Зображення, що містить текст

Автоматично згенерований опис

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 №3**

Системне програмне забезпечення

**Тема:** Програмування розгалужених алгоритмів

Виконав Перевірив:

студент групи ІП-11: Лісовиченко О. І

Панченко С. В. “7” березня 2023 р.

Київ 2023

Зміст

[1 Мета комп’ютерного практикуму 6](#__RefHeading___Toc3319_204808571)

[2 Завдання 7](#__RefHeading___Toc5937_3437555080)

[3 Текст програми 8](#__RefHeading___Toc4957_2404284615)

[4 Схема функціонування програми 17](#__RefHeading___Toc4963_2404284615)

[5 Приклад виконання 24](#__RefHeading___Toc2239_1691502584)

[6 Висновок 28](#__RefHeading___Toc4971_2404284615)

[6.1 Команда безумовного переходу та її особливості 28](#__RefHeading___Toc2131_2579377654)

[6.2 Команди умовного переходу 28](#__RefHeading___Toc2133_2579377654)

[6.3 Команда порівняння CMP 28](#__RefHeading___Toc2135_2579377654)

[6.4 Як здійснити умовний перехід на відстань, більшу за 128 байт? 29](#__RefHeading___Toc2137_2579377654)

# Мета комп’ютерного практикуму

Навчитись програмувати розгалужені алгоритми.

# Завдання

1. Можливість введення користувачем значень x, y, t, a, b за необхідності.
2. Обчислювати значення функції за введеними значеннями.
3. Виводити на екран результат обчислень.
4. Якщо є ділення, то результат дозволяється виводити:

* як дійсне число (наприклад: = 1,666667 ) – підвищена складність;
* окремо цілу частину та остачу (наприклад: = 1 остача 2 ) – середня складність;
* окремо цілу частину та остачу як дріб – середня складність.

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

# Текст програми

bits 64

; list of system calls

; https://filippo.io/linux-syscall-table/

SYS\_READ equ 0

SYS\_WRITE equ 1

; Descriptors

STDIN equ 0

STDOUT equ 1

; ASCII characters

NULL\_TERMINATOR equ 0

NEW\_LINE\_CHARACTER equ 10

PLUS\_SIGH equ 43

MINUS\_SIGN equ 45

DIGIT\_ZERO equ 48

DIGIT\_NINE equ 57

N\_LETTER equ 110

Y\_LETTER equ 121

; Other constants

BUFFER\_LENGTH equ 20

MAX\_LENGTH equ 10

section .data

; Errors

error\_incorrect\_symbol: db "Incorrect symbol in input", NEW\_LINE\_CHARACTER, 0

error\_incorrect\_symbol\_length: equ $-error\_incorrect\_symbol

error\_sign\_character\_not\_first db "Sign characters must be first", NEW\_LINE\_CHARACTER, 0

error\_sign\_character\_not\_first\_length equ $-error\_sign\_character\_not\_first

input\_a\_msg db "Input number:", NEW\_LINE\_CHARACTER, 0

input\_a\_msg\_length equ $-input\_a\_msg

continue\_msg db "Continue? y - yes, other - no:", NEW\_LINE\_CHARACTER, 0

continue\_msg\_length equ $-continue\_msg

max\_length\_error db "Max length is 10", NEW\_LINE\_CHARACTER, 0

max\_length\_error\_length equ $-max\_length\_error

error\_overflow db "Overflow", NEW\_LINE\_CHARACTER, 0

error\_overflow\_length equ $-error\_sign\_character\_not\_first

; Buffers

buffer: times BUFFER\_LENGTH db 0

inputtedLength: dq 0

section .text

global asm\_main

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

asm\_main:

push rax

push rdi

push rsi

push rdx

push r8

.loop:

mov rax, input\_a\_msg

mov rdi, input\_a\_msg\_length

call WriteToConsole

xor rdx, rdx

mov rax, buffer

mov rdi, BUFFER\_LENGTH

mov rsi, inputtedLength

call ReadIntoBuffer

mov rsi, [rsi]

call TryConvertStringToInteger

cmp r8, 0

je .Repeat

sub rdx, 34

call ClearBuffer

call TryConvertNumberToString

mov rdi, rsi

call WriteToConsole

call PrintEndl

.Repeat:

mov rax, continue\_msg

mov rdi, continue\_msg\_length

call WriteToConsole

mov rax, buffer

mov rdi, BUFFER\_LENGTH

call ClearBuffer

mov rsi, inputtedLength

call ReadIntoBuffer

xor rbx, rbx

mov bl, byte [rax]

cmp bl, Y\_LETTER

jne .End

jmp .loop

.End:

pop r8

pop rdx

pop rsi

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintEndl:

; Function printing enl

; void PrintEndl(char\* const buffer);

; Params:

; rax: char\* buffer

; Returns:

; void

push rdi

push rbx

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 bufferLength, int inputtedLength, int number);

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; rsi: int& inputtedLength

; rdx: int number

; Returns:

; r8: bool if true, no error, else throwed 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

.ReadingNumbersIntoStack:

