University of Guadalajara

Project Report

Traductores de Lenguajes I

NASM Introduction Manual for Mac

Author:
Abrahan Joseph Pinedo López

Assessor:
Noé Ortega-Sánchez

December, 2018

Contents

List of source codes

1

Introduction

The Netwide Assembler, NASM, is an 80x86 and x86-64 assembler designed for portability and modularity. It supports a range of object file formats, including Linux and BSD a.out, ELF, COFF, Mach-O, 16-bit and 32-bit OBJ (OMF) format, Win32 and Win64. It will also output plain binary files, Intel hex and Motorola S-Record formats. Its syntax is designed to be simple and easy to understand, similar to the syntax in the Intel Software Developer Manual with minimal complexity. It supports all currently known x86 architectural extensions, and has strong support for macros [1].

1.1 History

NASM was originally written by Simon Tatham with assistance from Julian Hall. As of 2018, it is maintained by a small team led by H. Peter Anvin. It is open-source software released under the terms of a simplified (2-clause) BSD license.

NASM was among the first of the Open-Source, freely available, assemblers available for the x86. The project was started in the 1996 time frame as a way of creating a portable x86 assembler that uses a "somewhat Intel Syntax" (as opposed to GNU's Gas, the only other truly portable x86 assembler available at the time). Originally, NASM started out as a copyrighted program similar to FASM. Recently, however, NASM's original authors released NASM to the open software community under the LGPL license.

1.2 Installation

The best way to install NASM on mac is trough homebrew [2]. For the installation of NASM under MS-DOS (Windows) or Unix consult the NASM Manual from the official website [1].

- Install **Homebrew** first: open a Terminal prompt and paste (just remove the backslash "\" before the dollar sign)

```
/usr/bin/ruby -e "\$(curl -fsSL https://raw.githubuserco
↪  ntent.com/Homebrew/install/master/install)"
```

- Install **NASM**

```
brew install nasm
```

1.3 Running NASM

In order to understand the nature of the running of NASM and the options available, we will present your first assembly program called helloFriend.s (Source Code 1), for the moment it's not necessary to know all the details of the file. To assemble a file, you issue a command of the form:

```
nasm -f <format> <filename> [-o <output>]
```

For example, for the file helloFriend.s

```
nasm -f macho64 helloFriend.s
```

will assemble helloFriend.s into an `macho64` object file helloFriend.o. Than to compile and execute the program we type

```
gcc -o hello helloFriend.o
./hello
Hello Friend!
```

Some of the most important options for running NASM are:

- **-o Option: Specifying the Output File Name**
  NASM will normally choose the name of your output file for you; precisely how it does this is dependent on the object file format. For Unix object file formats (aout, as86, coff, elf32, elf64, elfx32, ieee, macho32 and macho64) it will substitute .o. If the output file already exists, NASM will overwrite it, unless it has the same name as the

```
; ------------------------------------------------------------------
; 64-bit hello program, designed for OS X. To assemble and run:
;     nasm -fmacho64 helloFriend.asm && gcc helloFriend.o && ./a.out
; ------------------------------------------------------------------
        SECTION         .data
cad:    db              'Hello Friend!',10,0 ;

        SECTION         .text
        global          _main
        extern          _printf
        default         rel

_main:
        push        rbp
        mov         rbp, rsp ; Preparing the main function

        lea         rdi, [cad] ; The string is on RDI
        mov         rax, 0
        call        _printf

        mov         rax, 0 ; Return 0
        leave
        ret
```

Source Code 1 – Your first hello assembly program. helloFriend.asm

input file, in which case it will give a warning and use nasm.out as the output file name instead.

For situations in which this behaviour is unacceptable, NASM provides the -o command-line option, which allows you to specify your desired output file name. You invoke -o by following it with the name you wish for the output file, either with or without an intervening space. For example:

`nasm -f macho64 helloFriend.asm -o hello`

will create the output file hello.o

- **-f Option: Specifying the Output File Format**
  If you do not supply the -f option to NASM, it will choose an output file format for you itself. In the distribution versions of NASM, the default is always bin. Like -o, the intervening space between -f and the output file format is optional; so -f macho64 and -fmacho64 are both

valid. A complete list of the available output file formats can be given by issuing the command nasm -hf.

