AYUSH MAHANT
CSSS
2019353

ASSIGNMENT 0.2

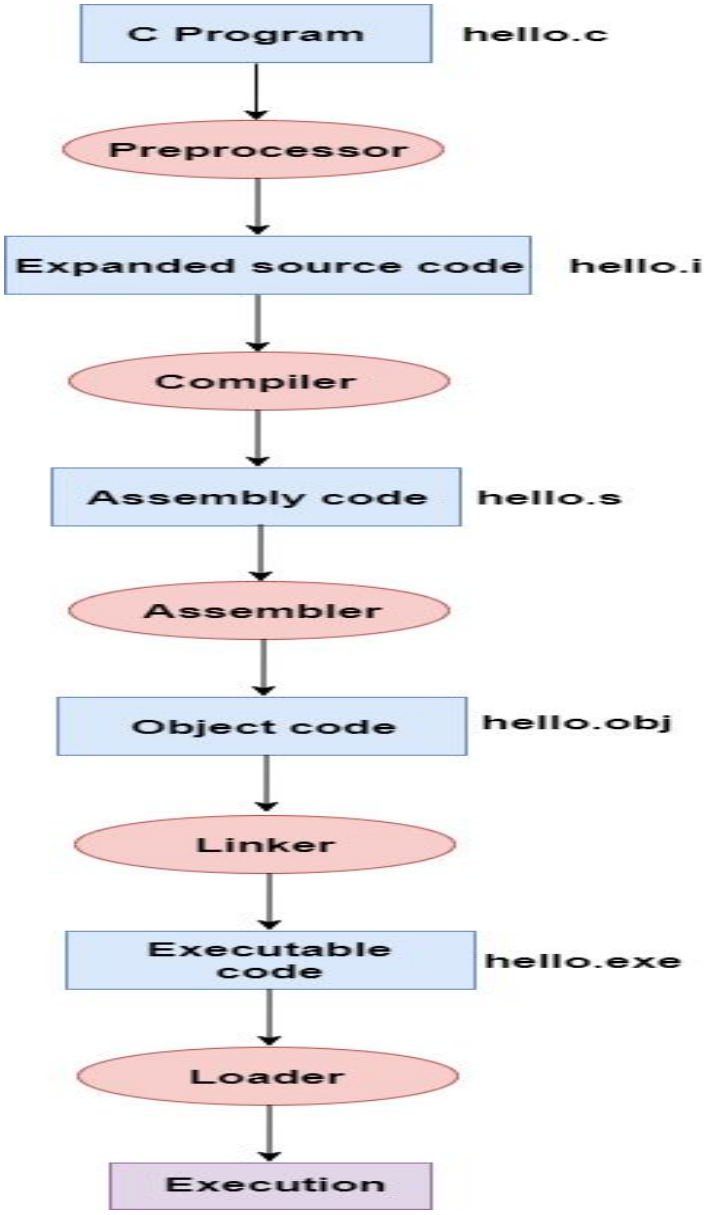
Understanding C program compilation process-

The most important task given is to understand the C program compilation process using Makefile. Makefile is a tool for organizing code for compilation. Suppose, we have multiple c language programs and we need to link them together. Whenever we do the compilation steps for creating the object files(.o file) and then the final executable files(.exe) on the terminal, it becomes rigorous as we need to compile it every time on the terminal and then link the object files.

Using makefile, we can create multiple targets for compilation which makes the task easier. We need to write only a few commands on the terminal like “make” and the final output is displayed on the executable file after linking. The major importance of the makefile is that when we make any changes in any of the c programs we don't need to recompile again at each step or recompile the other c programs too.

Instead we just need to run the “make clean” which removes the object files and write “make” again which automatically changes. In this assignment we are required to make only one c program file,an asm file and a makefile which shows steps of compilation which are shown in the diagram below.In the c program we input two integer numbers using scanf and pass them to the asm file using function add.

I have compiled each file one by one which explains the pause and compile.



Reference-compilation-process-in-c3.png

Source code(c lang)- In the source code we are asked to add two local integer variables taken from the user and pass them using the function “add”,which returns the addition of the two numbers and prints them using “printf”. We are making a prog-add.c file.
ASM code- the first register “ rdi” takes the first value passed and the second register ‘rsi” takes the second value passed. The addition is stored in register “rax” which is returned to the c program.

MAKEFILE AND COMPILATION STEPS-

Makefiles have a specific format(remember to put a tab space before the ACTION) -

TARGET:DEPENDENCIES
	ACTION

Making the object file of asm file-

When we write “make STEP1” on the terminal, the object file of the add.asm file is created which is of hexadecimal form. The object file is created using the command nasm -felf64 add.asm.

