

**“AZƏRBAYCAN HAVA YOLLARI” CJSC NATIONAL AVIATION ACADEMY**

**Course Work**

## **Profession :Computer Engineering**

**Faculty:Aerocosmic**

**Subject: System software and Operating systems-2**

**Group: 1459i**

**Teacher: Mehemmed Shahmaliyev**

**Student: Mammadov Ahmad**

## **Topic:** **Standard functions in assembly language and their usage**

**Date: 30.05.2022**

**Baku 2022**

**Contents**

Introduction 3

**CHAPTER 1. Standard functions in assembly language and their usage** 4

1.1 Functions 4

1.2 How functions work? 5

1.3. Assembler functions that work by calling C 6

1.4 Loss of registry prices 10

1.5 Recursive functions 11

1**.**6 Return the results of functions to the register 14

1.7 Return function results in a stack 15

**CHAPTER 2. Practise** .16

Result23

Source24

**Introduction**

In 1952, Grace Hopper, an American woman, created Assembler, the world's first mnemonic programming language. Its name is derived from the English word "assemble", which means "to assemble". It included a system of mnemonic commands (command lists), a library of procedures, and a special program to convert program texts into machine code. Such a procedure for obtaining machine code is called compiling, and the program that implements it is called a compiler. These were also invented by Grace M. Hopper.

Assembler is unlike any other programming language and is a completely different language. The fact that the Assembler language is a 2nd generation language and the development of 3rd generation languages ​​on the assembler does not mean that the Assembler language is outdated and is no longer used.

Assembler language is very important in programming today, and to imagine modern programming without it is like to imagine a computer without a processor. Assembler language is currently being developed, various standards are being developed and tested by various institutes and IT companies.

The possibilities of assembler language are endless. The possibilities of other languages ​​are finite.

For example, only a program written in Assembler can control the processor's response to external and internal signals. However, it is possible to manage memory access modes with a program written in assembler language.

Assembler is currently only used in system programming.

**CHAPTER1. Standard functions of Assembler language and their use.**

* 1. **Functions**

The purpose of a function is to return some open value. The purpose of the procedure is to perform some operations. We use proc / endp directives to declare a function in assembly language. All rules and methods applied to procedures apply to functions.

To ensure that programmers work together in groups, they need to be able to divide the program into different parts. This division is determined by a clearly defined agreement between them. In this way, different parts can be developed and tested by different creators, which allows several creators to work on the same project. Programmers use functions to divide their programs into parts.

A function is a piece of code that performs certain operations on a given data. For example, in the program I can set the function written\_herfe\_bax, which is activated every time the user enters any character. The information that the function performs is probably the input character and the document that the user is working on. The function will change the document according to the entered symbol.

The data transmitted to a function is called a parameter. A normal program usually consists of 100 or 1000 small and well-written functions.

* 1. **How functions work?**

The functions consist of several parts:

The name of the functions

A function name is an expression that contains the start address of the function code. To specify a function name, simply place the function name at the beginning of the function code.

Function parameters

The parameters of a function are the information given to the function when it is called. For example, consider the sinusoidal function we know from mathematics. If we ask the computer the sine of an angle of 30 degrees, then 30 degrees will be the parameter of this function. Other features may require more settings.

Local variables

Local variables are the part of memory used by a function during execution, which is discarded when the function returns. You can take it as a draft paper. Each time the function is activated, it takes up as much paper as it needs and is discarded when it finishes its work. One function does not see the local variables of another function.

Static variables

Static variables are parts of memory that the function uses, but does not discard it when the function ends, and uses it again the next time the function is called. The information in this section is not visible to the rest of the program, ie the information contained herein cannot be read or modified. Static variables are only used when needed because they can cause unexpected problems.

Global variables

Global variables are the part of memory that a function uses, but these variables are declared outside the function. Return address The return address is an "invisible" parameter of the function, so it is not used directly. This parameter indicates from which address the program should continue to run after the function is finished. The function needs to know this, because the function can be called from different parts of the program and must return the execution to the address to which it was called upon completion.

**1.3. Assembler functions that work by calling C**

The most basic concept of assembler language is STEK. We cannot write a function in assembly language without knowing how the computer stack works. For normal operation of functions, each program allocates a special place in the kernel memory called a stack.

The stack is also part of a certain memory. Each program is part of its own stack. You can post information to the header of the steak via pushl. The top of the stack is the bottom of the stack memory. Although this is confusing, we accept that something has been added to the stack or that the removal operations are performed on top of it.

For example, in a machine gun (magazine) bullets are added to the top of the comb (when filling the comb) and removed from the top (when firing). This is an example of how the stack works. Here all the processes take place at the top of the stack, we never add or remove anything from the bottom. However, on a computer, the top of the stack grows downwards. System architects have come up with such a design for the stack. Therefore, when we say the top of the stack, we mean the lower part of the stack memory. The popl instruction is used to remove the elements placed on the stack. This instruction deletes the top value of the stack from there and places it in the register or in a given memory address.

