



Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Комп’ютерний практикум №2

Системне програмне забезпечення

Тема: Засоби обміну даними

Виконав

студент групи ІІІ-11:

Панченко С. В.

Перевірив:

Лісовиченко О. І

“6” березня 2023 р.

Київ 2023

ЗМІСТ

1	Мета комп'ютерного практикуму.....	6
2	Завдання.....	7
3	Текст програми.....	8
4	Схема функціонування програми.....	16
5	Приклад виконання.....	20
6	Висновок.....	22
6.1	Представлення дійсних чисел у пам'яті комп'ютера.....	22
6.1.1	Цілі числа.....	22
6.1.2	Реальні числа.....	22
6.1.3	Символи.....	22
6.2	Вектори переривання, їх розташування у пам'яті.....	23
6.3	Особливості виконання команд множення MUL та IMUL.....	23
6.3.1	Інструкція MUL.....	23
6.3.2	Інструкція IMUL.....	23
6.4	Особливості виконання команд ділення DIV та IDIV.....	24
6.4.1	Інструкція DIV.....	24
6.4.2	Інструкція IDIV.....	24
6.5	Стек, і як він працює.....	25
6.6	Команда організації циклів loop та особливості її виконання.....	25

1 МЕТА КОМП'ЮТЕРНОГО ПРАКТИКУМУ

Навчитись застосовувати засоби обміну даними в асемблері.

2 ЗАВДАННЯ

1. Написати програму з використанням 2-х процедур

- Процедура введення і перетворення цілого числа. Після цього треба виконати математичну дію над числом (номер завдання вибирати за останніми двома числами номеру в заліковій книжці - Таблиця 2.1).
- Процедура переведення отриманого результату в рядок та виведення його на екран.

2. Програма повинна мати захист від некоректного введення вхідних даних (символи, переповнення, ділення на 0 і т.і.).


```
mov bl, byte [rax]

mov byte [rax], NEW_LINE_CHARACTER
mov rdi, 1

call WriteToConsole

mov byte [rax], bl
pop rbx
pop rdi
ret
;=====
<<
;=====
<<
ClearBuffer:
; Function clearing buffer
; void ClearBuffer(char* buffer, int length);
; Params:
;     rax:    char*   buffer
;     rdi:    int     length
; Returns:
;     void
push rcx
xor rcx, rcx

.loop:
    cmp rcx, rdi
    jbe .End
    mov byte [rax + rcx], 0
    jmp .loop
.End:
pop rcx
ret
;=====
<<
;=====
<<
TryConvertNumberToString:
; Function converting integer to string;
; bool TryConvertNumberToString(char* buffer, int bufferSize, int
inputtedLength, int number);
; Params:
;     rax:    char*   buffer
;     rdi:    int     bufferSize
;     rsi:    int&    inputtedLength
;     rdx:    int     number
; Returns:
;     r8:     bool    if true, no error, else thrown error
;
pushf
push rbx
push rcx
push r8
push r9

xor r9, r9
xor r8, r8
; counter = 0
mov rcx, 0
cmp rdx, 0
jge .ReadingNumbersIntoStack

.CheckForNegative:
    mov r9, 1
    neg rdx
    mov byte [rax], MINUS_SIGN
```


[illegible]

[illegible]

```

jl .NoError

mov bl, byte [rax + r9]

; if(IsNeLineCharacter()) {
;     break;
; }
.IsNewLineCharacter:
    cmp bl, NEW_LINE_CHARACTER
    je .NoError

; if(sign=='-' || sign=='+') {
;     if(index!=0) {
;         return Error;
;     }
;     if(sign=='-') {
;         *number *= -1;
;     }
;
;     --rcx;
;     ++r9;
;     continue;
; }
.CheckForSigns:
    .IsPlusCharacter:
        cmp bl, PLUS_SIGN
        je .CheckForSignBeingFirst
    .IsMinusCharacter:
        cmp bl, MINUS_SIGN
        je .OnEqualMinus
    jmp .CallIsDigit
    .OnEqualMinus:
        mov r10, 1
    .CheckForSignBeingFirst:
        cmp r9, 0
        jne .ErrorSignNotFirst
    jmp .OnIterationEnd

.CallIsDigit
    push rax
    push rdi

    xor rax, rax
    mov al, bl
    call IsDigit
    mov r8, rdi

    pop rdi
    pop rax

cmp r8, 0
je .ErrorIncorrectSymbol
sub bl, DIGIT_ZERO

.CallPow
    push rax
    push rdi
    push rsi

    mov rax, 10
    mov rdi, rcx
    call Pow

    imul rsi, rbx
    mov rdi, rdx
    add rdi, rsi
    mov rdx, rdi

