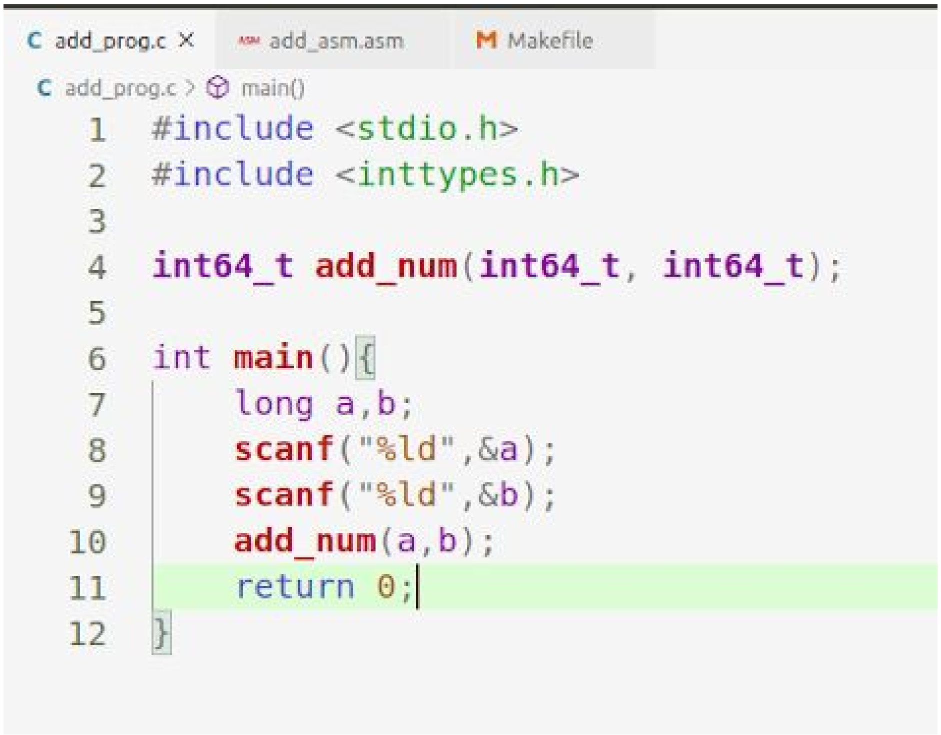
Tushar Mohan
2019393

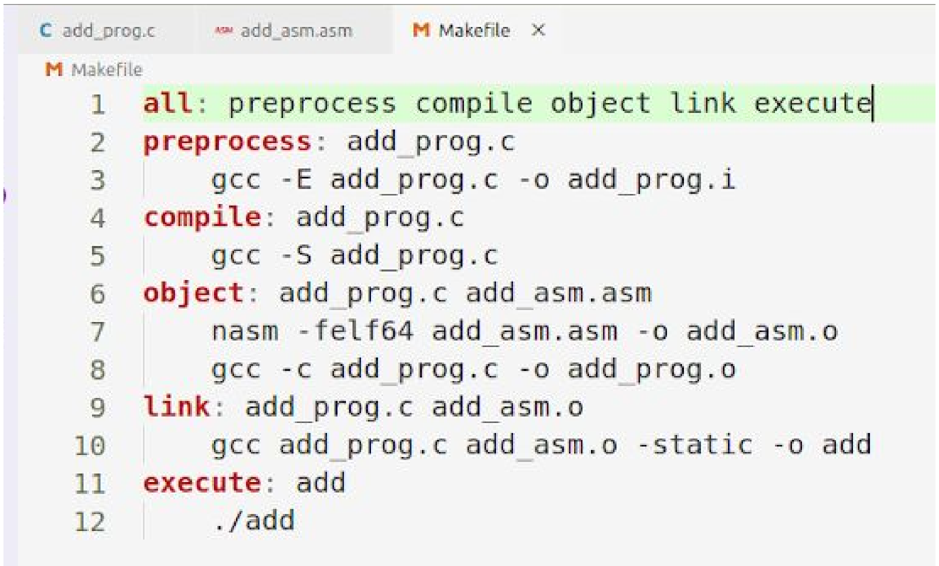
C and Assembly compilation Steps Write-up

