

**МИНИСТЕРСТВО ЦИФРОВЫХ ТЕХНОЛОГИЙ РЕСПУБЛИКИ УЗБЕКИСТАН
ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ
ТЕХНОЛОГИЙ имени МУХАММАДА АЛЬ-ХОРЕЗМИ**

**ПРЕЗЕНТАЦИЯ
ПО ДИСЦИПЛИНЕ: СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ**

**НА ТЕМУ : Написание программы на языке Ассемблера выполняющую обработку и вывод
символьных данных. Написание программы на языке Ассемблера выполняющую обработку и
вывод целочисленных данных.**

**Выполнил: студент 4-го курса гр. 321-20
факультета
Программный инжиниринг
Сахабиев Рослан**

Ташкент-2024



Функциональные требования

1. Ввод символьных данных:

- Возможность ввода как отдельных символов, так и строк.

2. Обработка символьных данных:

- Реализация алгоритма обработки, включающего базовые операции над символами.

3. Вывод результатов:

- Возможность вывода результатов обработки на экран консоли.

Символьные данные

В языке ассемблера символьные данные представляют собой последовательность ASCII-символов, хранящихся в памяти. Символьные данные используются для представления строк и символов.

Пример:

```
section .data
```

```
    ; определяем строку
```

```
msg_welcome db 'Здравствуйте, как вас зовут?', 0xA, 0
```

Здесь db используется для определения байтов (define byte)

Длина строки

Длина строки указывается одним из следующих двух способов:

- явное содержание длины строки;
- использование нуль-терминатора.

Мы можем явно хранить длину строки, используя символ счетчика местоположения \$, который предоставляет текущее значение счетчика местоположения строки.

Например:

```
msg_welcome db 'Здравствуйте, как вас зовут?', 0xA ; наша строка
msg_welcome_len equ $ - msg_welcome ; длина нашей строки
```

; Строка с завершающим нуль-терминатором

```
msg_with_name_user db ', приятно познакомиться.', 0xA, 0
```

Символ \$ указывает на byte после последнего символа строковой переменной `msg`. Следовательно, `$ - msg` равен длине строки.

Вычисление длины произвольной строки

```
1. ; Input: rax = string
2. ; Output: rax = string length
3. strlen:
4.     push rbx                ; RBX - callee-saved регистр.
5.     xor rbx, rbx            ; Счетчик длины строки.
6.     .next_iter:
7.         ; Сравниваем текущий символ с нуль-терминатором.
8.         cmp [rax+rbx], byte 0
9.         je .close
10.        inc rbx
11.        jmp .next_iter
12.     .close:
13.        mov rax, rbx
14.        pop rbx
15.        ret
```

Ввод строки с клавиатуры

```
1. ; Input:
2. ; rax = string buffer
3. ; rbx = string buffer size
4. input_string:
5.     push rax
6.     push rbx
7.     push rcx
8.     push rdx
9.
10.    mov rcx, rax    ; Копируем указатель на буфер в RCX.
11.    mov rdx, rbx    ; Копируем размер буфера в RDX.
12.    mov rax, 3      ; Системный вызов для чтения данных (sys_read).
13.    mov rbx, 0      ; Файловый дескриптор stdin.
14.    int 0x80
15.
16.    ; Добавляем нуль-терминатор в конец строки.
17.    mov [rcx+rax-1], byte 0
18.
19.    pop rdx
20.    pop rcx
21.    pop rbx
22.    pop rax
23.    ret
```

Ввод символов с клавиатуры

```
1.  ; Output: rax = char
2.  input_char:
3.      push rbx
4.      mov rax, buffer_char
5.      mov rbx, buffer_char_size
6.      call input_string
7.      mov rax, [rax]
8.      pop rbx
9.      ret
```

Вывод строки произвольной длины в консоль

```
1. ; Input: rax = string
2. print_string:
3.     push rax
4.     push rbx
5.     push rcx
6.     push rdx
7.
8.     mov rcx, rax      ; Копируем указатель на строку в RCX.
9.     call strlen
10.
11.    ; Копируем длину строки в RDX (для системного вызова write).
12.    mov rdx, rax
13.    mov rax, 4         ; Системный вызов для записи данных (sys_write).
14.    mov rbx, 1         ; Файловый дескриптор stdout.
15.    int 0x80
16.
17.    pop rdx
18.    pop rcx
19.    pop rbx
20.    pop rax
21.    ret
```