- **-l Option: Generating a Listing File**
  If you supply the -l option to NASM, followed (with the usual optional space) by a file name, NASM will generate a source-listing file for you, in which addresses and generated code are listed on the left, and the actual source code, with expansions of multi-line macros on the right For example:

  ```
  nasm -fmacho64 helloFriend.asm -l helloFriend.lst
  ```

- **-g Option: Enabling Debug Information**
  This option can be used to generate debugging information in the specified format. Using -g without -F results in emitting debug info in the default format, if any, for the selected output format. If no debug information is currently implemented in the selected output format, -g is silently ignored

2

NASM Language

2.1 Structure of a NASM Program

Like most assemblers, each NASM source line contains some combination of the four fields:

```
label: instruction operands ; comment
```

As usual, most of these fields are optional; the presence or absence of any combination of a label, an instruction and a comment is allowed. Of course, the operand field is either required or forbidden by the presence and nature of the instruction field. The structure of a NASM program is presented in the figure 2.1. Generally, you put code in a section called .text and your constant data in a section called .data.

2.2 How to write and read from console?

In some way we already saw how to write to console in the hello example from chapter 1 (Source Code 1). In the section we are going to be more specific.

Writing standalone programs with just system calls is cool, but rare. We would like to use the good stuff in the C library. Remember how in C execution "starts" at the function main? That's because the C library actually has the _start label inside itself! The code at _start does some initialization, then it calls main, then it does some clean up, then it issues the system call for exit. So you just have to implement main. We can do that in assembly!

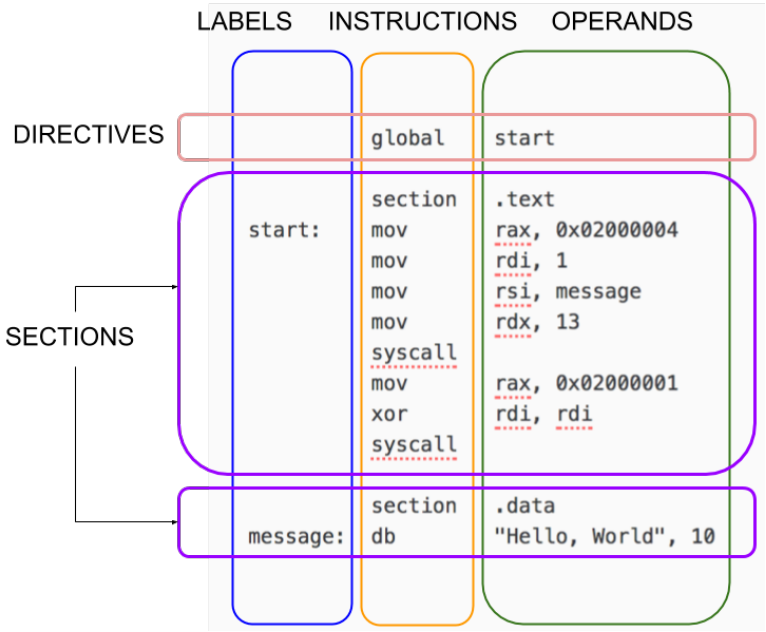

Figure 2.1 – Structure of a NASM program. *Tomado de [3]*

In macOS land, C functions (or any function that is exported from one module to another, really) must be prefixed with underscores. The call stack must be aligned on a 16-byte boundary. And when accessing named variables, a rel prefix is required.

As we can see from the program writeRead.asm (Source Code 2), we have three sections called .data, .bss and .text. We are going to explain the function of each of those.

2.2.1 Pseudo-Instructions

Pseudo-instructions are things which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them. The current pseudoinstructions are DB, DW, DD, DQ, DT, DO, DY and DZ; their uninitialized counterparts RESB, RESW, RESD, RESQ, REST, RESO, RESY and RESZ; the INCBIN command, the EQU command, and the TIMES prefix.

```
; ------------------------------------------------------------------
; Asking for name. Write and Read from console using the C library.
; To assemble and run:
;     nasm -fmacho64 writeRead.asm && gcc writeRead.o && ./a.out
; ------------------------------------------------------------------
        SECTION         .data
n_in:    db             'What is your first name? ',0
n_out:    db             'Your name is %s', 10, 0
f_str:    db             "%s",0

        SECTION         .bss
NAME:    resb        1

        SECTION         .text
        global         _main
        extern         _printf
        extern         _scanf
        default         rel

_main:
        push        rbp
        mov         rbp, rsp ; int main () {

        lea         rdi, [n_in] ; printf("What is your name?");
        mov         rax, 0
        call        _printf

        lea         rsi, [NAME] ; scanf("%s", &NAME);
        lea         rdi, [f_str]
        mov         rax, 0
        call         _scanf

        lea         rsi, [NAME] ; printf("Your name is %s\n", NAME);
        lea         rdi, [n_out]
        mov         rax, 1
        call        _printf

        mov         rax, 0
        mov         rsp, rbp
        pop         rbp
        ret                     ; return 0 };
```

Source Code 2 – Input and output. writeRead.asm

2.2.1.1 section .data, initialized varables

The data section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names, or buffer size, etc. The variables are defined using this directives

- `db:` Byte type variable, 8 bits.
- `dw:` Word type variable, 16 bits.
- `dd:` Double word type variable, 32 bits.
- `dq:` Quadruple word type variable, 64 bits.

In the given example (Source Code 2) the variables declared are:

```
n_in: db 'What is your first name? ',0 ;For name input
n_out: db 'Your name is %s', 10, 0 ; For name output, 10 is for breakline
f_str: db "%s",0 ;To store the name
```

2.2.1.2 section .bss, uninitialized variables

The bss section is used for declaring uninitialized variables. The variables are defined using thes corresponding directives

- `resb:` Reserve space in Bytes units, 1 byte
- `resw:` Reserve space in Word units, 2 bytes
- `resd:` Reserve space in Double Word units, 4 bytes
- `resq:` Reserve space in Quadruple Word units, 8 bytes

In the given example (Source Code 2) the variables are:

```
NAME: resq 1 ; to save the name
```

Notice how in this example we used `resq` since we only need the pointer to the start of the string. There are other ways to defined initialized and uninitialized data, you can have a look at the NASM docs to further explanation [1].

2.2.1.3 section .text, instructions of the program

Finally in this section we present the instructions for the actual program. In C programming would be something like this:

```
int main() {
    printf("What is your name? ");
    scanf("%s", &NAME);
    printf("Your name is %s", NAME);
    return 0;
}
```

2.2.1.4 Definition of other elements

Other element commonly present are:

- **extern**
  Declare a symbol like external. We use it if we want to access a symbol that is not defined in the file we are assembling, but in another source code file, in which it will have to be defined and declared with the global directive. For example when we use it to invoke the *printf* and *scanf* functions from the C library.

- **global**
  It is the complementary directive of extern. It makes visible a symbol defined in a source code file in other source code files; In this way, we can refer to this symbol in other files using the extern directive.

  For example a subroutine that takes two inputs and add them called sum.asm with a global symbol _sum can be called from other C program. In our example we have:

```
global     _main
extern     _printf
extern     _scanf
```

2.3 Instructions and Addressing Modes

An assembly instruction consists of an operation code (the name of the instruction) that determines what the instruction should do, plus a set of operands that directly express a data, a record or a memory address; the different ways of expressing an operand in an instruction and the associated procedure that allows obtaining the data is called addressing mode [4].

1. **Instructions without any explicit operand:** opcode

```
ret    ; return from a subroutine
```

2. **Instructions with one operand:** opcode destiny

```
push rax          ; save rax on stack
pop rax           ; load the top of stack on rax
call _printf      ; calls for printf subrutine
inc rax           ; rax=rax+1
```

3. **Instructions with two operands:** opcode destiny, source

```
mov rbp, rsp   ; move the stack to bas pointer
add rax, 4     ; rax=rax+4
```

2.3.1 Addressing Modes

When an operand is in memory, it is necessary to consider which mode of addressing we use to express an operand in an instruction to define the way to access a specific data. Next, we will look at the addressing modes that we can use in an assembly program:

1. **Immediate.** In this case, the operand refers to a data found in the instruction itself. There is no need to do any extra memory access to obtain it. We can only use an immediate address as a source operand. For example:

```
mov rax, 1                ; Immediate mode, loads 1 to rax
```

In the example showed, rax needed to be 1 because we have one parameter

```
printf("Your name is %s", name);
```

2. **Register Direct.** In this case, the operand refers to a data that is stored in a record. In this addressing mode we can specify any general purpose record (data records, index registers and pointer records).

```
mov rbp, rsp        ; Register direct mode, aligning the stack
```

3. **Memory Direct.** In this case, the operand refers to a data that is stored in a memory location. The operand must specify the name of a memory variable in brackets []; It should be remembered that in NASM syntax the name of a variable without curls is interpreted as the address of the variable and not as the content of the variable. For mac users the next line must be used.

```
 default rel
```

   In the writeRead.asm (2) example we have:

```
lea rsi, [NAME]     ; Memory direct mode from var [name] to rsi
```

   where `lea` means load effective address.

4. **Stack Mode.** Implicitly works with the top of the stack, by means of the register stack pointer; in the x86-64 architecture this record is called rsp. In the stack we can only store 16-bit and 64-bit values.

   There are only two specific instructions designed to work with the stack: `push` and `pop`.

```
push rbp         ; loads base pointer to the stack
pop rbp         ; extract base pointer from the stack
```

2.3.2 Types of instructions

We can organize the instructions according to the following types:

1. **Data Transfer Instructions.**

   - `mov dest, src`: generic instruction to move a data from a source to a destination.
   - `push src`: instruction that moves the operand of the instruction to the top of the stack.

- `pop dest`: moves the data that is on top of the stack to the operand destination.

2. **Arithmetic and Comparison Instructions.**

   - `add dest, src`: arithmetic sum of the two operands.
   - `sub dest, src`: arithmetic subtract of the two operands.
   - `inc dest`: increments the operand in one unit.
   - `dec dest`: decrements the operand in one unit.
   - `mul src`: integer multiplication without sign.
   - `imul src`: integer multiplication with sign.
   - `div src`: integer division without sign.
   - `idiv src`: integer division with sign.
   - `neg dest`: arithmetic negation, 2 complement's.
   - `cmp dest, src`: comparison between two operands, it subtracts them without saving the result.

3. **Logic Instructions.**

   - `and dest, src`: logic operation 'and'.
   - `or dest, src`: logic operation 'or'.
   - `xor dest, src`: logic operation 'exclusive or'.
   - `not dest, src`: logic negation bit by bit.

4. **Sequence Instructions.**
   Unconditional jump

   - `jmp label`: Unconditional jump to label.

   Jumps that query a concrete result bit

   - `je label / jz label`: jumps if equal, jumps if bit zero is active.
   - `jne label / jnz label`: jumps if not equal, jumps if bit zero is not active.

   Conditional jump without considering the sign

   - `jb label`: jumps if below.
   - `jbe label`: jumps if below or equal.

- `ja label`: jumps if above.
- `jae label`: jumps if above or equal.

Conditional jump considering the sign

- `jl label`: jumps if lesser.
- `jle label`: jumps if lesser or equal.
- `jg label`: jumps if greater.
- `jge label`: jumps if greater or equal.

Other sequence instructions

- `loop label`: decrements rcx and jumps if rcx is not zero.
- `call label`: call to a subroutine.
- `ret`: returns from a subroutine.