```

[illegible]

4 СХЕМА ФУНКЦІОНУВАННЯ ПРОГРАМИ

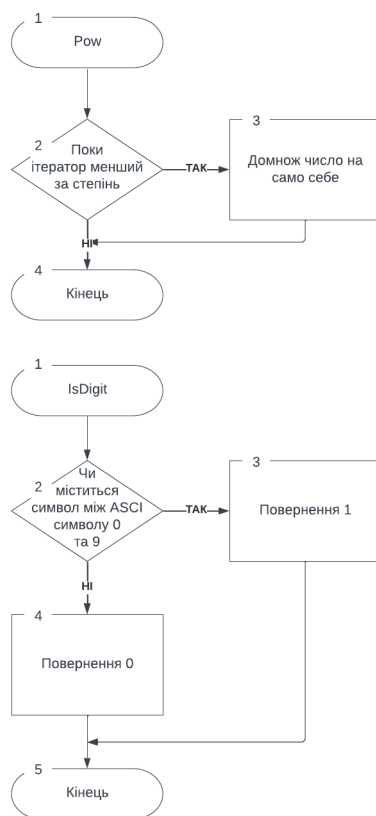


Рисунок 4.1 — схема функцій Pow та IsDigit

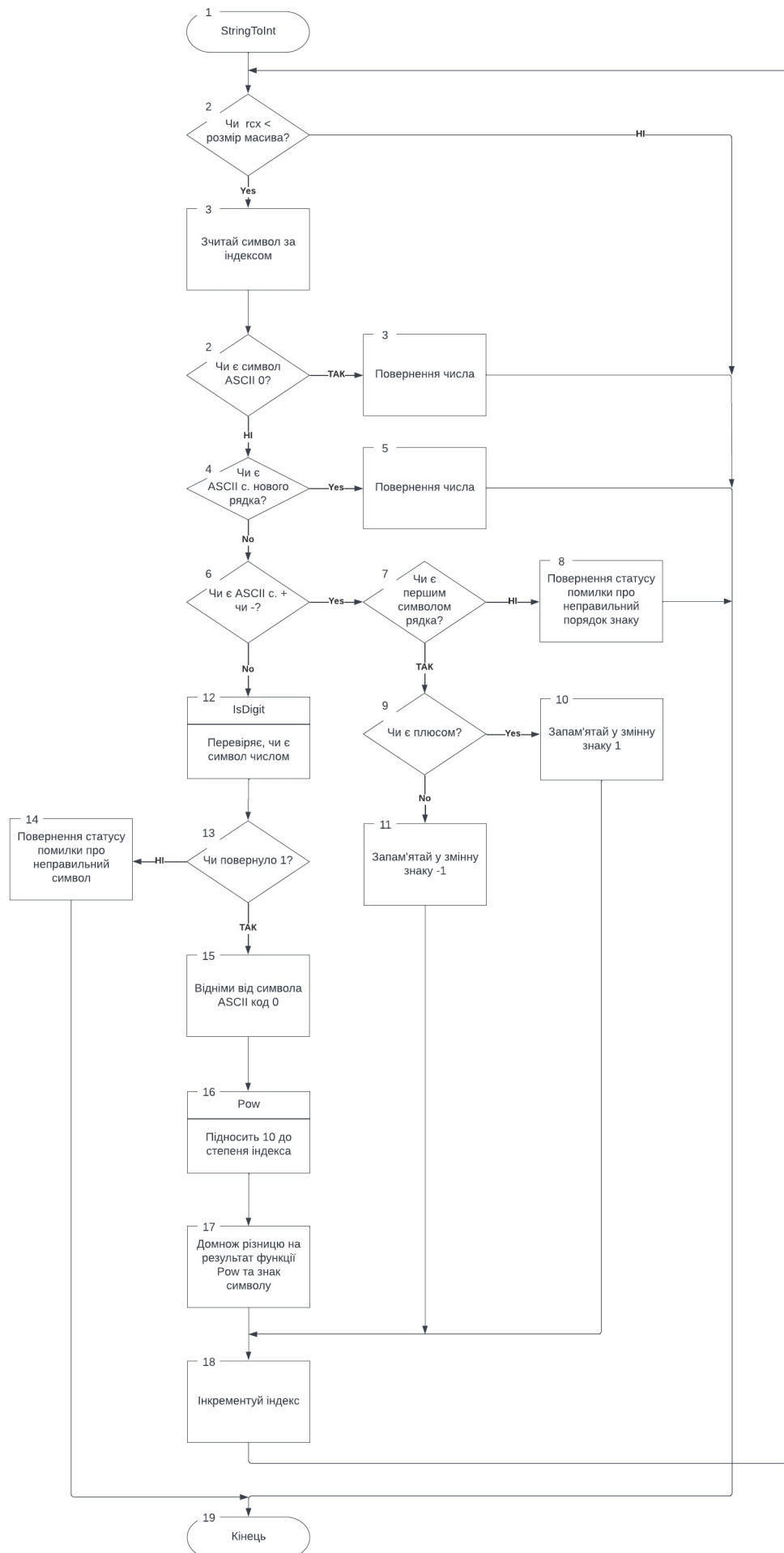


Рисунок 4.2 — схема функції StringToInt

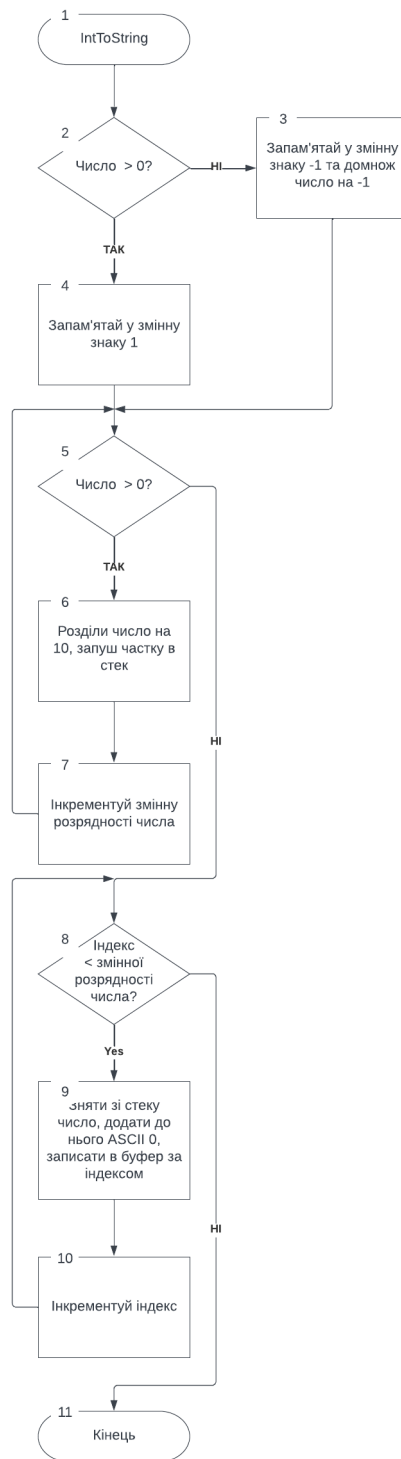


Рисунок 4.3 — схема функції IntToStr