; The idea behind this is to read number into stack

; For example, 123 into stack like "3","2","1"

; and we counted digits. In this, example count = 3

; so we need to do smth like that:

; while(index<count) {

; pop stack into var

; var = var + ZERO\_CODE

; \*buffer[index] = var

; ++index

; }

; copy value to rbx

mov rbx, rdx

.loop:

cmp rbx, 0

jle .ReadingNumbersFromStackToBuffer

.ReadDigit:

push rax

push rdx

; rax\_rbx\_copy = rbx;

mov rax, rbx

; rdx = 0; rbx = 10

xor rdx, rdx

mov rbx, 10

; rax\_rbx\_copy, rdx\_remaindex = rax\_rbx\_copy / rbx

idiv rbx

; r8 = rdx\_remainder; rbx = rax\_rbx\_copy

mov r8, rdx

mov rbx, rax

pop rdx

pop rax

push r8

inc rcx

jmp .loop

.ReadingNumbersFromStackToBuffer:

xor rbx, rbx

xor r8, r8

cmp r9, 0

je .Preparation

.IncrementIfNegative:

inc r8

inc rcx

.Preparation:

mov rsi, rcx

.loop2:

cmp r8, rcx

jge .NoError

pop rbx

add rbx, DIGIT\_ZERO

mov [rax + r8], bl

inc r8

jmp .loop2

.NoError:

pop r9

pop r8

pop rcx

pop rbx

popf

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

WriteToConsole:

; Function writing string into STDOUT

; void WriteToConsole(char\* buffer, int bufferLength)

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; Returns:

; void

push rax

push rdi

push rsi

push rdx

mov rsi, rax

mov rdx, rdi

mov rax, SYS\_WRITE

mov rdi, STDOUT

call DoSystemCallNoModify

pop rdx

pop rsi

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

ReadIntoBuffer:

; Function reading string into buffer;

; void ReadIntoBuffer(char\* buffer, int bufferLength, int\* inputtedLength);

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; rsi: int\* inputtedLength

; Returns:

; void

push rax

push rdi

push rdx

push r8

mov qword [rsi], 0

mov r8, rsi

.loop:

mov rsi, rax

mov rdx, rdi

mov rax, SYS\_READ

mov rdi, STDIN

call DoSystemCallNoModify

cmp rax, MAX\_LENGTH

jle .NoError

mov rax, max\_length\_error

mov rdi, max\_length\_error\_length

call WriteToConsole

jmp .loop

.NoError:

mov rsi, r8

mov [rsi], rax

pop r8

pop rdx

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

DoSystemCallNoModify:

; Function doing system call without

; modifying rcx and r11 registers after the call.

; type(rax) sys\_call(rax, rdi, rsi, rdx, r8, r9...);

; The reason behind this function is that in x64 NASM

; system call neither stores nor loads any registers

; it just uses and modifies them.

; https://stackoverflow.com/questions/47983371/why-do-x86-64-linux-system-calls-modify-rcx-and-what-does-the-value-mean

; http://www.int80h.org/bsdasm/#system-calls

; https://docs.freebsd.org/en/books/developers-handbook/x86/#x86-system-calls

pushf

push rcx

push r11

syscall

pop r11

pop rcx

popf

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

TryConvertStringToInteger:

; Function converting string to integer;

; bool TryConvertStringToInteger(char\* buffer, int bufferLength, int inputtedLength, int\* number);

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; rsi: int inputtedLength

; rdx: int& number

; Returns:

; r8: bool if true, no error, else throwed error

;

pushf

push rbx

push rcx

push r9

push r10

xor r10, r10

xor r9, r9

mov rdx, 0

mov rcx, rsi

xor rbx, rbx

xor r8, r8

dec rcx

cmp byte [rax + rcx], NEW\_LINE\_CHARACTER

jne .loop

dec rcx

.loop:

cmp rcx, 0

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\_SIGH

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

pop rsi

pop rdi

pop rax

; whole\_digit = digit\*(10^counter)

.OnIterationEnd:

dec rcx

inc r9

jmp .loop

.ErrorSignNotFirst:

mov r8, 0

push rax

push rdi

mov rax, error\_sign\_character\_not\_first

mov rdi, error\_sign\_character\_not\_first\_length

call WriteToConsole

pop rdi

pop rax

jmp .End

.ErrorIncorrectSymbol

; print error message

mov r8, 0

push rax

push rdi

mov rax, error\_incorrect\_symbol

mov rdi, error\_incorrect\_symbol\_length

call WriteToConsole

pop rdi

pop rax

jmp .End

.NoError:

cmp r10, 1

jne .GeneralNoError

push rax

neg rax

mov rdx, rax

pop rax

.GeneralNoError:

mov r8, 1

jmp .End

.End:

pop r10

pop r9

pop rcx

pop rbx

popf

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

IsDigit:

; Function checking whether byte value

; is in digit codes range

; bool IsDigit(char c);

; Params:

; rax: char c

; Returns:

; rdi: bool if true, then it is digit, else not

pushf

cmp al, DIGIT\_ZERO

jl .False

cmp al, DIGIT\_NINE

jg .False

jmp .True

.False:

mov rdi, 0

jmp .End

.True:

mov rdi, 1

jmp .End

.End:

popf

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Pow:

; Function powing number to a certain degree.