- C program



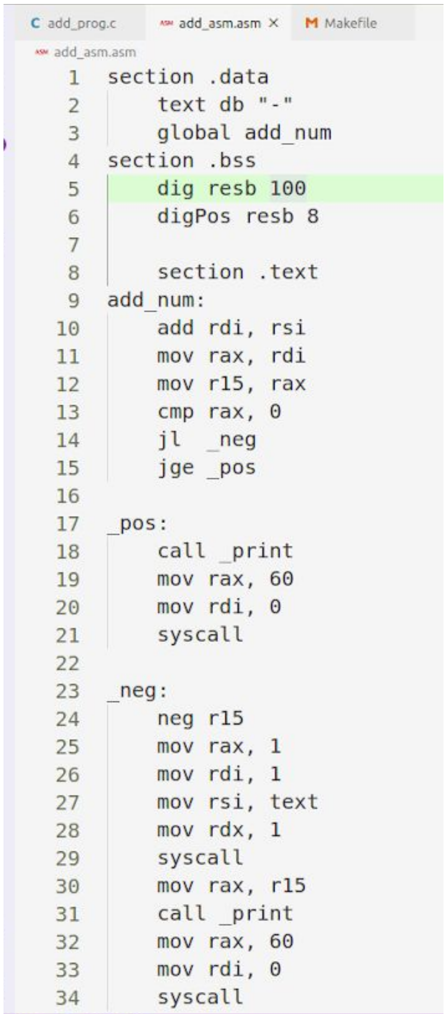
```c
#include <stdio.h>
#include <inttypes.h>

int64_t add_num(int64_t, int64_t);

int main(){
    long a,b;
    scanf("%ld",&a);
    scanf("%ld",&b);
    add_num(a,b);
    return 0;
}
```

- Makefile

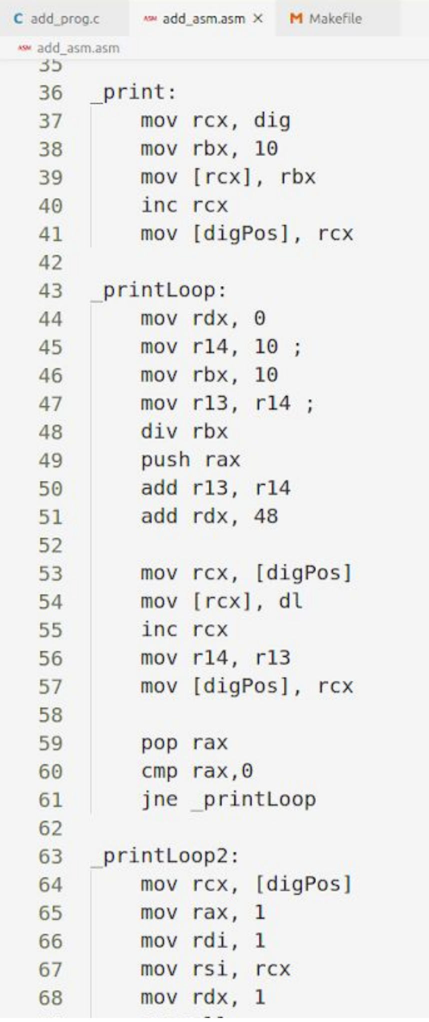


```makefile
all: preprocess compile object link execute
preprocess: add_prog.c
	gcc -E add_prog.c -o add_prog.i
compile: add_prog.c
	gcc -S add_prog.c
object: add_prog.c add_asm.asm
	nasm -felf64 add_asm.asm -o add_asm.o
	gcc -c add_prog.c -o add_prog.o
link: add_prog.c add_asm.o
	gcc add_prog.c add_asm.o -static -o add
execute: add
	./add
```

