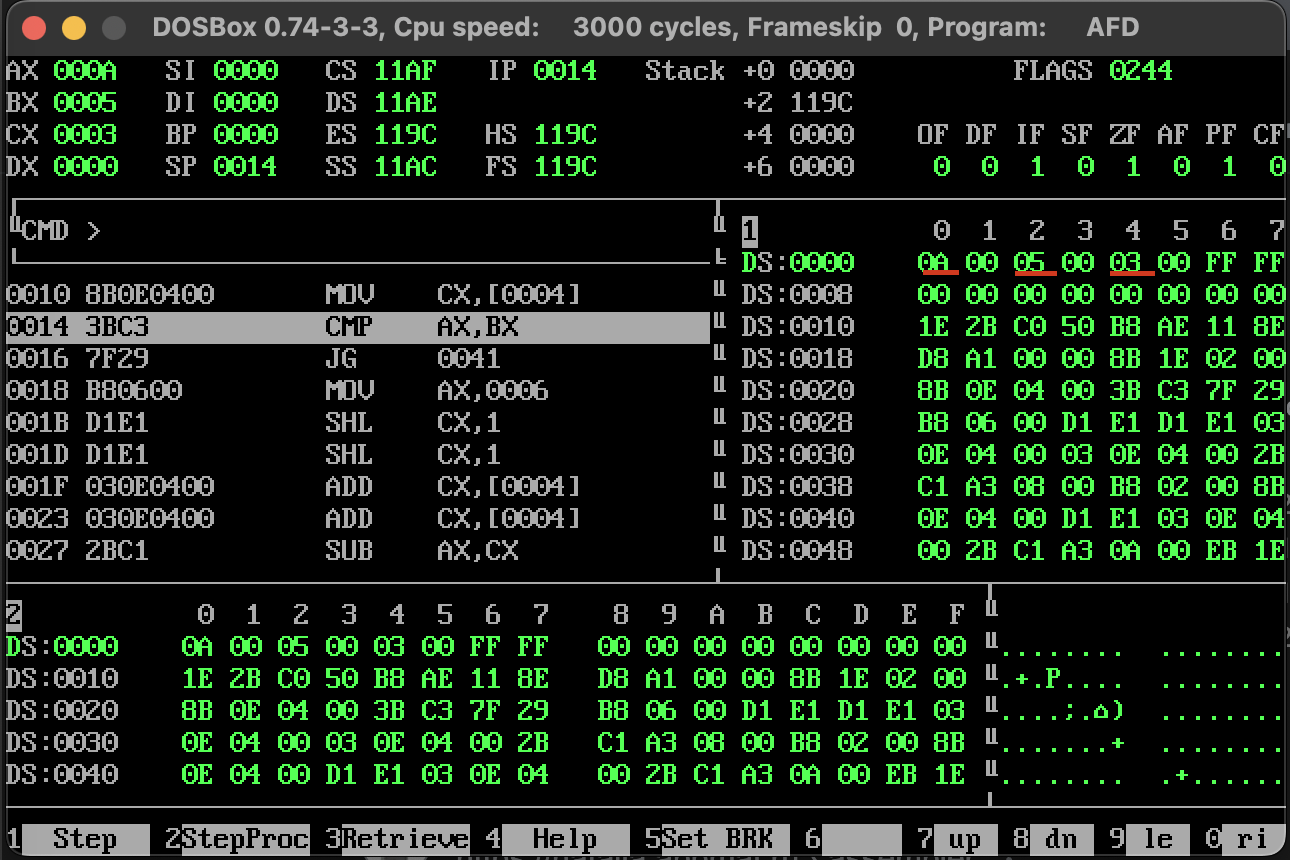
