**LABORATORY WORK 08** (due to end of the class *with possibility to be extended*)

**INTRODUCTION**

This assignment will teach you how to work with loops and arrays in Assembly code.

Learning Objectives:

* Practice conditional instructions
* Practice Labels in Assembly
* Discover how loops are organized in Assembly
* Understand the addressing mode with arrays



To solve the problems, you must know:

* C programming language
* Memory image for processes
* **GCC** compiler

**RULES**

* Remember that: You are not allowed to search for help online!
* Any time you receive help from your teachers, you should acknowledge that in your code with a comment starting with the string “assistance from”.
* You are not allowed to ask other students for help, show them your code, or discuss the specifics of the solution.
* Reconsideration requests must be made within one week of our release of grades for the assignment.

Be aware that you may be asked to explain your code to a member of our course staff using only what you have submitted: your comments in the code should be such that you can determine what your code does and why a few weeks later, if needed.

**INTRO**

**A key to understanding** how the generated assembly code relates to the original source code is to find a mapping between **program** **values** and **registers**. The C compiler will often rearrange the computations, **so that some variables in the C code have no counterpart** in the machine code, and new values are introduced into the machine code that do not exist in the source code. **Moreover, it will often try to minimize register usage** by mapping multiple program values onto a single register.

**Good practice for reverse engineering** is to look at how registers are initialized before the loop, updated and tested within the loop, and used after the loop.

**Note**

Be prepared for surprising transformations, some of which are clearly cases where the compiler was able to optimize the code, and others where it is hard to explain why the compiler chose that particular strategy.

**INSTRUCTIONS**

1. Run Unix OS or similar (for tasks 4)
2. Compile the C source codes with -Og -S command
3. Examine your program output

Your assignment is to complete each task, listed below:

**TASKS**

**All code for your tasks should be well documented (commented) and created as a function.**

**Absence of comments [-10%]**

**Task 1 [25%]:** Copy the following code and save it as lab8\_task1.c.

Compile it till Assembly code:

$ gcc -Og -S lab8\_task1.c

Use the file **lab8\_task1.s** and map each passed argument to the function so\_fun() with used register. (Распишите – прокомментируйте через какие регистры были переданы аргументы в функцию)

#include <stdio.h>

#include <stdlib.h>

int so\_fun (short x, short y, int c, char b) {

int sum =0;

for(int i=0; i<y; i++)

{

sum = sum + (x \* c);

sum+=b;

}

return sum;

}

int main()

{

short x=10, y=25;

int c = 22, res;

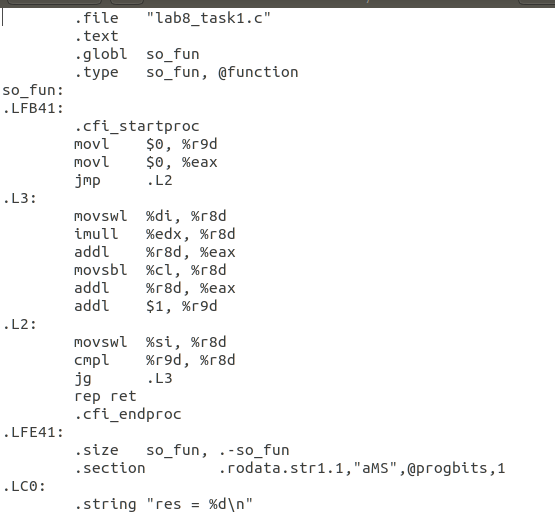
char b = 8;

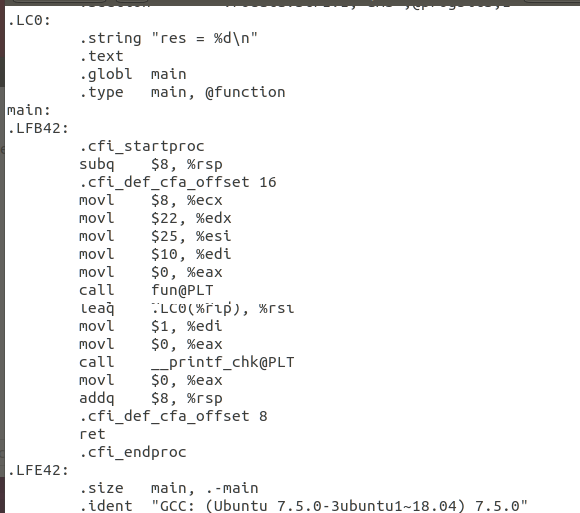
res = fun (x, y, c, b);

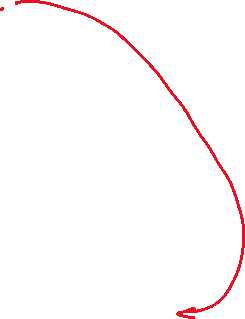
printf("res = %d\n", res);

return 0;}

**lab8\_task1.s :**







%edi – for x

%esi – for y

%edx – for c

%ecx – for b

**Task 2 [35%]:** Loops, loops, loops.

Copy the following code and save it as **lab8\_task2\_1.c file.** The code is very trivial, but we need it for learning purposes.