; Params:

; rax: int number

; rdi: int degree

; Returns:

; rsi: Powed number

pushf

push rcx

mov rsi, 1

xor rcx, rcx

.loop:

cmp rcx, rdi

jge .End

imul rsi, rax

inc rcx

jmp .loop

.End:

pop rcx

popf

ret

# Схема функціонування програми

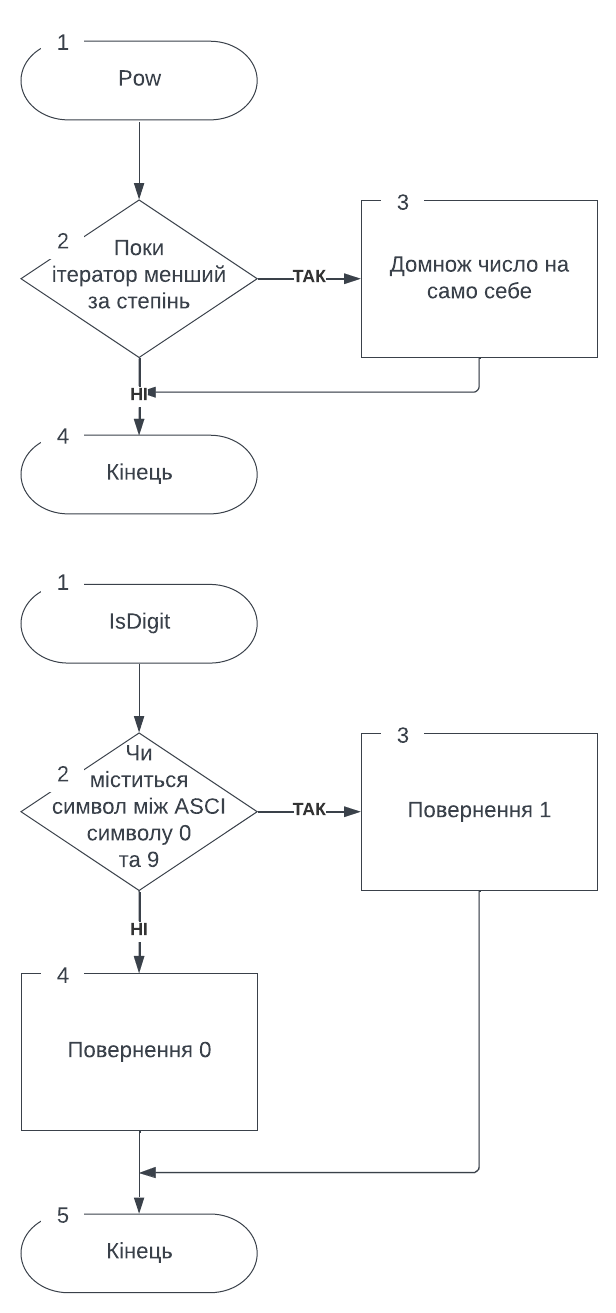


Рисунок 4.1 — схема функцій Pow та IsDigit

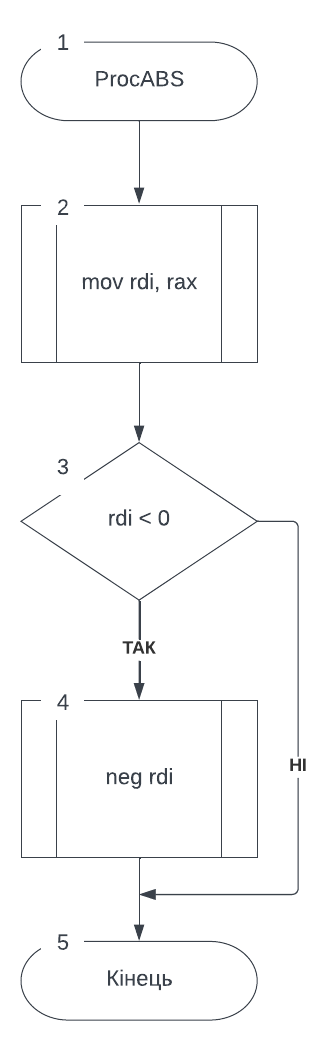


Рисунок 4.2 — схема функції ProcABS

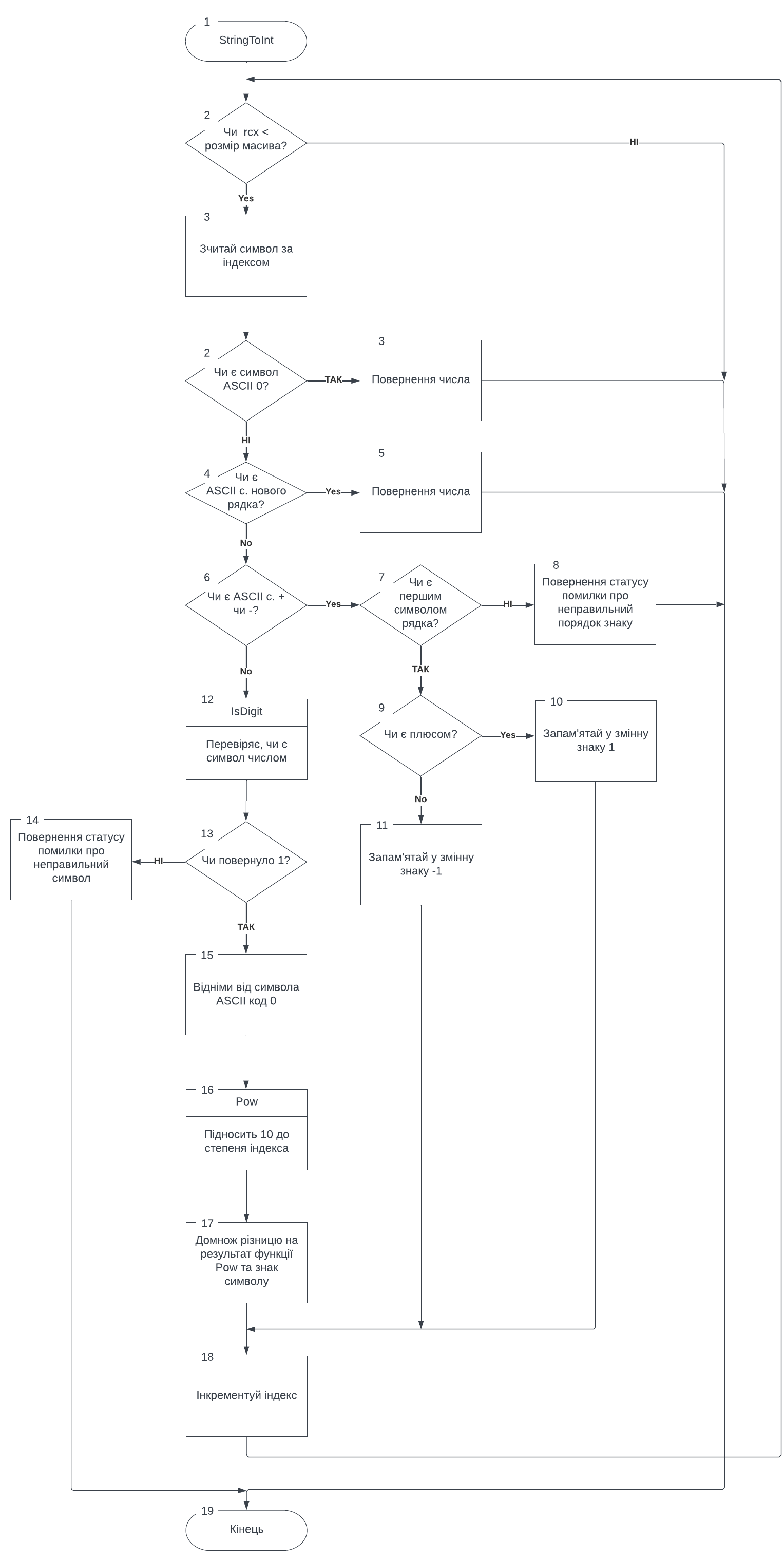


Рисунок 4.3 — схема функції StringToInt

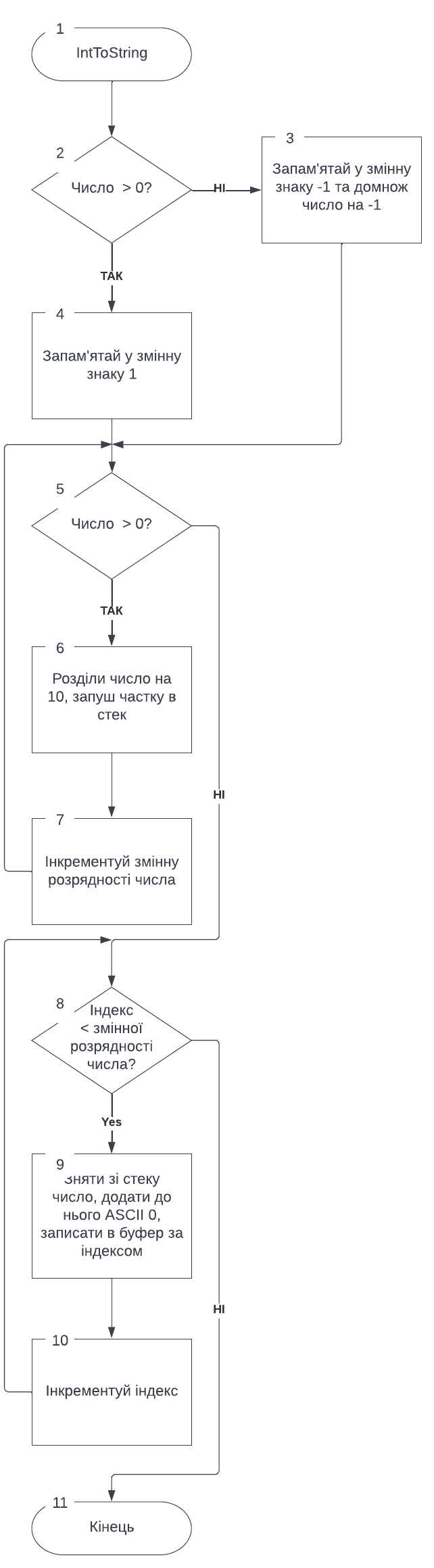


Рисунок 4.4 — схема функції IntToStr

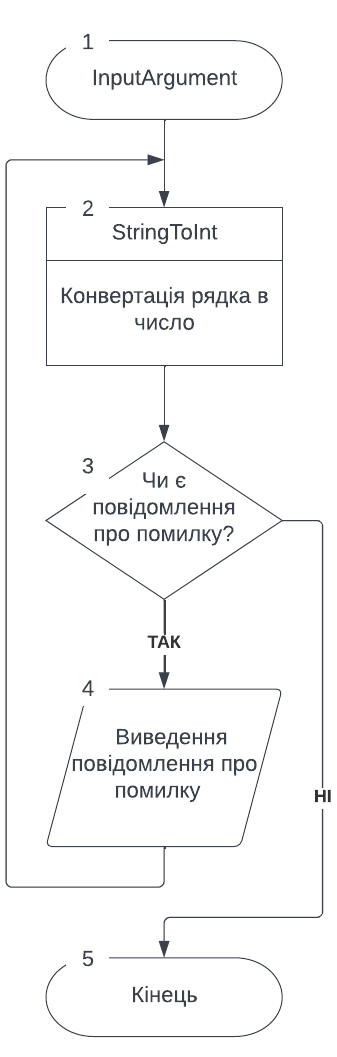


Рисунок 4.5 — схема функції InputArgument

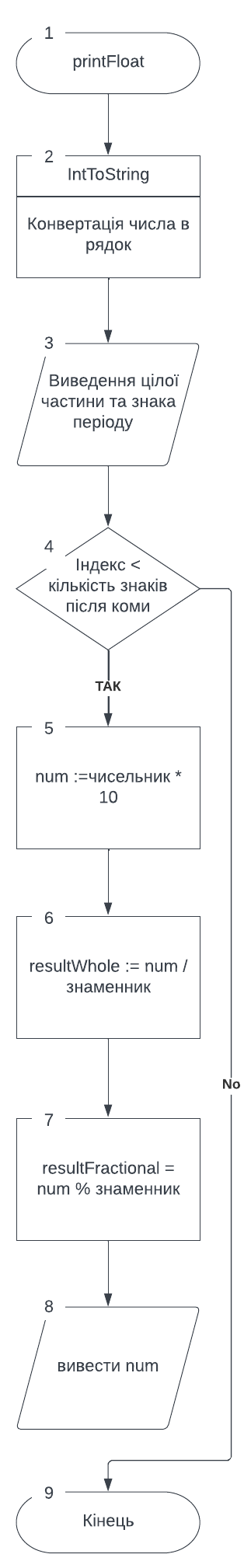