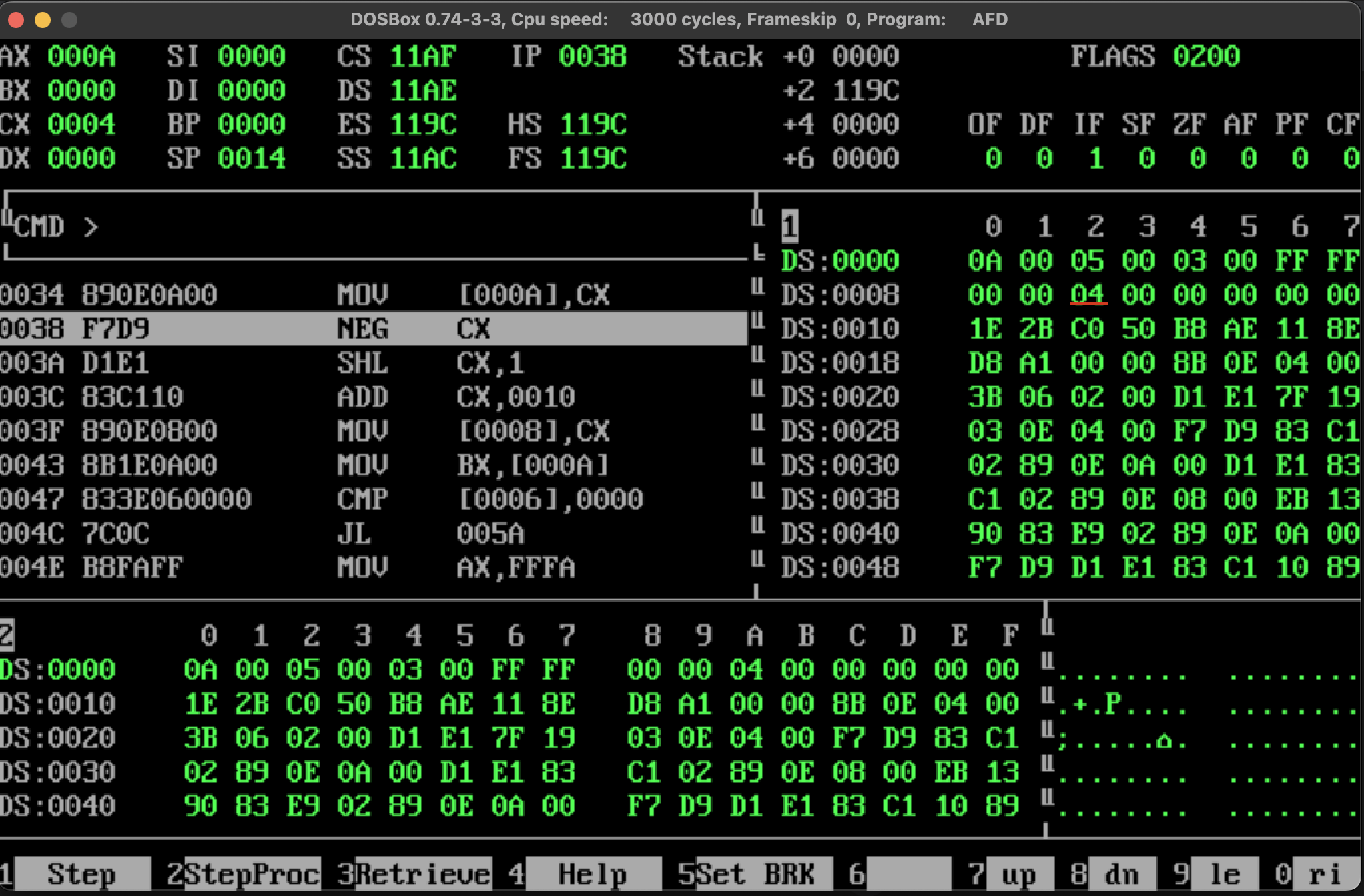
