

# Code Constructs

Miguel Valencia Z.

September 2020

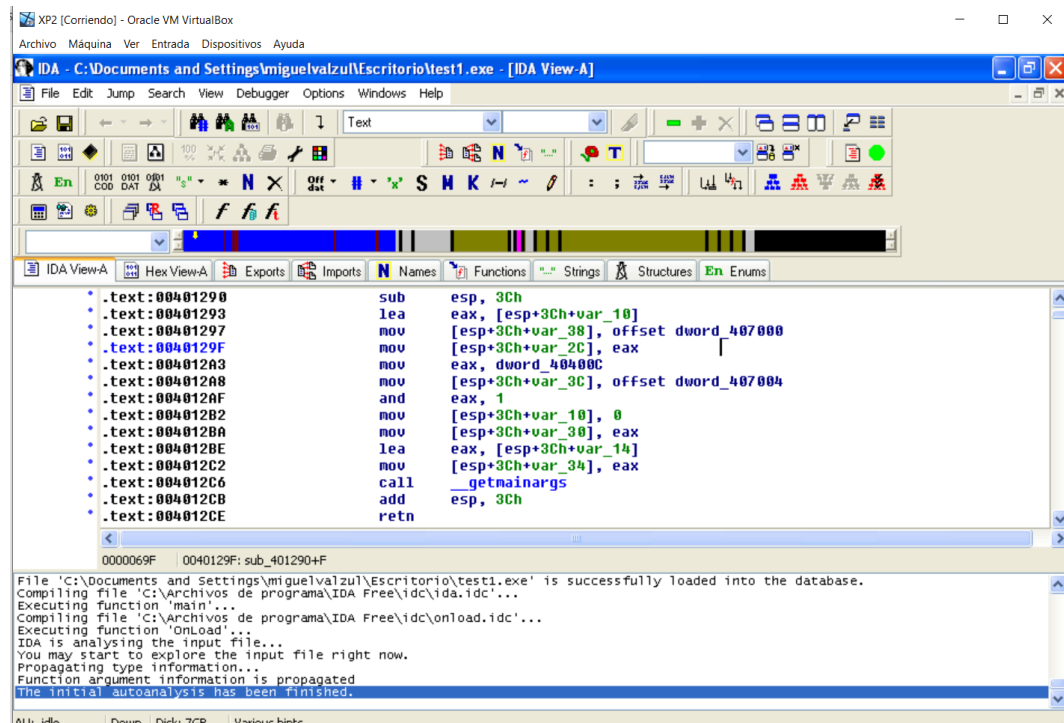
In this document we will answer some questions about how the assembly code is created.

After reading the section "Recognizing C Code Constructs in Assembly" we can say that a Code Construct is a "code abstraction level that defines a functional property but not the details of its implementation", i.e, it's a functional high-level definition of a piece of a program. Some examples of these are if statements, switch statements, loops, linked lists, etc. Different aspects may affect the way assembly code is generated. Some examples are:

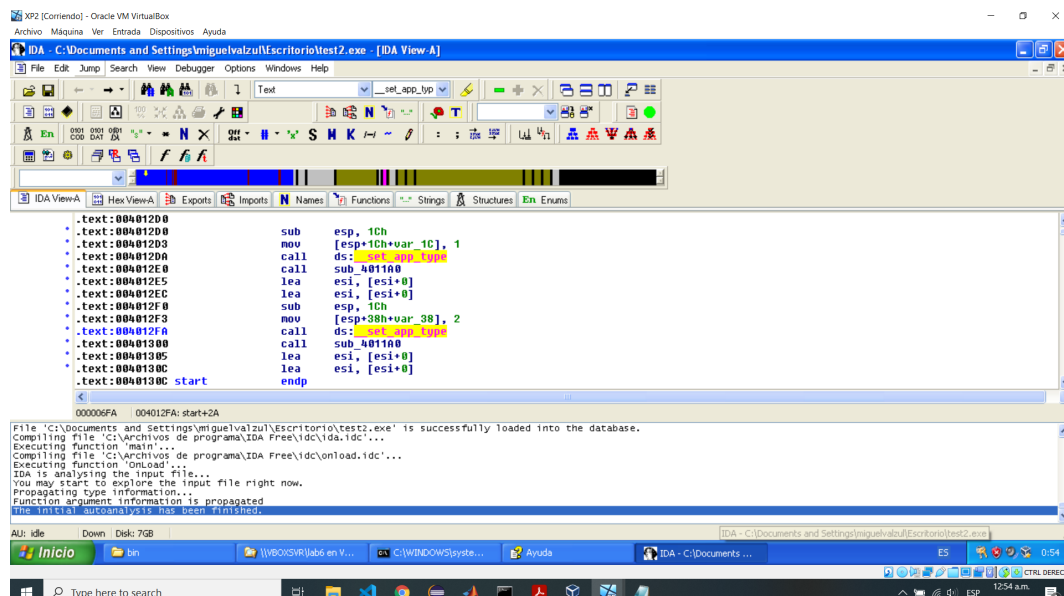
- The compiler. Depending on this, the generated assembly code's appearance will change.
- The programming language. For example, a C++ generated assembly code will differ from a Delphi one.
- The computer architecture. Some programs are made for a 32 bits architecture; others for a 64 bits one. The two will generate a different assembly code.

After reading the section "Global vs Local Variables" we can say that the differences in the compilation of a code that employs global vs one that employs local Variables are that the global variables are referenced by memory addresses, whilst the local variables are referenced by the stack addresses.

We compiled the C code exposed in the book and analysed it with IDApro. This is what we got for the global variables:



And for the local variables:

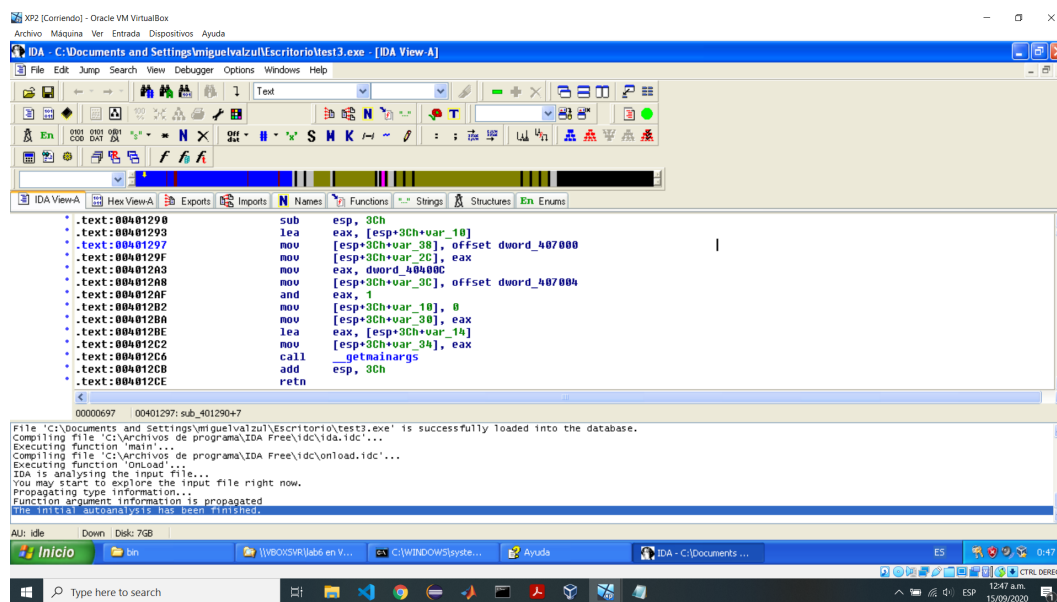


After reading the section "Disassembling Arithmetic Operations", we can

explain how the different operations (addition, subtraction, increment, decrement and modulo) are represented in assembly code.