When we place a new element on the stack, the top of the stack slides appropriately to accept the new element. If we continue to place values on the stack indefinitely, it will continue to grow until it reaches other parts of the data or code in memory.

But how do we know where the top of the stack is currently located in memory? The stack register **% esp** always contains the address of the top of the stack. Whenever we post any information to the stack using pushl, the value of **% esp** is reduced by 4 units to indicate the updated address of the top of the stack. If we want to get any information from the stack, we use popl, which in turn renews the value of **% esp**, ie increases by 4 units. Both pushl and popl instructions are single-operand instructions. A register that contains information that can be retrieved from the stack or placed on the stack. If we just want to know the information at the top of the stack and do not want to delete it from the stack, then we use the indirect routing rule and the **% esp** register. For example, this code writes the information at the top of the stack (4 bytes) to **% eax** and has no effect on the stack.

movl (%esp), %eax

If we wrote movl **% esp,% eax**, then **% eax** would contain the address of the top of the stack instead of the information in that section. An indirect reference is used to enclose **% esp** in parentheses, and we obtain information from the address referenced by **% esp**. If we want to write the information in the section immediately after the start of the stack to **% eax**, we use the following instruction: **movl 4 (% esp),% eax** This instruction uses the "base pointer addressing" method. There are only 4 on **% esp** and it looks at the resulting address and copies the information to **% eax.**

movl 4(%esp), %eax

This instruction uses the "base pointer addressing" method (see $ 2), which returns only 4% to% esp and transfers the data to% eax by looking at the resulting address..

The stack is the most basic element for working with local variables, return addresses, and parameters of a function according to the C call order. Before performing a function, the program places all the parameters of the function on the stack in reverse order. The program then performs the required function using the call instruction. call instruction does 2 jobs. First, it places the address (return address) of the next instruction to be executed on the stack.

It then updates the value of the instruction pointer (% eip) to refer to the beginning of that function. Thus, at the start of the function, the stack image looks like this (the stack header is at the end of this example):

Parameter #N

...

Parameter 2

Parameter 1

Return Address <--- (% esp)

The program code that calls the function first places all the parameters to be passed to the function, then the return address on the stack and calls the function (passes the execution to the function). Thus the execution is transferred to the function. The first thing the function will do is place the% ebp register on the stack.

The% ebp register is a special register used to refer to the local variables and parameters of a function placed on the stack. The function then writes the value of% esp to% ebp.

movl %esp, %ebp

You can now access the local variables and parameters of the function stacked using the% ebp. You can assume that from now on,% ebp replaces% esp. However, when a function is executed, you can place other information on the stack, such as setting the parameters of other functions, which in turn will change the value of% esp.

Currently, the condition of the stack is as follows:

Parameter #N <--- N\*4+4(%ebp)

...

Parameter 2 <--- 12(%ebp)

Parameter 1 <--- 8(%ebp)

Return Address <--- 4(%ebp)

Old %ebp <--- (%esp) and (%ebp) !! (movl %esp, %ebp -before)

As you can see, each parameter can be accessed via% ebp using the "base pointer addressing" reference method. The function then allocates space on the stack for the local variables it needs. Let's say we need 8 bytes (2 words) of memory to perform a function. For this purpose, we can simply slide the price of the stack down by 8 units to allocate space in memory.

Parameter #N <--- N\*4+4(%ebp)

...

Parameter 2 <--- 12(%ebp)

Parameter 1 <--- 8(%ebp)

Return Adress <--- 4(%ebp)

Old %ebp <--- (%esp) and (%ebp) !! (movl %esp, %ebp -before)

Local variables 1 <--- -4(%ebp)

Local variables 2 <--- -8(%ebp) and (%esp)

Thus, we can now use the "base pointer addressing" addressing method to access the information we want as a result of% ebp and the corresponding slip. % ebp was created specifically for this purpose. You can use other registers for the same purpose, but in the x86 architecture, access to% ebp is much faster.

Performs the following 3 tasks while completing the function:

1. Places the return address in% eax.

2. Returns the stack to its previous state (when the function is called).

3. Returns the execution to the place where the function is called.

The execution is returned to the previous address via the ret instruction.

ret instruction The information in the stack header (remember the address of the next instruction to be executed, the last one in the stack when we call the function

what did we include?) and writes to% eip.

Thus, before the function returns, it must return the stack to its current state when the function is called. In other words,% esp and% ebp when the function returns

The values of must be the same as when the function was called. To do this, as a rule, we complete the function as follows:

movl %ebp, %esp

popl %ebp

ret

At this point, it can be assumed that all local variables are gone. This is because you have returned the stack pointer to its previous state, and the next data to be placed on the stack will be overwritten by those that are there (your previous local variables).

