НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет программной инженерии и компьютерной техники

Низкоуровневое программирование

Лабораторная работа № 1

Выполнил студент

Куприянов Артур Алексеевич

Группа № P33113

Преподаватель: Жирков Игорь Олегович

г. Санкт-Петербург

2020

Лабораторная работа

В данной работе было необходимо написать код на ассемблере, реализующую базовые функции, который в последующем тестировались на отдельной программе

Вывод

При разработке программы на ассемблере разработчик сталкивается с множествами проблем связанных с различием ассемблера от высокоуровневых языков. В ассемблере пользователю предоставляется очень много возможностей, но и не в лучшую сторону тоде. Например, пользователю самому приходится соблюдать правила для callee и caller saved registers

Тем не менее, программирование на ассемблере (помимо высокой скорости) также дает основные знания об исполнении скомпилированной программы в машине, а также отвечает на вопрос почему в некоторых языках необходимо компилировать отдельно для каждой машины (за исключение кросс-платформенных, например, на JVM)

github.com/AppLoidx/Assembly/

```
; some system code variables
%define sys_exit 60
%define sys_read 0
%define sys_write 1

; input/output
%define stdin 0
%define stdout 1

; result codes
%define success 0

; const
%define new_line 0xA
%define base_10 10

section .text


; Принимает код возврата и завершает текущий процесс
exit:
    xor rax, rax,
    mov rax, sys_exit
    ; rdi already set // i hope :)
    ret

sys_stdout:
    mov rax, sys_write
    mov rdi, stdout
    syscall
    ret

; Принимает указатель на нуль-терминированную строку,
возвращает её длину
string_length:
    xor rax, rax
    ; rax = 0
    ; rdi -> value1 , ... value2, value3 ...
    .str_len_loop:
        cmp byte[rdi + rax], 0  ; check for null sym
        je .end                 ; if equals
        inc rax                 ; increment counter
        jmp .str_len_loop       ; back to loop
    .end:
        ret

; Принимает указатель на нуль-терминированную строку,
выводит её в stdout
print_string:
    xor rax, rax
```

```
    mov rsi, rdi                ; put argument // lol

    call string_length          ; length of passed string
    mov rdx, rax

    call sys_stdout
    ret

; Принимает код символа и выводит его в stdout
print_char:
    xor rax, rax

    push rdi

    mov rsi, rsp    ; pointer to stack
    mov rdx, 1      ; print one char
    call sys_stdout


    add rsp, 8
    ret

; Переводит строку (выводит символ с кодом 0xA)
print_newline:
    xor rax, rax
    push rsi

    mov rsi, new_line
    mov rdx, 1
    call sys_stdout

    pop rsi

    ret

; Выводит беззнаковое 8-байтовое число в десятичном
формате
; Совет: выделите место в стеке и храните там результаты
деления
; Не забудьте перевести цифры в их ASCII коды.
print_uint:
    push r12
    mov r12, rsp
    mov rax, rdi
    dec rsp
    mov byte[rsp], 0
    .loop:
        xor rdx, rdx
        dec rsp                                 ; move
pointer
        push r13
        mov r13, 10
```

```
        div  r13                ; div by 10 (sys base),
quotient stored to rdx
        pop r13
        add rdx, 0x30           ; convert to ASCII
        mov  byte[rsp], dl      ; save value
        test rax, rax           ; set flags (zf)
        jz .print
        jmp .loop
    .print:
        mov rdi, rsp
        call print_string
        mov rsp, r12
        jmp .end
    .end:
        pop r12                                         ;
восстановим регистры
        ret

; Выводит знаковое 8-байтовое число в десятичном формате
print_int:
    xor rax, rax
    mov rax, rdi
    test rax, rax       ; set flags
    jns .not_minus

    push rax
    mov  rdi, '-'       ; print "-"
    call print_char
    pop rax

    neg rax             ; convert to non-negative number
    mov rdi, rax

    .not_minus:
        call print_uint ; print as unsigned value ("-"
char already should be printed)
        ret

; Принимает два указателя на нуль-терминированные строки,
возвращает 1 если они равны, 0 иначе
string_equals:
        xor rax, rax
        push r10
        push r13
        .loop:
            mov r10b, byte[rsi]
            mov r13b, byte[rdi]
            inc rsi
            inc rdi
            cmp r10b, r13b
            jne .ret_zero
            cmp r13b, 0
```

```
            jne .loop
            inc rax
        .ret_zero:
            pop r13
            pop r10


    ret
; Читает один символ из stdin и возвращает его. Возвращает
0 если достигнут конец потока
read_char:
    push rdi
    push rdx
    dec rsp
    mov rax, sys_read
    mov rdi, stdin
    mov rdx, 1
    mov rsi, rsp
    syscall
    test rax, rax
    je .return
    xor rax, rax
    mov al, [rsp]       ; save stack[rsp] value

    .return:
        ; restore values
        inc rsp
        pop rdx
        pop rdi
        ret


; Принимает: адрес начала буфера, размер буфера
; Читает в буфер слово из stdin, пропуская пробельные
символы в начале, .
; Пробельные символы это пробел 0x20, табуляция 0x9 и
перевод строки 0xA.
; Останавливается и возвращает 0 если слово слишком
большое для буфера
; При успехе возвращает адрес буфера в rax, длину слова в
rdx.
; При неудаче возвращает 0 в rax
; Эта функция должна дописывать к слову нуль-терминатор
read_word:
    push r13
    push r14
    xor r14, r14
    mov r10, rsi
    mov r13, rdi
    .init_loop:
        call read_char
        cmp al, 0x20
```

```
        je .init_loop
        cmp al, 0x9
        je .init_loop
        cmp al, 0xA
        je .init_loop
        cmp al, 0x0
        je .read_end
        jmp .write_char
    .read_start:
        call read_char
        cmp al, 0x0
        je .read_end
        cmp al, 0x20
        je .read_end    ; starts new word, thus we need to
stop read
        jmp .write_char
    .write_char:
        mov byte [r13 + r14], al
        inc r14
        cmp r14, r10    ; overflow check
        je .overflow
        jmp .read_start
    .read_end:
        mov rax, r13
        mov byte [r13 + r14], 0

        mov rdx, r14
        jmp .return
    .overflow:
        xor rax, rax
        xor rdx, rdx
    .return:
        pop r14
        pop r13
        ret


; Принимает указатель на строку, пытается
; прочитать из её начала беззнаковое число.
; Возвращает в rax: число, rdx : его длину в символах
; rdx = 0 если число прочитать не удалось
parse_uint:
    xor rax, rax
    push r13
    mov r13, 10
    xor rax, rax
    xor rcx, rcx
    xor rdx, rdx
    xor rsi, rsi
    .parse_char:
        mov sil, [rdi + rcx]
        cmp sil, 0x30
```

```
        jl .return
        cmp sil, 0x39
        jg .return
        inc rcx
        sub sil, 0x30
        mul r13
        add rax, rsi
        jmp .parse_char
    .return:
        mov rdx, rcx
        pop r13
        ret



; Принимает указатель на строку, пытается
; прочитать из её начала знаковое число.
; Если есть знак, пробелы между ним и числом не разрешены.
; Возвращает в rax: число, rdx : его длину в символах
(включая знак, если он был)
; rdx = 0 если число прочитать не удалось
parse_int:
    xor rax, rax
    xor rax, rax
    cmp byte [rdi], 0x2d
    je .parse_ng
    call parse_uint
    ret
    .parse_ng:
        inc rdi
        call parse_uint
        cmp rdx, 0
        je .return
        neg rax
        inc rdx
    .return:
        ret

; Принимает указатель на строку, указатель на буфер и
длину буфера
; Копирует строку в буфер
; Возвращает длину строки если она умещается в буфер,
иначе 0
string_copy:
    ; by convention rdi, rsx, rdx, ...
    ; rdi - pointer to string
    ; rsi - pointer to buffer
    ; rdx - len of buffer

    ; lets use r12 register as inner buffer
    ; but first we have to save it, because of callee-
saved register
```

```
    push r12
    xor rcx, rcx
    .main_loop:
        ; for range 0 to buffer_len
        cmp rcx, rdx    ; counter for loop
        je .overflow    ; buffer overflowed if they are
equal
        mov r12, [rdi + rcx]    ; start + offset    (auto-
incremented counter)
        mov [rsi, rcx], r12     ; move from inner buffer
to target

        ; check for end of string
        cmp r12, 0
        je .exit

        inc rcx
        jmp .main_loop

    .overflow:
        mov rax, 0
        jmp .exit

    .exit:
        pop r12
        ret
```