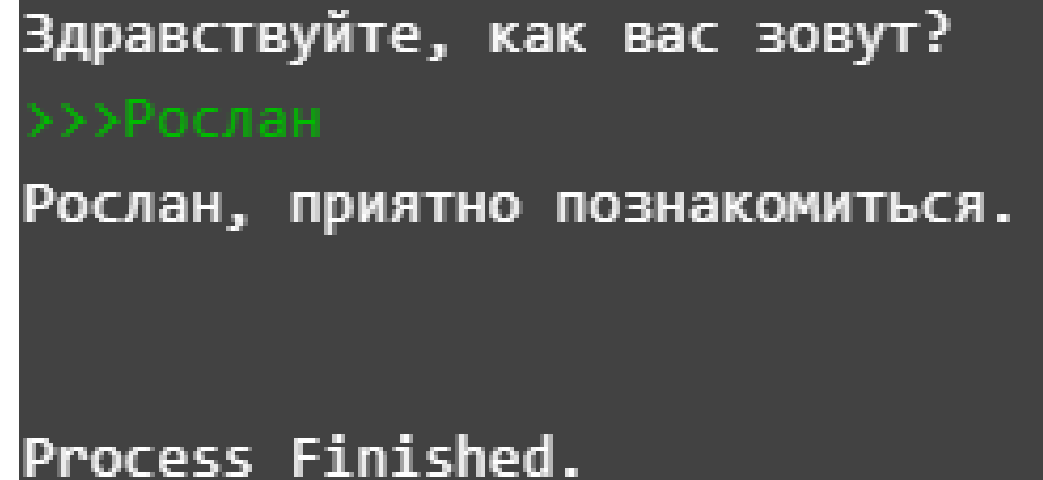

Вывод символов в консоль

```
1. ; Input: rax = char
2. print_char:
3.     push rdx
4.     push rcx
5.     push rbx
6.     push rax
7.
8.     mov [bss_char], al
9.
10.    mov rax, 4
11.    mov rbx, 1
12.    mov rcx, bss_char
13.    mov rdx, 1
14.    int 0x80
15.
16.    pop rax
17.    pop rbx
18.    pop rcx
19.    pop rdx
20.    ret
```

Результат работы программы

```
1. section .data
2.   msg_welcome db 'Здравствуйте, как вас зовут?', 0xA, 0
3.   msg_with_name_user db ', приятно познакомиться.', 0xA, 0
4.   user_name_size equ 20
5.
6. section .bss
7.   user_name resb user_name_size
8.
9. section .text
10. global _start
11.
12._start:
13.   mov rax, msg_welcome
14.   call print_string
15.   mov rax, user_name
16.   mov rbx, user_name_size
17.   call input_string
18.   mov rax, user_name
19.   call print_string
20.   mov rax, msg_with_name_user
21.   call print_string
22.
23.   mov rax, 1           ; Системный вызов sys_exit
24.   mov rbx, 0           ; Код возврата 0
25.   int 80h;
```

Результат:



```
Здравствуйте, как вас зовут?
>>>Рослан
Рослан, приятно познакомиться.

Process Finished.
```

Функциональные требования для программы, выполняющую обработку и вывод целочисленных данных.

1. Ввод данных:

- Программа должна поддерживать ввод целочисленных данных пользователем.

2. Обработка целочисленных данных:

- Выполнение арифметических операций над введёнными данными (сложение, вычитание, умножение, деление).

3. Вывод результатов:

- Возможность вывода результатов обработки на экран консоли.

Секция data для обработки и вывода целочисленных данных

```
1. section .data
2.     msg1 db 'Введите первое число: ', 0xA, 0
3.     msg2 db 'Введите второе число: ', 0xA, 0
4.     msg3 db 'Введите математическую операцию (+, -, *, /): ', 0xA, 0
5.     msg_result db 'Результат: ', 0
6.     buffer_number_size equ 20
7.     buffer_char_size equ 2
8. section .bss
9.     buffer_number resb buffer_number_size
10.    number1 resb buffer_number_size
11.    number2 resb buffer_number_size
12.    bss_char resb 1
13.    buffer_char resb buffer_char_size
```

Ввод чисел с клавиатуры

```
1.  ; Output: rax = number
2.  input_number:
3.      push rbx
4.      mov rax, buffer_number
5.      mov rbx, buffer_number_size
6.      call input_string
7.      call atoi
8.      pop rbx
9.      ret
```

Реализация функции atoi

Input: rax = string

Output: rax = number

```
1.  atoi:
2.      push rbx
3.      push rcx
4.      push rdx
5.      push rdi
6.      xor rdi, rdi
7.      xor rdx, rdx
8.      xor rcx, rcx
9.      xor rbx, rbx
10.     cmp byte [rax], '-'
11.     jne .next_iter
12.     inc rdi
13.     inc rbx
14.     .next_iter:
15.         movzx rdx, byte [rax+rbx]
16.         cmp rdx, 0
17.         je .close
18.         sub rdx, '0'
19.         imul rcx, 10
20.         add rcx, rdx
21.         inc rbx
22.         jmp .next_iter
23.     .close:
24.         mov rax, rcx
25.         cmp rdi, 1
26.         jne .is_not_negative
27.         neg rax
28.         .is_not_negative:
29.         pop rdi
30.         pop rdx
31.         pop rcx
32.         pop rbx
33.         ret
```

Реализация функции print_integer

Input: rax = number

```
1. print_integer:
2.     push rax
3.     push rbx
4.     push rcx
5.     push rdx
6.     xor rcx, rcx
7.     cmp rax, 0
8.     jnl .next_iter
9.     neg rax
10.    push rax
11.    mov rax, '-'
12.    call print_char
13.    pop rax
14.    .next_iter:
15.        mov rbx, 10
16.        xor rdx, rdx
17.        div rbx
18.        add rdx, '0'
19.        push rdx
20.        inc rcx
21.        cmp rax, 0
22.        je .print_iter
23.        jmp .next_iter
24.    .print_iter:
25.        cmp rcx, 0
26.        je .close
27.        pop rax
28.        call print_char
29.        dec rcx
30.        jmp .print_iter
31.    .close:
32.        pop rdx
33.        pop rcx
34.        pop rbx
35.        pop rax
36.        ret
```

Реализация функции calc

Input:

rax = number1

rbx = number2

rcx = operator

Output:

rax = result

```
1. calc:
2.     cmp rcx, '-'
3.     je .substruct
4.
5.     cmp rcx, '+'
6.     je .sum
7.
8.     cmp rcx, '*'
9.     je .multiply
10.
11.    cmp rcx, '/'
12.    je .divide
13.
14.    .sum:
15.    add rax, rbx
16.    jmp .end_calc
17.
18.    .substruct:
19.    sub rax, rbx
20.    jmp .end_calc
21.
22.    .multiply:
23.    imul rax, rbx
24.    jmp .end_calc
25.
26.    .divide:
27.    idiv rbx
28.
29.    .end_calc:
30.    ret
```


Перевод каретки

```
1. print_line:  
2.     push rax  
3.     mov rax, 0xA  
4.     call print_char  
5.     pop rax  
6.     ret
```

Секция Text

```
1. section .text
2.     global _start
3.
4. _start:
5.     .loop:
6.     mov rax, msg1
7.     call print_string
8.
9.     mov rax, buffer_number
10.    call input_number
11.    mov [number1], rax
12.
13.    mov rax, msg2
14.    call print_string
15.
16.    mov rax, buffer_number
17.    call input_number
18.    mov [number2], rax
19.
20.    mov rax, msg3
21.    call print_string
22.
23.    call input_char
24.    mov [buffer_char], rax
25.
26.    mov rax, msg_result
27.    call print_string
28.
29.    mov rax, [number1]
30.    mov rbx, [number2]
31.    mov rcx, [buffer_char]
32.    call calc
33.
34.    ; Вывод результата
35.    call print_integer
36.    call print_line
37.    jmp .loop
38.
39.    ; Завершаем программу
40.    mov rax, 1                ; Системный вызов sys_exit
41.    xor rbx, rbx             ; Код возврата 0
42.    int 0x80
```

Результат

Введите первое число:

>>>5

Введите второе число:

>>>3

Введите математическую операцию (+, -, *, /):

>>>+

Результат: 8

Введите первое число:

>>>-5

Введите второе число:

>>>6

Введите математическую операцию (+, -, *, /):

>>>*

Результат: -30

Введите первое число:

>>>20

Введите второе число:

>>>5

Введите математическую операцию (+, -, *, /):

>>>/

Результат: 4