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \* argv[])

{

long sum = 0;

long i;

for (i = 0; i < 10; i++) {

if (i & 1)

continue;

sum += i;

}

printf("Sum =%d", sum);

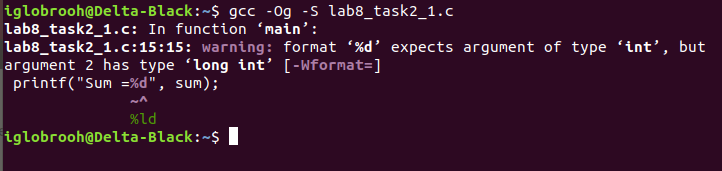
return 0;

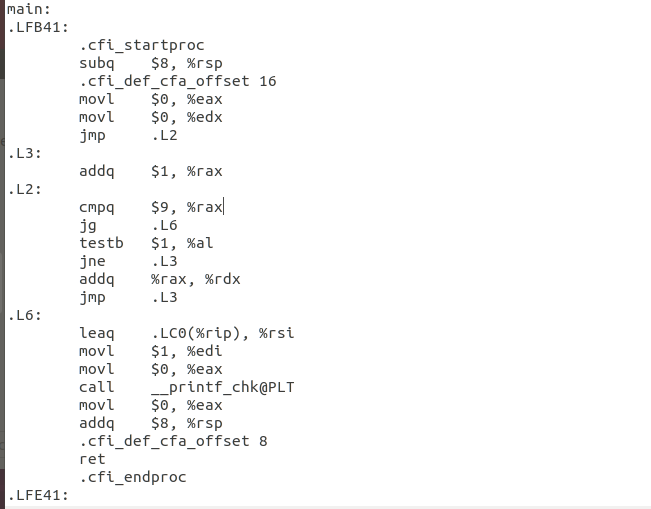
}

Compile this file as the following:

$ gcc -Og -S lab8\_task2\_1.c

**Put the output of your code here:**





1. **Annotate** and **map** each assembly line to the corresponding C code line.

.file "lab8\_task2\_1.c"

.text

.section .rodata.str1.1,"aMS",@progbits,1

.LC0:

.string "Sum =%d"

.text

.globl main

.type main, @function

main:

.LFB41:

.cfi\_startproc

subq $8, %rsp

.cfi\_def\_cfa\_offset 16

movl $0, %eax //long sum = 0;

movl $0, %edx //long i = 0;

jmp .L2

.L3:

addq $1, %rax // continue; (i++)

.L2:

cmpq $9, %rax // i<10

jg .L6

testb $1, %al // if( i&1 )

jne .L3

addq %rax, %rdx //sum += I;

jmp .L3 // i++

.L6:

leaq .LC0(%rip), %rsi

movl $1, %edi

movl $0, %eax

call \_\_printf\_chk@PLT // printf("Sum =%d", sum);

movl $0, %eax

addq $8, %rsp

.cfi\_def\_cfa\_offset 8

ret

.cfi\_endproc

.LFE41:

.size main, .-main

.ident "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"

.section .note.GNU-stack,"",@progbits

1. Re-write the C code using **while loop** instead **of for loop name the file lab8\_task2\_2.c**

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \* argv[])

{

long sum = 0;

long i = 0;

while (i < 10){

if (i & 1){

i++;

continue;

}

sum += i;

i++;

}

printf("Sum = %ld", sum);

return 0;

}

**Awesome!**

Now, compile your file **lab8\_task2\_2.c** till Assembly:

$ gcc -Og -S lab8\_task2\_2.c

Compare two Assembly files - **lab8\_task2\_1.s** and **lab8\_task2\_2.s,**

**Answer the question:**

1. What are the differences in implementation of while and for loops?

There is an additional addq in the **lab8\_task2\_2.s**

**The other is almost identical**

**Task 3 [40%]:** Write any code that works with an array (nested array or two-dimensional) and explore its Assembly code – annotate how array elements are addressed.