**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

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

Тема: Изучение организации ветвлений в программах на языке ассемблера

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Еникеев А.А. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2024

## Цель работы

Изучить организацию ветвелния в программе на языке ассемблера. Написать программу, которая с помощью ветвлений находит значения функций.

## Задание

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

а) 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 нужные ячейки памяти в режиме отладки. При этом следует рассмотреть все возможные комбинации параметров a, b и k, позволяющие проверить различные маршруты выполнения программы.

|  |  |
| --- | --- |
| N вар. | Ширф задания |
| 9 | 2.4.7 |

Формулы для fn1, fn2, fn3:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**│**

/-(4\*i+3), при a>b **│** /-(6\*i-4), при a>b

f2 = < **│** f4 = <

\ 6\*i-10 , при a<=b **│** \ 3\*(i+2), при a<=b

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**│**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**│**

/ (|i1|+|i2|), при k<0 **│**

f7 = < **│**

\ max(6,|i1|), при k>=0  **│**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_│\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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.

## Основные теоритические положения

**CMP**: Сравнивает содержимое двух полей данных. Фактически команда CMP вычитает второй операнд из первого, но содержимое полей не изменяет. Операнды должны иметь одинаковую длину: байт или слово. Команда CMP может сравнивать содержимое регистра, памяти или непосредственное значение с содержимым регистра; или содержимое регистра или непосредственное значение с содержимым памяти.

Флаги: Команда воздействует на флаги AF, CF, OF, PF, SF и ZF.

**TEST**: Выполняет проверку байта или слова на определенную битовую комбинацию. Команда TEST действует аналогично команде AND, но не изменяет результирующий операнд. Операнды могут иметь однобайтовые или двухбайтовые значения. Второй операнд может иметь непосредственное значение.

Флаги: Команда воздействует на флаги CF, OF, PF, SF и ZF (флаг AF

не определен)

Рассмотрим флаги в последовательности справа налево.

**CF** (Carry Flag) - флаг переноса. Содержит значение "переносов" (0 или 1) из старшего разряда при арифметических операциях и некоторых операциях сдвига и циклического сдвига.

**PF** (Parity Flag) - флаг четности. Проверяет младшие восемь бит

pезультатов операций над данными. Нечетное число бит приводит к установке этого флага в 0, а четное - в 1. Не следует путать флаг четности с битом контроля на четность.

**AF** (Auxiliary Carry Flag) - дополнительный флаг переноса. Устанавливается в 1, если арифметическая операция приводит к переносу

четвертого справа бита (бит номер 3) в регистровой однобайтовой команде.

Данный флаг имеет отношение к арифметическим операциям над символами кода ASCII и к десятичным упакованным полям.

**ZF** (Zero Flag) - флаг нуля. Устанавливается в качестве результата

aрифметических команд и команд сравнения. Как это ни странно, ненулевой

результат приводит к установке нулевого значения этого флага, а нулевой -

к установке единичного значения. Кажущееся несоответствие является,

однако, логически правильным, так как 0 обозначает "нет" (т.е. результат

не равен нулю), а единица обозначаeт "да" (т.е. результат равен нулю).

Команды условного перехода JE и JZ проверяют этот флаг.

**SF** (SIgn Flag) - знаковый флаг. Устанавливается в соответствии со

знаком результата (старшего бита) после арифметических опеpаций:

положительный результат устанавливает 0, а отрицательный - 1. Команды

условного перехода JG и JL проверяют этот флаг.

**TF** (Trap Flag) - флаг пошагового выполнения. Этот флаг вам уже

приходилось устанавливать, когда использовалась команда Т в отладчике

DEBUG. Если этот флаг установлен в единичное cостояние, то процессор

переходит в режим пошагового выполнения команд, т.е. в каждый момент

выполняется одна команда под пользовательским управлением.

**IF** (Interrupt Flag) - флаг прерывания. При нулевом состоянии этого

флага прерывания запрещены, при единичном - разрешены.

**DF** (DIrection Flag) - флаг направления. Используется в строковых

операциях для определения направления передачи данных. При нулевом

состоянии команда увеличивает содержимое регистров SI и DI, вызывая

передачу данных слева направо, при нулевом - уменьшает содержимое этих

регистров, вызывая передачу данных справа налево.

**OF** (Overflow Flag) - флаг переполнения. Фиксирует арифметическое

переполнение, т.е. перенос вниз старшего (знакового) бита при знаковых

арифметических операциях.

Табл. 1 - Условные переходы

|  |  |  |
| --- | --- | --- |
| **Переходы для беззнаковых чисел** | | |
| Мнемоника | Описание | Проверяемые флаги |
| JE/JZ | Переход, если равно/нуль | ZF |
| JNE/JNZ | Переход, если не равно/не  нуль | ZF |
| JA/JNBE | Переход, если выше/не  ниже или равно | ZF,CF |
| JAE/JNB | Переход, если выше или  равно/не ниже | CF |
| JB/JNAE | Переход, если ниже/не  выше или равно | CF |
| JBE/JNA | Переход, если ниже или равно/не выше | CF,AF |
| **Переходы для знаковых данных** | | |
| JE/JZ | Переход, если равно/нуль | ZF |
| JNE/JNZ | Переход, если не равно/не  нуль | ZF |
| JG/JNLE | Переход, если больше/не  меньше или равно | ZF,SF,OF |
| JGE/JNL | Переход, если больше или  равно/не меньше | SF,OF |
| JL/JNGE | Переход, если меньше/не  больше или равно | SF,OF |
| JLE/JNG | Переход, если меньше или  равно/не больше | ZF,SF,OF |
| **Специальные арифметические проверки** | | |
| JS | Переход, если есть знак  (отрицательно) | SF |
| JNS | Переход, если нет  знака(положительно) | SF |
| JC | Переход, если есть перенос  (аналогично JB) | CF |
| JNC | Переход, если нет переноса | CF |
| JO | Переход, если есть  переполнение | OF |
| JNO | Переход, если нет  переполнения | OF |
| JP/JPE | Переход, если паритет  четный | PF |
| JNP/JP | Переход, если паритет  нечетный | PF |

## Выполнение работы

1. Преобразуем формулы, по которым будут выполняться вычисления значений функций:

Функция fn1(a, b, i):

* Если a > b: i1 = -(4\*i + 3) = -(i << 2) - 3
* Иначе: i1 = 6\*i — 10 = (i << 2) + (i << 1) — 10

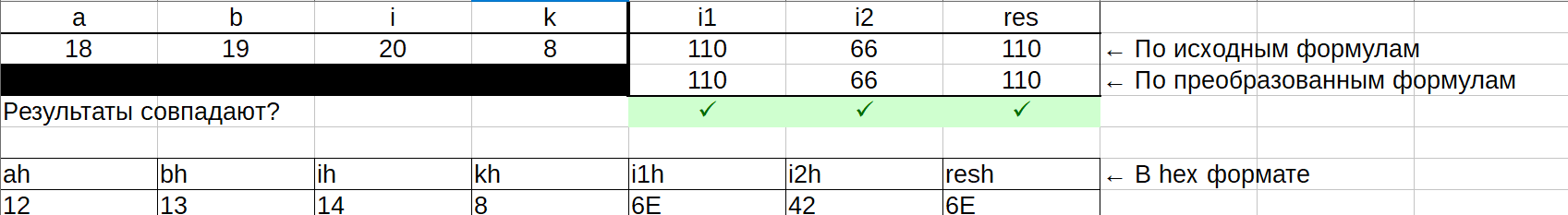
Функция fn2(a, b, i):

* Если a > b: i2 = -(6\*i — 4) = -(i << 2) - (i << 1) + 4
* Иначе: i2 = 3\*(i + 2) = (i << 1) + i + 6

Функция fn3(i1, i2, k):

* Если k < 0: res = |i1| + |i2|
* Иначе: res = max(6, |i1|)

1. Создадим документ в LibreOffice Calc для проверки преобразованных формул и значений, полученных программой, см. рис. 1.

  
Рисунок 1 - Таблица расчетов

1. Напишем программу LR3.ASM, которая предназначена для вычисления значений функций fn1, fn2, и fn3 на основе входных данных a, b, i, и k.

Краткое описание программы:

Сегмент данных (DATA SEGMENT) используется для статического выделения памяти для переменных a, b, i, k, i1, i2, и res и доступа к ним. Каждая переменная занимает одно слово (2 байта). Сегмент кода (CODE SEGMENT) содержит сам код программы, включая вычисления и управление потоком. В программе не используются процедуры или функции. Вместо этого код написан линейно, с использованием меток и условных переходов для управления потоком выполнения. Программа использует сдвиги (shl) и сложение для выполнения умножения. Условные операции реализованы с помощью сравнения и условных переходов. Программа завершает выполнение с помощью системного вызова int 21h, который инициирует выход из программы. Ввод данных и контроль результатов осуществляется в режиме отладки.

Исходный код программы см. в прил. A.

1. Результаты тестирования см.в табл. 2.

Табл. 2. - Реультаты тестирования

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Исходные данные** | | | | | **Результат** | | **Ожидаемый результат** |
| a | b | i | k | res | |  | |
| 2 | 1 | 5 | 8 | 23 | | 23 | |
| 2 | 1 | 5 | -1 | 49 | | 49 | |
| 1 | 2 | 5 | 1 | 20 | | 20 | |
| 1 | 2 | 5 | -1 | 41 | | 41 | |

**Выводы**

Были изучены ветвления на ассемблера iX86: команды сравнения и команды уловного перехода. Разработана и протестирована программа, которая находит значения функций от нескольких переменных.

# Приложение A

**Исходный код программы**

**Файл LR3.ASM:**

AStack SEGMENT STACK

DW 12 DUP(?)

AStack ENDS

DATA SEGMENT

a dw 0

b dw 0

i dw 0

k dw 0

i1 dw ?

i2 dw ?

res dw ?

DATA ENDS

CODE SEGMENT

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

start:

mov ax,DATA ; Инициализация

mov ds,ax ; cегмента DS

; Ввод данных (a, b, i, k) осуществляется в режиме отладки

; Общая часть кода для fn1 и fn2

mov ax, i ; i

shl ax, 1 ; 2 \* i

mov bx, ax ; 2 \* i -> bx

shl ax, 1 ; 4 \* i

mov dx, a ; a -> dx для сравнения

cmp dx, b ; Сравниваем a и b

jle A\_NotGreater\_B ; Если a <= b

; ---- Вычисление fn1 и fn2 (Вар. 9) ----

; fn1 = -(4\*i + 3), если a > b; 6\*i - 10, если a <= b

; fn2 = -(6\*i - 4), если a > b; 3\*(i + 2), если a <= b

A\_Greater\_B: ; a > b

;fn1

add ax, 3 ; 4 \* i + 3

neg ax ; -(4 \* i + 3)

mov i1, ax ; Сохранение i1

;fn2

sub ax, bx ; - 6 \* i - 3

add ax, 7 ; - 6 \* i + 4 = -(6 \* i - 4)

mov i2, ax ; Сохранение i2

jmp fn\_3 ; К вычислению fn3

A\_NotGreater\_B: ; a <= b

;fn1

add ax, bx ; 6\*i

sub ax, 10 ; 6\*i - 10

mov i1, ax ; Сохранение i1

;fn2

add bx, i ; 3 \* i

add bx, 6 ; 3 \* i + 6

mov i2, bx ; Сохранение i2

; ---- Вычисление fn3 ----

; fn3 = (|i1| + |i2|), если k < 0; max(6, |i1|), если k >= 0

fn\_3:

mov bx, i1 ; Получаем значение i1 -> bx

mov ax, i2 ; Получаем значение i2 -> ax

Abs\_i1:

neg bx ; Изменяем знак

js Abs\_i1 ; Если bx < 0

; Получили |i1|

cmp k, 0 ; Сравниваем k и 0

jl k\_negative ; Если k < 0

k\_positive: ; k >= 0

cmp bx, 6 ; Сравниваем bx и 6

jg store\_res ; Если bx > 6

mov bx, 6 ; Записываем 6 -> bx

jmp store\_res ; К сохранению res

k\_negative: ; k < 0

neg ax ; Меняем знак ax

js k\_negative ; Если ax < 0

; Получили |i2|

add bx, ax ; |i2| + |i1|

store\_res:

mov res, bx ; Сохранение bx -> res

; ---- Завершение программы ----

mov ah, 4Ch

int 21h

CODE ENDS

end start

; 34 команды в главной части