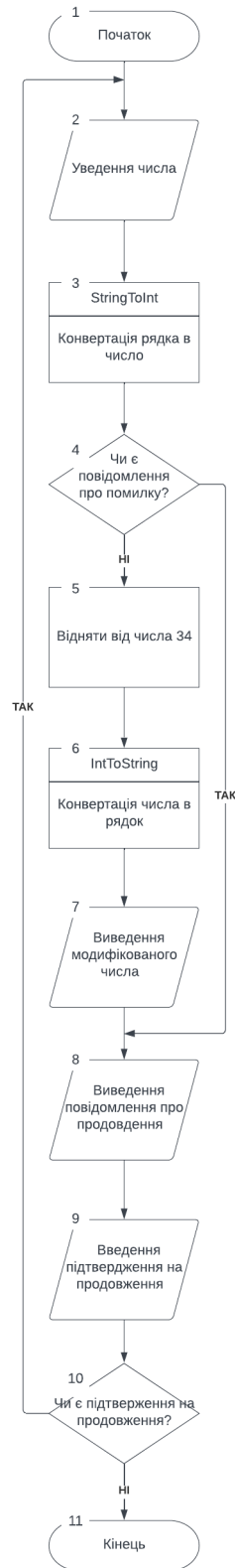


Рисунок 4.4 — схема функції Main

5 ПРИКЛАД ВИКОНАННЯ

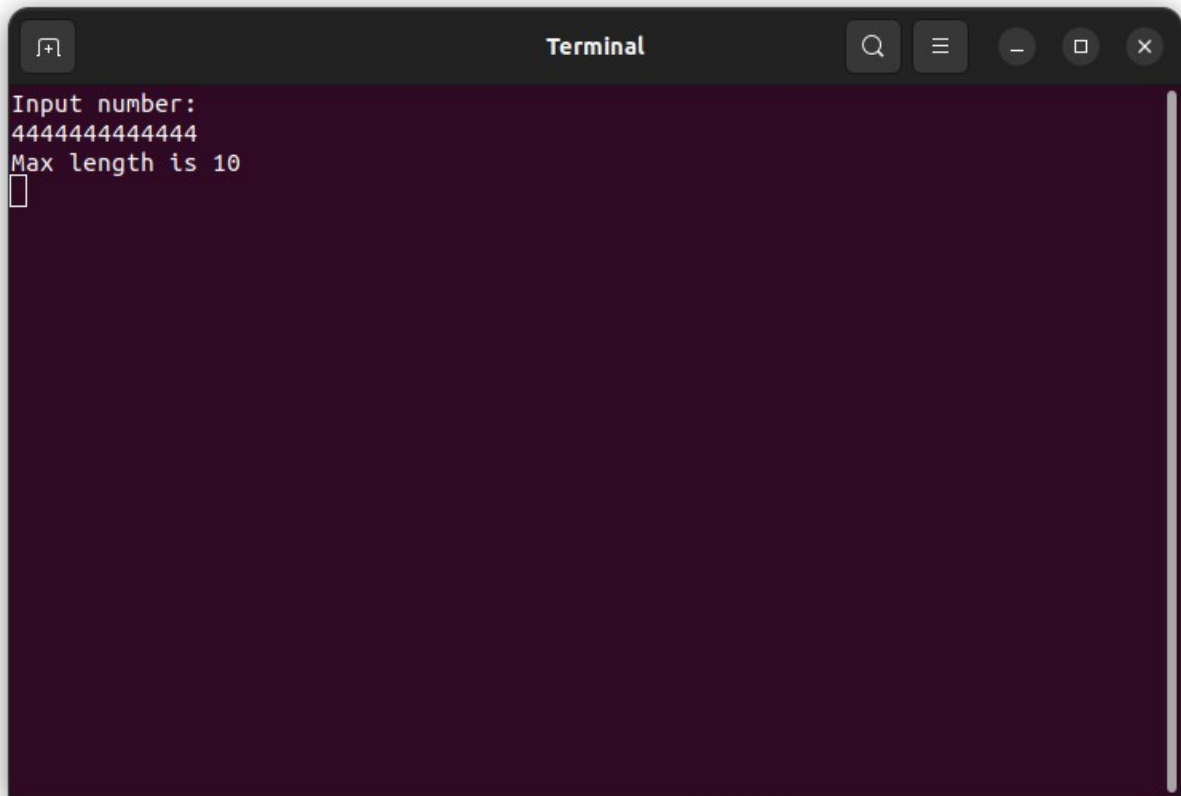


Рисунок 5.1 — Виведення повідомлення про переповнення буфера

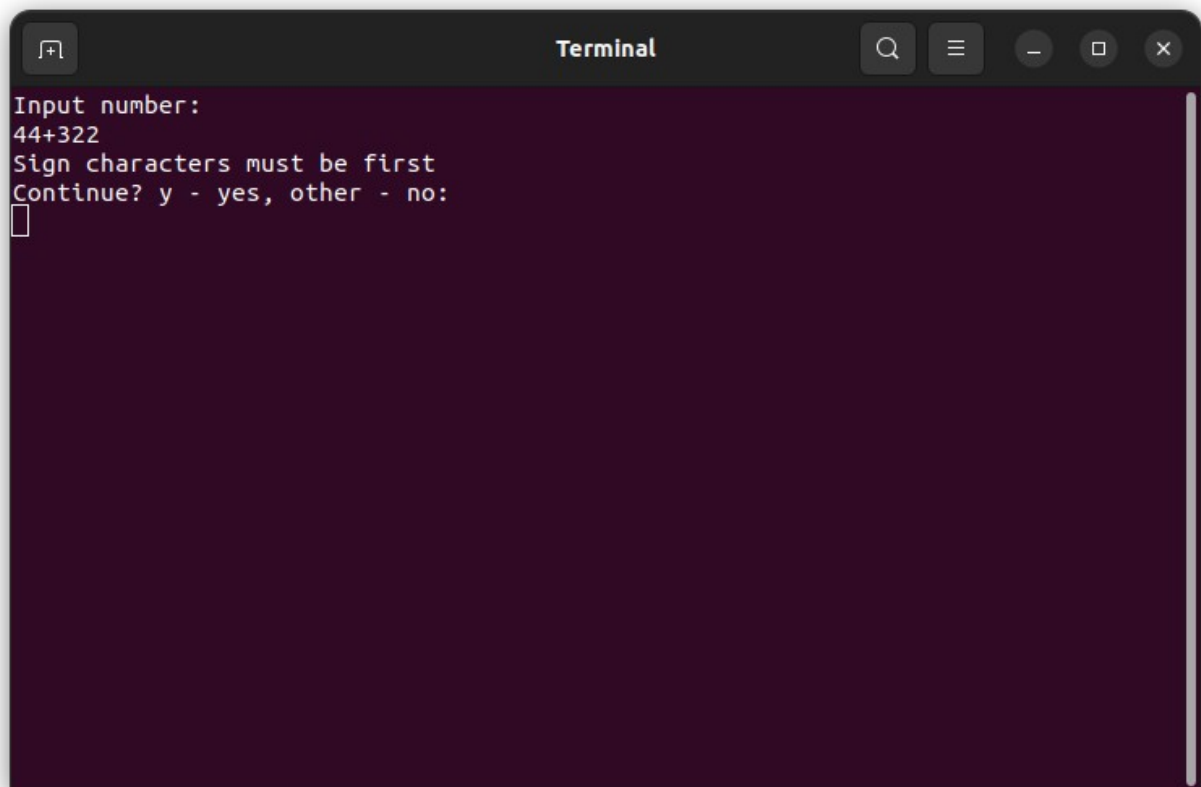
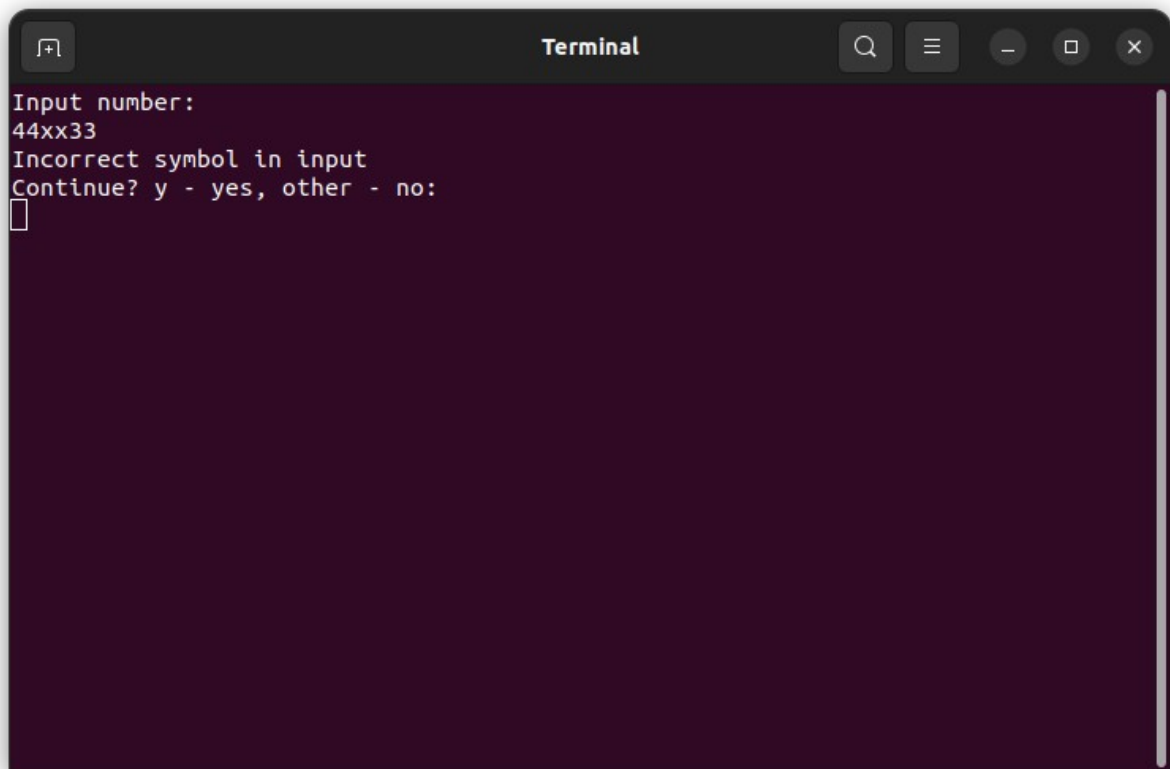