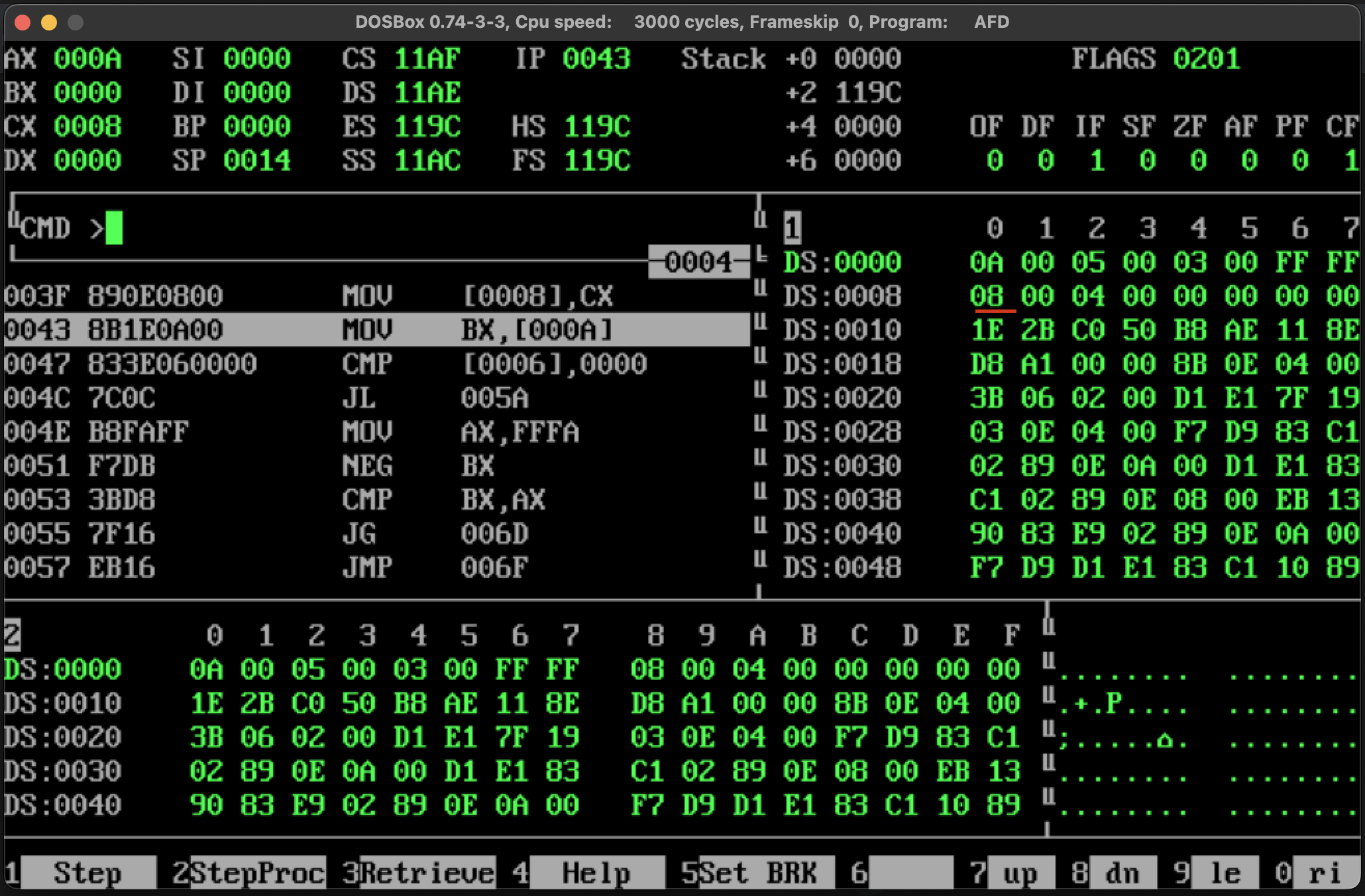