- Addition. In assembly code, it is represented by the "add" command. In the book's example, the variable a is stored in eax, and then 0x0b is added to eax, thus performing the addition of a and 11.
- Subtraction. In assembly code, it is represented by the "sub" command.
- Increment. This is normally represented by the "inc" command. However, as the book shows, the compiler can choose to use the "add" command instead. It adds 1 to eax.
- Decrement. This is normally represented by the "dec" command. However, as the book shows, the compiler can choose to use the "sub" command instead. It subtracts 1 from edx.
- Modulo. Finally, this function is represented by the "idiv" command. We are dividing edx:eax by the operand and storing the result in eax and the remainder in edx.

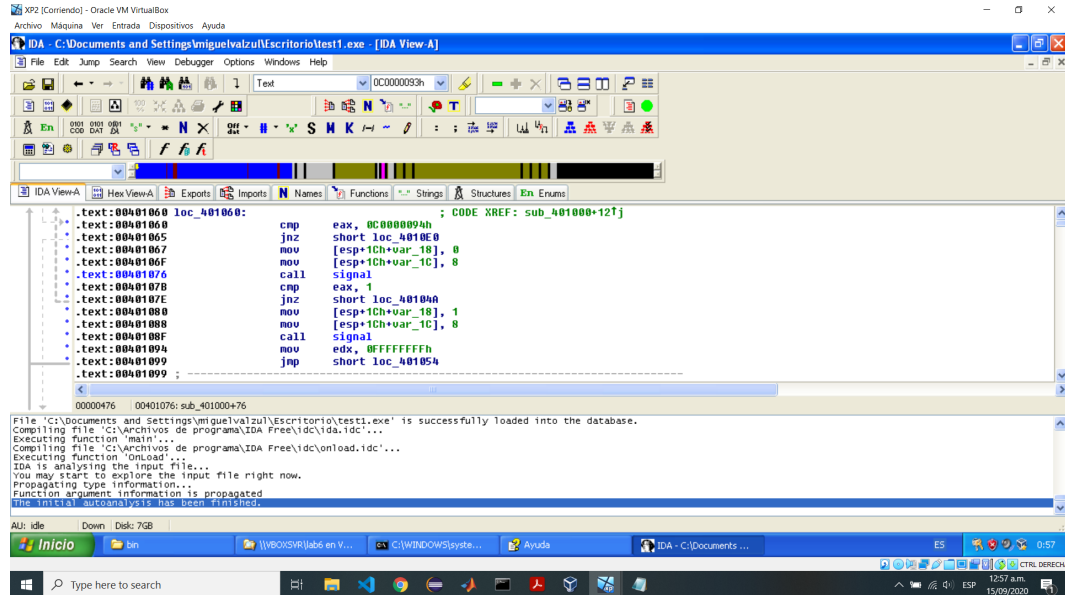
This is what we got after analysing the book's example with IDApro:



After reading the section "Recognizing if Statements", we now can recognize and if/else structure in assembly code. First, we must have a comparison. This is done by the "cmp" command. Then, we have a conditional jump "jnz", which takes a decision based on the previous comparison. If the comparison results in False, the jump occurs. Otherwise, the code continues its path of execution.

