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 №2 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 №2

Варіант 18

Тема: засоби обміну даними. Завдання:

- 1. Написати програму з використанням 2-х процедур:
- 1. 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).

- 2. 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.)

Таблипя 2.1

| № в зал. книжці |
|-----------------|-----------------|-----------------|-----------------|
| ариф. дія | ариф. дія | ариф. дія | ариф. дія |
| 1. +67 | 2. +15 | 3. +5 | 478 |
| 532 | 6. +78 | 7. *9 | 823 |
| 9. *4 | 10. +23 | 1188 | 1299 |
| 13. +124. | 14. *7 | 15. *3 | 16. *7 |
| 178 | 18. +6 | 1934 | 2023 |
| 21. 6.*3 | 22. *2 | 23. +47 | 2496 |
| 25. *23 | 2678 | 27. +78 | 28. *5 |
| 29. +33 | 3044 | 31. +55 | 3299 |
| 33. *15 | 3477 | 3577 | 36. +90 |

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
                 eau 0
NEW_LINE_CHARACTER equ 10
                 eau 43
PLUS_SIGH
MINUS_SIGN
                 equ 45
DIGIT_ZERO
                 eau 48
DIGIT_NINE
                 equ 57
   Other constants
BUFFER_LENGTH
            egu 5
section .data
   Errors
error_incorrect_symbol:
                                          db
                                              "Incorrect symbol in input",
NEW_LINE_CHARACTER, 0
error_incorrect_symbol_length:
                                          equ $-error_incorrect_symbol
                                          db "Sign characters must be
error_sign_character_not_first
first", NEW_LINE_CHARACTER, 0
error_sign_character_not_first_length
                                          equ $-
error_sign_character_not_first
   Buffers
buffer:
                        times BUFFER_LENGTH db
inputtedLength:
                                          dq
                                             0
section .text
global asm main
<<<<<<
<<<<<<
asm_main:
   push rax
   push rdi
   push rsi
   push rdx
   push r8
   mov rax, buffer
   mov rdi, BUFFER_LENGTH
   call ReadIntoBuffer
   mov rsi, [rsi]
   call TryConvertStringToInteger
   cmp r8, 0
   je .End
   mov rdx, qword [rdx]
   add rdx, 6
   call ClearBuffer
   call TryConvertNumberToString
   mov rdi, rsi
   call WriteToConsole
   call PrintEndl
   .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
                       number
         rdx:
                int
      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
    .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:
           char*
                 buffer
     rax:
     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
           int*
     rsi:
                 inputtedLength
  Returns:
     void
  push rax
  push rdi
  push rdx
  push r8
  mov r8, rsi
  mov rsi, rax
  mov rdx, rdi
  mov rax, SYS_READ
  mov rdi, STDIN
  call DoSystemCallNoModify
```

```
cmp rax, rdx
   jle .End
   mov rax, rdx
   .End:
   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 qword [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
```

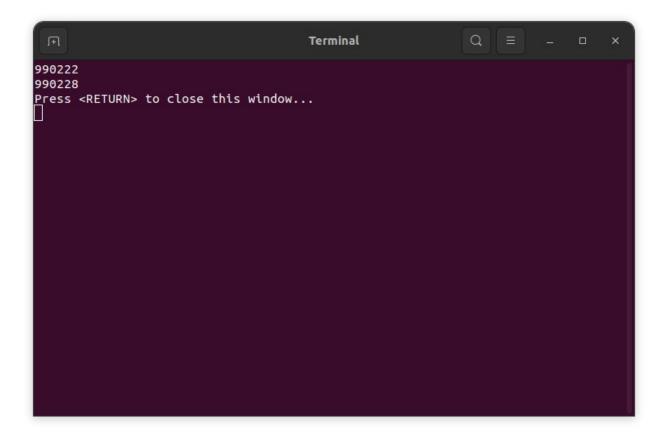
```
push rax
        push rdi
        push rsi
        mov rax, 10
        mov rdi, rcx
        call Pow
        imul rsi, rbx
        mov rdi, qword [rdx]
        add rdi, rsi
        mov qword [rdx], rdi
        pop rsi
        pop rdi
        pop rax
        whole_digit = digit*(10^counter)
    .OnIterationEnd:
    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, qword [rdx]
    neg rax
   mov qword [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
     rax:
  Returns:
                if true, then it is digit, else not
     rdi:
          bool
  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
```

Приклад виконання

```
Terminal Q = - - ×

-10678
-10672
Press <RETURN> to close this window...
```



2.3 Контрольні питання

Цілі числа, дійсні числа та символи по-різному представлені в пам'яті комп'ютера.

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.

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.

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.

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).

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.

Інструкція MUL:

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

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

Інструкція IMUL:

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

Особливості виконання команд ділення DIV та IDIV:

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

Інструкція DIV:

Бере один операнд, який є дільником. Розділяє вміст регістра АХ (для 8-або 16-розрядних операндів) або регістрів DX:АХ (для 32-розрядних операндів) на дільник. Частка зберігається в регістрі АХ (для 8-розрядних або 16-розрядних операндів) або регістрах DX:АХ (для 32-розрядних операндів). Залишок зберігається в регістрі АН (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.

Інструкція IDIV:

Бере один операнд, який є дільником. Розділяє вміст регістра АХ (для 8-розрядних операндів), регістрів DX:АХ (для 16-розрядних операндів) або регістрів EDX:EAX (для 32-розрядних операндів) на дільник. Частка зберігається в регістрі АХ (для 8-розрядних операндів), регістрах DX:АХ (для 16-розрядних операндів) або регістрах EDX:EAX (для 32-розрядних операндів). Залишок зберігається в регістрі АН (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 регістрі призначення, генерується виняток.

Що таке 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 регістра, 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.

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
```

