# Міністерство освіти і науки України Національний технічний університет України "Київський політехнічний інститут" Факультет інформатики та обчислювальної техніки Кафедра інформатики та програмної інженерії

# 3ВІТ про виконання лабораторної роботи №3 з дисципліни " СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ"

Прийняв доцент кафедри IПI Лісовиченко О.І. "…" ………… 20хх р. Виконав Студент 2 курсу групи IП-11 Панченко Сергій

### Комп'ютерний практикум №3

### Варіант 19

Тема: засоби обміну даними. Завдання:

19. 
$$Z = \begin{cases} ax^2 + b / x & \text{якщо } x > 0 \\ a + 2b & \text{якщо } x = 0 \\ ax^2 - bx & \text{якщо } x < 0 \end{cases}$$

Написати програму, яка повинна мати наступний функціонал:

- 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:
- а) як дійсне число (наприклад: = 1,666667) підвищена складність;
- б) окремо цілу частину та остачу (наприклад: = 1 остача 2 ) середня складність;
- в) окремо цілу частину та остачу як дріб (наприклад: = 1 ) середня складність.
- 5. 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
PERIOD equ 46
DIGIT_ZERO equ 48
```

```
DIGIT_NINE
                 eau 57
   Other constants
BUFFER_LENGTH equ 20
MAX_LENGTH equ 10
N LETTER
                  equ 110
Y_LETTER
                  equ 121
; Imported functions
extern CFunction
section .data
   Errors
                                           db
                                              "Incorrect symbol in input",
error_incorrect_symbol:
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
max_length_error db "Max length is 10", NEW_LINE_CHARACTER, 0
max_length_error_length equ $-max_length_error
input_a_msg db "Input a:", NEW_LINE_CHARACTER, 0
input_a_msg_length equ $-input_a_msg
input_b_msg db "Input b:", NEW_LINE_CHARACTER, 0
input_b_msg_length equ $-input_b_msg
input_x_msg db "Input x:", NEW_LINE_CHARACTER, 0
input_x_msg_length equ $-input_x_msg
continue_msg db "Continue? y - yes, other - no:", NEW_LINE_CHARACTER, 0
continue_msg_length equ $-continue_msg
   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
   push rbx
   .loop:
       .inputA:
          mov rax, input_a_msg
          mov rdi, input_a_msg_length
          call WriteToConsole
          mov rax, buffer
          mov rdi, BUFFER_LENGTH
```

```
call InputArgument
    push rsi
.inputB:
   mov rax, input_b_msg
   mov rdi, input_b_msg_length
   call WriteToConsole
   mov rax, buffer
   mov rdi, BUFFER_LENGTH
    call InputArgument
    push rsi
.inputX:
   mov rax, input_x_msg
   mov rdi, input_x_msg_length
   call WriteToConsole
   mov rax, buffer
   mov rdi, BUFFER_LENGTH
   call InputArgument
    push rsi
.OnLoopEnd:
    pop rdx
    pop rsi
   pop rdi
   xor rax, rax
   call CFunction
   mov rax, rdi
   mov rdi, rsi
   mov rsi, rdx
   call Function
   mov rax, buffer
mov rdi, BUFFER_LENGTH
    cmp r11, 1
    je .printPlus
        .printMinus:
            call PrintMinus
            jmp .printFractional
        .printPlus:
            call PrintPlus
            jmp .printFractional
    .printFractional:
        mov rsi, r8
        mov rdx, r9
        mov r8, r10
        mov r9, 10
        call printFloat
        call PrintEndl
        call ClearBuffer
        mov rax, continue_msg
        mov rdi, continue_msg_length
        call WriteToConsole
```

```
xor rbx, rbx
      mov bl, byte [rax]
      cmp bl, Y_LETTER
      jne .End
      jmp .loop
      .End:
         pop rbx
         pop r8
         pop rdx
         pop rsi
         pop rdi
         pop rax
         ret
<<<<<<
>>>>>>>>
InputArgument:
; void (rax-buffer, rdi-bufferLength, rsi- int&_out number)
   push rax
   push rdi
   push rdx
   push r8
   push r9
   push r10
   mov r9, rax
   mov r10, rdi
   xor rsi, rsi
   .loop:
      mov rax, r9
      mov rdi, r10
      call ReadIntoBuffer
      call TryConvertStringToInteger
      call ClearBuffer
      cmp r8, 0
      je .loop
   call ClearBuffer
   mov rsi, rdx
   pop r10
   pop r9
   pop r8
   pop rdx
   pop rdi
   pop rax
   ret
```

mov rax, buffer
mov rdi, BUFFER\_LENGTH

call ReadIntoBuffer