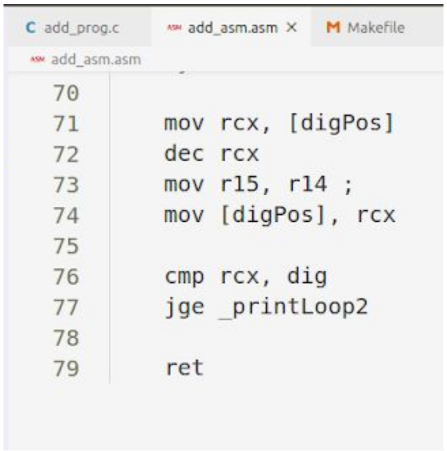
- Assembly program



```
section .data
    text db "-"
    global add_num
section .bss
    dig resb 100
    digPos resb 8

    section .text
add_num:
    add rdi, rsi
    mov rax, rdi
    mov r15, rax
    cmp rax, 0
    jl  _neg
    jge _pos

_pos:
    call _print
    mov rax, 60
    mov rdi, 0
    syscall

_neg:
    neg r15
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 1
    syscall
    mov rax, r15
    call _print
    mov rax, 60
    mov rdi, 0
    syscall
```



```

_print:
    mov rcx, dig
    mov rbx, 10
    mov [rcx], rbx
    inc rcx
    mov [digPos], rcx

_printLoop:
    mov rdx, 0
    mov r14, 10 ;
    mov rbx, 10
    mov r13, r14 ;
    div rbx
    push rax
    add r13, r14
    add rdx, 48

    mov rcx, [digPos]
    mov [rcx], dl
    inc rcx
    mov r14, r13
    mov [digPos], rcx

    pop rax
    cmp rax,0
    jne _printLoop

_printLoop2:
    mov rcx, [digPos]
    mov rax, 1
    mov rdi, 1
    mov rsi, rcx
    mov rdx, 1
```



```

    mov rcx, [digPos]
    dec rcx
    mov r15, r14 ;
    mov [digPos], rcx

    cmp rcx, dig
    jge _printLoop2

    ret
```

1. **Preprocessing step:**

This step includes preprocessing directives mentioned in the code written in C language which are the include lines, typedefs, the define, etc. It also removes all the comments from the code.

Command used to do the above mentioned step:

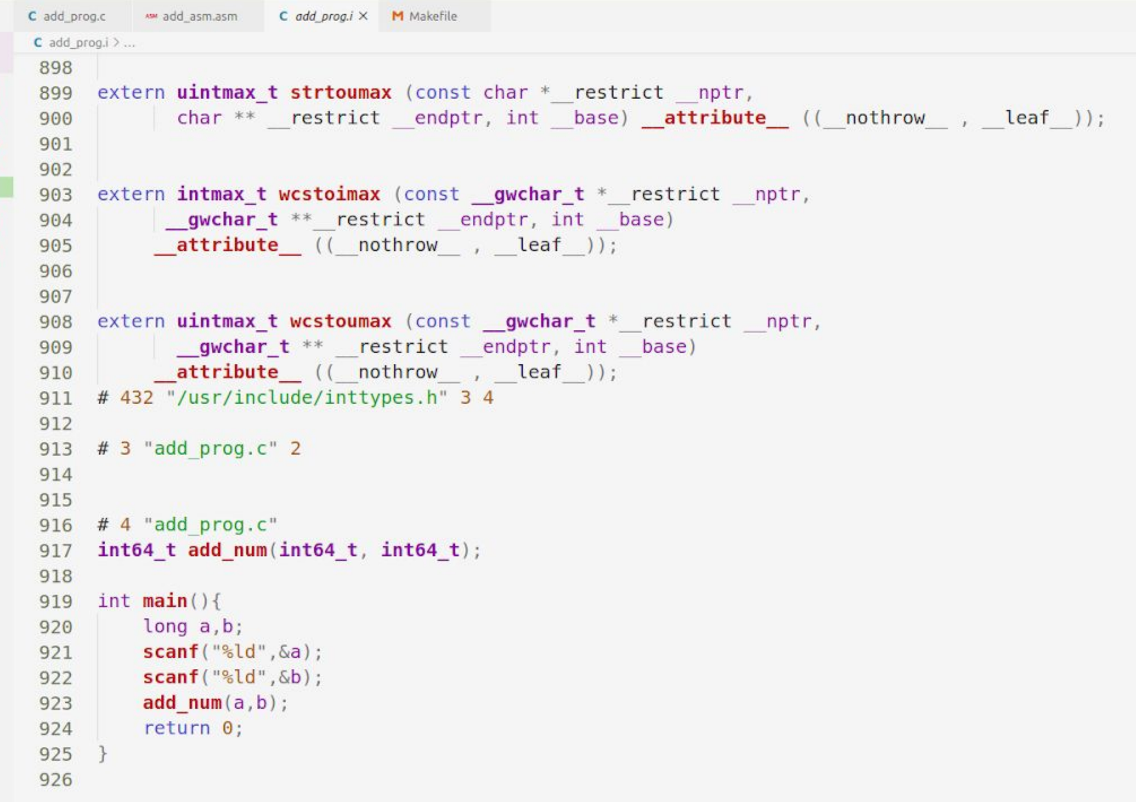
gcc -E add_prog.c -o add_prog.i

Makefile command for the same :

make preprocess

Result of the above command is preprocessed file named “hello.i” which is readable and itself written in C language, but the difference between this and the original C program is that all the preprocessing has been done and it is free from any comments that might be present in the C code. This file includes various new lines introduced by the command we used above.

- An excerpt from add_prog.i:



```

extern uintmax_t strtoumax (const char *__restrict __nptr,
       char ** __restrict __endptr, int __base) __attribute__ ((__nothrow__ , __leaf__));


extern intmax_t wcstoimax (const __gwchar_t *__restrict __nptr,
       __gwchar_t **__restrict __endptr, int __base)
     __attribute__ ((__nothrow__ , __leaf__));


extern uintmax_t wcstoumax (const __gwchar_t *__restrict __nptr,
        __gwchar_t ** __restrict __endptr, int __base)
     __attribute__ ((__nothrow__ , __leaf__));
# 432 "/usr/include/inttypes.h" 3 4

# 3 "add_prog.c" 2


# 4 "add_prog.c"
int64_t add_num(int64_t, int64_t);

int main(){
    long a,b;
    scanf("%ld",&a);
    scanf("%ld",&b);
    add_num(a,b);
    return 0;
}

```

2. **Compilation step:**

This step includes compiling the code written in a high level language. It essentially converts the high level code to assembly language code which will later be converted to machine language code.

Command used to do the above mentioned step:

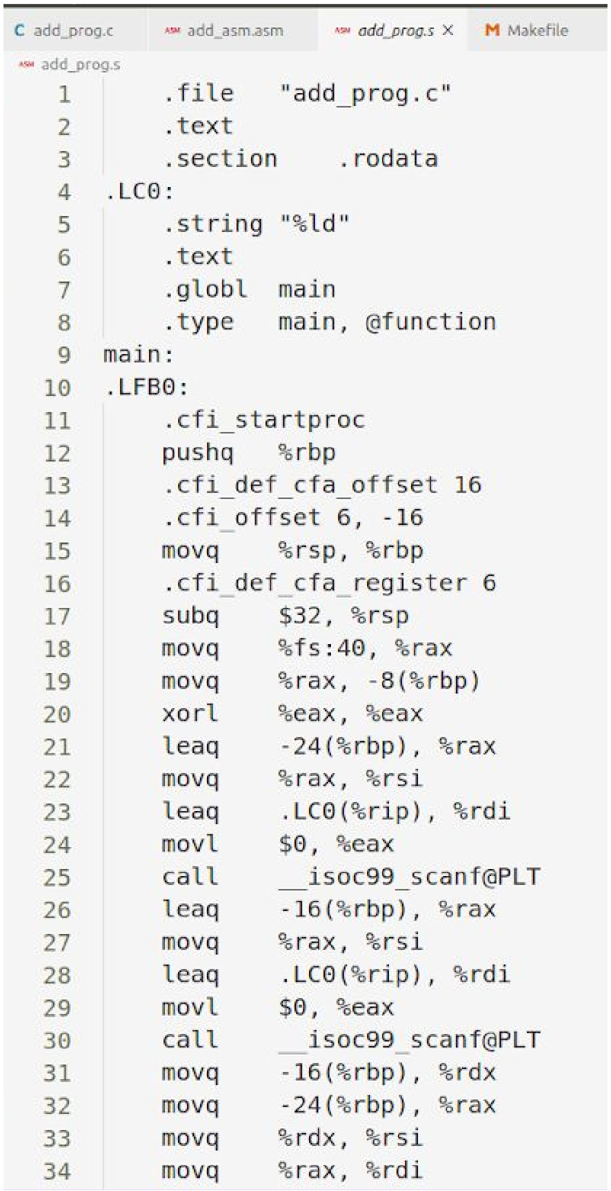
gcc -S add_prog.c

Makefile command for the same:

make compile

Result of the above command is a file named “hello.s” which if viewed will show a code written in assembly language using mnemonics. It has all the dependencies intact and gives an insight as to what is happening at an assembly level.

- An excerpt from add_prog.s:



```
    .file   "add_prog.c"
    .text
    .section    .rodata
.LC0:
    .string "%ld"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    leaq    -24(%rbp), %rax
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    __isoc99_scanf@PLT
    leaq    -16(%rbp), %rax
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    __isoc99_scanf@PLT
    movq    -16(%rbp), %rdx
    movq    -24(%rbp), %rax
    movq    %rdx, %rsi
    movq    %rax, %rdi
```

3. **Assembly step:**

This step includes converting the assembly code into machine language code, i.e., object code. This file is non-readable by a human being.

Command used to do the above mentioned step:

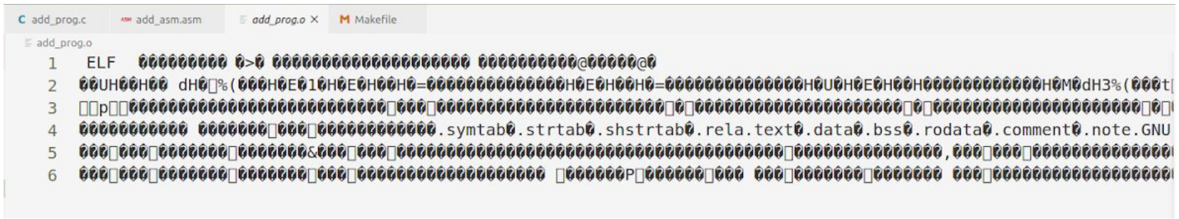
nasm -felf64 add_asm.asm -o add_asm.o
gcc -c add_prog.c -o add_prog.o

Makefile command for the same:

make object

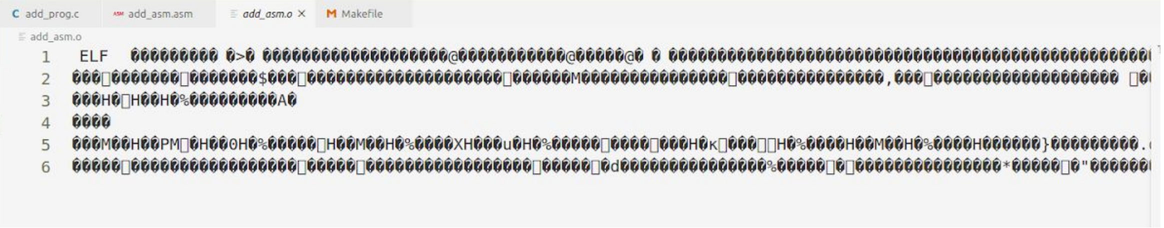
Result of the above commands are 2 files namely “add_asm.o” and “add_prog.o”. The first file is the object file of the assembly language code while the second is the object file of the C program code which are essentially binary codes.

- add_prog.o as available to be read:



```
 ELF   ��������� �>� ���������������������� ����������@�����@�
��UH��H�� dH�□%(���H�E�1�H�E�H��H�=��������������H�E�H��H�=��������������H�U�H�E�H��H������������H�M�dH3%(���t[
□□p□□����������������������������□���□�������������������������□�□����������������������□�□����������������������□�□
����������� �������□���□������������.symtab�.strtab�.shstrtab�.rela.text�.data�.bss�.rodata�.comment�.note.GNU
���□���□�������□�������&���□���□����������������������������������������□���������������,���□���□���������������
���□���□�������□�������□���□��������������������� □������P□������□��� ���□�������□������� ���□�������������������
```

- Add_asm.o as available to be read:



```
 ELF   ��������� �>� ��������������������@������������@�����@� � ����������������������������������������������������
���□��������□�������$���□����������������������□������M����������������□����������������,���□�������������������� □�
���H�□H��H�%���������A�
����
���M��H��PM□�H��0H�%�����□H��M��H�%����XH���u�H�%�����□����□���H�κ□���□□H�%����H��M��H�%����H������}���������.
�����□������������������□�����□�����������������□�����□�d����������������%�����□�□����������������*�����□�"������
```

4. **Linking step:**

This step includes taking into account making a single independent executable file which includes the whole binary code needed to perform the specific function for which different codes were written, here, “add_prog.c” and “add_asm.asm”.

