Assembly x64 bits

1. Edit
2. Assemble
3. Link
4. Run (or Debug)

There are many good text editors on the market, both free and commercial. Look for one that supports syntax highlighting for NASM 64-bit. In most cases, you will have to download some kind of plugin or package to have syntax highlighting.

We think you will agree that syntax highlighting makes the assembler code a little bit easier to read. When we write assembly programs, we have two windows open on our screen—a
window with gedit containing our assembler source code and a window with a command prompt in the project directory—so that we can easily switch between editing and manipulating the project files (assembling and running the program, debugging, and so on). We agree that for more complex and larger projects, this is not feasible; you will need an integrated development environment (IDE).

1. sudo apt install gcc build-essential nasm
2. makefile for assembly code

```
#makefile for hello.asm
hello: hello.o
	gcc -o hello hello.o -no-pie
hello.o: hello.asm
	nasm -f elf64 -g -F dwarf hello.asm -l hello.lst
```

Save this file as makefile in the same directory as hello.asm and quit the editor.

A makefile will be used by make to automate the building of our program. Building a program means checking your source code for error, adding all necessary services from the operation system, and converting your code into a sequence of machine-readable instructions.

You read the makefile from the bottom up to see what it is doing. Here is a simplified explanation: the make utility works with a dependency tree. It notes that Hello depends on hello.o
It then sees that hello.o depends on hello.asm and that hello.asm depends on nothing else. Make compares the last modification dates of hello.asm with hello.o, which is hello.asm. Then make restarts reading the makefile and find that the modification date of hello.o is more recent than the date from hello. So, it executes the line after hello, which is hello.o.

In the bottom line of our makefile, NASM is used as the assembler. The -f is followed by the output format, in our case elf64, which means Executable and Linkable Format for 64-bit. The -g means that we want to include debug information in a debug format specified after the -F option. We use the dwarf debug format.
DWARF => Debug With Arbitrary Record Format
The -l tells NASM to generate a .lst file. We will use .lst files to examine the result of the assembly. NASM will create an object file with an .o extension. That object file will next be used by a linker.

When you finally convinced NASM to give you an object file, this object file is then linked with a linker. A linker takes your object code and searches the system for other files that are needed, typically system services or other object files. These files are combined with your generated object code by the linker, and an executable file is produced. Of course, the linker will take every possible occasion to complain to you about missing things and so on. If that is the case, have another coffee and check your source code and makefile.
In our case, we use the linking functionality of GCC (repeated here for reference):

```
hello: hello.o
        gcc -o hello hello.o -no-pie
```

The recent GCC linker and compiler position-independent executables (PIES) by default. This is to prevent hackers from investigating how memory is used by a program and eventually interfering with program execution. At this point, we will not build position-independent executables; it would really complicate the analysis of our program (on purpose, for security reasons). So, we add the parameter -no-pie in the makefile.

Finally, you can insert comments in your makefile by preceding them with the pound symbol, #.

```
#makefike for hello.asm
```

Structure of an Assembly Program

This first program illustrates the basic structure of an assembly program. The following are the main parts of an assembly program:

1. section .data
2. section .bss
3. section .text

Section .data

In section .data, initialized data is declared and defined, in the following format:

<variable name> <type> <value>

When a variable is included in section .data, memory is allocated for that variable when the source code is assembled and linked to an executable. Variable names are symbolic names, and references to memory locations and a variable can take one or more memory locations. The variable name refers to the start address of the variable in memory.

Variable names start with a letter, followed by letters or numbers or special characters.

1. db 8 bits byte
2. dw 16 bits word
3. dd 32 bits double word
4. dq 64 bits quadword

In the example program, section .data contains one variable, msg, which is a symbolic name pointing to the memory address of ‘h’, which is the first byte of the string “hello, world”, 10. So, msg points to the letter ‘h’, msg+1 points to the letter ‘e’, and so on. This variable is called a string, which is a contiguous list in memory that can be considered a string; the characters can be human readable or not, and the string can be meaningful to humans or not.

It is convenient to have a zero indicating the end of a human-readable string. You can omit the terminating

zero at your own peril. The terminating 0 we are referring to is not an ASCII 0; it is a numeric 0, and the memory place at the 0 contains eight 0 bits.
If you frowned at the acronym ASCII, do some Googling.
Having a grasp of what ASCII means is important in programming. Here is the short explanation: characters for use by humans have a special code in computers. Capital A has code 65, B has code 66, and so on. A line feed or new line has code 10, and NULL has code 0. Thus, we terminate a string with NULL.
When you type man ascii at the CLI, Linux will show an ASCII table. section .data can also contain constants, which are values that cannot be changed in the program. They are declared in the following format:

```
<constant name>  equ <value>
```

Here's an example:

```
    PI equ 3.1416
```

section .txt

section .txt is where all the actions is.
This section contains the program code and starts with the following:

```
    global main
main:
```

The main: part is called a label. When you have a label on a line without anything following it, the word is best followed by a colon; otherwise, the assembler will send you a warning. And you should not ignore warnings! When a label is followed by other instructions, there is no need for a colon, but it is best to make it a habit to end all labels with a colon. Doing so will increase the readability of your code.
In our hello.asm code, after the main: label, registers such as rdi, rsi and rax are prepared for outputting a message on the screen. Here, we will display a string on the screen using a system call. That is, we will ask the operating system to do the work for us.

1. The system call code 1 is put into the register rax, which means “write”.

2. To put some value into a register, we use the instruction mov. In reality, this instruction does not move anything; it makes a copy from the source to the destination. The format is as follows:
   > mov destination, source

3. The instruction mov can be used as follows:
   > mov register, immediate value
   > mov register, memory
   > mov memory, register
   > illegal: mov memory, memory

4. In our code, the output destination for writing is stored into the register rdi, and 1 means standard output (in this case, output to your screen).

5. The address of the string to be displayed is put into register rsi.

6. In register rdx, we place the message length. Count the characters of hello, world. Do not count the quotes of the string or the terminating 0, If you count the terminating 0, the program will try to display a NULL byte, which is a bit senseless.

7. Then the system call, syscall, is executed, and the string, msg, will be displayed on the standard output. A syscall is a call to functionality provided by the operating system.

8. To avoid error messages when the program finishes, a clean program exit is needed. We start with writing 60 into rax, which indicates “exit”. The “success” exit code 0 goes into rdi, and then a system call is executed. The program exits without complaining.

   System Call (Syscalls) are used to ask the operating system to do specific actions. Every operating system has a different list of system call parameters, and the system calls for Linux are different from Windows or MacOS.

You have a column with the line numbers and then a column with eight digits. This column represents memory locations. When the assembler built the object files, it didn’t know yet what memory locations would be used. So, it started at location 0 for the different sections. The section .bss part has no memory.