Рис. 1 – Трансляция и запуск программы

Программа выполнена в пошаговом режиме под управлением отладчика afd. На рисунках 2-5 представлены скриншоты отладчика с подчеркнутым красным результатами.







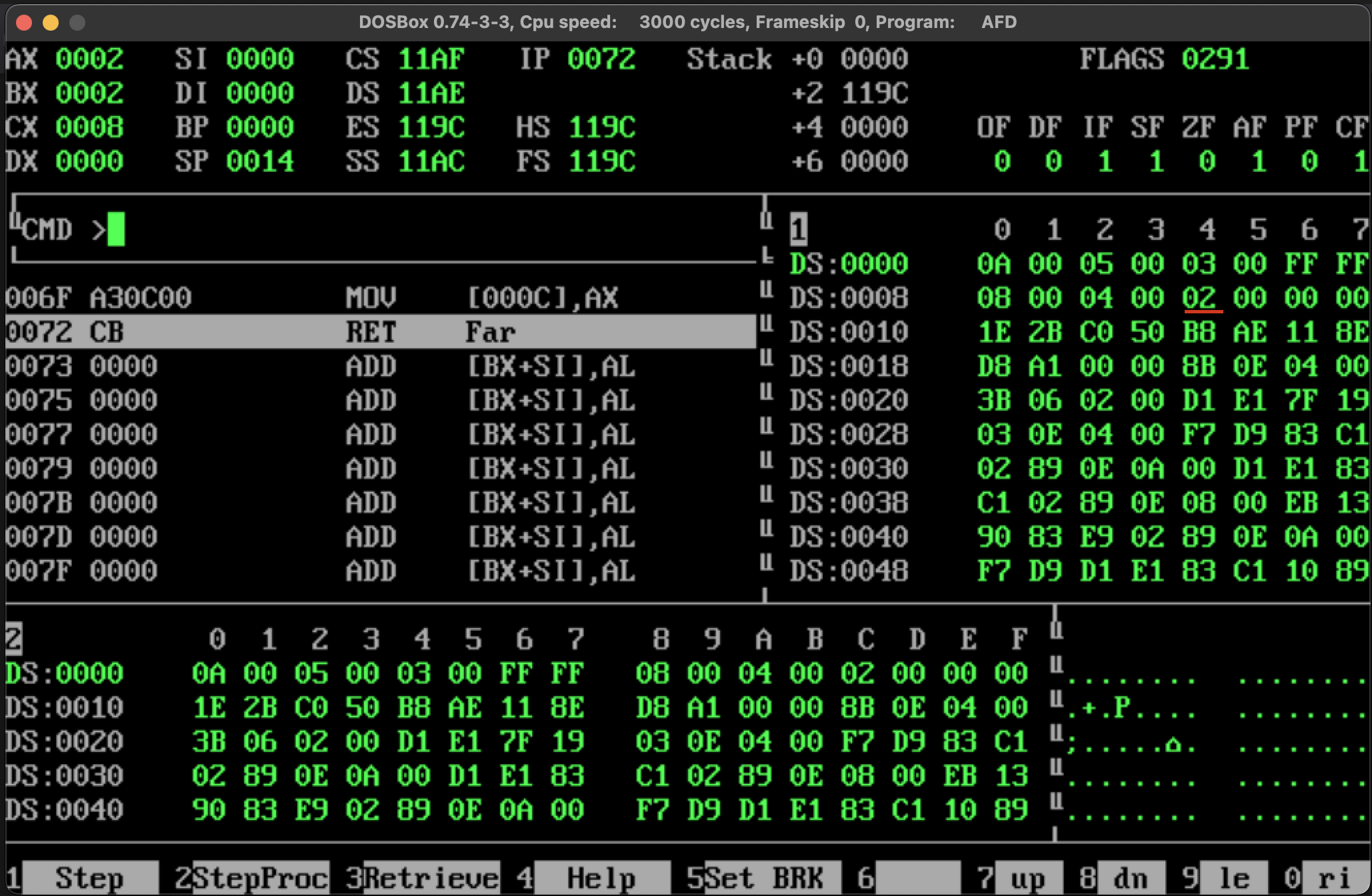


Рис. 2-5 – Тестирование на разных входных данных

Результаты тестирования программы представлены в Таблице 1.