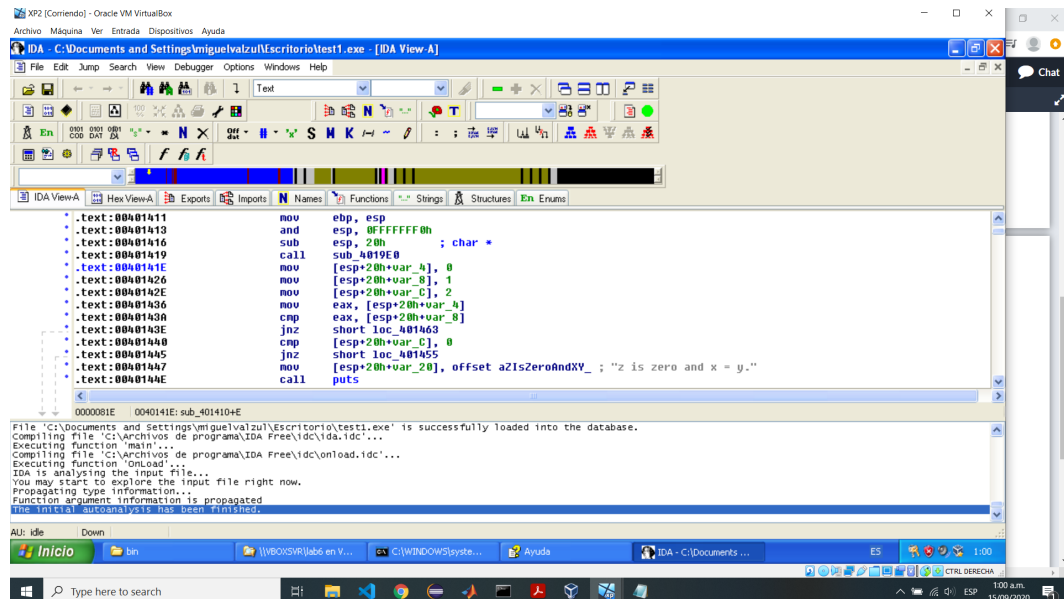
There is also a "jmp" commando for the else section. Only one of these two code paths can be taken.

This is what an if looks like after compiling it in our XP virtual machine:

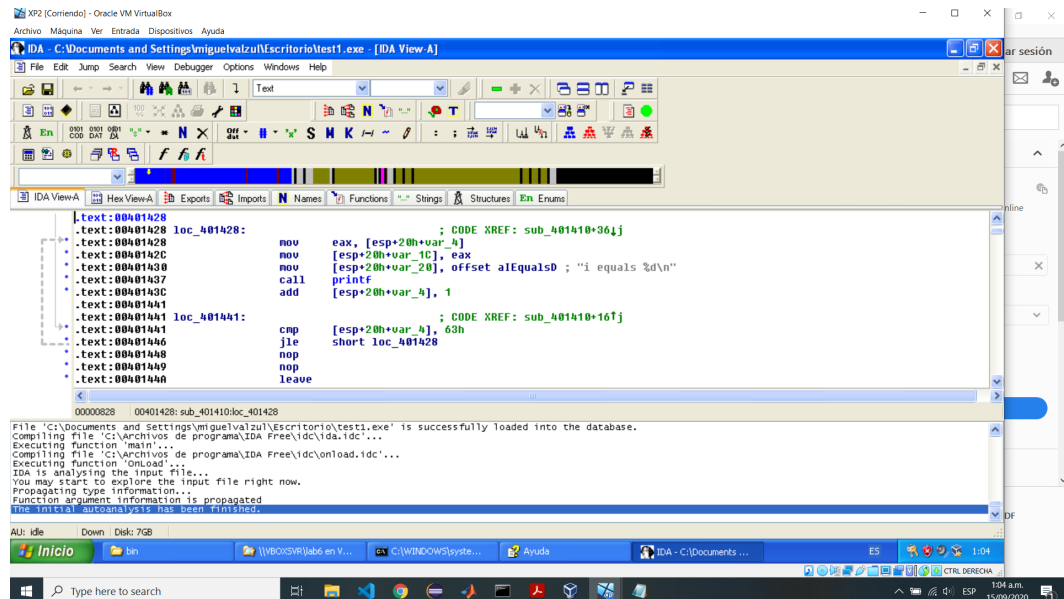


```
.text:00401060 loc_401060:  cmp     eax, 0C0000094h ; CODE XREF: sub_401000+127j
.text:00401060  jnz     short loc_4010E0
.text:00401065  mov     [esp+1Ch+var_18], 0
.text:00401067  mov     [esp+1Ch+var_1C], 8
.text:0040106F  call    signal
.text:00401076  cmp     eax, 1
.text:0040107E  jnz     short loc_4010A0
.text:00401080  mov     [esp+1Ch+var_18], 1
.text:00401088  mov     [esp+1Ch+var_1C], 8
.text:0040108F  call    signal
.text:00401094  mov     edx, 0FFFFFFFh
.text:00401099  jmp     short loc_401054
.text:00401099 ;
```