3

Programming Concepts in Assembly

In this chapter we will see how the different control structures of the C language can be translated into assembler language.

3.1 Conditional Statement IF

In C programming we have:

```
if (a > b) {
    maxA = 1;
    maxA = 0;
}
```

We can translate this code in assembly as follows:

```
    mov rax, qword [a]     ; Loads 'a' and 'b' into registers rax and rbx
    mov rbx, qword [b]
    cmp rax, rbx           ; The comparison is made
    jg L1                  ; If condition is true jumps to L1
    jmp L2                 ; If condition is false jumps to L2
L1:
    mov byte [maxA], 1     ; Only executed if condition was true
    mov byte [maxB], 0
L2:
```

3.2 Conditional Statement IF-ELSE

In C programming we may have:

```
if (a > b) {
    max = a;
} else {
    max = b;
}
```

We can translate this code in assembly as follows:

```
    mov rax, qword [a]     ; loads 'a' and 'b' to registers rax and rbx
    mov rbx, qword [b]
    cmp rax, rbx           ; The comparison is made
    jg L1                  ; If condition is true, jumps to L1
    mov byte [max], 'b'    ; else (a <= b)
    jmp L2
L1:
    mov byte [max], 'a'    ; if (a > b)
L2:
```

3.3 Conditional Statement WHILE

In C programming we have the next factorial example:

```
result=1;
while (num > 1){
    result = result * num;
    num--;
}
```

which can be written in assembly as follows:

```
    mov rax, 1               ; rax its going to be the result
while:
    cmp qword [num], 1       ; The comparison is made
    jle L1
    imul rax, qword [num]    ; rax=rax*[num]
    dec qword [num]
    jmp while
L1:
    mov qword [result], rax
```

3.4 Conditional Statement DO-WHILE

Another example in C...

```
result=1;
do {
    result = result * num;
    num--;
} while (num > 1)
```

The difference with the WHILE structure is subtle

```
    mov rax, 1            ; rax its going to be [result]
    mov rbx, qword [num]  ; Loads num in rbx
while:
    imul rax, rbx
    dec rbx
    cmp rbx, 1            ; the comparison is made
    jg while              ; It it is true then jumps tu while
    mov qword [result], rax
    mov qword [num], rbx
```

3.5 Conditional Statement FOR

Finally the FOR structure example in C

```
result=1;
for (i = num; i >1; i--) {
    result = result * i;
}
```

The difference with the WHILE structure is subtle

```
    mov rax, 1            ; rax its going to be [result]
    mov rcx, qword [num] ; rcx [i] initialized with [num]
for:
    cmp rcx, 1            ; The comparison is made
    jg cierto             ; If true jumps to L1
    jmp L2
L1:
    imul rax,rcx
    dec rcx
    jmp for
L2:
    mov qword [result], rax
    mov qword [i], rcx
```

4

Functions and Macros

Procedures or subroutines are very important in assembly language, as the assembly language programs tend to be large in size [5]. In this chapter we are going to present the use of functions (subroutines) along with the passing of parameters in C and we will finish with the explanation of macros.

4.1 Functions

A subroutine is a unit of self-contained code, designed to carry out a specific task and has a determining role in the development of programs in a structured manner.

An assembler subroutine would be equivalent to a function in C. In this section we will see how to define assembler subroutines and how we can use them later from a C program. First we will describe how to work with assembler subroutines:

- Definition of assembler subroutines.
- Call and return of subroutine.
- Passing of parameters to the subroutine and return of results.

Next we will see how to make calls to subroutines made in assembler from a program in C and what implications it has in the passing of parameters.

4.1.1 Subroutines

Basically, a subroutine is a set of instructions that start its execution in a point of code identified with a label that will be the name of the subroutine, and ends with the execution of a `ret` instruction, subroutine return instruction, which causes a jump to the next instruction from where the call was made (call).

Important considerations when defining a subroutine:

- We must store the modified records within the subroutine to leave them in the same state they were in at the time the call was made to the subroutine, except for the records that are used to return a value. To store the modified records we will use the stack.
- To maintain the structure of a subroutine and for the program to work properly, you can not perform jumps to instructions of the subroutine; we will always finish the execution of the subroutine with the instruction ret.

```
subrutine:
    ; Save on the stack
    ; the modified registers inside of the routine
    ;
    ; Routine instructions
    ;
    ; Restore the state of the modified registers
    ; Recover the initial value stored on the stack
    ret
```

4.1.1.1 Passing of parameters

A subroutine may need to be transferred parameters; the parameters can be passed through registers or the stack. The same happens with the return of results, which can be done by means of registration or the stack. We will consider the cases in which the number of input and return parameters of a subroutine is fixed.

As a working example we will show the use of the conditional statements and the way to define subroutines using the next example to calculate the greatest common divisor (Source Code 3).

```
; ------------------------------------------------------------------
; Greatest Common Divisor (gcd) subroutine. It takes two integers
; and returns the gcd. int gcd(int a, int b)
; To assemble:
;     nasm -fmacho64 gcd.asm
; ------------------------------------------------------------------
        global          _gcd

        SECTION     .text
_gcd:
        mov         rbx, rdi        ; rbx = a
        mov         rdi, rsi        ; rdi = b
        test         rbx, rbx        ; a == 0 ?
        jne         L1                ; if a != 0 jmp L1
        test         rsi, rsi        ; b == 0
        jne         L1                ; if b != 0 jmp L1
        mov         rdi, 1            ; a == b == 0 entonces b = 1
        mov         rax, rdi        ; Ready to return
        ret
L1:
        test         rdi, rdi        ; b == 0 ?
        jne         L2                ; if b != 0 jmp L2
        mov         rax, rbx        ; Ready to return
        ret
L2:
        test         rbx, rbx        ; a == 0 ?
        je            L5
L3:
        cmp         rbx, rdi
        je            L5
        jae         L4
        sub         rdi, rbx
        jmp         L3
L4:
        sub         rbx, rdi
        jmp         L3
L5:
        mov         rax, rdi        ; Ready to return
        ret
```

Source Code 3 – Conditional statements. gcd.asm

4.1.1.2 Calling assembly functions from C

To use assembler functions within a C program we must define them in the C program, but without implementing them; only the header of the functions is included. Headers include the return type of the function, the name of the function, and the data types of each function parameter.

Once the functions have been defined, they can be called like any other program function. To show this we are going to assembly two codes: gcd.asm (Source Code 3) and factorial.asm (Source Code 4).

```
; ------------------------------------------------------------------
; Factorial. It takes an integer input and returns its factorial
; To assemble:
;       nasm -fmacho64 factorial.asm
; ------------------------------------------------------------------
        global  _factorial

        SECTION .text
_factorial:
        cmp     rdi, 1          ; n <= 1?
        jnbe    L1              ; If not, make a recursive call
        mov     rax, 1          ; Other way return 1
        ret
L1:
        push    rdi             ; save n on stack
        dec     rdi             ; n-1
        call    _factorial      ; factorial(n-1), rax = result
        pop     rdi             ; retrieve n
        imul    rax, rdi        ; n * factorial(n-1), save in rax
        ret
```

Source Code 4 – Cyclic statements. factorial.asm

Now we can write a short code in C to use our assembly codes gcd.asm and factorial.asm, but first we need to assembly the codes.

```
nasm -fmacho64 gcd.asm
nasm -fmacho64 factorial.asm
```

This will generate the object files gcd.o and factorial.o and then we can link this with a C program for the passing of arguments (Source Code 5).

```
gcc gcd.o factorial.o menu.c && ./a.out
```

```c
/*****************************************************************
 * C program menu for gcd.o and facto.o
 * To assembly and run:
 *     gcc gcd.o facto.o menu.c && ./a.out
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

int factorial(int n);
int gcd(int a, int b);

int main() {
    int n, a, b, option, continueMenu = 1;
    do {
        system("clear");
        printf("\nMENU\n");
        printf("1) Factorial\n");
        printf("2) Greatest Common Divisor\n");
        printf("3) EXIT\n");
        printf("Choose an option: ");
        scanf("%d",&option); getchar();
        switch(option){
            case 1: printf("\n--- FACTORIAL ---\n");
                printf("Enter an integer number: ");
                scanf("%d", &n); getchar();
                printf("factorial(%d) = %d \n", n, factorial(n)); break;
            case 2: printf("\n--- GREATEST COMMON DIVISOR ---\n");
                printf("Enter the first number: ");
                scanf("%d",&a); getchar();
                printf("Enter the second number: ");
                scanf("%d",&b); getchar();
                printf("GCD = %d \n", gcd(a,b)); break;
            case 3:
                continueMenu = 0; break;
            default:
                printf("\nIncorrect option\n");
        }
        if(continueMenu){
            printf("\nPress any key to continue..."); getchar();
        }
    } while (continueMenu);
    return 0;
}
```

Source Code 5 – Passing arguments. menu.c

4.2 Macros

Writing a macro is another way of ensuring modular programming in assembly language.

- A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.
- In NASM, macros are defined with %macro and %endmacro directives.
- The macro begins with the %macro directive and ends with the %endmacro directive.

The syntax fro the macro definition is like follows:

```
%macro macroName  numberParams
    ; macro body
%endmacro
```

Where, numberParams specifies the number parameters, macroName specifies the name of the macro. The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

For example, a very common need for programs is to write a string of characters in the screen. For displaying a string of characters, you need the following sequence of instructions

```
    mov    edx, len          ;message length
    mov    ecx, msg          ;message to write
    mov    ebx, 1          ;file descriptor (stdout)
    mov    eax, 4          ;system call number (sys_write)
    int    0x80            ;call kernel
```

In the above example of displaying a character string, the registers EAX, EBX, ECX and EDX have been used by the INT 80H function call. So, each time you need to display on screen, you need to save these registers on the stack, invoke INT 80H and then restore the original value of the registers from the stack. So, it could be useful to write two macros for saving and restoring data.

We have observed that, some instructions like IMUL, IDIV, INT, etc., need some of the information to be stored in some particular registers and even return values in some specific register(s). If the program was already using those registers for keeping important data, then the existing data from these registers should be saved in the stack and restored after the instruction is executed (Source Code 6).

```
; A macro with two parameters
; Implements the write system call
   %macro write_string 2
      mov   rax, 4
      mov   rbx, 1
      mov   rcx, %1
      mov   rdx, %2
      int   80h
   %endmacro

   SECTION  .data
msg1 db  'Hello, programmers!',0xA,0xD
len1 equ $ - msg1

msg2 db 'Welcome to the world of,', 0xA,0xD
len2 equ $- msg2

msg3 db 'Linux assembly programming! '
len3 equ $- msg3

   SECTION    .text
   global _start            ;must be declared for using gcc

_start:                     ;tell linker entry point
   write_string msg1, len1
   write_string msg2, len2
   write_string msg3, len3

   mov rax,1                ;system call number (sys_exit)
   int 0x80                 ;call kernel
```

Source Code 6 – Macro example. macro.c

Bibliography

[1] The NASM development team. Nasm - the netwide assembler. `https://www.nasm.us/xdoc/2.14/html/nasmdoc0.html`. Accessed: December, 2018.

[2] Homebrew. `https://brew.sh/`. Accessed: December, 2018.

[3] Nasm tutorial. `http://cs.lmu.edu/~ray/notes/nasmtutorial/`. Accessed: December, 2018.

[4] Miquel Albert Orenga and Gerard Enrique Manonellas. *Estructura de Computadores*. Universitat Oberta de Catalunya, 2011.

[5] Nasm tutorials point. `https://www.tutorialspoint.com/assembly_programming/index.htm`. Accessed: December, 2018.