Таблица 1 – Результаты тестирования программы

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Исходные данные | | | | Результат | | | Ожидаемый результат | | |
| a | b | i | k | i1 | i2 | res | i1 | i2 | res |
| 10 | 5 | 3 | -1 | 0008 | 0004 | 0002 | 8 | 4 | 2 |
| 10 | 5 | 1 | 0 | 0010 | 0000 | 0000 | 16 | 0 | 0 |
| 5 | 5 | 2 | 0 | FFFA | FFFC | 0004 | -6 | -4 | 4 |
| 2 | 10 | 3 | -1 | FFF4 | FFF9 | 0002 | -12 | -7 | 2 |

Замечание: если перевести полученные результаты из 16-ичной системы счисления в 10-ичную, то будет видно, что значения совпадают, то есть программа отработала корректно.

**Вывод**

В ходе выполнения лабораторной работы изучена организация ветвлений и сравнений на языке Ассемблер. Написана программа на языке Ассемблер, которая реализует расчёт значения res по заданным функциям.

**ПРИЛОЖЕНИЕ**

Исходный код написанной программы

Название файла: lb3.ASM

ASSUME CS:CODE, DS:DATA, SS:AStack

AStack SEGMENT STACK

DW 12 DUP('?')

AStack ENDS

DATA SEGMENT

var\_a DW 2

var\_b DW 10

var\_i DW 3

var\_k DW -1

var\_i1 DW 0

var\_i2 DW 0

var\_res DW 0

DATA ENDS

CODE SEGMENT

Main PROC FAR

push DS

sub AX,AX

push AX

mov AX,DATA

mov DS,AX

mov ax, [var\_a] ; закидываем в ax значение a

mov cx, [var\_i] ; закидываем в cx значение i

cmp ax, [var\_b] ; сравниваем a и b

shl cx, 1 ; cx = 2i

jg a\_b\_greater ; если a > b перейти к выполнению a\_b\_greater ; a <=b -> i1 = f1 = 6 - 6\*i : i2 = f2 = 2 - 3\*i

add cx, [var\_i] ; cx = 3i

neg cx ; cx = -3i

add cx, 2 ; cx = 2 - 3i

mov [var\_i2], cx ; перемещаем в var\_i1 cx

shl cx, 1 ; cx = 4 - 6i

add cx, 2 ; cx = 6 - 6i

mov [var\_i1], cx ; перемещаем в var\_i2 cx

jmp calc\_res\_first ; переходим к вычислению f3

a\_b\_greater: ; a > b : a > b i1 = f1 = 20 - 4\*i : i2 = f2 = 2\*i - 2

sub cx, 2 ; cx = 2i - 2

mov [var\_i2], cx ; var\_i2 = cx = 2i - 2

neg cx ; cx = 2 - 2i

shl cx, 1 ; cx = 4 - 4i

add cx, 16 ; cx = 20 - 4i

mov [var\_i1], cx ; var\_i1 = cx = 20 - 4i

calc\_res\_first: ; f3 ; k >= 0 res = f3 = max(-6, -i2)

mov bx, [var\_i2] ; bx = i2

cmp [var\_k], 0 ; сравниваем k с нулем

jl calc\_res\_second ; если k < 0 переходим к calc\_res\_second

mov ax, -6 ; ax = -6

neg bx ; bx = -i2

cmp bx, ax ; сравниваем bx с ax (-i2,-6)

jg set\_ax ; переход если bx > ax (-i2 > -6)

jmp store\_res ; переход к store\_res

calc\_res\_second: ; k < 0 res = f3 = min(|i1 - i2|, 2)

mov ax, [var\_i1] ; ax = i1

sub ax, bx ; ax = i1 - i2

module:

neg ax ; ax = -ax

js module ; если ax < 0 то переходим к module

mov bx , 2 ; bx = 2

cmp bx, ax ; сравниваем bx и ax (2, |i1-i2|)

jl set\_ax ; переходим если bx < ax (2 < |i1-i2|)

jmp store\_res ; переход к store\_res

set\_ax:

mov ax, bx ; если bx(-i2) был больше чем ax(-6) то в ax перемещаем -i2 в случае calc\_res\_first;в случае calc\_res\_second bx(2) < ax(|i1-i2|) в ax будет min(ax,bx) т.е 2

store\_res:

mov [var\_res], ax ; помещаем в ячейку памяти var\_res значение ax~f3

ret

Main ENDP

CODE ENDS

END Main