Execution is now in the code section that calls the function again. This part of the code places the parameters that must be passed to the function on the stack when calling the function (the function, in turn, places some local variables on the stack, but returns the stack to its previous state when it returns) and

should remove all these parameters from the stack.

**1.4 Loss of registry costs**

When calling the function, you must accept that the data in the registers will be lost. It is only clear that the value of the% ebp register will not change. The price of% eax will change precisely, and it is likely that others will change as well.

If the value of any register is required after the function returns, the values of these registers must be placed on the stack before the function parameters. After you return the function and remove the parameters from the stack, we can restore the previous values of these registers using the popl instruction (as opposed to the input sequence: shown in the example).

**1.5 Recursive functions**

The program calculates the factorial of a given number. The factorial is the product of a given number and all numbers between that number and 1. For example, the factorial of 7 is 7 \* 6 \* 5 \* 4 \* 3 \* 2 \* 1, and the factorial of 4 is 4 \* 3 \* 2 \* 1.

The factorial of a given number is equal to the product of the factorial of that number and 1 unit smaller than that number. For example, a factorial of 4 is equal to a product of a factor of 4 to 3, a factor of 3 is equal to a product of a factor of 3 to 2, a factor of 2 is equal to a product of a factor of 2 to 1, and a factor of 1 is equal to 1.

This appointment is called recursion. That is, within the function, the function itself is referenced. However, in this case, the end of the function must be taken into account, otherwise when the function is called, it will also appeal to itself, and it will itself ... and as a result, an infinite process will be obtained. Completion is the condition that determines the cessation of recursion.

The end state for the factorial function is the case for calculating the factorial of 1. When we reach 1, we stop the factorial and say that the factorial of 1 is 1.

Now let's look at how to construct a factorial function.

1. Check the number

2. Is this number equal to 1?

3. If so, the answer is 1.

4. Otherwise the answer is equal to the product of the factorial of this number (this\_number - 1).

We can note the following advantages of the functions:

1. In order to use functions, other programmers do not need to know anything about them, they just need to know the arguments passed to the function.

2. Functions can be accessed from anywhere in the program, in any number.

These are the main advantages of the functions. Functions are widely used in the design of large programs. In this case, large programs are divided into smaller parts - functions. In fact, almost all programming consists of the compilation and reference of individual functions.

Now let's look at how the factorial function itself is created.

The following line appears before the function starts

.type factorial, @ function

factorial:

The expression .type indicates to the coordination program that it is a factorial function. If we do not want to use this function from other programs, then there is no need to write it. We write it so as not to break the generality.

factorial: The string assigns the address of the instruction following this line to the factorial symbol. Therefore, the call instruction can determine where to go when typing call factorial anywhere in the program.

The first real instructions of the function are as follows.

pushl% ebp

movl% esp,% ebp

As we already know, these instructions prepare to use the stack. You must start all your functions with these two lines.

The next instruction is as follows:

movl 8 (% ebp),% eax

This instruction writes the first parameter in the stack to% eax using the base reference rule. Remember (% ebp) refers to the old% ebp, 4 (% ebp) refers to the return address, and 8 (% ebp) refers to the first parameter. If we recall a little earlier, then we see that this value is 4. So before we called the function, we placed 4 on the stack (pushl $ 4).

Then we check whether the completion condition is met (if the parameter is equal to 1)? If so, we switch to end\_factorial, which contains the return code of the function. We place the value returned by the function in% eax, in which case it is already in% eax. All this is done with the following instructions.

cmpl $ 1,% eax

is end\_factorial

What, though, if the deadline is not met? We had to reduce the value of the parameter by one and return to the factorial function. Therefore, we first reduce% eax by 1 unit.

decl% eax

decl is a reduction instruction. It reduces the value of the operand by 1 unit. incl, on the other hand, increases a unit. After reducing% eax by 1 unit, we place it on the stack, because it is a parameter of the next function we will call. Then we return to the factorial function again!

pushl% eax

call factorial

We called the factorial function. Now we transfer the information we placed on the stack to the required registers.

movl 8 (% ebp),% ebx

Now we have to multiply the result of this numerical factorial function. We just mentioned that the result of the previous function is in% eax. Therefore, we must multiply% eax by% ebx.

imull% ebx,% eax

We store this result in% eax, and our goal is to keep the result of the function in% eax.

At the beginning of the function, we placed% ebp on the stack and wrote the value of% esp to% ebp. Now, by performing the opposite operation, we leave the last stack and activate the previous stack.

end\_factorial:

movl% ebp,% esp

popl% ebp

Now we are ready to return, so we follow the ret instruction.

ret

It takes the information in the stack header and goes to the code section at that address.

While analyzing the call instruction, we said that the call instruction places the address of the next instruction on the stack before calling the given function.

The ret instruction, on the other hand, takes the information (address) placed in the header of the stack (placed by the call) and refers to this address.

**1.6 Return the results of functions to the register**

Like settings, 80x86 notes are the best place to return function results. The getc routine in the UCR Standard Library is a good example of a function. The following notes return the function results:

Use First Last

Bytes: al, ah, dl, dh, cl, ch, bl, bh

Words: ax, dx, cx, si, di, bx

Double words: dx:ax eax, On pre-80386

edx, ecx, esi, edi, ebx On 80386 and later.

16-bitOffsets: bx, si, di, dx

32-bit Offsets: ebx, esi , edi, eax, ecx, edx

Segmented Pointers:es:di, es:bx, dx:ax, es:si Do not use DS.

**1.7 Return function results in a stack**

Another good place where we can return function results is the stack. The main idea here is to click on a set of fake values to create space for the function result.

The following code sequence shows how to return a set of values:

function PasFunc(i,j,k:integer):integer;

begin

PasFunc := i+j+k;

end;

m := PasFunc(2,n,l);

Assemblerda:

PasFunc\_rtn equ 10[bp]

PasFunc\_i equ 8[bp]

PasFunc\_j equ 6[bp]

PasFunc\_k equ 4[bp]

PasFunc proc near

push bp

mov bp, sp

push ax

mov ax, PasFunc\_i

add ax, PasFunc\_j

add ax, PasFunc\_k

mov PasFunc\_rtn, ax

pop ax

pop bp

ret 6

PasFunc endp

**CHAPTER 2. Practise.**

1) We will consider the force function here. To the power function

we will pass the parameter: the number and the force we want to increase this number. For example, if we give the numbers 2 and 3 as parameters, the function will increase the power of 2 from the 3rd power, ie 2 \* 2 \* 2 and return 8 as a result. In order to simplify the program, we will use only numbers greater than 1 and 1. The text code of the program is given below. Then, as usual, space is set aside for explanation.

Remember the text of the program in the pow.s file.

#Purpose: An example of a program for using functions

# This program will calculate 2 ^ 3 + 5 ^ 2

#

.section .data

.section .text

.globl \_start

\_start:

pushl $3

pushl $2

call pow

addl $8, %esp

pushl %eax

pushl $2

pushl $5

call pow

addl $8, %esp

popl %ebx

addl %eax, %ebx

movl $1, %eax

int $0x80

#Purpose: This function increases the given numerical power

.type pow, @function

pow:

pushl %ebp

movl %esp, %ebp

subl $4, %esp

movl 8(%ebp), %ebx

movl 12(%ebp), %ecx

movl %ebx, -4(%ebp)

pow\_per\_start:

cmpl $1, %ecx

je pow\_end

movl -4(%ebp), %eax

imull %ebx, %eax

movl %eax, -4(%ebp)

decl %ecx

jmp pow\_per\_start

pow\_end:

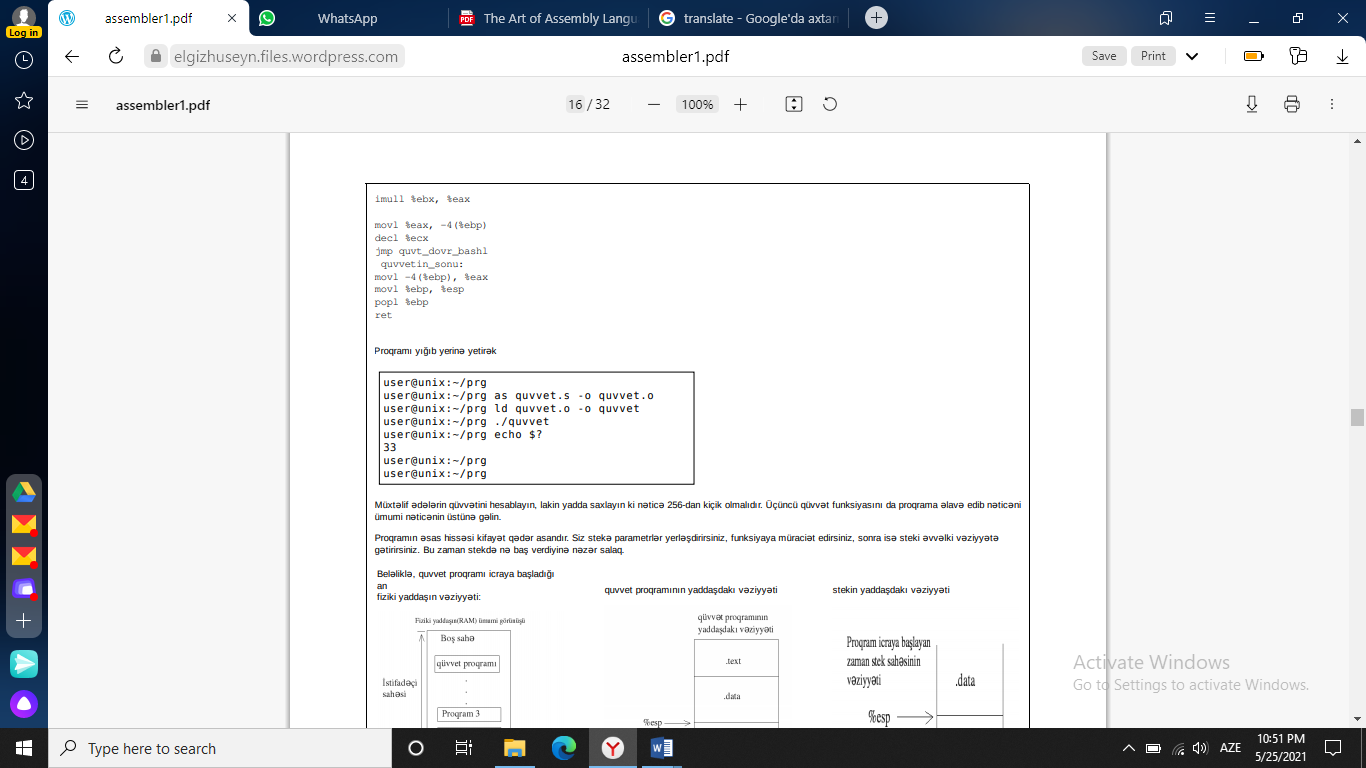
movl -4(%ebp), %eax

movl %ebp, %esp

popl %ebp

ret

Let's assemble and run the program



1. #factorial.s

#Purpose – A program that calculates the factorial of a given number

.section .data

.section .text

.globl \_start

.globl factorial

\_start:

pushl $4

call factorial

addl $4, %esp

movl %eax, %ebx

movl $1, %eax

int $0x80

.type factorial,@function

factorial:

pushl %ebp

movl %esp, %ebp

movl 8(%ebp), %eax

cmpl $1, %eax

je end\_factorial

decl %eax

pushl %eax

call factorial

movl 8(%ebp), %ebx

imull %ebx, %eax

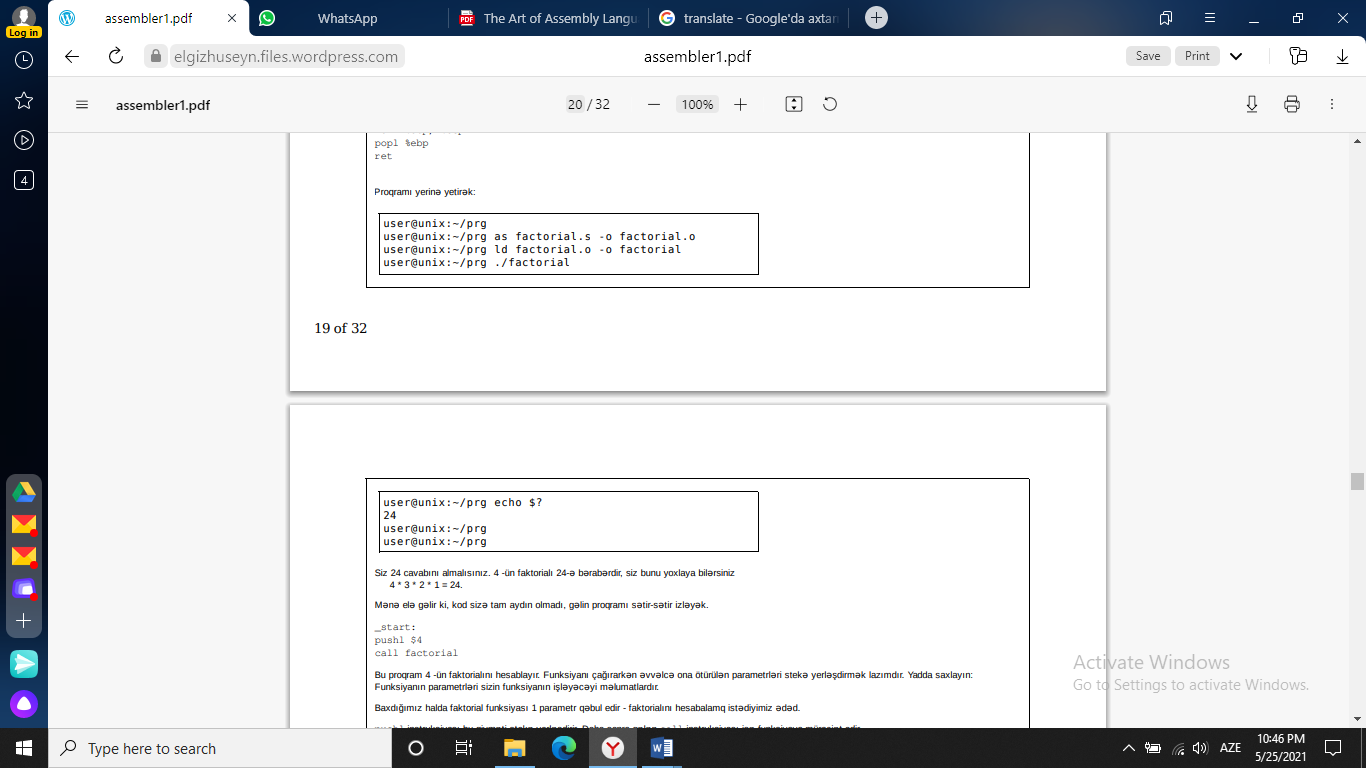
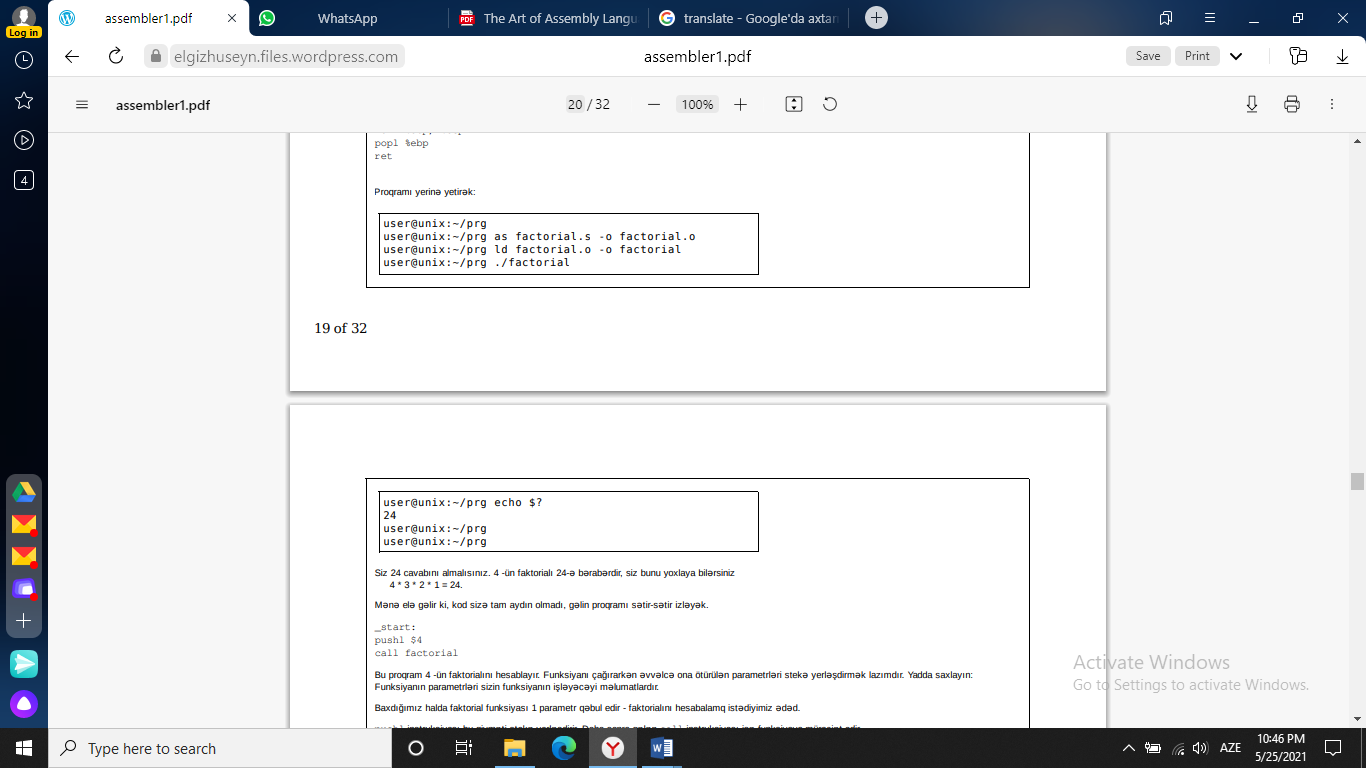
end\_factorial:

movl %ebp, %esp

popl %ebp

ret

Let's run the program:

Answer: 24.

Factorial of 4 is equal 24

4 \* 3 \* 2 \* 1 = 24.

1. #exit.s

#Purpose: Sadə proqram hansı ki, icrasını bitirir və Linux nüvəsinə çıxış kodunu qaytarır

.section .data

.section .text

.globl \_start

\_start:

movl $0, %ebx

movl $1, %eax

int $0x80

The text above is called the source code of the program. Source code is a human-readable form of software. To turn it into a form that a computer can execute, we need to assemble and coordinate it.

First we have to do the assembly. The compilation process is the process of converting what you write into machine instructions. The machine works with numbers, and people prefer words. Assembler language is a state of instruction that the computer understands is close to human language. The compilation operation converts a file in human language into a file in machine language. To perform the assembly process

as exit.s -o exit.o

We enter the command.

Here as is the program that performs the compilation process, exit.s is the source file of the program. The -o exit.o part of the command tells the as program to write the result of the assembly to the exit.o file. The exit.o file is an object file. The object file is a file in machine language, but not yet fully finished. The coordinate program gathers object files together so that the kernel can download and execute them. In our case, since there is only one object file, the coordination program will simply add the necessary information to it so that it can be downloaded. Enter the following command to link the object file.

ld exit.o -o exit

ld is an authentication program, exit.o is the name of the file we want to associate. -o exit ld to write the result to a file named exit. If this

If errors appear on the screen when entering commands, we have made a mistake either in the commands or in the text of the program. It is necessary

After making adjustments, you must re-enter the commands. In general, if you make any changes to the text of the program, then repeat the commands

we have to re-enter. to execute the exit program

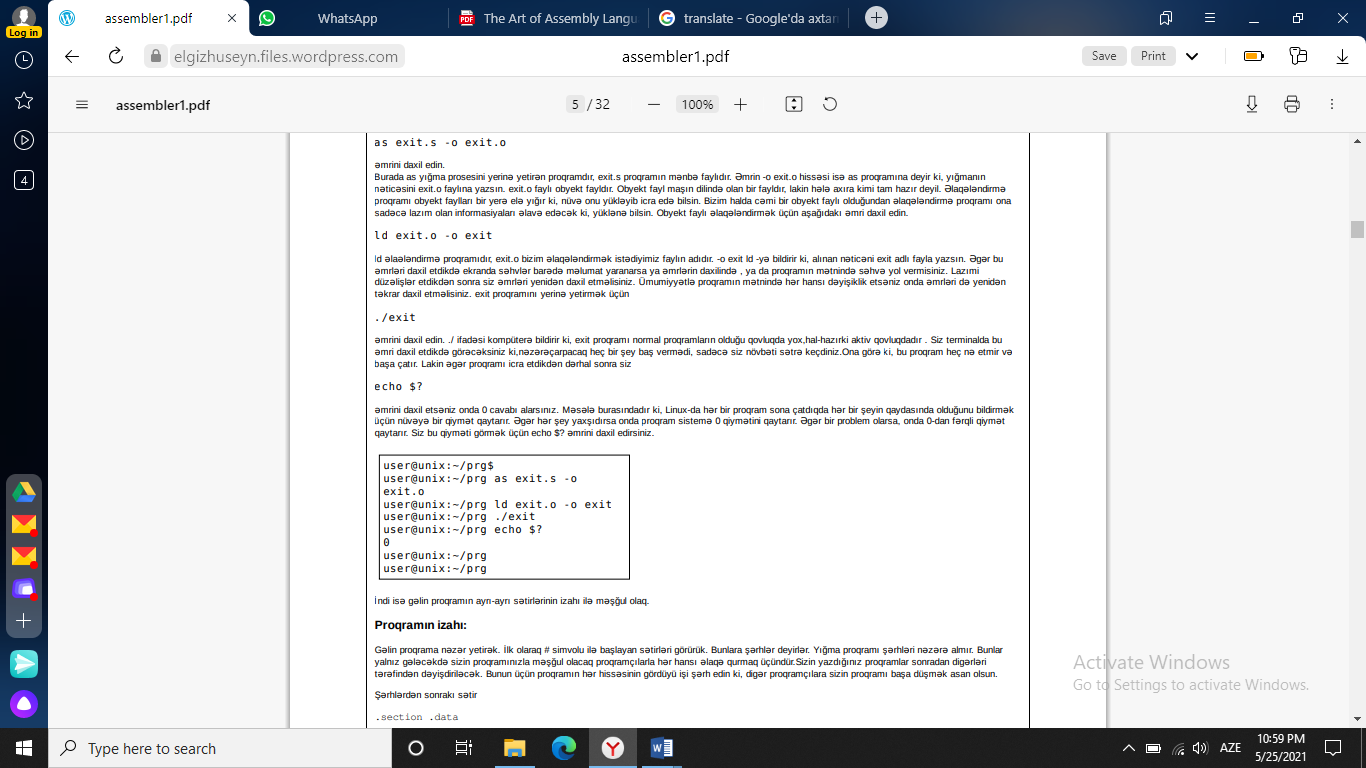
./exit

We enter the command. The ./ statement tells the computer that the exit program is not in the folder where the normal programs are, but in the currently active folder. When we enter this command in the terminal, we will see that nothing noticeable happened, we just went to the next line.

ends. But if immediately after the execution of the program we

echo $?

If we enter the command, we get 0 answers. The problem is that in Linux, each program returns a value to the kernel to indicate that everything is in order when it finishes. If all goes well, the program returns a value of 0 to the system. If there is a problem, it returns a value different from 0. You echo $ to see this price? enter the command.



1. Find the large number

We write the following program to the file maximum.s.

#Purpose: This program finds the bigest of the given numbers

.section .data

data\_items:

.long 3,67,34,222,45,75,54,34,44,33,22,11,66,0

.section .text

.globl \_start

\_start:

movl $0, %edi

movl data\_items(,%edi,4), %eax

movl %eax, %ebx

start\_loop:

cmpl $0, %eax

je loop\_exit

incl %edi

movl data\_items(,%edi,4), %eax

cmpl %ebx, %eax

jle start\_loop

movl %eax, %ebx

jmp start\_loop

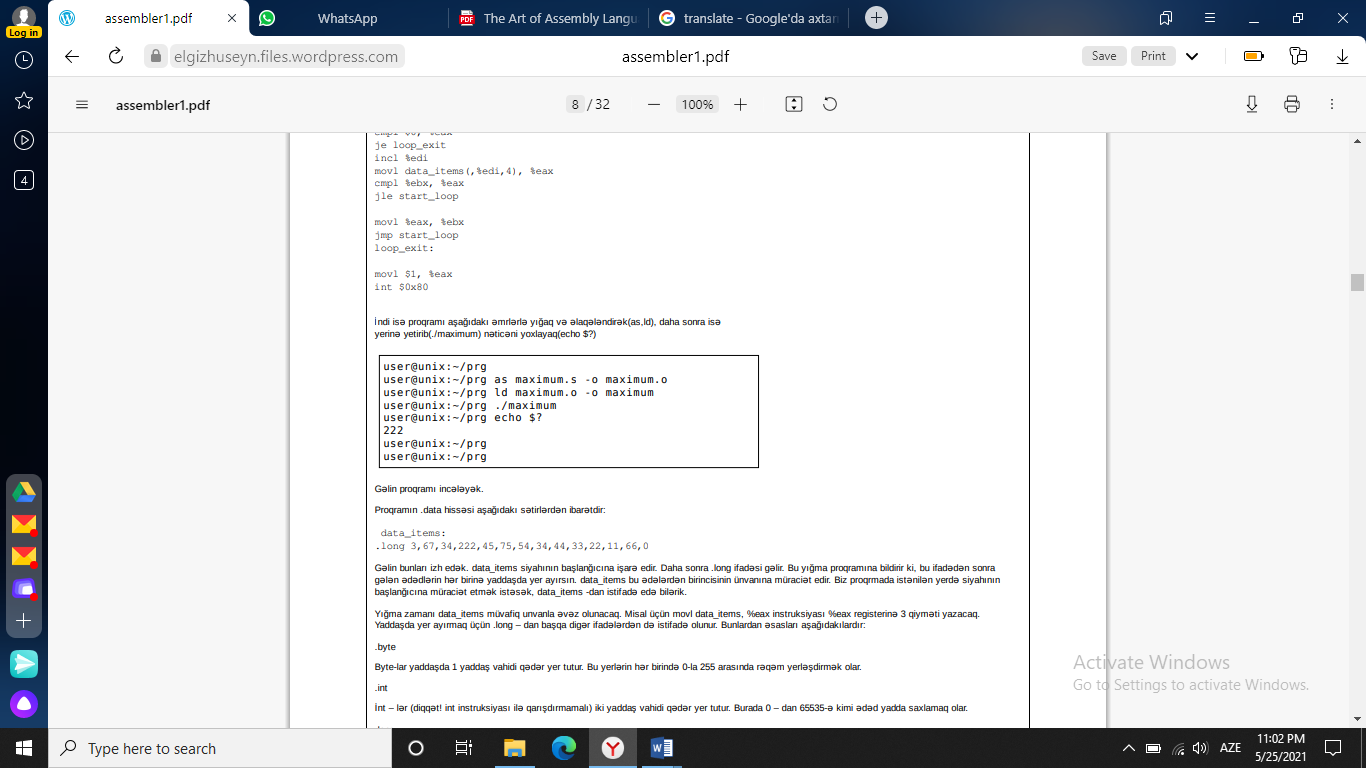
loop\_exit:

movl $1, %eax

int $0x80

Now let's assemble and link the program with the following commands (as, ld), and then

Let's do (./ maximum) check the result (echo $?)



**RESULT**

In this course, we talked about the standard functions of assembler language and their use.

Here we talked about different functions and solved examples on them.

**SOURCE**

1. History of programming languages səh.182 Bakı 2008
2. <https://cs.lmu.edu/~ray/notes/x86assembly/>
3. <https://elgizhuseyn.files.wordpress.com/2010/11/assembler1.pdf>