An assembly language program

There is a sample assembly language program file for you on the remote Unix machine. It is a program with a dozen assembler instructions. When you run it, it prints "Hello, world!" on the screen. It illustrates:

- how programs are created as sequences of machine instructions
- how those instruction are lodged in memory
- how they utilize the system calls offered by the operating system

Here is the source code of the program. Note that any line whose first character is a hash mark (#) is a comment and plays no operational role in the program. And here is an explanation of the program in technical terms.

Assignment:

Here is a list of six commands I want you to execute after logging in to the remote Unix machine. (Don't type the numerals, which are only for identification of the lines.) Please do it:

1. cd
2. cp /home/public/hello-world.s ~
3. as -o hello-world.o hello-world.s
4. ld -o hello-world -O0 hello-world.o
5. ./hello-world
6. objdump -d hello-world.o | tee assignments/disassembly

Here is a screenshot of what appeared on my screen when I executed this list of commands. You will see the same thing when you do so. Let me first explain the commands and what they do. And then, explain screenshot's results.

What the commands do

1. the first command assures that you are within your home directory.
2. the second makes a copy of the sample assembly language program source code file in your home directory. "hello-world.s" is the name of that file.
3. the third command "assembles" the source code found in the file you just copied. "as" is the name of the assembler tool. When you run it on a file containing assembly language source code, it produces a file containing something called "object code." Command number 3 tells it to place that object code into a new file named "hello-world.o." Converting an assembly language source code file into a runnable executable file is a 2-step process. The first step is "assembling" and the second is "linking."
4. the fourth command links the intermediate object code file, generating an executable program file as the final result. "ld" is the name of the linker tool. Command number 4 tells it to place the executable code into a new file named "hello-world."
5. command number 5 proceeds to run the freshly created executable program.
6. command number 6 reveals the actual machine code bytes that were created in this process, showing the sequence in which they line up for execution in memory. "objdump" is the name of the tool that can do this. It will show the machine code to you on the screen. As a side effect

of command number 6, its latter half produces a file named "disassembly" in your existing "assignments" subdirectory, for me to grade.

What the screenshot results show

The machine language of which this program consists (produced by the assembler when you ran it) is shown. It consists of 31 bytes (count them). The first instruction:

```
xor %ebx,%ebx
```

in assembly language got converted into 2 bytes of machine code for the CPU:

```
31 db
```

The second got converted into 5 bytes, b8 04 00 00 00. And so forth. In each case the machine code's first byte is the opcode for the instruction, from the Pentium's instruction set. You can confirm that if you look look up the documentation defining each assembly language instruction. Here is the documentation for several of the instructions in the sample program:

xor

inc

lea

So for example, the documentation for "inc" tells us it's a 1-byte instruction. Its first 5 bits are always 01000. Its final 3 bits vary depending on which register is the target to be incremented. If we're talking about the AX register, the last 3 bits are 000 so the full command becomes 01000000. In hexadecimal that's 40. If instead we're targeting the BX register, the last 3 bits are 011. Then, the full command becomes 01000011 or 43 in hex. Note that the screenshot shows an instance each. The "inc %ebx" in the 4th line is an instruction to increment the bx register, and lo and behold the disassembled value show to its left is 43. The "inc $eax" instruction in the 9th line shows 40. Consistent with the documentation! You can go through these instructions and map the assembly mnemonics like "xor," "inc," and "lea" to the corresponding machine instructions like "31," "43," and "8d" for example.