Command used to do the above mentioned step:

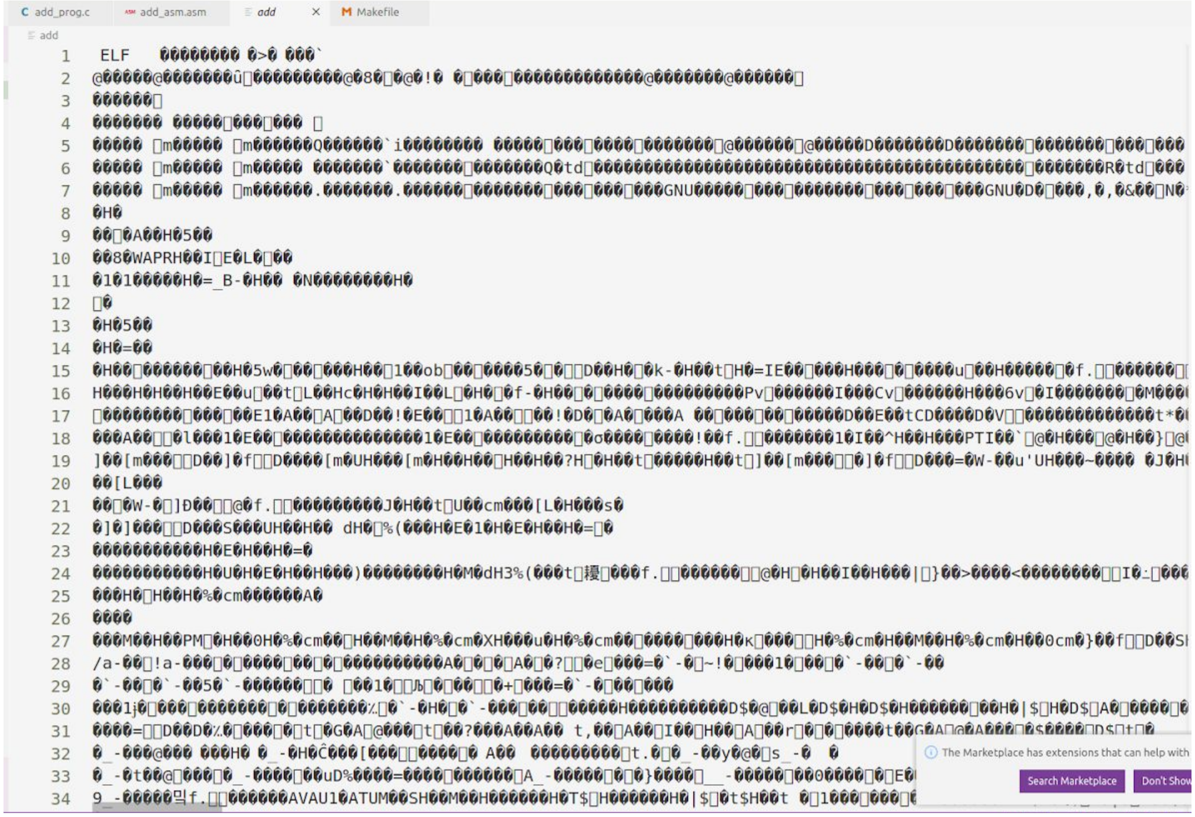
gcc add_prog.c add_asm.o -o add

Makefile command for the same:

make link

Result is “add” file which can then be executed with command: make execute

- An excerpt from add file:



```
 ELF   ��������� �>� ���`
@�����@�������ũ□���������@�8�□�@�!� �□���□��������������@�������@�������□
������□
������� �����□���□��� □
����� □m����� □m������Q������`i��������� �����□���□����□�������□@�������□@�����D��������D��������□��������□���□���
����� □m����� □m����� �������`�������□�������Q�td□�����������������������������������������□�������R�td□���
����� □m����� □m�������.�������.������□�������□���□���□���GNU������□���□��������□���□���□���GNU�D�□���,�,�&��□N�
�H�
��□�A��H�5��
��8�WAPRH��I□E�L�□��
�1�1�����H�=_B-�H�� �N���������H�
□�
�H�5��
�H�=��
�H��□������□��H�5w�□��□���H��□1��ob□��□����5�□�□□D��H�□�k-�H��t□H�=IE��□���H���□�□����u□��H�����□�f.□□�������□
H���H�H��H��E��u□��t□L��Hc�H�H��I��L□�H�□�f-�H��□�□����□���������Pv□������I���Cv□������H���6v□�I�������□�M���
□��������□���□��E1�A��□A□��D��!�E��□□1�A��□□��!�D�□�A�□���A ��□���□��□�����D��E��tCD����D�V□□�������������t*�
���A��□□�l���1�E��□��������������1�E��□���������□�σ����□����!��f.□□�������1�I��^H��H���PTI��`□@�H���□@�H��}□@
]��[m���□□D��]�f□□D����[m�UH���[m�H��H��□H��H��?H□�H��t□�����H��t□]��[m���□□�]�f□□D���=�W-��u'UH���~���� �J�H
��[L���
��□�W-�□]Đ��□□@�f.□□���������J�H��t□U��cm���[L�H���s�
�]�]���□□D���S���UH��H�� dH�□%(���H�E�1�H�E�H��H�=□�
�����������H�E�H��H�=�
�����������H�U�H�E�H��H���)��������H�M�dH3%(���t□擾□���f.□□�������□□@�H□�H��I��H���|□}��>����<��������□□I�□���
���H�□H��H�%�cm������A�
����
���M��H��PM□�H���H�%�cm��□H��M��H�%�cm�XH���u�H�%�cm��□����□���H�κ□���□□H�%�cm�H��M��H�%�cm�H���cm�}��f□□D��S
/a-��□!a-���□�□����□��□�□����������A�□�□A�□�?□□�e□���=�`-�□~!�□���1�□��□�`-��□�`-��
�`-��□�`-��5�`-������□□� □��1�□□Љ□�□��□□�+□���=�`-�□���□���
���1ј�□���□�������□�□�������⁒□�`-�H�□�`-���□��□□�����H����������D$�@□��L�D$�H�D$�H������□��H�|$□H�D$□A�□����□�
����=□□D��D�⁒�□���□�□t□�G�A□@���□t□��?���A��A�� t,��□A��□I��□H��□A□��r□�□�□����t��G�A□@�A���□�S����□D$□t□�
�_-���@��� ���H� �_-�H�Ĉ���[���□□����□� A�� ���������□t.�□�_-��y�@�□s_-� �
�_-�t��@����□�_-����□��uD%����=����□������□A_-�����□�□�}����□__-�����□��0�����□�□E�
9_-�����믜f.□□������AVAU1�ATUM��SH��M��H������H�T$□H�������H�|$□�t$H��t �□1���□���□�
```

Description of add_prog.c:

It has 2 include statements, one required to take input and the other required to define int64_t data type. It includes 2 input statement with function “scanf” which stores values in variables namely, a and b, which are in turn stored in rdi and rsi registers. And ultimately it include a function call to the assembly language code.

Description of add_asm.asm:

It is the assembly language code which is written with the help of mnemonics. It has a label called add_num which is used to add numbers stored in rdi and rsi registers. It also has a label which handles positive results named “_pos” and a label which handles negative results named “_neg”. Labels namely “_print”, “_printLoop” and “_printLoop2” are used to print the integer form of the result.

I used a brute force method to print the numbers from the assembly code. As the integer stored in the register after the addition cannot be directly printed so using the three labels namely “_print”, “_printLoop” and “_printLoop2”, I kept on dividing the number by 10 and pushed its remainder into the stack. Then I called the print function for that very digit. In case of a negative number, label “_neg” makes a call to print “-” before the number.