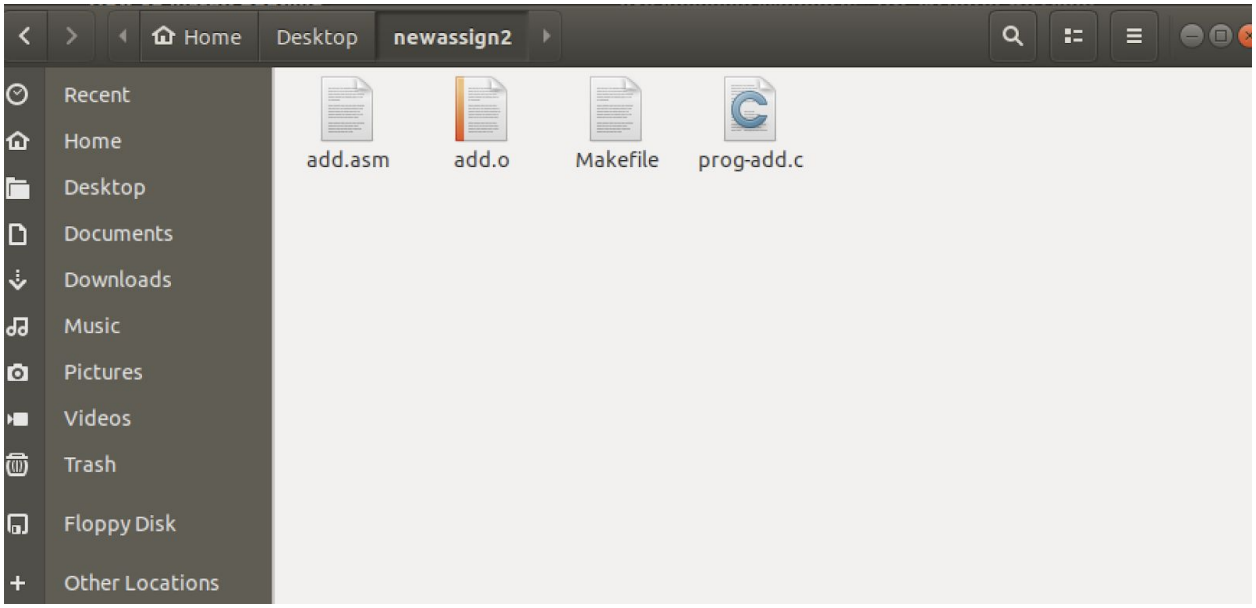
The add.asm file has the directive “global add” and the section as “add: “.

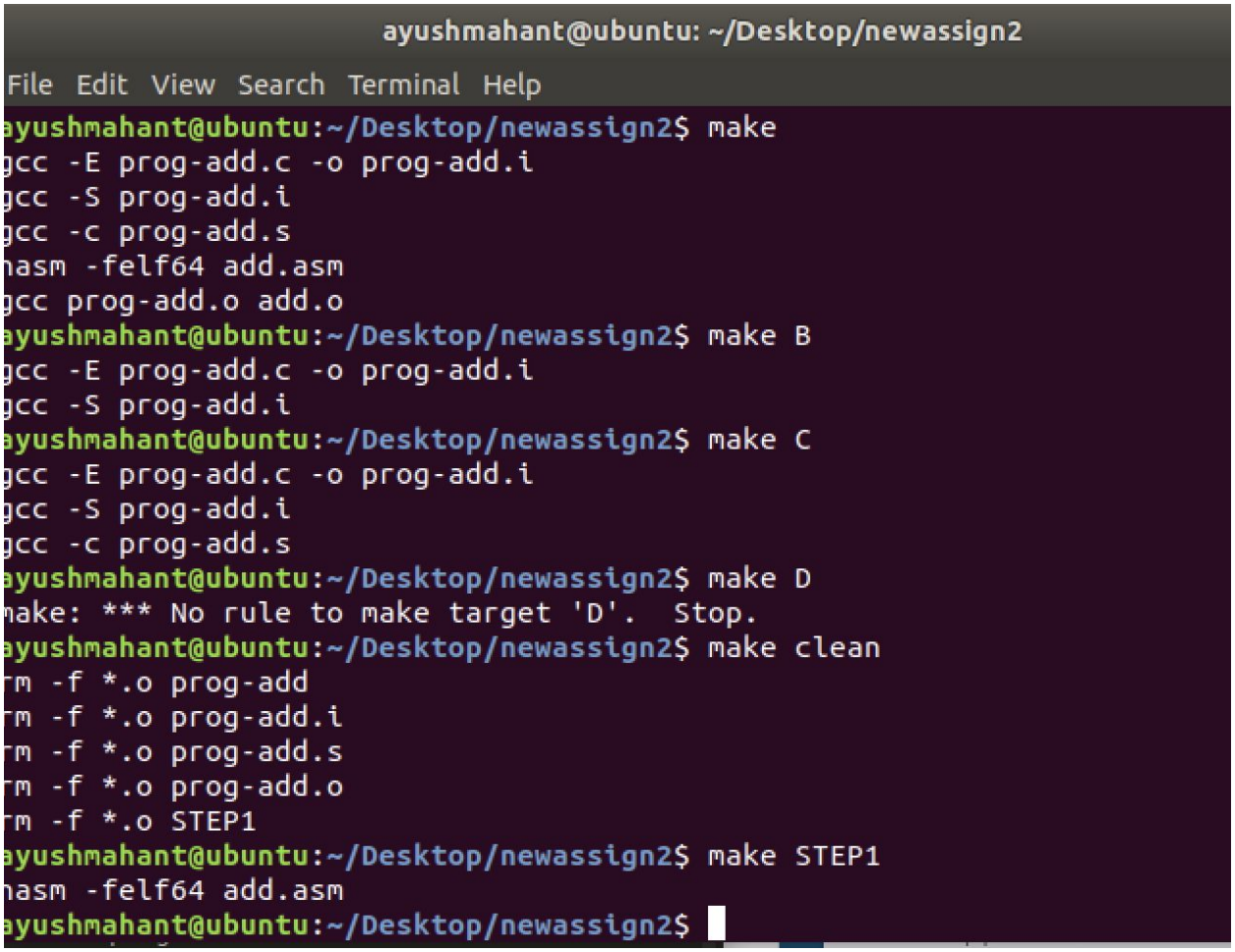
NASM- NASM stands for netwide assembler for x86 architecture.It can be used to write 16bit, 32 bit and 64 bit (x86-64) programs.

X86-64 is the 64 bit version of the x86 instruction set.It introduces two new modes of operation, **64**-bit mode and compatibility mode, along with a new 4-level paging mode.There were 8 registers in x86 (eax, ..) and in x86-64 there are 16 registers(rbi,rsi..)

-felf64 -the format of the object file is elf and -f specifies the format.

OUTPUT ON THE TERMINAL AND OUTPUT FILE-





PREPROCESSING-

The preprocessing step:
(taken from my makefile code)

A: prog-add.c
	gcc -E prog-add.c -o prog-add.i

Step A is the first step which depends on the prog-add.c file and converts prog-add.c file to prog-add.i(which is the pre-processed file). On the terminal we need to write gcc -E prog-add.c -o prog-add.i for the creation of the prog-add.i file but now on the terminal we simply write “ make A” which creates the output file, i.e, the prog-add.i file.
The preprocessor handles the #include, #define and strip out the comments.
Explaining the command “ gcc -E prog-add.c -o prog-add.i ”

gcc- gcc is the compiler which takes the human readable c file and converts it to machine code in macOs(which currently I am working on). Gcc stands for GNU compiler collection.

-E - option which saves the result of the preprocessing stage or the preprocessed source code .
prog-add.c - this is the file passed to the preprocessor which is further saved in prog-add.i
-o prog-add.i - the result of preprocessing is stored in the prog-add.i file.

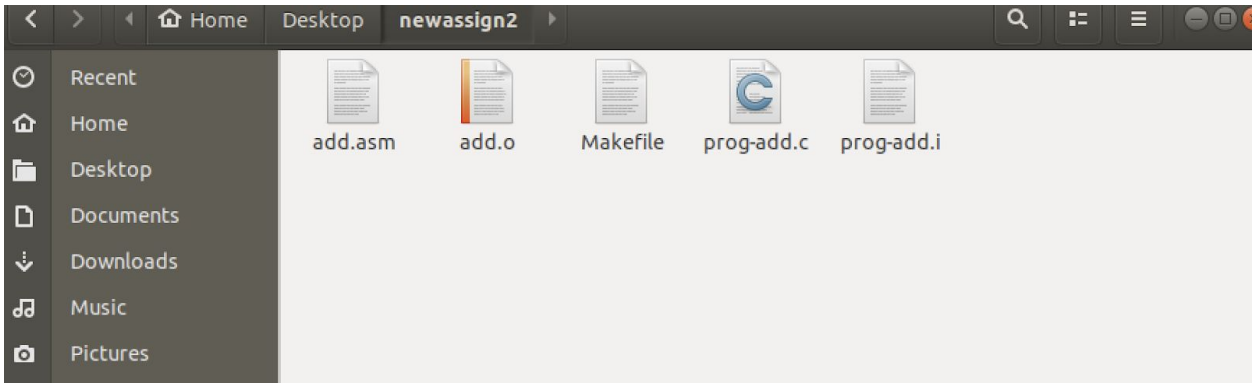
For a general understanding, the #include or the #define statements in the c program are replaced by the preprocessed statements or code.

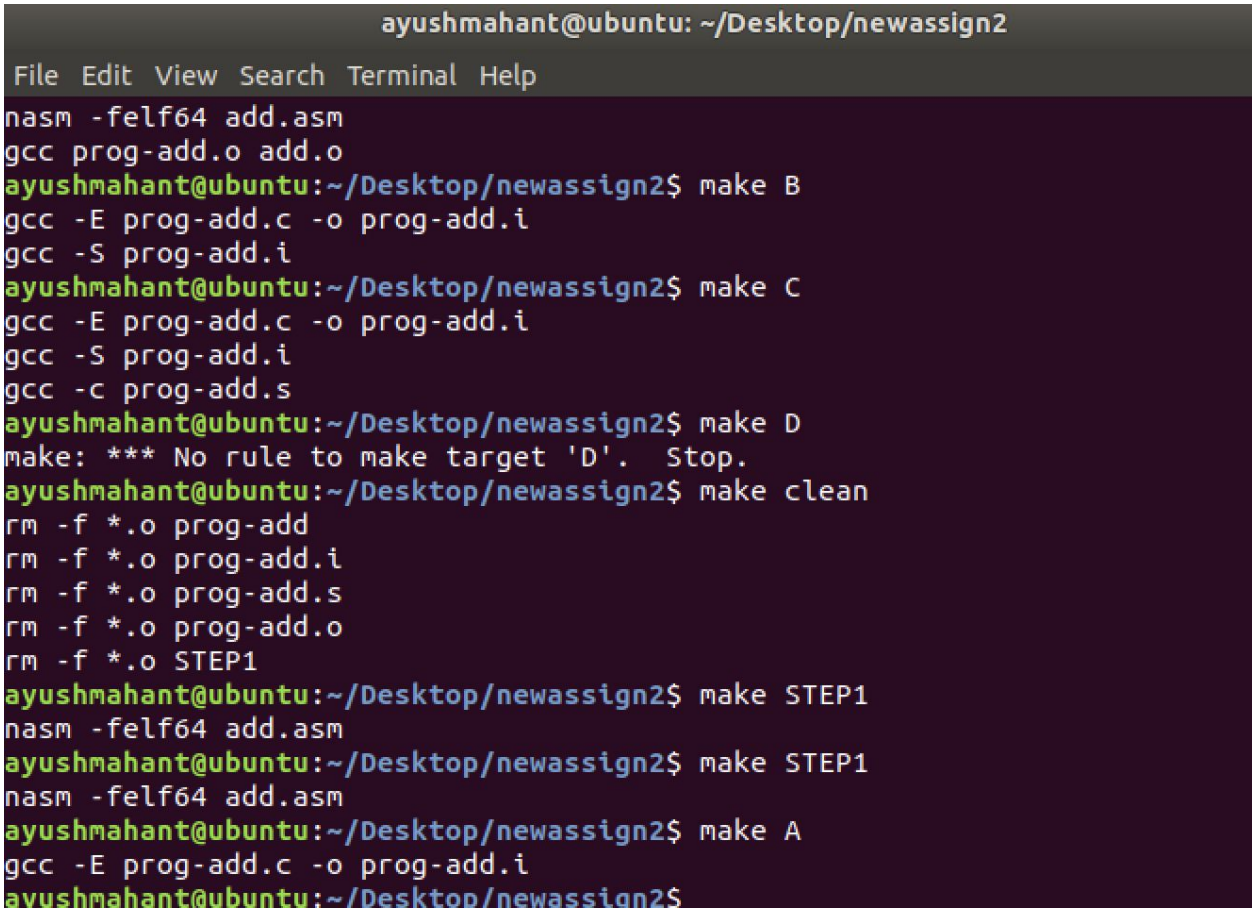
Just by writing make A we are stopping at the preprocessing step which runs the " gcc -E prog-add.c -o prog-add.i " command and does the task.

Make is basically the command utility which is used to process the makefile and when we write a target in front of the make it only runs at that target step and performs the action.

Output and the output file-

When we write "make A" for the processing output, on the terminal, "gcc -E prog-add.c -o prog-add.i" which creates the output file(prog-add.i) in the current directory as shown below .





Before describing the output file , the task of the preprocessing step is to replace the #include or #define statements in the .c file with the processing statements.

COMPILING-

The Assembly code file step-

(taken from makefile)

B:A

gcc -S prog-add.i

B is the target which depends on A, i.e, the prog-add.i file.The action is creation of an assembly file(.s file)

On the terminal we need to write gcc -S prog-add.c for the creation of the prog-add.s file but now on the terminal we simply write “ make B” which creates the output file, i.e, the prog-add.s file.

Even if we write “make B” it creates both the preprocessing and the assembly file because the assembly code file depends on the processing file.
Explaining the command “gcc -S prog-add.i”
gcc- gcc is the compiler which takes the human readable c file and converts it to machine code in macOs(which currently I am working on). Gcc stands for GNU compiler collection.

-S - option which saves the result of the preprocessing stage or the preprocessed source code in the .s file .
prog-add.i - this is the file compiled which produces the intermediate compiled output file -prog-add.s.

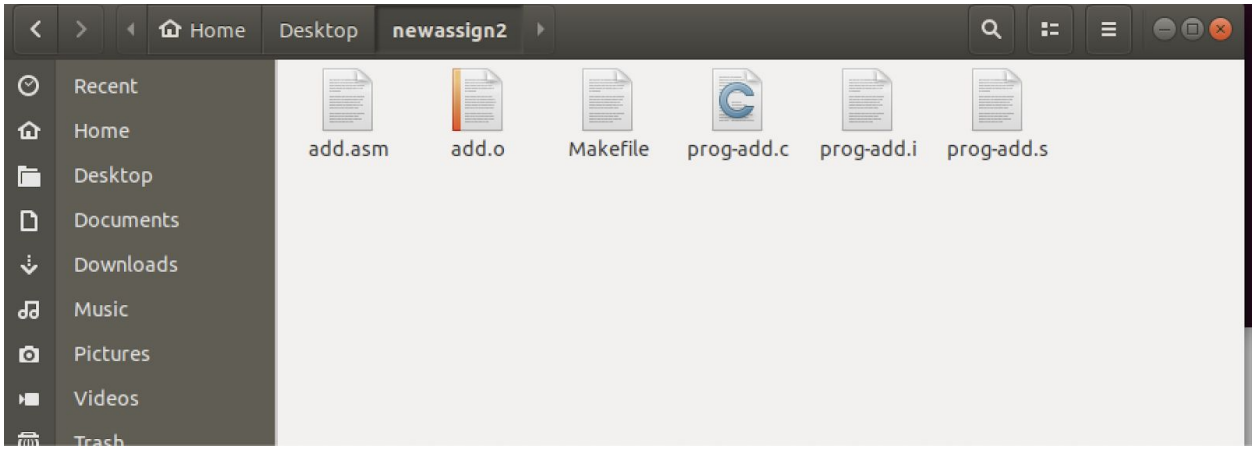
Output file and output shown on the command

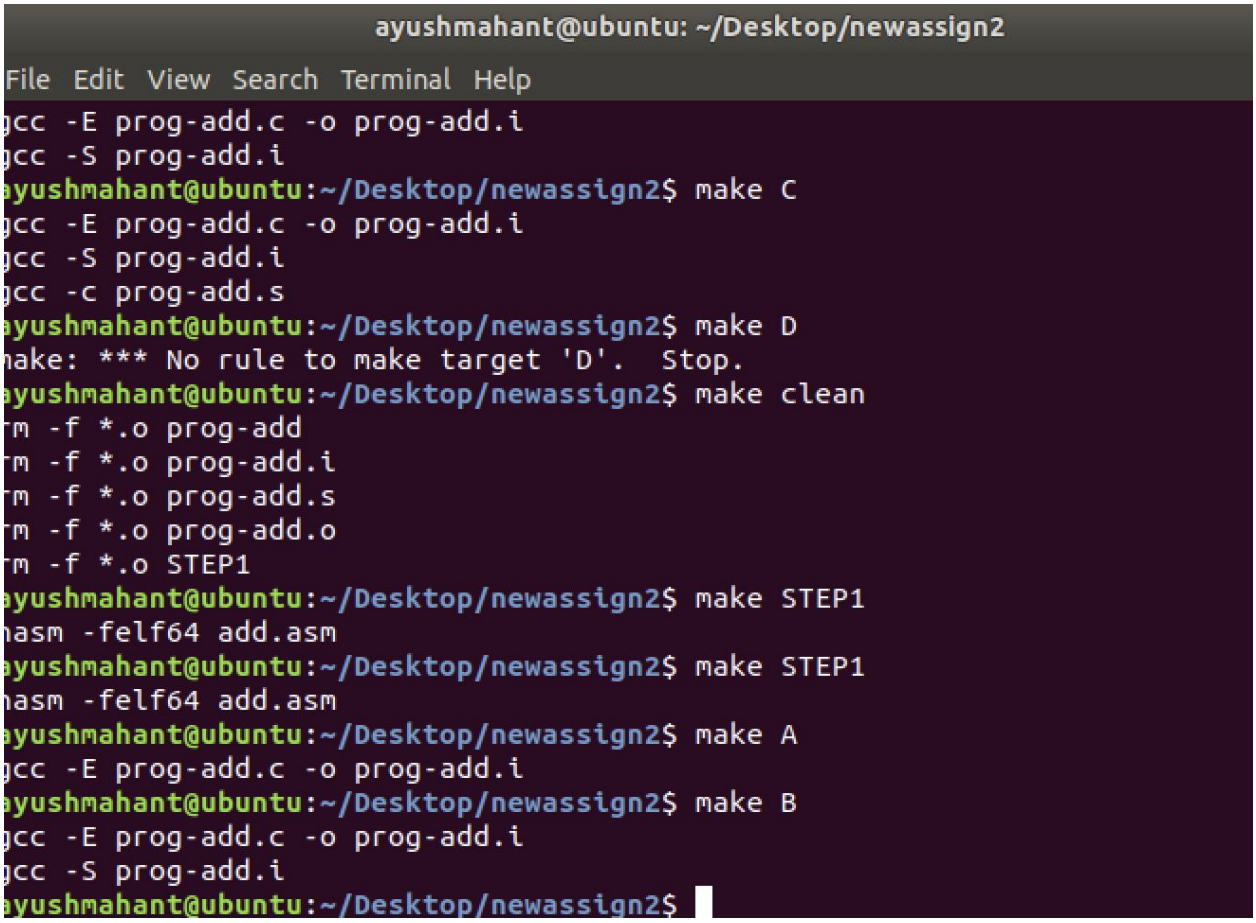
Output on the terminal-

```
gcc -E prog-add.c -o prog-add.i
gcc -S prog-add.i
```

It is basically showing the output or the gcc commands which the “make” is generating ,
Like here I wrote “make B” so it generates the gcc commands for the assembly code file and the preprocessing file as target B depends on target A.

Output file-





Each intended line in the assembly code output file is corresponding to a single machine instruction. The pushq instruction states that the contents of the register %rbp is to be stored in the program stack.

ASSEMBLY-

Object file step:

C:B

gcc -c prog-add.s

C is the target which depends on B, i.e, the prog-add.s file.The action is creation of an object file(.o file)

On the terminal we need to write gcc -c prog-add.s for the creation of the prog-add.s file but now on the terminal we simply write " make C" which creates the output file, i.e, the prog-add.o file.

Even if we write “make C” it creates both the preprocessing and the assembly file and the object file because the object code file depends on the preprocessing file and the assembly file.
THe assembler converts the assembly code to the object code or the machine level code.
Explaining the command “gcc -c prog-add.s”
gcc- gcc is the compiler which takes the human readable c file and converts it to machine code in macOs(which currently I am working on). Gcc stands for GNU compiler collection.

-c - option which creates the object file from the (.s file)
prog-add.s - this is the file compiled which produces the object code output file prog-add.o.

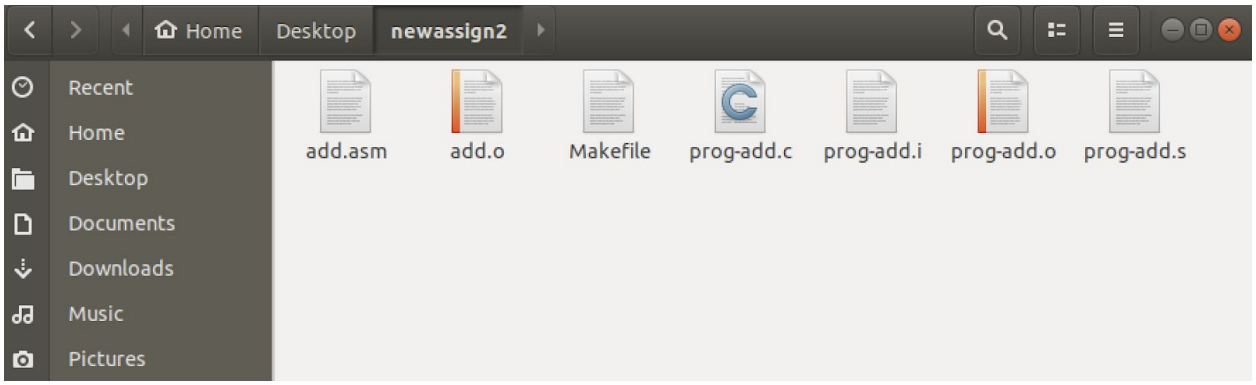
Output file and output shown on the command

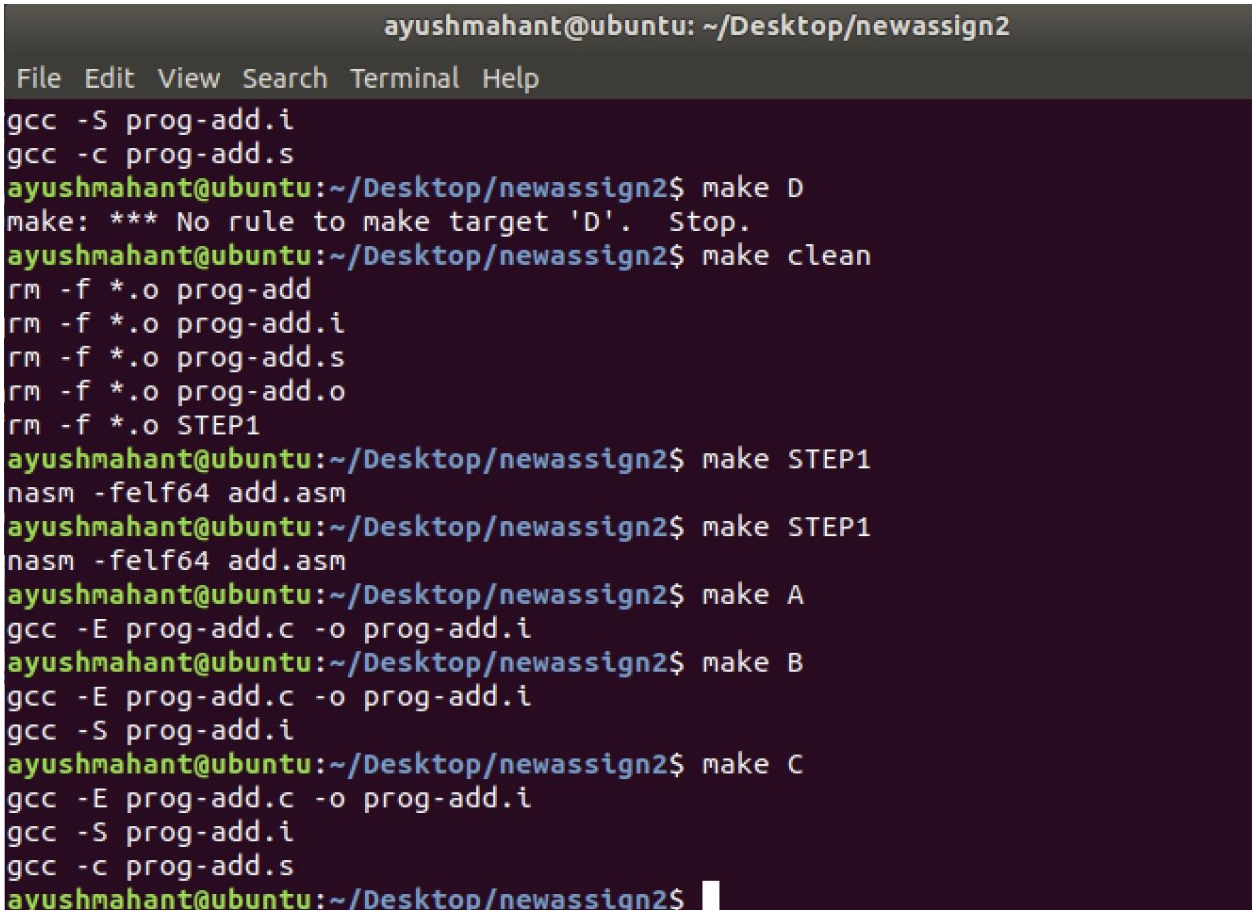
Output on the terminal-

gcc -E prog-add.c -o prog-add.i
gcc -S prog-add.i
gcc -c prog-add.s

It is basically showing the output or the gcc commands which the “make” is generating ,
Like here I wrote “make C” so it generates the gcc commands for the object code and the assembly code file and the preprocessing file as target C depends on target B and B on A.

OUTPUT FILE-





The object file created consists of functions defined in hexadecimal form but they are not executable themselves.
Each intended line in the object code output file is corresponding to a bunch of hexadecimal conversions.

LINKER-

The executable file:

prog-add: C step1
	gcc prog-add.o add.o

Step D or the target “prog-add” is the step which depends on the prog-add.o file and the add.o and converts them to prog-add(which is the executable file). On the terminal we need to write

gcc prog-add.o add.o. for the creation of the prog-add file but now on the terminal we simply write “ make prog-add” which creates the output file, i.e, the prog-add file.
Explaining the command “ gcc prog-add.o add.o ”

gcc- gcc is the compiler which takes the human readable c file and converts it to machine code in macOs(which currently I am working on). Gcc stands for GNU compiler collection.

prog-add.o - c program object file.
add.o - asm object file

./a.out- the execution begins after writing this.