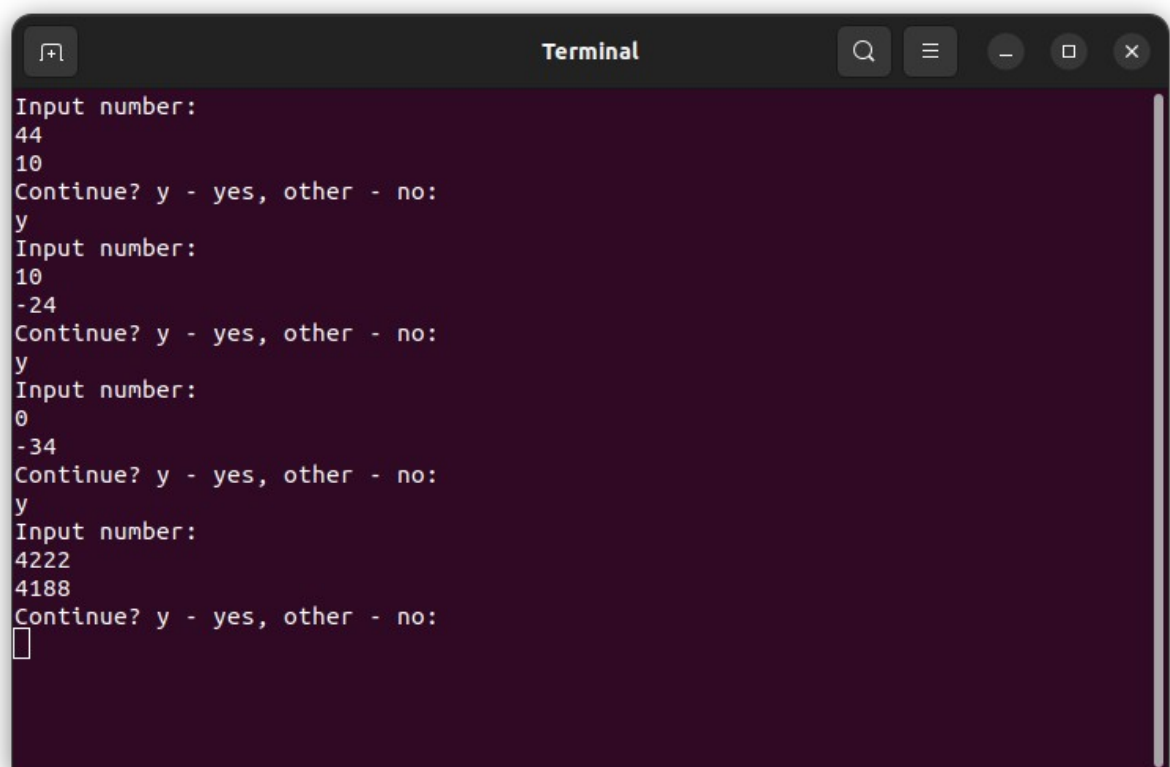
Рисунок 5.2 — Виведення повідомлення про неправильне розташування знаку



```
Terminal
Input number:
44xx33
Incorrect symbol in input
Continue? y - yes, other - no:

```

Рисунок 5.3 — Виведення повідомлення про некоректний символ



```
Terminal
Input number:
44
10
Continue? y - yes, other - no:
y
Input number:
10
-24
Continue? y - yes, other - no:
y
Input number:
0
-34
Continue? y - yes, other - no:
y
Input number:
4222
4188
Continue? y - yes, other - no:

```

Рисунок 5.4 — Виведення результату та повідомлення про запит на повторне введення числа

6 ВИСНОВОК

6.1 Представлення дійсних чисел у пам'яті комп'ютера

6.1.1 Цілі числа

Цілі числа зазвичай представлені в пам'яті комп'ютера за допомогою фіксованої кількості двійкових цифр або «бітів». Кількість бітів, які використовуються для представлення цілого числа, визначає діапазон значень, які можна зберегти. Наприклад, 32-розрядне ціле число може представляти значення від -2 147 483 648 до 2 147 483 647. Найпоширенішим представленням цілих чисел є система доповнення до двох, що дозволяє виконувати ефективні арифметичні операції.

6.1.2 Реальні числа

Реальні числа зазвичай представлені в пам'яті комп'ютера за допомогою представлення з плаваючою комою. Це передбачає розбиття числа на компоненти експоненти та мантиси та збереження їх в окремих бітових полях. Стандарт IEEE 754 є найбільш поширеним форматом представлення чисел з плаваючою комою в сучасних комп'ютерах. Цей стандарт визначає кілька різних форматів, включаючи одинарну точність (32 біти) і подвійну точність (64 біти), які використовуються для представлення дійсних чисел із різним ступенем точності.

6.1.3 Символи

Символи, такі як літери, знаки пунктуації та інші спеціальні символи, зазвичай представлені в пам'яті комп'ютера за допомогою кодів символів. Існує багато різних схем кодування символів, але найпоширенішою є ASCII (американський стандартний код для обміну інформацією), яка призначає унікальний 7-бітний код кожному символу. Розширені версії ASCII, такі як Юнікод, використовують більші коди (до 32 бітів) для підтримки більшого діапазону символів із різних систем письма.

Загалом, представлення даних у пам'яті комп'ютера – це складна тема, яка залежить від конкретної архітектури та дизайну комп'ютера. Однак наведені вище описи надають загальний огляд того, як зазвичай представлені цілі числа, дійсні числа

та символи.

6.2 Вектори переривання, їх розташування у пам'яті.

Вектори переривань - це механізм, який використовується комп'ютерами для обробки різних типів запитів на переривання, наприклад апаратних або програмних переривань. Вектор переривання - це вказівник на область пам'яті, де знаходиться код обробки конкретного переривання. Коли спрацьовує переривання, процесор шукає відповідний вектор переривання, щоб знайти розташування коду обробника переривань. Розташування векторів переривань у пам'яті залежить від архітектури комп'ютера та операційної системи, що використовується. Однак у більшості випадків вектори переривань зберігаються у виділеному розділі пам'яті, відомому як таблиця векторів переривань (IVT).