Рисунок 4.6 — схема функції printFloat

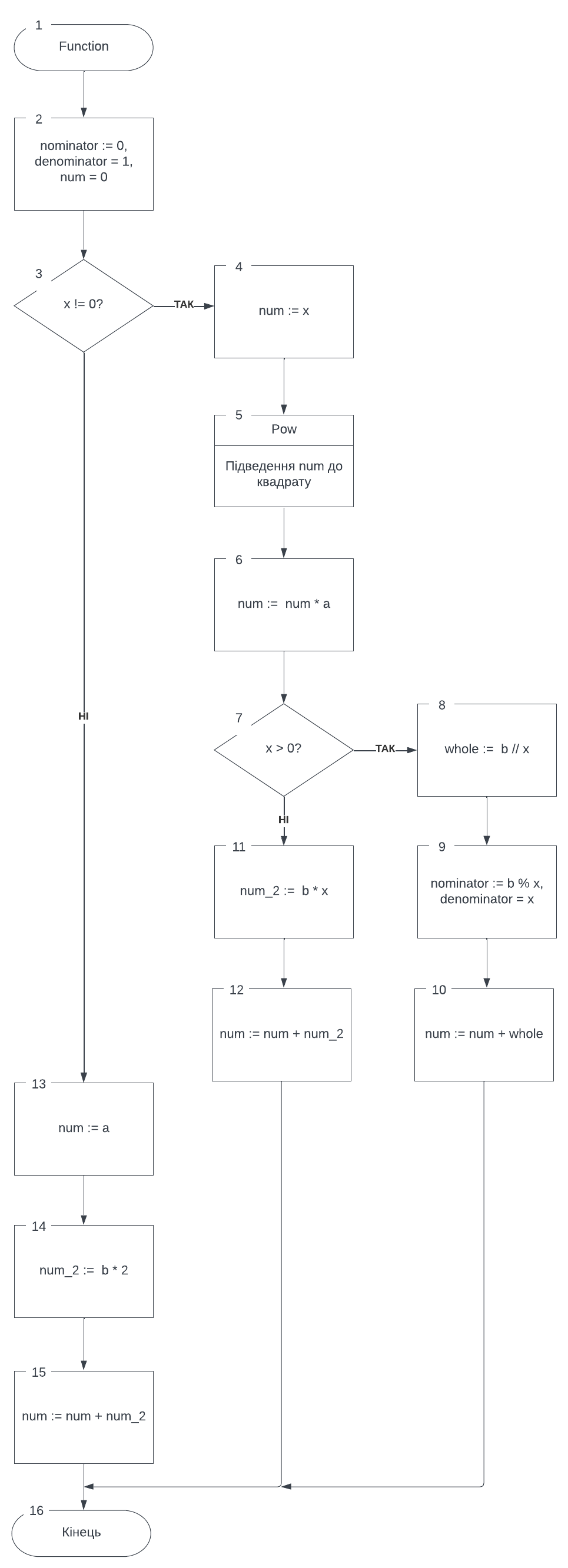


Рисунок 4.7 — схема функції Function

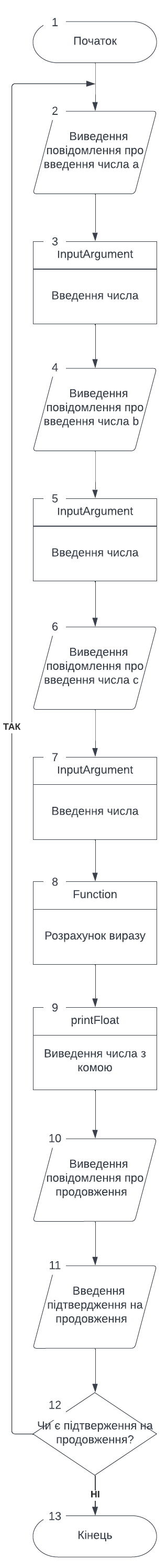


Рисунок 4.7 — схема функції Main

# Приклад виконання

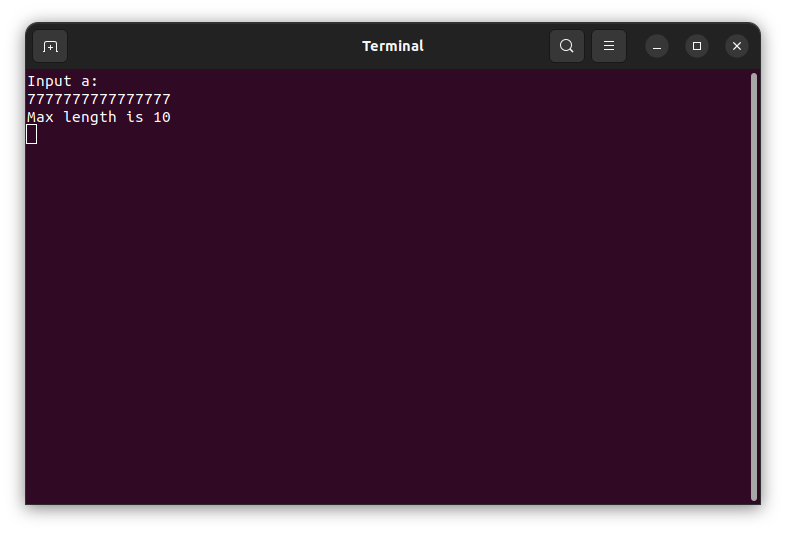


Рисунок 5.1 — виведення повідомлення про переповнення буфера

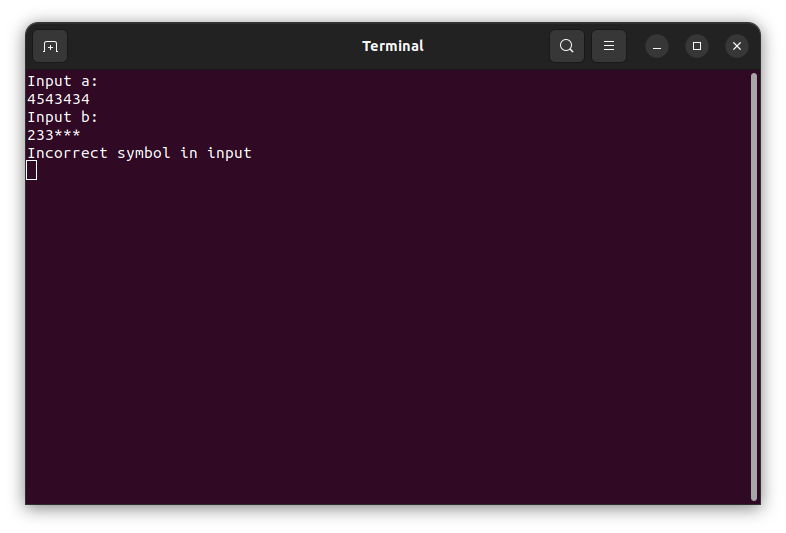


Рисунок 5.2 — виведення повідомлення про некоректний символ

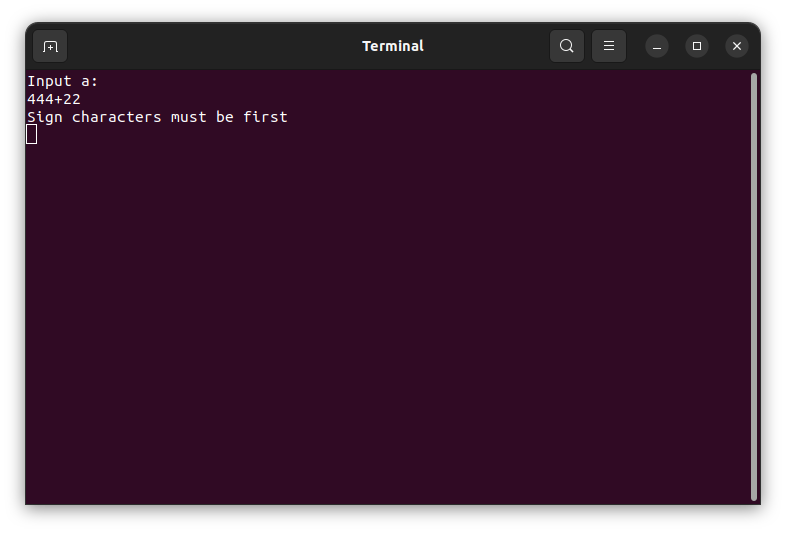


Рисунок 5.3 — виведення повідомлення про неправильну позицію знака

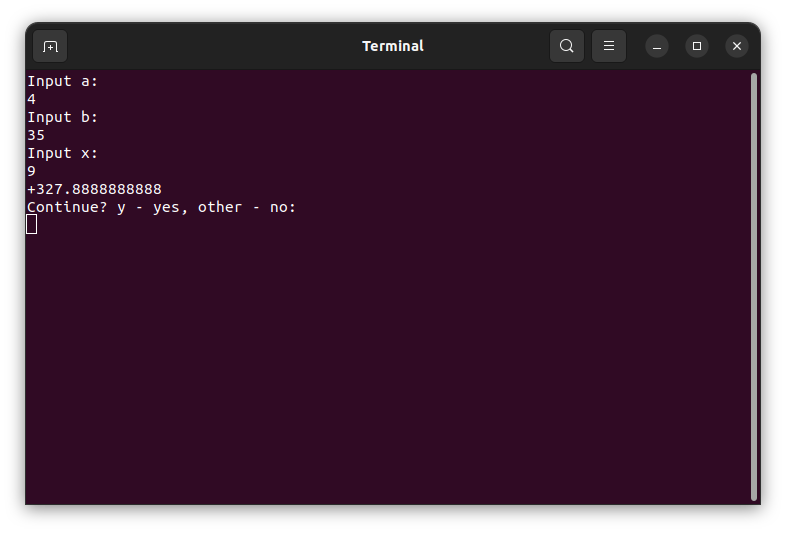


Рисунок 5.4 — виведення результату при x>0

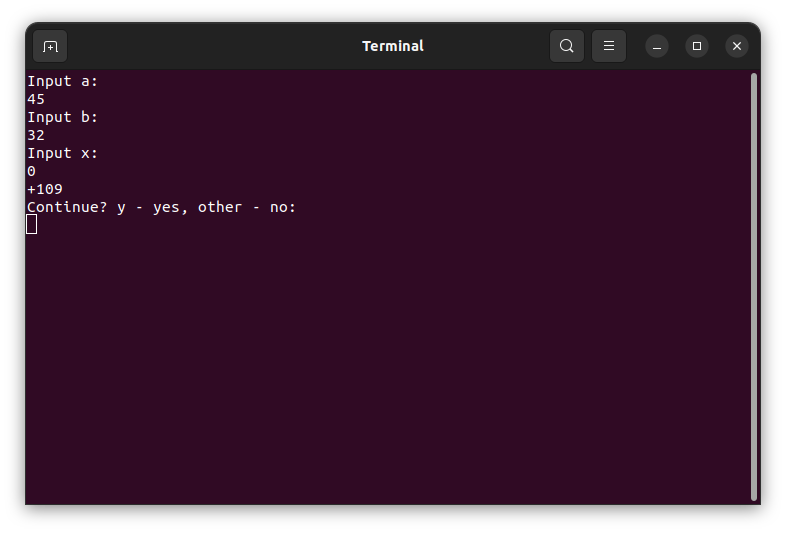


Рисунок 5.5 — виведення результату при x=0

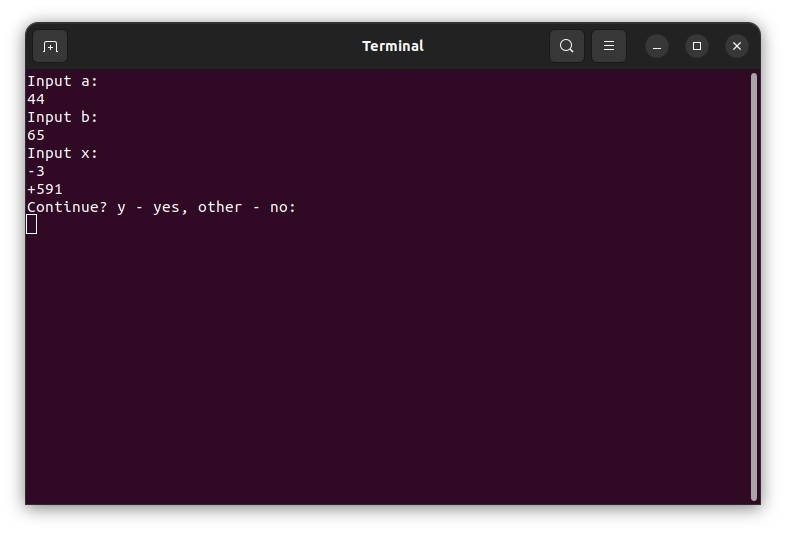


Рисунок 5.6 — виведення результату при x<0

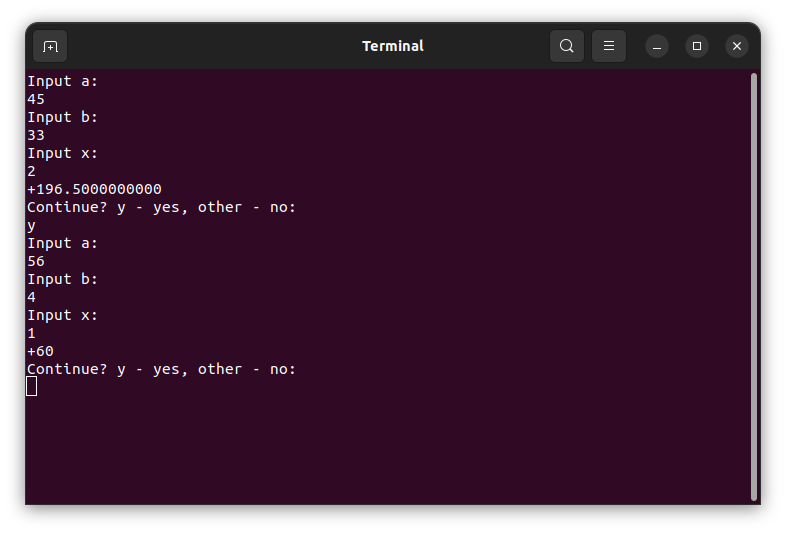


Рисунок 5.7 — виведення повідомлення про запит на продовження

# Висновок

## Команда безумовного переходу та її особливості

У NASM (Netwide Assembler) безумовні переходи — це інструкції, які передають керування іншій частині програми без будь-яких умов чи обмежень. Вони дозволяють програмі переходити до вказаної адреси пам’яті або мітки, незалежно від будь-яких попередніх інструкцій чи умов. Найпоширенішою інструкцією безумовного переходу в NASM є інструкція JMP. Ця інструкція приймає один операнд, який може бути адресою пам’яті або міткою, визначеною в програмі. Коли інструкція JMP виконується, процесор передасть управління в область пам'яті, визначену операндом. Безумовні переходи можуть бути корисними для реалізації циклів, реалізації викликів функцій або переходу до різних частин програми на основі введення користувача або інших умов. Однак вони також можуть ускладнити читання та налагодження коду, особливо якщо їх використовувати надмірно.

## Команди умовного переходу

У NASM (Netwide Assembler) умовні переходи — це інструкції, які передають керування іншій частині програми на основі певної умови. Вони дозволяють програмі перевірити певну умову, а потім перейти до вказаної адреси пам’яті або мітки, якщо умова виконується. Умовні переходи корисні для реалізації потоку керування в програмах. Вони дозволяють програмі виконувати різні шляхи коду на основі значення регістра, розташування пам’яті чи іншої умови.

## Команда порівняння CMP

У NASM (Netwide Assembler) інструкція CMP (порівняння) використовується для порівняння двох операндів і встановлення прапорів процесора на основі результату порівняння. Інструкція CMP не змінює жодного операнда, вона лише оновлює регістр прапорів на основі порівняння. Прапори, які встановлюються інструкцією CMP, залежать від результату віднімання. Якщо результат дорівнює нулю, встановлюється позначка нуля (ZF). Якщо результат негативний, встановлюється позначка (SF). Якщо результат переповнюється, встановлюється прапор переповнення (OF). Якщо результат вимагає запозичення або перенесення, встановлюється прапор перенесення (CF).

## Як здійснити умовний перехід на відстань, більшу за 128 байт?

Якщо вам потрібно виконати умовний стрибок на відстань більше 128 байт, можна використовувати комбінацію інструкцій JMP і міток для створення багаторівневого переходу. Використовуючи кілька міток і інструкцій JMP, можливо ефективно переходити до інструкції, яка знаходиться далі, ніж 128 байт від поточної інструкції.

cmp eax, 1

je section1

cmp eax, 2

je section2

cmp eax, 3

je section3

jmp end

section1:

jmp end

section2:

jmp end

section3:

jmp end

end: