Университет ИТМО, кафедра ВТ

Лабораторная работа №2 по
“Языкам Системного Программирования”

Работу выполнил

студент группы P3200

Рогов Я. С.

Преподаватели:

Жирков И. О.

Балакшин П. В.

Санкт-Петербург, 2016

Задание: реализовать на языке ассемблера (синтаксис Intel) интерпретатор языка Forth.

Содержимое файлов macroses.mcrs и изменения в библиотеке ввода/вывода libio.inc:

```
; Добавлены 2 функции в библиотеку libio.inc

next_word:

    mov rsi, [startpos]
    mov al, [isended]
    test al, al
    jz .loop ; there're words left in buffer

    mov rdi, yournameplease ; print invitation
    call print_string

    mov byte[isended], 0
    xor rax, rax
    mov rdi, 0
    mov rsi, word_buffer
    mov rdx, 255
    syscall

    .loop:
        mov dil, byte[rsi]
        call is_space
        test rax,rax
        jns .notnul
        mov rax, rsi
        mov byte[isended], 1
        xor rdx, rdx
        ret           ; No word
    .notnul:
        jz .foundword
        inc rsi
        jmp .loop

.foundword:
    mov rdx, rsi

    .loop2:
        inc rdx
        mov dil, byte[rdx]
        call is_space
        test rax,rax ;
        jz .loop2
        jns .haswords
        jmp .bufferend

    .bufferend:
    mov byte[isended], 1
    mov qword[startpos], word_buffer
    jmp .wordend

    .haswords:
    lea rax, [rdx+1]
    mov qword[startpos], rax
    mov byte[isended], 0

.wordend:
    mov byte[rdx], 0
    mov rax, rsi
    sub rdx, rax
    ret



; returns rax : -1 if NUL, 1 if space character,
-3 if newline character, else 0
; is_not_regular? test rax, 1 -> jz/jnz
; is_nul_or_newline? test rax, rax -> js/jns
is_space:

    xor rax, rax

    test dil, dil
    jnz .notnul
    dec rax; NUL : -1
    ret

.notnul:
    cmp dil, 0x20
    ja .regular
    je .space
    cmp dil, 0xa
    je .newline
    ja .regular
    cmp dil, 0x9
    jne .regular
    jmp .space
.newline:
    mov rax, -3
    ret
.space:
    inc rax
.regular:
    ret

; Макросы для forth-слов в dictionary.asm из
macroses.mcrs:

%define prev 0

; word word_suffix flags
%macro native 3
section .data
w_%2:
       %%prev: dq prev
       db %1, 0
       db %3

xt_%2:
       dq impl_%2

section .text
       impl_%2:
%define prev %%prev
%endmacro

; word word_suffix
%macro native 2
native %1, %2, 0
%endmacro


; word word_suffix flags [executed_words]
; It push xt_exit in the end automatically
%macro colon 3-*
section .data
w_%2:
       %%prev: dq prev
       db %1, 0
       db %3
docol_%2:
       dq impl_docol
%rep %0-3
       dq xt_%4
       %rotate 1
%endrep
       dq xt_exit

xt_%5:
       dq docol_%5
%define prev %%prev
%endmacro

%macro colon 2
colon %1, %2, 0
%endmacro
```

Содержимое файла dictionary.asm:

```
%include 'macroses.mcrs'

%define WFLAG_IMMEDIATE 1
%define WFLAG_COMPILEONLY 2
%define WFLAG_BRANCH 4

section .data
        times 32 dq 0
        rstackend:

section .data
        user_memory: times USERMEMSIZE dq 0

section .data
        ; messages
        stackendmsg: db '[ERROR] End of stack',
10, 0
        usMemAError: db '[ERROR] Out of memory
bounds', 10, 0

; WORDS

native 'quit', quit
        lea rsp, [STACK_END+8]
        pop r15
        pop r14
        pop r13
        pop r12
        pop rbx
        mov rax, 60
        xor rdi, rdi
        syscall


; Advanced Arithmetic

native '/', divmod
        call safepop
        mov rcx, rax
        call safepop
        idiv rcx
        push rax
        push rdx
        jmp next

native '*l', lmultiply
        call safepop
        mov rcx, rax
        call safepop
        mul rcx
        push rdx
        push rax
        jmp next


; Memory Words

native 'mem', getmemadd
        push USERMEMSTART
        jmp next

native '!', write
        call safepop
        mov rdi, rax
        call check_um_address
        call safepop
        mov qword[rdi], rax
        jmp next

native '@', read
        call safepop
        mov rdi, rax
        call check_um_address
        push qword[rdi]
        jmp next


; Compiling Words

native 'branch', branch, WFLAG_COMPILEONLY |
WFLAG_BRANCH
        mov rax, [PC]
        lea PC, [PC + rax*CELLSIZE + CELLSIZE]
        jmp next
        mov rdi, rax
        call parse_int
        add PC, rax
        jmp next

native 'branch0', branch0, WFLAG_COMPILEONLY |
WFLAG_BRANCH
        call safepop
        test rax, rax
        jz impl_branch
        add PC, CELLSIZE
        jmp next

native 'lit', lit, 2
        push qword[PC]
        add PC, CELLSIZE
        jmp next

native ':', compilestart
        call next_word
        test rdx, rdx
        jz next

        mov rdi, qword[lastword]
        mov qword[HERE], rdi
        mov qword[lastword], HERE
        add HERE, 10
        lea rsi, [HERE-2] ; str_wordname pointer
        mov rdi, rax
        add HERE, rdx
        call string_copy
        mov byte[HERE-1], 0
        mov qword[HERE], impl_docol
        add HERE, 8
        mov byte[state], 1
        jmp next

native ';', compileend, WFLAG_IMMEDIATE
        mov byte[state], 0
        mov qword[HERE], xt_exit
        add HERE, 8
        jmp next


; IO Words

native 'key', readchar
        call read_char ;
        push rax
        jmp next

native 'emit', printchar
        call safepop
        mov rdi, rax
        call print_char ;
        jmp next

native 'number', readint
        call read_word ;
        mov rdi, rax
        call parse_int ;
        push rax
        jmp next


; Bitwise Words

native '&', andb
        call safepop
        mov rcx, rax
        call safepop
        and rax, rcx
        push rax
        jmp next

native '|', orb
        call safepop
        mov rcx, rax
        call safepop
        or rax, rcx
        push rax
        jmp next

native '^', xor
        call safepop
        mov rcx, rax
        call safepop
        xor rax, rcx
        push rax
        jmp next

native '<<', lshift
        call safepop
        mov cl, al
        call safepop
        shl rax, cl
        push rax
        jmp next

native '>>', rshift
        call safepop
        mov cl, al
        call safepop
        shr rax, cl
        push rax
        jmp next

native '~', notb
        call safepop
        not rax
        push rax
        jmp next; Stack Words


; Stack Operations

native 'dup', dup
        call safepop
        push rax
        push rax
        jmp next

native 'drop', drop
        call safepop
        jmp next

native 'swap', swap
        call safepop
        mov rcx, rax
        call safepop
        push rcx
        push rax
        jmp next

native 'rot', rot
        call safepop
        mov rcx, rax
        call safepop
        mov rdx, rax
        call safepop
        push rdx
        push rcx
        push rax
        jmp next

native '.S', printstack
        mov SPBOUNDSAFE, STACK_END
        .loop:
                cmp STACK_END, rsp
                jl .stop
                mov rdi, [STACK_END]
                call print_int
                call print_space
                sub STACK_END, 8
                jmp .loop
        .stop:
        mov STACK_END, SPBOUNDSAFE
        call print_newline
        jmp next

native '.', popnprint
        call safepop
        mov rdi, rax
        call print_int ;
        call print_newline
        jmp next


; Boolean Words

native 'and', and

        call safepop
        test rax, rax
        jnz .continue
        mov qword[rsp], 0
        jmp next
        .continue:
        mov rcx, rax
        call safepop
        test rax, rax
        jz .false
        mov rax, 1
        .false:
        push rax
        jmp next

colon 'or', or, 0, not, swap, not, and, not

native 'not', not
        call safepop
        test rax, rax
        jnz .true
        inc rax
        jmp .move
        .true:
        xor rax, rax
        .move:
        push rax
        jmp next

; Basic Comparison

native '=', equal
        call safepop
        mov rcx, rax
        call safepop
        xor rax, rcx
        jnz .false
        inc rax
        jmp .move
        .false:
        xor rax, rax
        .move:
        push rax
        jmp next

native '<', less
        call safepop
        mov rcx, rax
        call safepop
        cmp rcx, rax
        jge .true
        xor rax, rax
        jmp .move
        .true:
        mov rax, 1
        .move:
        push rax
        jmp next

colon '>', greater, 0, less, not

; Basic Arithmetic

native '++', inc
        call safepop
        inc rax
        push rax
        jmp next

native '--', dec
        call safepop
        dec rax
        push rax
        jmp next

native '//', div
        call safepop
        mov rcx, rax
        call safepop
        idiv rcx
        push rax
        jmp next

native '*', multiply
        call safepop
        mov rcx, rax
        call safepop
        mul rcx
        push rax
        jmp next

native '-', minus
        call safepop
        mov rcx, rax
        call safepop
        sub rax, rcx
        push rax
        jmp next

native '+', plus
        call safepop
        mov rcx, rax
        call safepop
        add rax, rcx
        push rax
        jmp next

; Colon Words

section .text
impl_docol:
        sub RSTACK, 8
        mov [RSTACK], PC
        add W, CELLSIZE
        mov     PC, W
        jmp     next


section .data
        lastword: dq prev

section .data
xt_exit: dq impl_exit
section .text

impl_exit:
        mov PC, [RSTACK]
        add RSTACK, 8
        jmp next

; Safety functions

; checks stack bound and pops value to rax if in bounds
; returns:
; rax : value (if in range)
; prints error message and returns to imploop otherwise
safepop:
        mov rax, STACK_END
        xor rax, rsp
        jnz .ok

        mov rdi, stackendmsg
        call print_string
        add rsp, 8

; check if colon word
        cmp RSTACK, rstackend
        jz next
; stops execution of colon word
        mov RSTACK, rstackend
        mov PC, [RSTACK-8]
        jmp next

        .ok:
        add rsp, 8
        pop rax
        jmp [rsp-16]

; checks if User Memory Address isn't out of bound
; and writes error if out of bound
; rdi - address
check_um_address:
        mov rax, rdi
        sub rax, USERMEMSTART
        js .error
        cmp rax, USERMEMSIZE - 1
        jg .error
        ret

        .error:
        push rdi
        mov rdi, usMemAError
        call print_string
        pop rdi
        jmp next

section .data
user_words:
        times USERDICTSIZE dq 0
```

Содержимое файла forth.asm:

```
%define HERE rbx
%define PC r12
%define W r13
%define RSTACK r14
%define STACK_END r15

%define USERMEMSTART user_memory
%define USERMEMSIZE 65536
%define USERDICTSTART user_words
%define USERDICTSIZE 65536

%define CELLSIZE 8
%define SPBOUNDSAFE [spboundsafe]

; 0 - Interpreter
%define STATE_COMPILER 1
%define STATE_BRANCH 2

%include 'libio.inc'
%include 'dictionary.asm'

section .data
        spboundsafe: dq 0

; Messages
        nowordmsg: db '[ERROR] Wrong word: ', 0
        nobranchnummsg: db '[ERROR] Wrong usage of branch. Usage: branch/branch0 n', 10, 0

state: db 0
program_stub: dq 0
xt_interpreter: dq .interpreter
.interpreter: dq main_loop

section .text

global _start

_start:
        push rbx
        push r12
        push r13
        push r14
        push r15
        mov HERE, USERDICTSTART
        mov PC, xt_interpreter
        mov RSTACK, rstackend
        lea STACK_END, [rsp-8]

        jmp next


main_loop:
        call next_word
        test rdx, rdx

        jz main_loop ; continue if no word

        mov rdi, rax
        call parse_int
        mov sil, byte[state]
        test rdx, rdx
; jump to command processing branch
        jz .command
```

```
; tests if compile mode
        test sil, STATE_COMPILER
        jz .itrp_num
; tests if last command was branch/branch0
        test sil, STATE_BRANCH
        jnz .comp_num
        mov qword[HERE], xt_lit
        add HERE, CELLSIZE
        .comp_num:
        mov qword[HERE], rax
        add HERE, CELLSIZE
        and byte[state], 255-2
        jmp main_loop

        .itrp_num:
        push rax
        jmp main_loop

        .command:
        test sil, STATE_BRANCH
        jz .nobranchstate

; writes error message for wrong using of branch
        mov rdi, nobranchnummsg
        call print_string
; and rollbacks all changes
        mov byte[state], 0
        mov HERE, [lastword]
        mov rax, [HERE]
        mov [lastword], rax
        .skiptillend:
                call next_word
                mov al, [rax]
                xor al, ';'
                jnz .skiptillend
        jmp main_loop

        .nobranchstate:
        call find_word
        test rax, rax
        jz .noword

; saves str_word pointer (for .noword)
        mov rsi, rdi
        mov rdi, rax
        call cfa
; restores str_word pointer
        mov rdi, rsi
        mov cl, [state]
        mov ch, [rax-1] ; loads word flags
; tests if compile mode
        test cl, STATE_COMPILER
        jz .itrp_cmd
; checks if immediate and interpretes if it is
        test ch, WFLAG_IMMEDIATE
        jnz .itrp_cmd
; checks if word is branch*
        test ch, WFLAG_BRANCH
        jz .notbranch
        or byte[state], STATE_BRANCH

        .notbranch:
        mov qword[HERE], rax
        add HERE, 8
```

```
        jmp main_loop

        .itrp_cmd:
; checks if word is for compile mode only
        test ch, WFLAG_COMPILEONLY
        jnz .noword
        mov [program_stub], rax
        mov PC, program_stub
        jmp next

        .noword:
        push rdi
        mov rdi, nowordmsg
        call print_string
        pop rdi
        call print_string

        call print_newline
        jmp main_loop

; rdi - 'word' string pointer
; returns: rax - word pointer
find_word:

        mov r8, [lastword]

        .loop: ; searches for ford
                test r8, r8
                jz .noword
                lea rsi, [r8+8]
                call string_equals
                test rax, rax
                jnz .found
                mov r8, [r8]
                jmp .loop

        .found:
        mov rax, r8
        ret
        .noword:
        xor rax, rax
        ret

; rdi - word pointer
cfa:
        add rdi, 8

        .loop: ; skips entire string - wordname
                mov al, byte[rdi]
                inc rdi
                test al, al
                jnz .loop

        lea rax, [rdi+1] ; skips flags
        ret

next:
        mov W, PC
        add PC, CELLSIZE
        mov W, [W]
        jmp [W]
```

Вывод: в ходе выполнения данной лабораторной работы я познакомился с концепцией конечных автоматов и их применения в программировании: в данном случае, для реализации интерпретатора языка Forth, диалект которого также был изучен.