6.3 Особливості виконання команд множення MUL та IMUL

У мові асемблера x86 є дві основні інструкції множення: MUL та IMUL. Інструкція MUL використовується для множення без знаку, тоді як інструкція IMUL використовується для множення зі знаком.

6.3.1 Інструкція MUL

Бере один операнд, який є операндом джерела. Помножує вихідний операнд на вміст регістра AX (для 8- або 16-розрядних операндів) або регістрів DX:AX (для 32-розрядних операндів). Результат зберігається в регістрі AX (для 8- або 16-бітних операндів) або регістрах DX:AX (для 32-бітових операндів).

Якщо результат множення не поміщається в регістрі призначення, генерується виняток.

6.3.2 Інструкція IMUL

Бере один операнд, який є операндом джерела. Помножує вихідний операнд на вміст регістра AX (для 8- або 16-бітних операндів) або регістрів EDX:EAX (для 32-бітових операндів). Результат зберігається в регістрі AX (для 8-розрядних операндів),

регістрах DX:AX (для 16-розрядних операндів) або регістрах EDX:EAX (для 32-розрядних операндів). Інструкція IMUL може виконувати як знакове, так і беззнакове множення. Для множення зі знаком результат розширюється за знаком до розміру регістра призначення.

Якщо результат множення не поміщається в регістрі призначення, генерується виняток.

6.4 Особливості виконання команд ділення DIV та IDIV

Інструкція DIV використовується для беззнакового ділення, тоді як інструкція IDIV використовується для ділення зі знаком.

6.4.1 Інструкція DIV

Бере один операнд, який є дільником. Розділяє вміст регістра AX (для 8- або 16-розрядних операндів) або регістрів DX:AX (для 32-розрядних операндів) на дільник. Частка зберігається в регістрі AX (для 8-розрядних або 16-розрядних операндів) або регістрах DX:AX (для 32-розрядних операндів). Залишок зберігається в регістрі AH (для 8-розрядних операндів) або регістрі DX (для 16-розрядних або 32-розрядних операндів). Якщо дільник дорівнює 0 або частка не вміщується в регістрі призначення, генерується виняток.

6.4.2 Інструкція IDIV

Бере один операнд, який є дільником. Розділяє вміст регістра AX (для 8-розрядних операндів), регістрів DX:AX (для 16-розрядних операндів) або регістрів EDX:EAX (для 32-розрядних операндів) на дільник. Частка зберігається в регістрі AX (для 8-розрядних операндів), регістрах DX:AX (для 16-розрядних операндів) або регістрах EDX:EAX (для 32-розрядних операндів). Залишок зберігається в регістрі AH (для 8-розрядних операндів), регістрі DX (для 16-розрядних операндів) або регістрі EDX (для 32-розрядних операндів). Інструкція IDIV може виконувати як знакове, так і беззнакове ділення. Для ділення зі знаком приватне розширюється за знаком до розміру регістра призначення. Якщо дільник дорівнює 0 або частка не вміщується в

реєстрі призначення, генерується виняток.

6.5 Стек, і як він працює

На мові асемблера стек — це тип пам'яті, яка використовується для зберігання тимчасових даних під час виконання програми. Він працює як набір елементів, де в будь-який час доступний лише верхній елемент. Це відоме як принцип «Останній прийшов, перший вийшов», що означає, що остання частина даних, додана до стеку, першою буде видалена.

Керування стеком здійснюється за допомогою спеціального реєстра, який називається покажчиком стека. Коли програма запускається, покажчик стека встановлюється на вершину стека. Під час виклику функції її аргументи та адреса повернення надсилаються в стек, а коли функція завершується, стек витягується, щоб повернути дані функції, що викликає.

6.6 Команда організації циклів loop та особливості її виконання

У мові асемблера цикли можуть бути реалізовані за допомогою команди організації циклу, яка є типом інструкції переходу. Команда організації циклу дозволяє повторювати частину коду задану кількість разів без необхідності створення кількох копій коду. Вона зазвичай використовується в поєднанні зі змінною лічильника, яка використовується для відстеження кількості виконання циклу.

```
mov cx, 5
.loop:
    dec cx
    cmp cx, 0
    jl .loop
```