```
<<<<<<
<<<<<<
procABS:
   ; rdi Abs(rax);
   mov rdi, rax
   cmp rdi, 0
   jge .End
       neg rdi
   .End:
       ret
printFloat:
      Function printing float;
      bool TryConvertNumberToString(char* buffer, int bufferLength, int whole,
int numerator, int denominator, int precision);
      Params:
                 char*
                         buffer
         rax:
         rdi:
                 int
                         bufferLength
                 int
                         whole
         rsi:
                 int
                         numerator
         rdx:
         r8:
                 int
                        denominator
                 int
         r9:
                         precision
      Returns:
          void
 print whole
;int resultWhole = numerator / denominator;
 int resultFractional = numerator % denominator;
  if (0 < precision) {</pre>
    cout << resultWhole << '.';</pre>
    for (int i = 0; i < precision; ++i)
      int num = resultFractional * 10;
      resultWhole = num / denominator;
      resultFractional = num % denominator;
      cout << resultWhole;</pre>
  }
   push rcx
   push rbx
   push rsi
   push rdx
   mov rdx, rsi
   call TryConvertNumberToString
   mov rdi, rsi
   call WriteToConsole
   call ClearBuffer
   pop rdx
   pop rsi
   cmp rdx, 0
   je .End
       call PrintPeriod
       xor rcx, rcx
       mov rbx, rdx
       .loop:
          cmp rcx, r9
          jge .End
              push rax
              mov rax, rbx
              mov rbx, 10
              imul rbx
              mov rbx, rax
```

```
pop rax
           push rdx
           push rsi
           push rax
           mov rax, rbx
           idiv r8
           mov rbx, rdx
           mov rdx, rax
           pop rax
           call TryConvertNumberToString
           mov rdi, rsi
           call WriteToConsole
           call ClearBuffer
           pop rsi
           pop rdx
           inc rcx
           jmp .loop
   .End:
     pop rbx
     pop rcx
     ret
<<<<<<
PrintEndl:
  Function printing enl
  void PrintEndl(char* const buffer);
  Params:
           char*
                 buffer
     rax:
  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
<<<<<<
<<<<<<
PrintPlus:
  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], PLUS_SIGH
  mov rdi, 1
  call WriteToConsole
  mov byte [rax], bl
  pop rbx
  pop rdi
  ret
PrintMinus:
```

```
Function printing enl
  void PrintEndl(char* const buffer);
  Params:
           char*
                buffer
     rax:
  Returns:
     void
  push rdi
  push rbx
  mov bl, byte [rax]
  mov byte [rax], MINUS_SIGN
  mov rdi, 1
  call WriteToConsole
  mov byte [rax], bl
  pop rbx
  pop rdi
  ret
<<<<<<
PrintPeriod:
  Function printing enl
  void PrintEndl(char* const buffer);
  Params:
           char*
                 buffer
     rax:
  Returns:
     void
  push rdi
  push rbx
  mov bl, byte [rax]
  mov byte [rax], PERIOD
  mov rdi, 1
  call WriteToConsole
  mov byte [rax], bl
  pop rbx
  pop rdi
  ret
<<<<<<
<<<<<<
Function:
; rax - a, rdi - b, rsi - x
  push rdx
  xor r8, r8 ; Whole
  xor r9, r9 ; Numerator
  xor r10, r10 ; Denominator
  mov r11, 1 ; Sign
  cmp rsi,0
  jg .GreaterZero
     je .EqualZero
        call FunctionCallAMulXSquared
        push rax
        xor rdx, rdx
        mov rax, rdi
        imul rsi
        neg rax
        add r8, rax
        pop rax
        jmp .SetSign
      .EqualZero:
        push rax
        mov rax, rdi
        add rax, rdi
```

```
mov r8, rax
            pop rax
            add r8, rax
            jmp .SetSign
    .SetSign:
        cmp r8, 0
        jge .SetSignNegative
                jmp .End
            .SetSignNegative:
                mov r11, -1
                jmp .End
    .GreaterZero:
        call FunctionCallAMulXSquared
        push rax
        xor rdx, rdx
        mov rax, rdi
        https://stackoverflow.com/questions/51717317/dividing-with-a-negative-
number-gives-me-an-overflow-in-nasm
        idiv rsi
        ; SUM r8
        ; rax CILE
        ; SUM - CILE
        add r8, rax
        ;numerator
        mov r9, rdx
        ;denominator
        mov r10, rsi
        ; COMPARE NUMERATOR AND SAVE IT INTO STACK
        cmp r9, 0
        pushf
        ;GET ABS( UNSIGNED VALUES )
        ;OF NUMERATOR AND DENOMINATOR
        push rax
        push rdi
        mov rax, r9
        call procABS
        mov r9, rdi
        mov rax, r10
        call procABS
        mov r10, rdi
        pop rdi
        pop rax
        ; POP NUMERATOR COMPARE RESULT FLAG FROM STACK
        popf
        jg .NumeratorPositive
            jl .NumeratorNegative
                jmp .localEnd
            .NumeratorNegative:
                cmp r8, 0
                jle .NumeratorNegativeSumLessEqualZero
                     .NumeratorNegativeSumGreaterZero:
                         push rbx
```