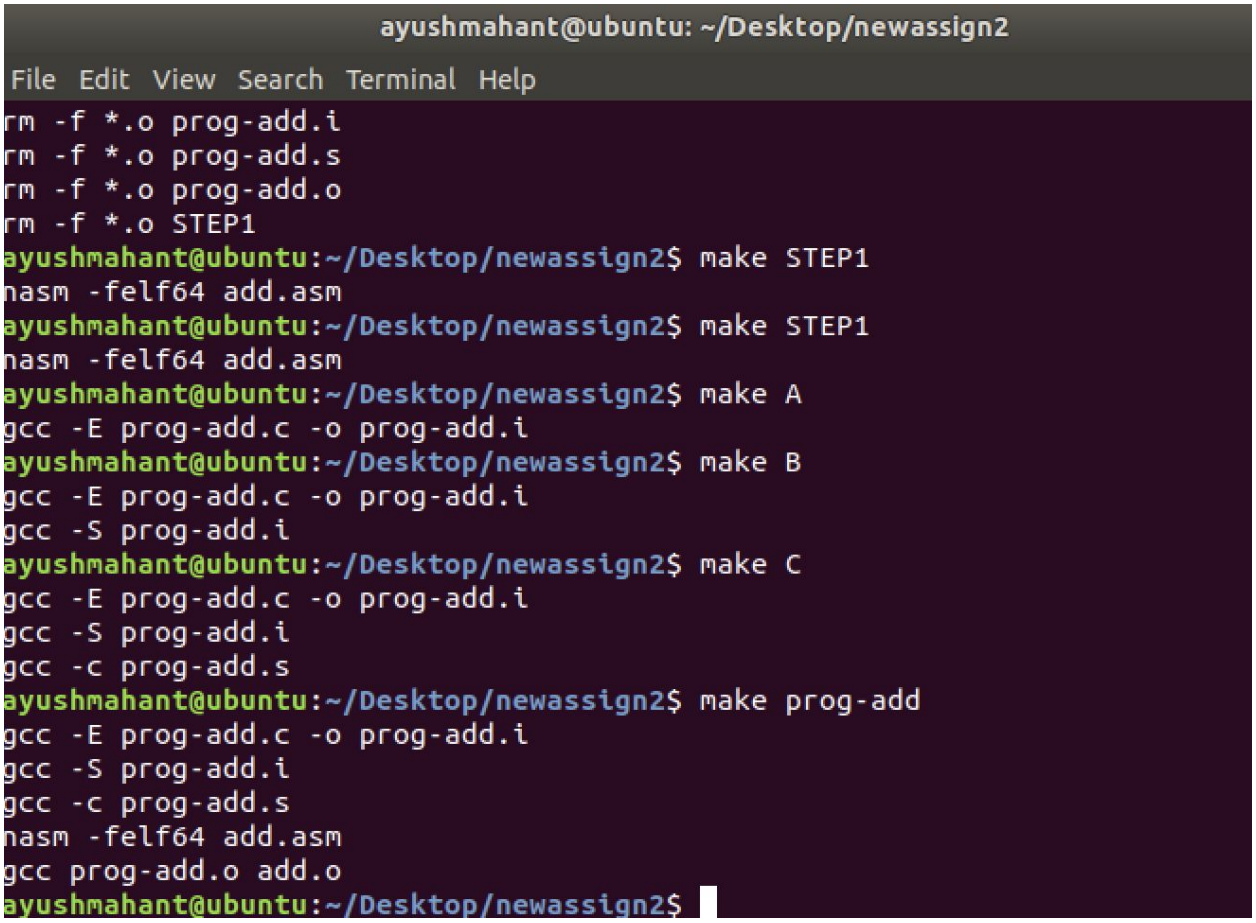
OUTPUT AND THE OUTPUT FILE-

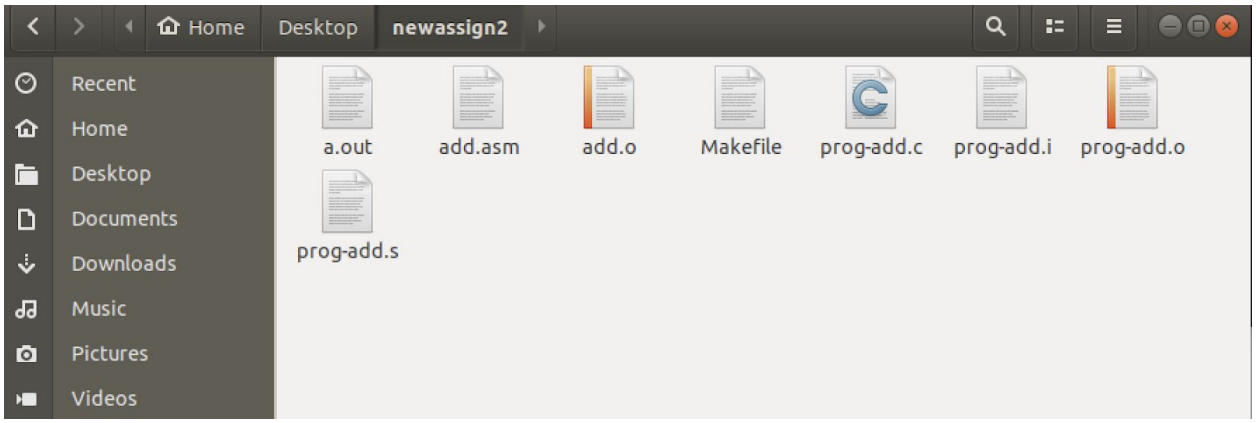
Output on the terminal-

```
nasm -felf64 add.asm
gcc -E prog-add.c -o prog-add.i
gcc -S prog-add.i
gcc -c prog-add.s
gcc prog-add.o add.o
```

It is basically showing the output or the gcc commands which the “make” is generating ,
Like here I wrote “make prog-add” so it generates the gcc commands for the object code and the assembly code file,the preprocessing file and the executable file as target prog-add depends on target C and C on B and B on A.

Output file-





When we open the executable file it opens the terminal displaying the two numbers to be displayed.

make clean:

rm -f *.o prog-add

```
rm -f *.o prog-add.i
rm -f *.o prog-add.o
rm -f *.o prog-add.s
rm -f *.o STEP1
```

It cleans all the executions or the gcc commands and the files made and makes the program ready for execution again.

HOW THE PROGRAM WORKS(in short because in above stances I have explained)-

The c program inputs two integers using scanf and the c program calls the asm routine “add” Which stores the first number in register “rdi” and the second number in second register “rsi” and then adds them using register rax which is returned back to the c program. The returned value is stored in a format which is printed.
Global- It tells the kernel where the execution begins.

Section.text- It is used to keep the actual code.
Move written in the section moves the value in rdi to rax.
Add written in the section adds the rax register with rsi register and stores the value in the rax register which is returned to the main program.

Each instruction consists of an operation code. Each executable instruction generates one machine language instruction.
Also the use of pointers is initiated here for better understanding.

–