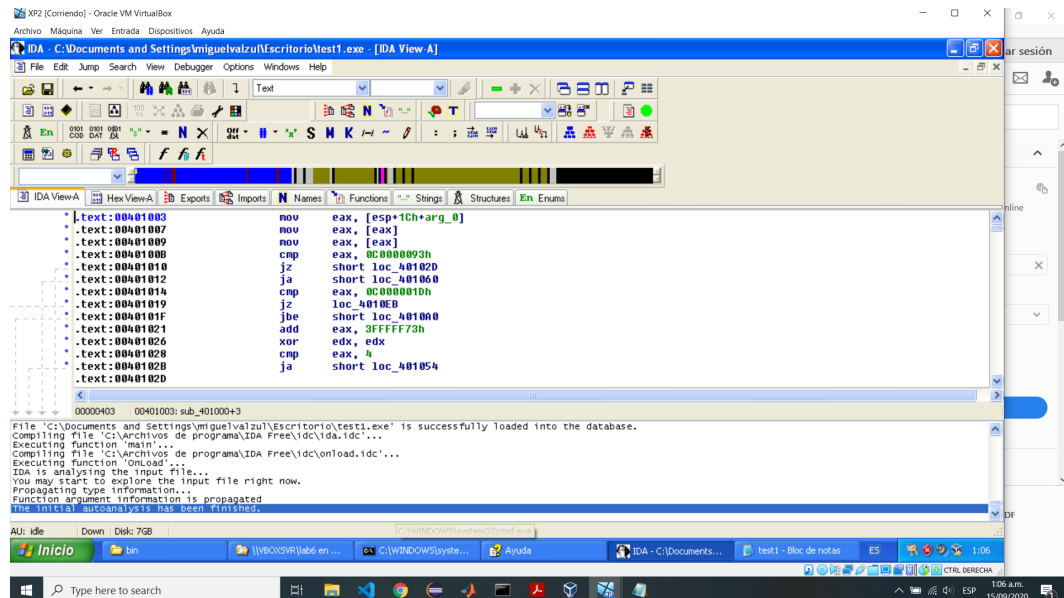
When we have nested if/else statements, more comparisons have to be made. Hence, we see more "jnz", "cmp" and "jmp" commands in the assembly code. And this is the assembly code for the nested if/else we obtained:



Now, let's talk about recognizing loops in assembly. For this, we have to locate the four components of this loop: initialization, comparison, execution instructions and increment/decrement. The initialization is done by a simple "mov" command, by moving the initial value to the stack. The increment is done by the "mov" and "add" commands. We first move `eax` to the variable, we then add 1 to `eax` and we finally move the variable to `eax`. The comparison is made by the "cmp" command and, in this case, as we're comparing the variable in each loop if it's greater or equal than 100, the assembly code shows a "jge" command, which stands for "jump if greater or equal to". If the jump is not taken, the code inside the for loop is executed. At the end, we have an unconditional jump with "jmp". This is what we got



Finally, let's talk about the while loop. The assembly code generated by this loop is very similar to the one generated by the for loop, except that it doesn't have an increment section. We have a conditional jump "jnz" and an unconditional jump "jmp". The while stops executing when the unconditional jump occurs. We got the following:



Note that we didn't get the exact same results as in the book for the different

assembly codes. This is because of what we talked earlier. We probably have a different compiler than the book's author, so the assembly code generated looks different. Also, our compiler is for a 32 bits machine and we don't know what computer architecture the author might have used. However, when analysing assembly code, we shouldn't get stuck with individual commands, as there's a lot of them, and no one has the time nor the will to take a look to every single one of them. The important thing is to understand the overall functionality of the assembly code.