

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
 Перевірив:

 студент групи ІП-11:
 Лісовиченко О. І

 Панченко С. В.
 "6" березня 2023 р.

3MICT

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. 6.1.1 Цілі числа. 6.1.2 Реальні числа. 6.1.3 Символи.	22 22
	6.2 Вектори переривання, їх розташування у пам'яті	23
	6.3 Особливості виконання команд множення MUL та IMUL 6.3.1 Інструкція MUL 6.3.2 Інструкція IMUL	23
	6.4 Особливості виконання команд ділення DIV та 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.).

3 ТЕКСТ ПРОГРАМИ

```
bits 64
   list of system calls
   https://filippo.io/linux-syscall-table/
SYS_READ
          equ 0
SYS_WRITE
          egu 1
   Descriptors
      equ 0
STDIN
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
             egu 20
MAX LENGTH
           egu 10
section .data
   Errors
error_incorrect_symbol:
                                              "Incorrect symbol in input",
NEW_LINE_CHARACTER, 0
error_incorrect_symbol_length:
                                           equ $-error_incorrect_symbol
                                              "Sign characters must be first",
error_sign_character_not_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
                            db "Overflow", NEW_LINE_CHARACTER, 0
error_overflow
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:
             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
<<
<<
ClearBuffer:
  Function clearing buffer
  void ClearBuffer(char* buffer, int length);
           char*
                 buffer
      rax:
      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:
                    buffer
        rax:
              char*
        rdi:
              int
                    bufferLength
              int&
                    inputtedLength
        rsi:
              int
                    number
        rdx:
     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
               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
```

```
<<
<<
WriteToConsole:
   Function writing string into STDOUT
   void WriteToConsole(char* buffer, int bufferLength)
   Params:
            char*
                   buffer
      rax:
                   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:
            char*
      rax:
                   buffer
                   bufferLength
      rdi:
            int
            int*
      rsi:
                   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:
                   buffer
      rax:
            char*
      rdi:
            int
                   bufferLength
            int
                   inputtedLength
      rsi:
            int&
                   number
      rdx:
   Returns:
                   if true, no error, else throwed error
       r8:
            bool
   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
```

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:
           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

4 СХЕМА ФУНКЦІОНУВАННЯ ПРОГРАМИ

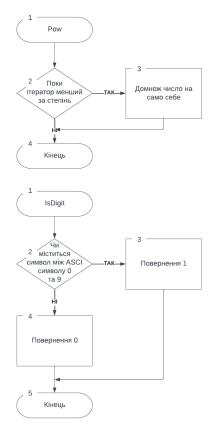


Рисунок 4.1 — схема функцій Pow та IsDigit

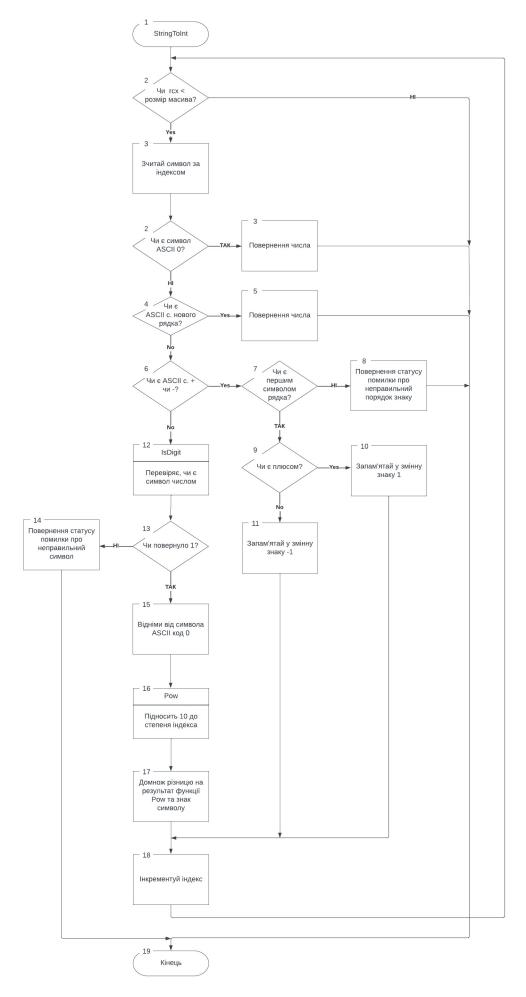


Рисунок 4.2 — схема функції StringToInt

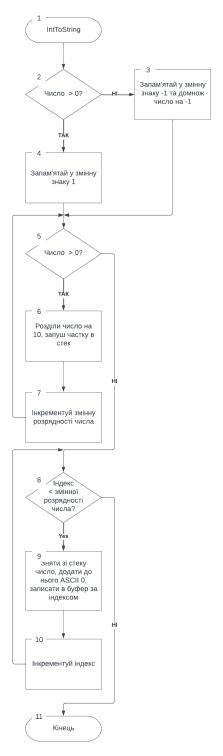


Рисунок 4.3 — схема функції IntToStr



Рисунок 4.4 — схема функції Маіп

5 ПРИКЛАД ВИКОНАННЯ

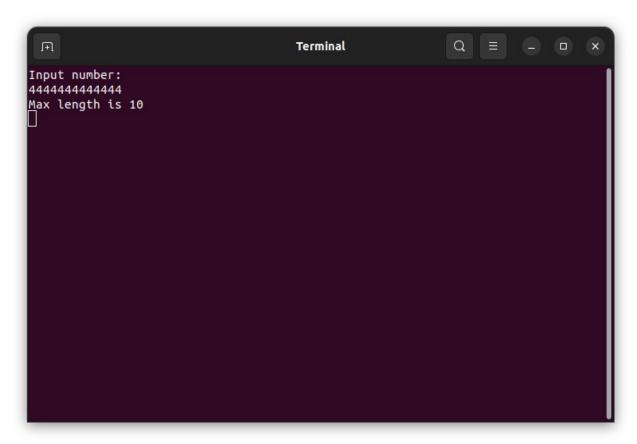


Рисунок 5.1 — Виведення повідомлення про переповнення буфера

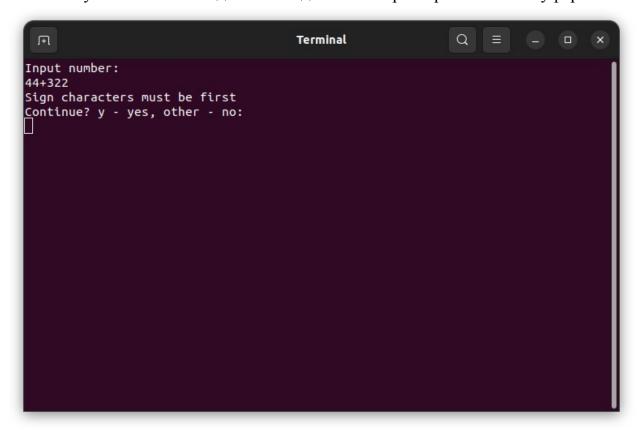


Рисунок 5.2 — Виведення повідомлення про неправильне розташування знаку

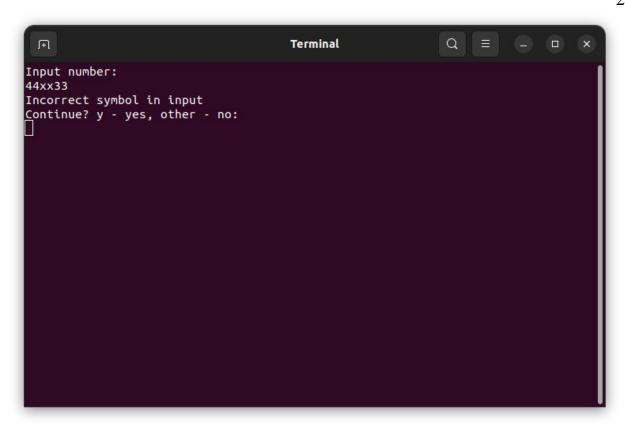


Рисунок 5.3 — Виведення повідомлення про некоректний символ

```
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 ІЕЕЕ 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

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 х $86 \ \epsilon$ 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 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.

6.3.2 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.

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 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.

6.4.2 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:ЕАХ (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:ЕАХ (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.

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 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.

6.6 Команда організації циклів Іоор 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
il.loop
```