```
mov rbx, r10
                   sub rbx, r9
                   mov r9, rbx
                   pop rbx
                   mov r11, 1
                   jmp .localEnd
                . \\ Numerator \\ Negative \\ Sum \\ Less \\ Equal \\ Zero:
                   mov r11, -1
                   jmp .localEnd
          .NumeratorPositive:
             cmp r8, 0
             jl .NumeratorPositiveSumLessZero
                .NumeratorPositiveSumGreaterEqualZero:
                   mov r11, 1
                   jmp .localEnd
                .NumeratorPositiveSumLessZero:
                   push rbx
                   inc r8
                   mov rbx, r10
                   sub rbx, r9
                   mov r9, rbx
                   pop rbx
                   mov r11, -1
                   jmp .localEnd
      .localEnd:
      pop rax
      jmp .End
   .End:
      push rax
      push rdi
      mov rax, r8
      call procABS
      mov r8, rdi
      pop rdi
      pop rax
      pop rdx
      ret
<<<<<<
<<<<<<
FunctionCallAMulXSquared:
   push rax
   push rdi
   push rsi
```

dec r8

```
; rsi = x^2
  mov rax, rsi
mov rdi, 2
  call Pow
  mov rdx, rsi
  pop rsi
  pop rdi
  pop rax
  push rax
  push rcx
  mov rcx, rax
  mov rax, rdx
  imul rcx
  mov r8, rax
  pop rcx
  pop rax
  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
<<<<<<
<<<<<<
TryConvertNumberToString:
     Function converting integer to string;
     bool TryConvertNumberToString(char* buffer, int bufferLength, int
inputtedLength, int number);
     Params:
              char*
                    buffer
        rax:
                    bufferLength
        rdi:
              int
        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) {</pre>
    pop stack into var
    var = var + ZERO_CODE
    *buffer[index] = var
    ++index
}
    copy value to rbx
    mov rbx, rdx
    cmp rbx, 0
    je .zero
        .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
        jmp .ReadingNumbersFromStackToBuffer
    .zero:
        push 0
        inc rcx
.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:
           char*
                 buffer
                 bufferLength
     rdi:
           int
  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
                 bufferLength
     rdi:
           int
     rsi:
           int&
                 inputtedLength
  Returns:
     void
  push rax
  push rdi
  push rdx
  push r8
  push r9
  mov r8, rax
  mov r9, rdi
  mov rsi, 0
   .loop:
     mov rsi, r8
     mov rdx, r9
     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, rax
   pop r9
   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:
            char*
                   buffer
      rax:
      rdi:
            int
                   bufferLength
      rsi:
            int
                   inputtedLength
      rdx:
            int&
                   number
   Returns:
                   if true, no error, else throwed error
       r8:
            bool
   pushf
   push rbx
   push rcx
```

```
push r9
push r10
xor rdx, rdx
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]
        .IsNewLineCharacter:
            cmp bl, NEW_LINE_CHARACTER
            je .NoError
        .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
         mov rax, rdx
         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:
            char
                   С
      rax:
   Returns:
                   if true, then it is digit, else not
      rdi:
            bool
   pushf
   cmp al, DIGIT_ZERO
   jl .Invalid
      cmp al, DIGIT_NINE
      jg .Invalid
         mov rdi, 1
         jmp .End
   .Invalid:
      mov rdi, 0
      jmp .End
   .End:
      popf
      ret
<<<<<<
Pow:
   Function powing number to a certain degree.
   Params:
      rax:
            int number
            int degree
      rdi:
   Returns:
            Powed number
      rsi:
   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
```

### Приклад виконання

```
Input a:
24
Input b:
-5
Input x:
+1175.2857142857
Continue? y - yes, other - no:
Input a:
24
Input b:
-5
Input x:
+14
Continue? y - yes, other - no:
Input a:
24
Input b:
Input x:
-7
+1141
Continue? y - yes, other - no:
Press <RETURN> to close this window...
```

```
Input a:
12
Input b:
-6781
Input x:
7
-380.7142857142
Continue? y - yes, other - no:
```

# Контрольні питання:

# 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 ЈМР. Ця інструкція приймає один операнд, який може бути адресою пам'яті або міткою, визначеною в програмі. Коли інструкція ЈМР 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 СМР

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

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, 2 je section2

cmp eax, 3
je section3

jmp end

section1: jmp end

section2: jmp end

section3: jmp end

end: