Tutorial 2: Creating testbenches for combinational circuits

In this tutorial, we will see how to test combinational blocks using Xilinx and the ModelSim simulator.

In order to learn how to create testbenches and performing simulations we rely on 4 of the basic functions of SHA-256[1] *Ch, Maj*, Σ_0 and Σ_1. We will create an independent block for each function and then we will create a testbench in order to verify its correctness. This is related to the methodology is expected that you would follow when implementing a hash function or a block cipher in hardware: first, implement the basic functions of the algorithm as independent blocks and follow a bottom-up approach till the top block where every module can be tested.

After finishing this tutorial is expected that you could implement and simulate the basic functions that are part of the round of the block cipher have you have selected for your project.

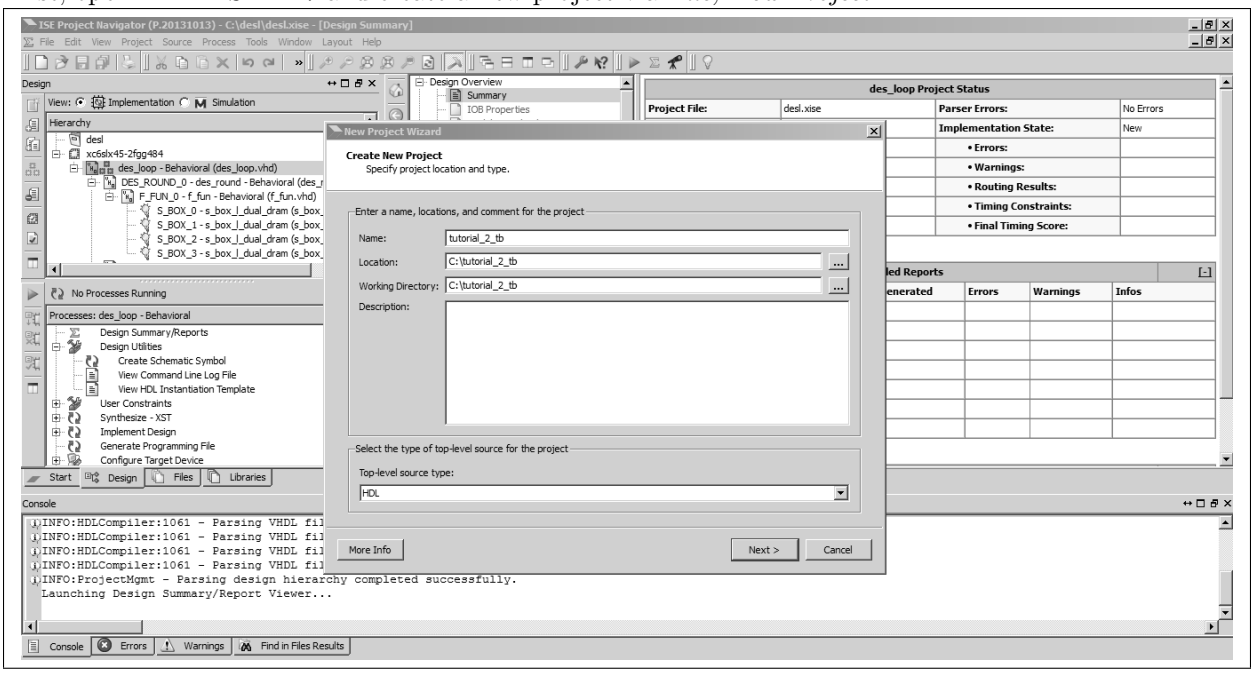
The *Ch, Maj*, Σ_0 and Σ_1 functions

These functions are part of the SHA-256 algorithm and are applied over operands of 32 bits. Consider inputs the variables x, y and z:

1. $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$
2. $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
3. $\Sigma_0(x) = (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22)$
4. $\Sigma_1(x) = (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25)$

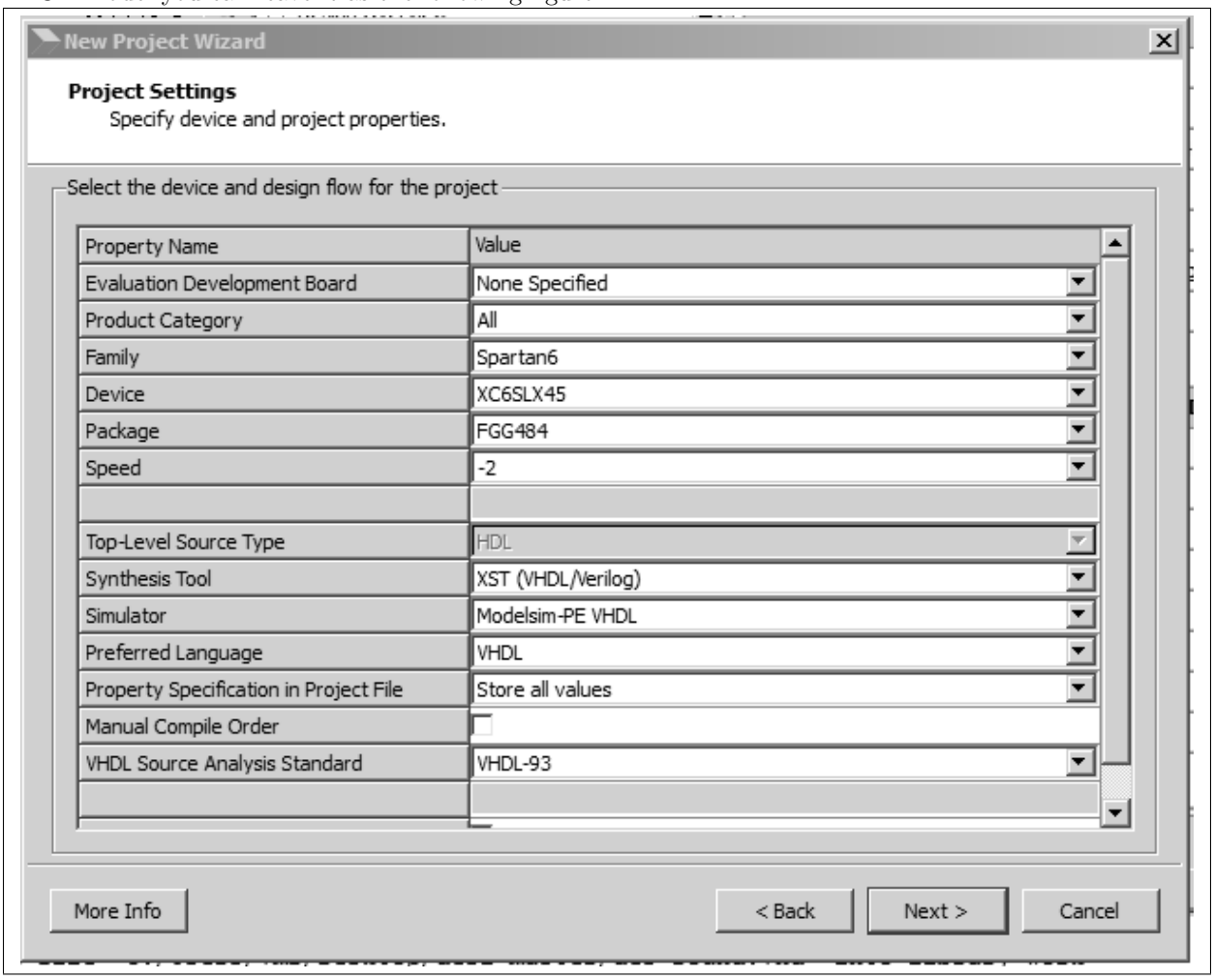
Creating a new project

First, open Xilinx ISE 14.7 and create a new project via *File, New Project....*

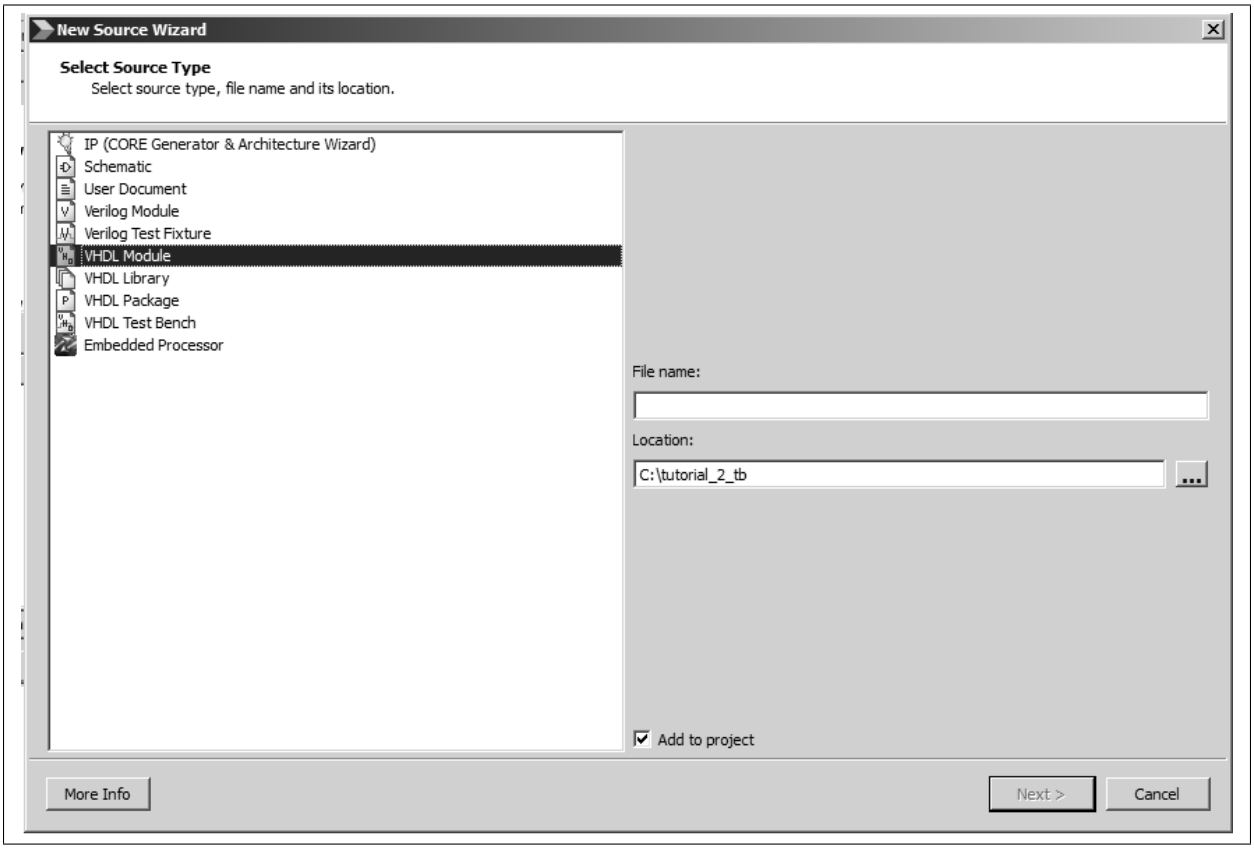


[1] FIPS PUB 180-4 Secure Hash Standard (SHS), available at `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`

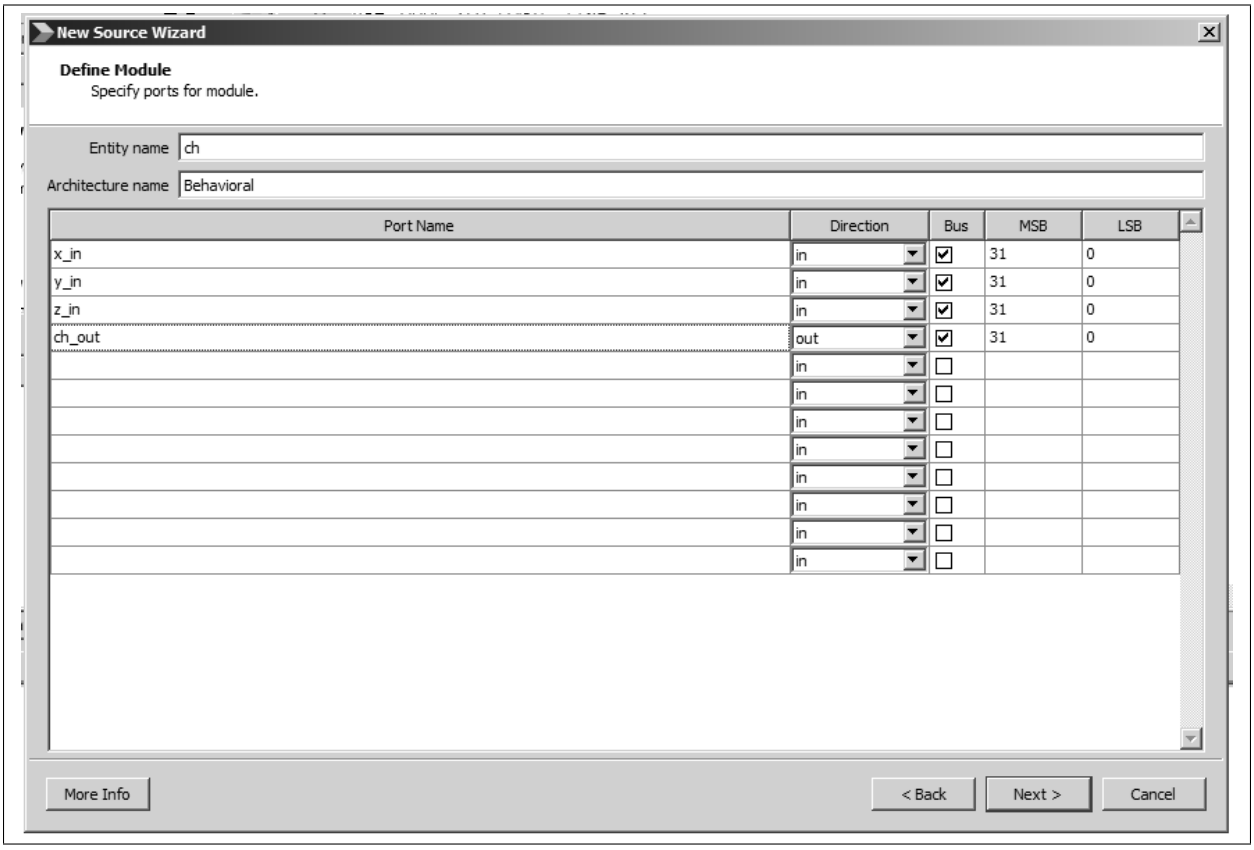
Select VHDL as a language and ModelSim PE as simulator (or other simulator if that works for you). As FPGA model you can leave it as the following figure:



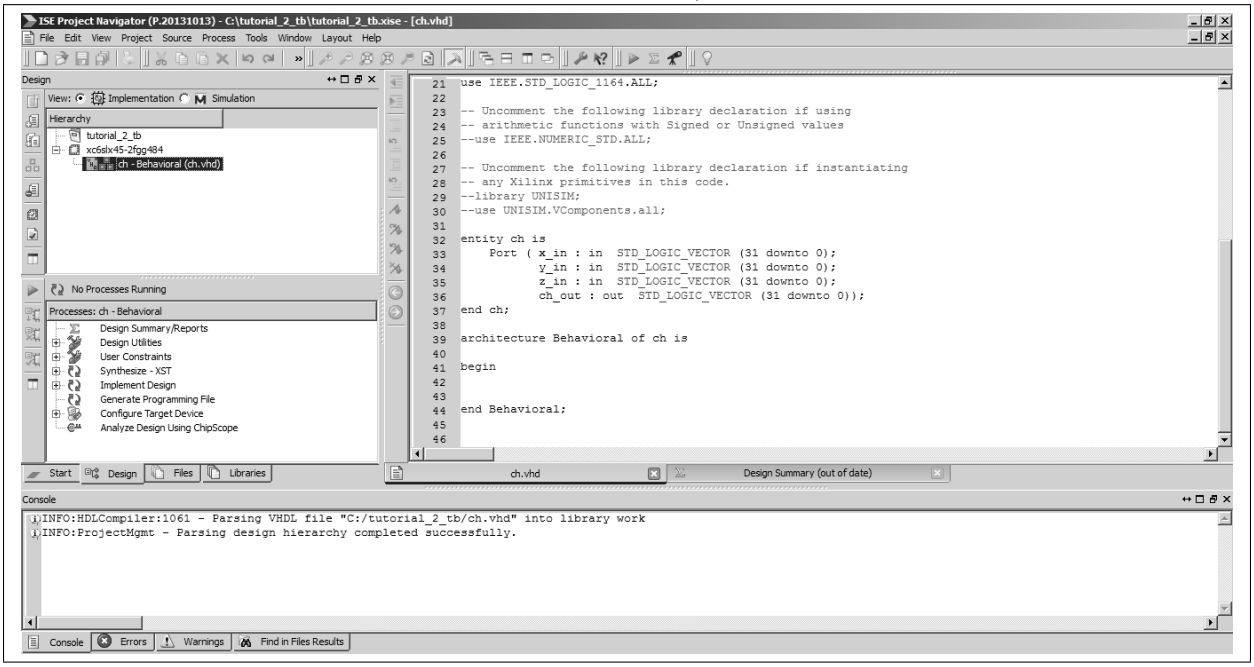
In the Implementation view, right-click on the name of your project, select *New Source...*. Then we will create a new VHDL module:



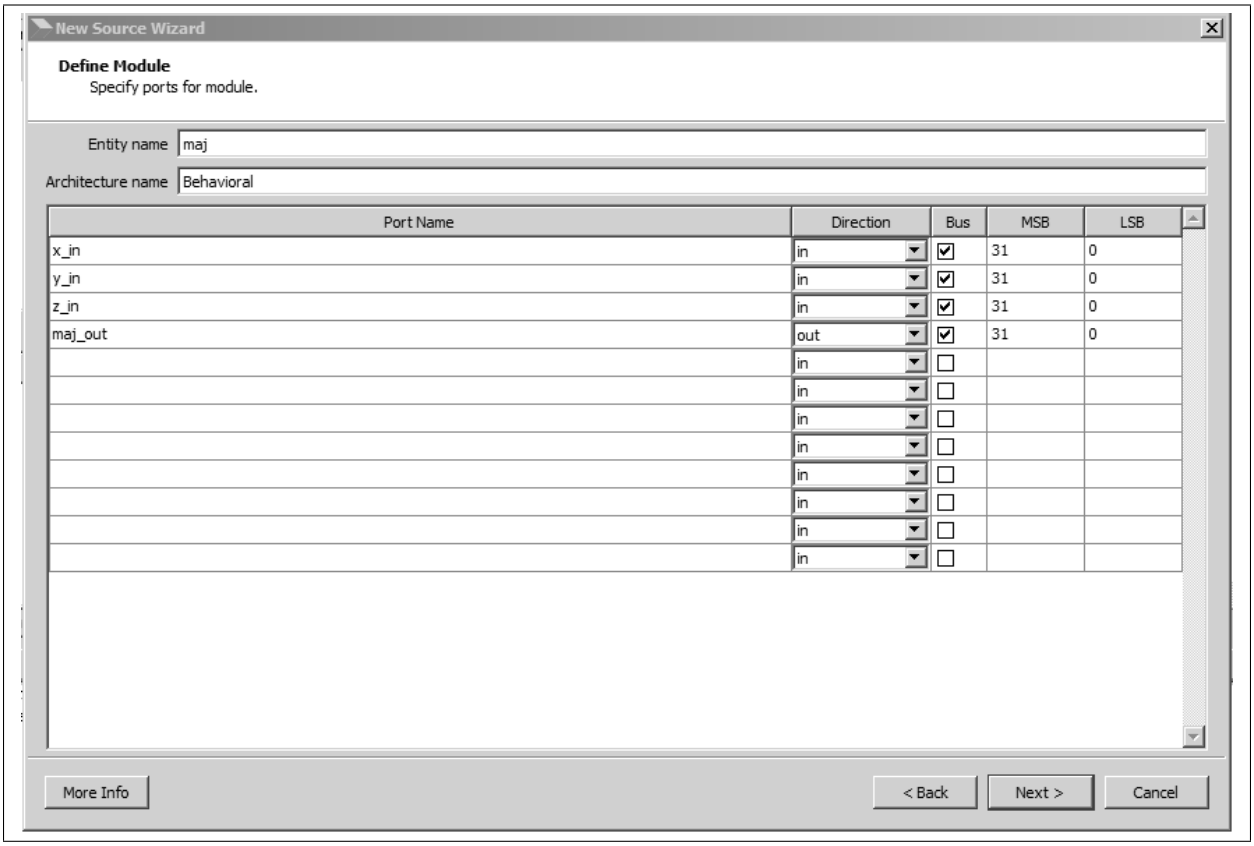
We start with the first function, *Ch*. Consequently, the name of the file will be "ch". This function requires 3 inputs of 32 bits and as output, the result of applying *Ch* (32 bits). Therefore, in the next window we can create 3 inputs (*x_in, y_in, z_in*) and one input *ch_out*.



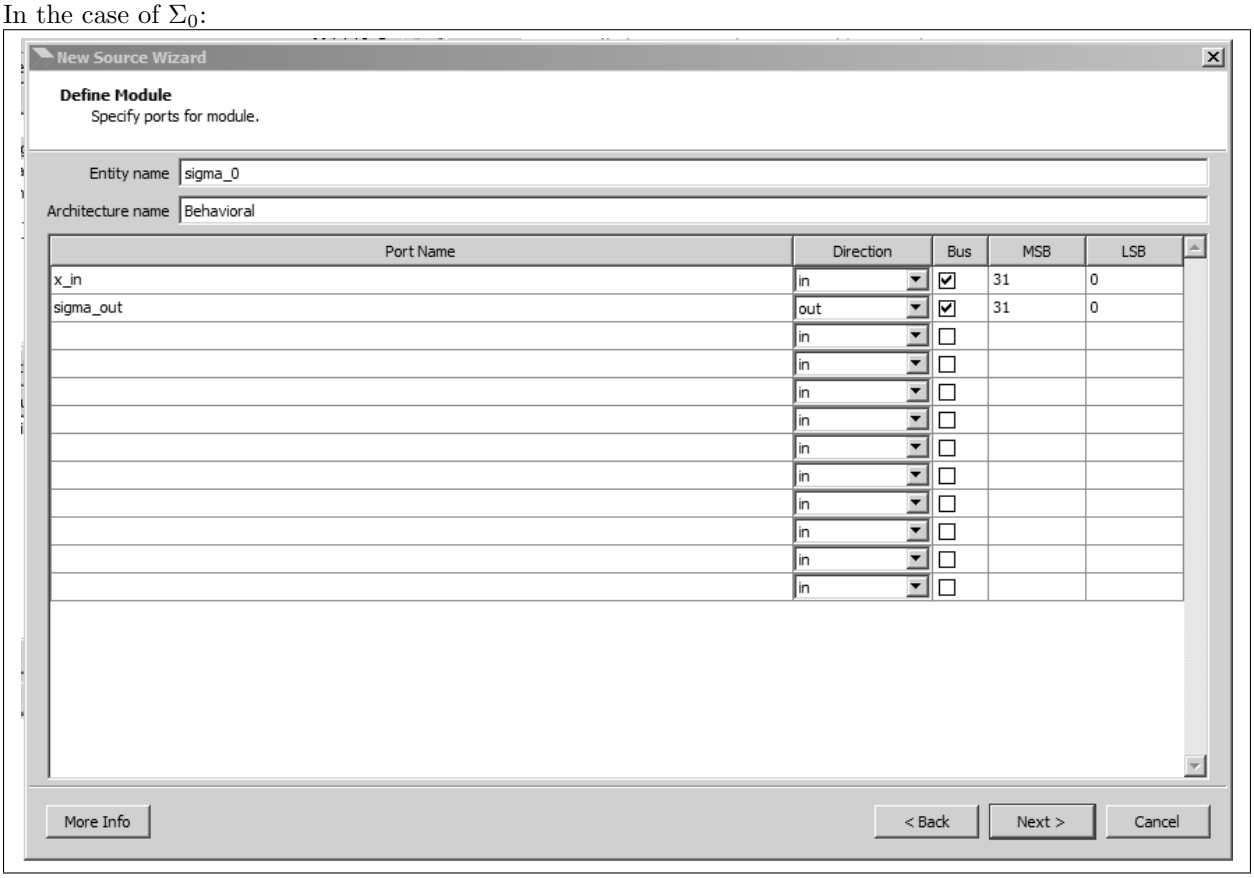
Note that we set the direction of *ch_out* as out. Afterwards, our module looks like:



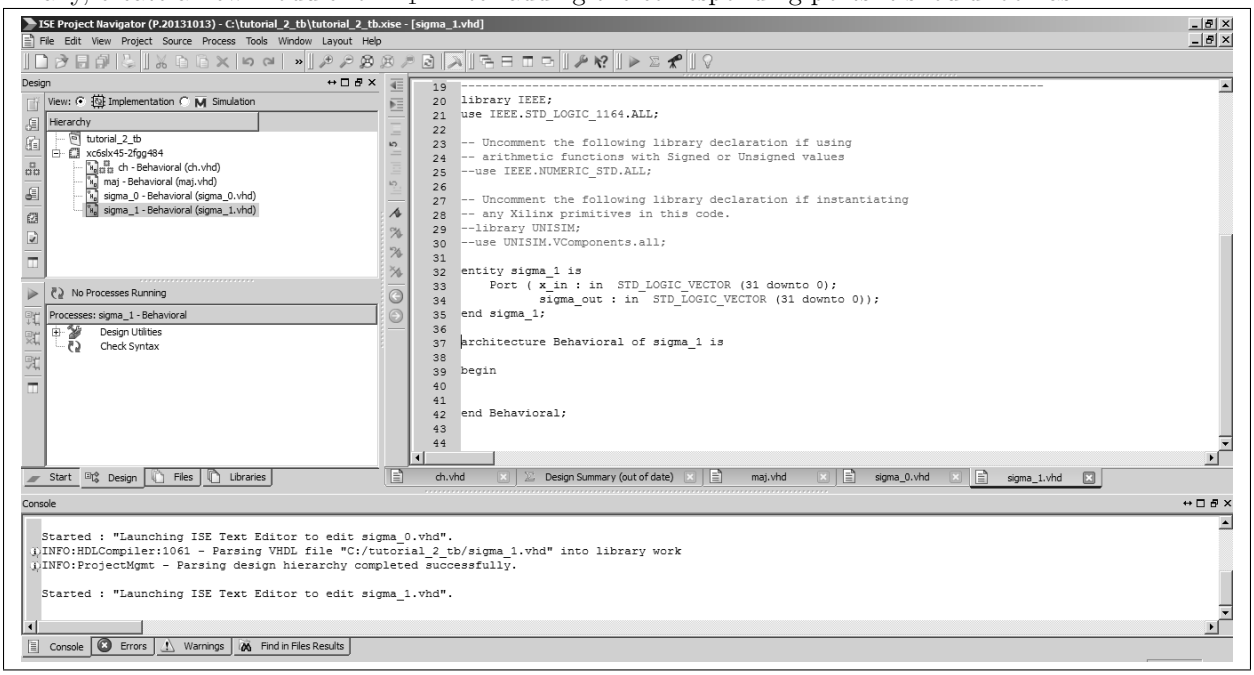
This is still an empty module with an empty architecture. We will add the code later. Now, we continue adding new VHDL modules to our project as we did with the first one. For *Maj*:



In the case of Σ_0:



Finally, create a new module for Σ_1. After adding the corresponding ports it should look as:

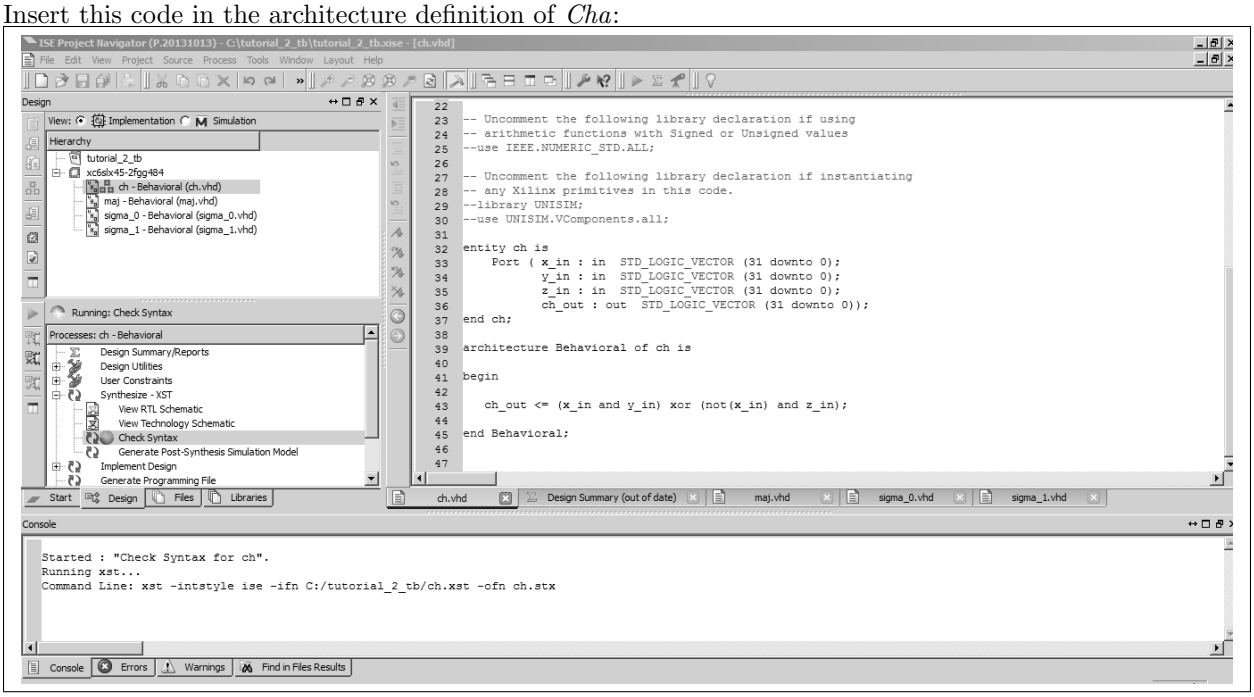


Implementing the *Ch*, *Maj*, Σ_0 and Σ_1 functions

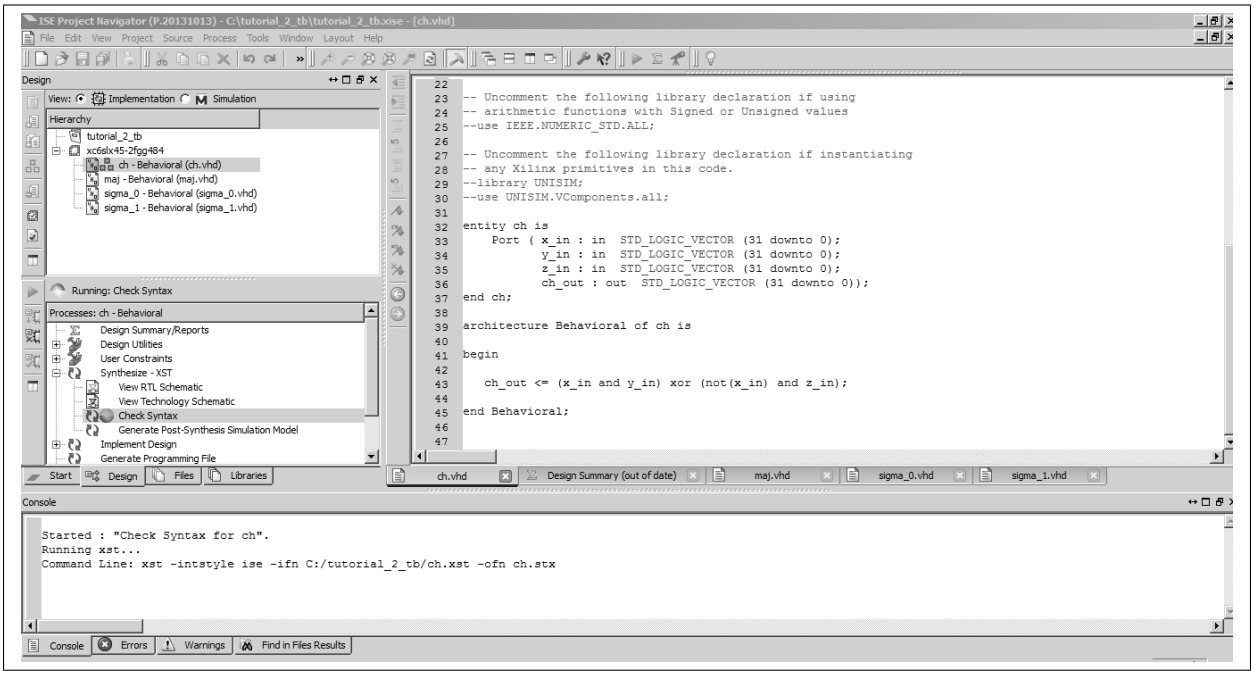
First, we will implement the *Ch* function. It can be implemented as:

```
ch_out <= (x_in and y_in) xor (not(x_in) and z_in);
```

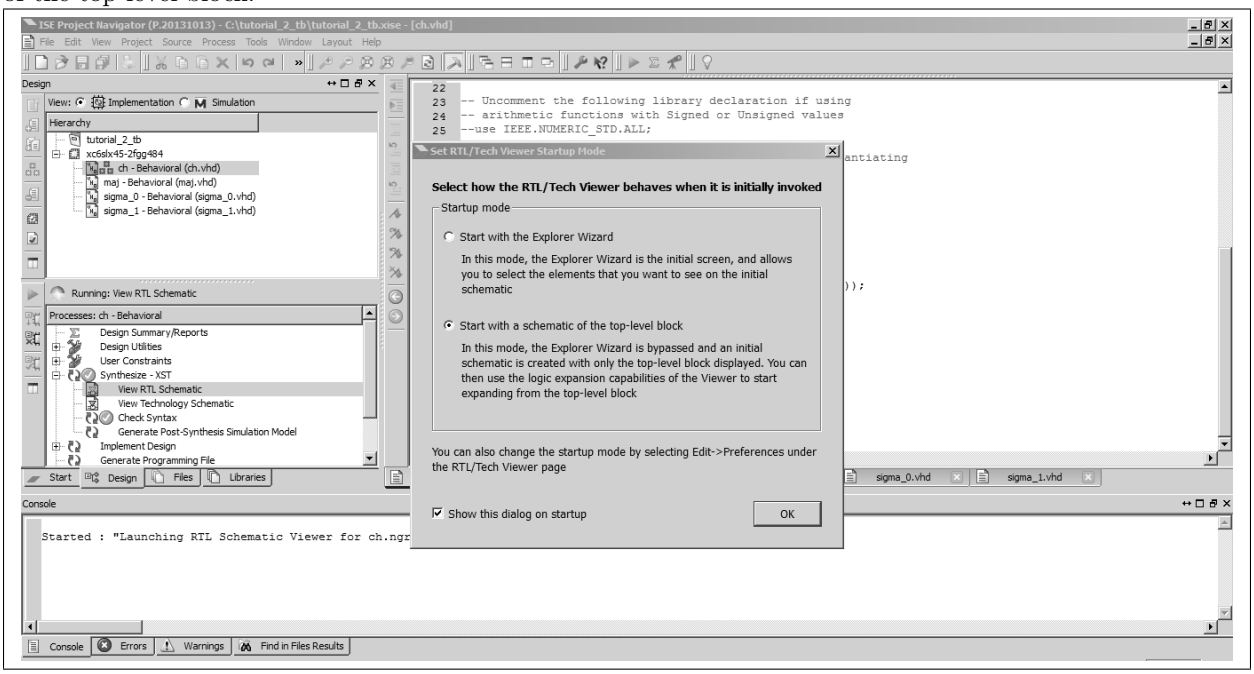
Insert this code in the architecture definition of *Cha*:



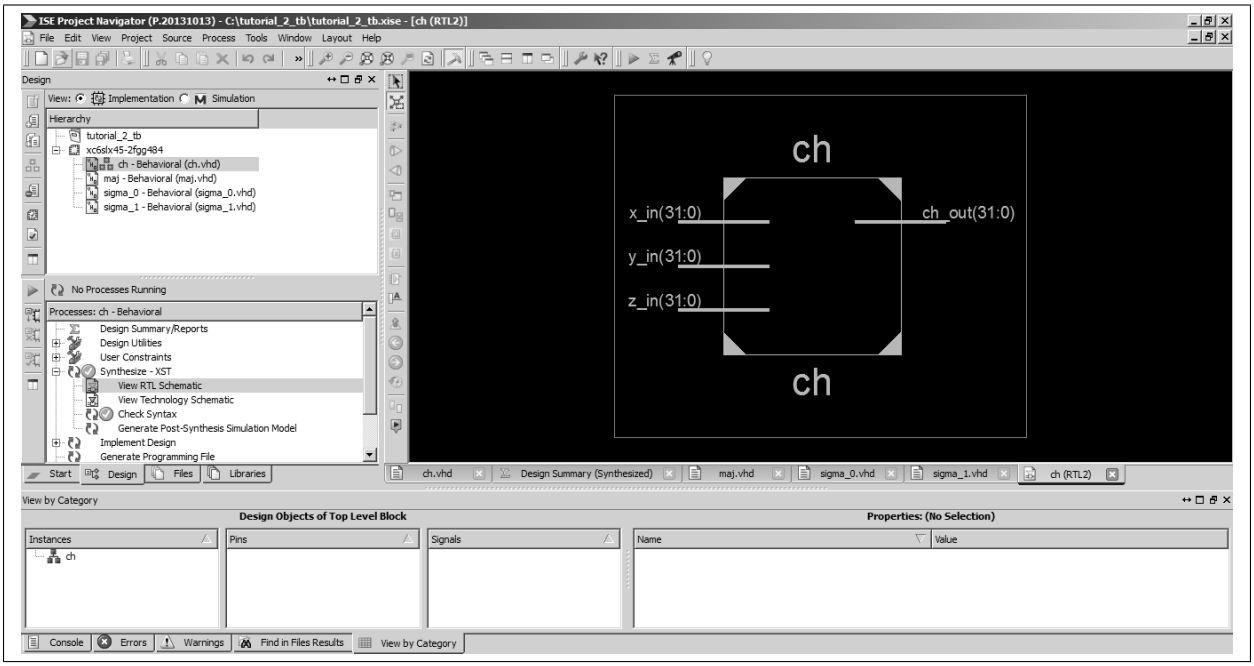
Sometimes, it is useful to check the syntax of your VHDL code before continuing working in your project. From the Implementation view, under Synthesize - XST, double-click Check Syntax:



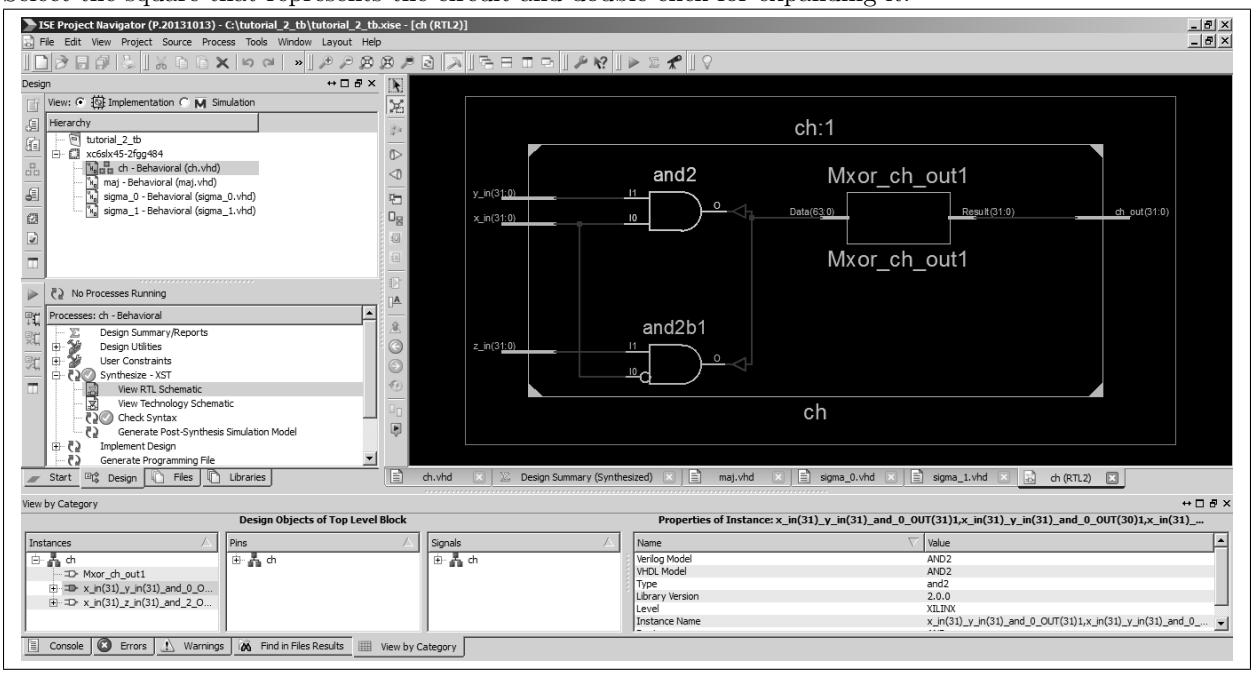
Besides, we can generate the register-transfer level (RTL) diagram corresponding to the circuit we have defined. Under Synthesize - XST, now double-click View RTL Schematic. This operation synthesize the description of the circuit into elements that can be instantiated into the FPGA. Select Start with a schematic of the top level block:



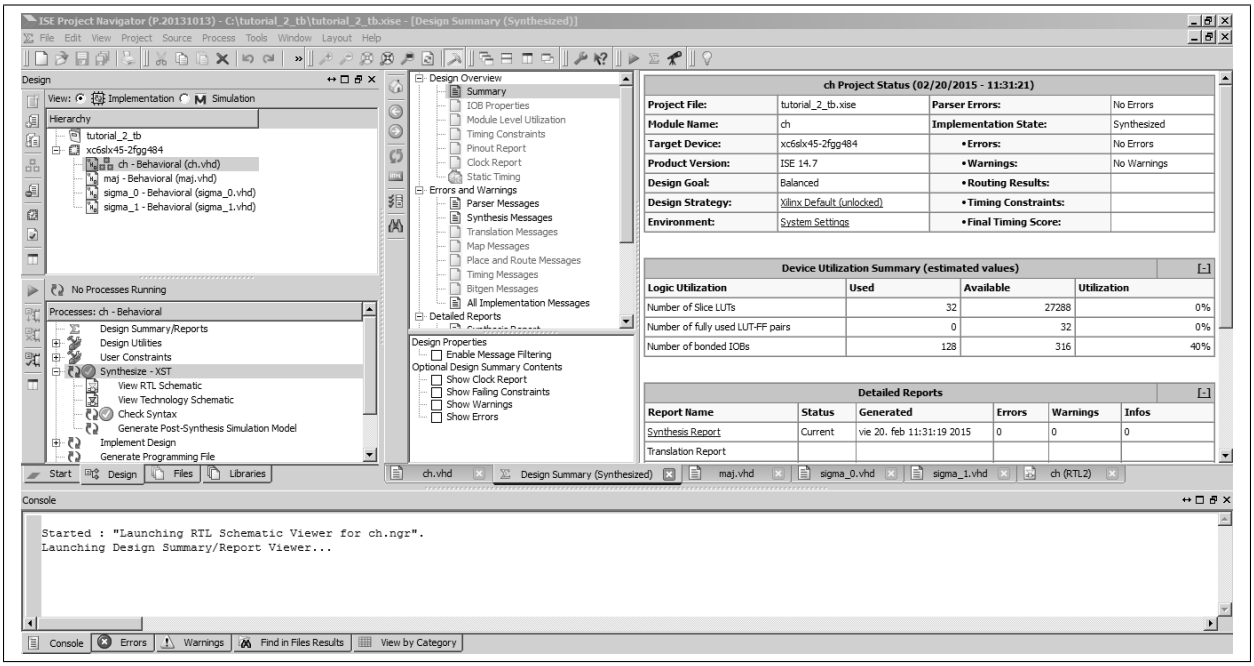
Then, you will see the top block of your circuit (Ch), with its corresponding inputs and outputs of 32 bits:



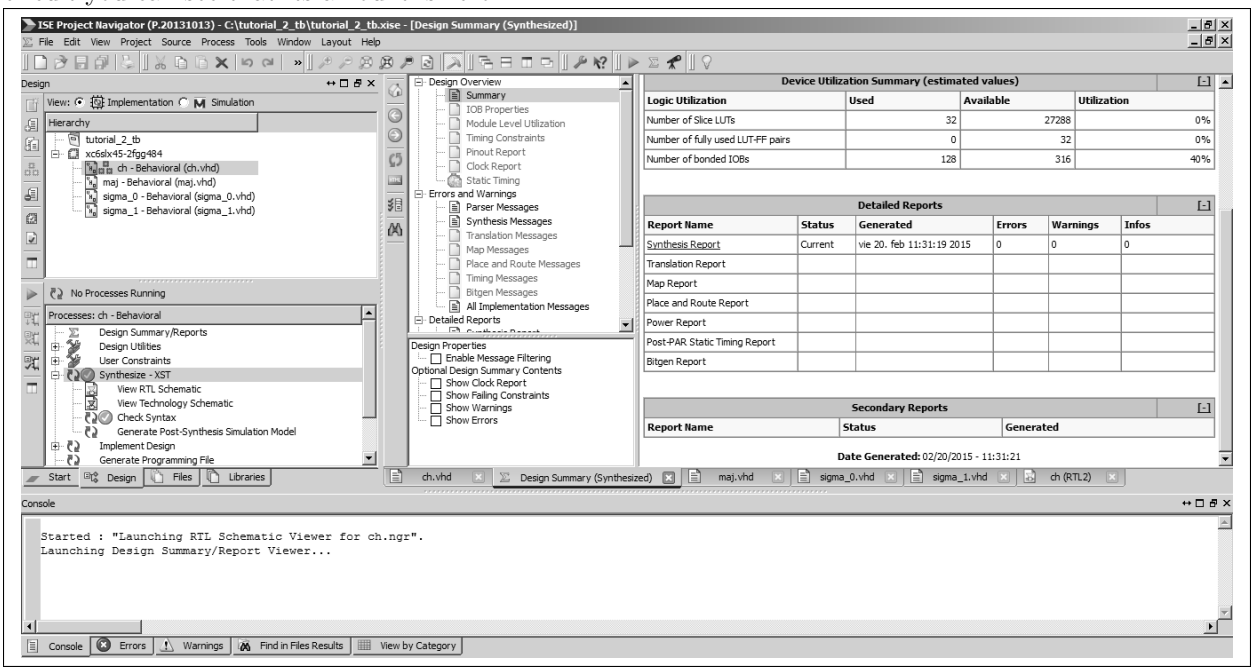
Select the square that represents the circuit and double-click for expanding it.



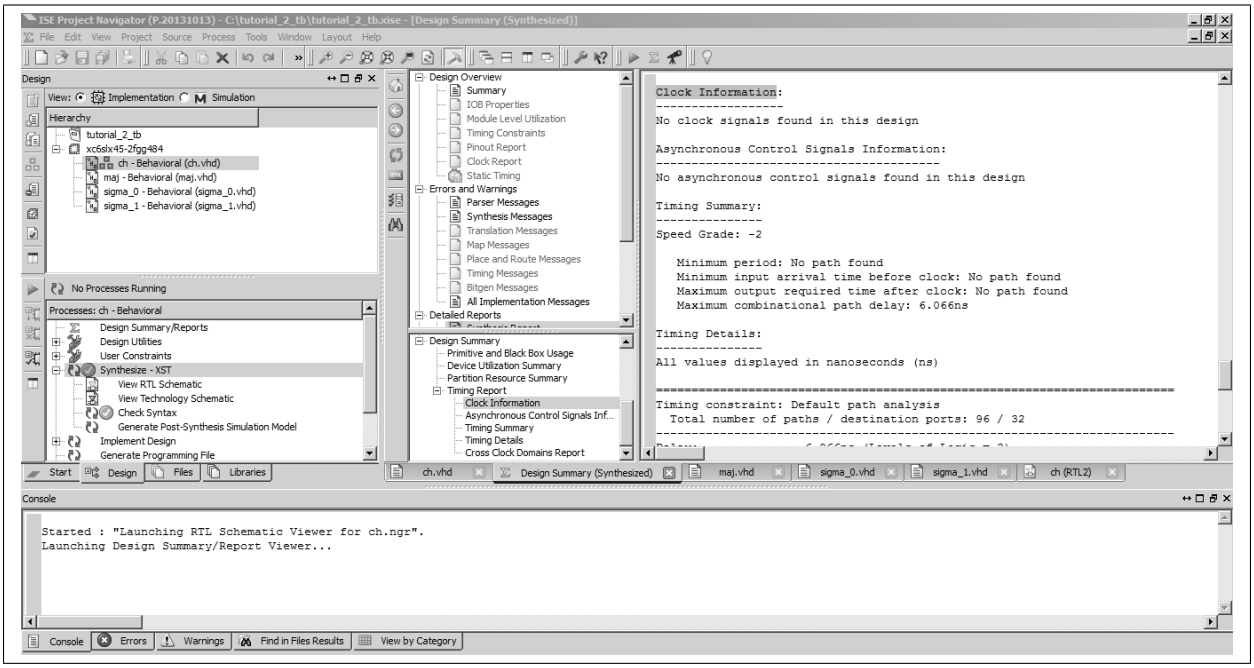
This is the circuit you have described using VHDL.



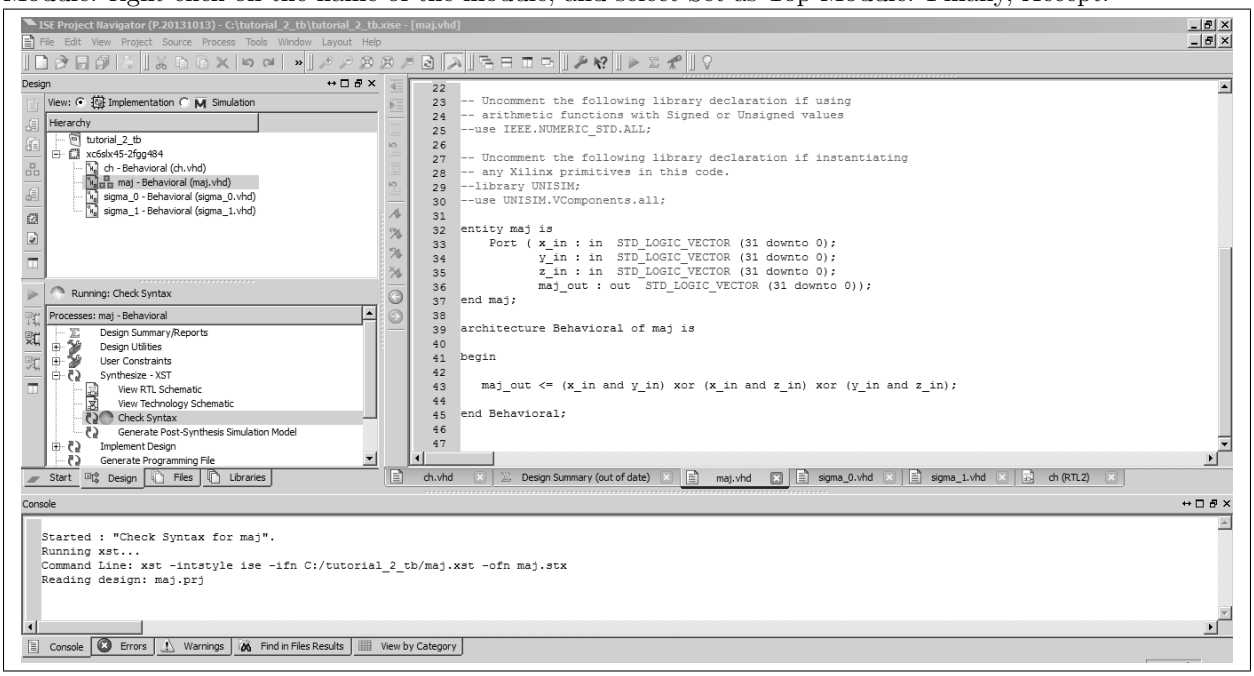
You can also open the synthesis report corresponding to your circuit. Go to the Project menu and select Design summary/Report. Select Summary. You can see that you are using 32 LUTs (Look-Up Tables) of a total of 27,208. The look-up tables are the basic elements of an FPGA. They encode Boolean functions as a table with a certain number of inputs and one output. For complex operations, several LUTs are connected (routed) into the FPGA. An FPGA also contains FFs but since we are designing a simple combinational circuit you can see that its amount is zero.



Click on synthesis report inside the Detailed report table. Select Clock Information within the Timing Report menu. Here you can obtain the combinational path delay of your circuit i.e. the time you would need to see an output in ch_out: 6.060 ns.



Continue writing the implementation of the other circuits. Since this project contains independent circuits, in order to work with another module e.g. for synthesis, simulation, etc. you will have to select it as Top Module: right-click on the name of the module, and select Set as Top Module. Finally, Accept.



Implement the Σ_0 function as seen below:

```
entity sigma_0 is
        port (x_in: in std_logic_vector(31 downto 0);
                      sigma_out: out std_logic_vector(31 downto 0));
end sigma_0;

architecture Behavioral of sigma_0 is
        signal tmp_0 : std_logic_vector(31 downto 0);
        signal tmp_1 : std_logic_vector(31 downto 0);
        signal tmp_2 : std_logic_vector(31 downto 0);
begin

        tmp_0 <= x_in(1 downto 0) & x_in(31 downto 2);
        tmp_1 <= x_in(12 downto 0) & x_in(31 downto 13);
```

```
        tmp_2 <= x_in(21 downto 0) & x_in(31 downto 22);

        sigma_out <= tmp_0 xor tmp_1 xor tmp_2;

end Behavioral;
```

And finally continue with Σ_1:

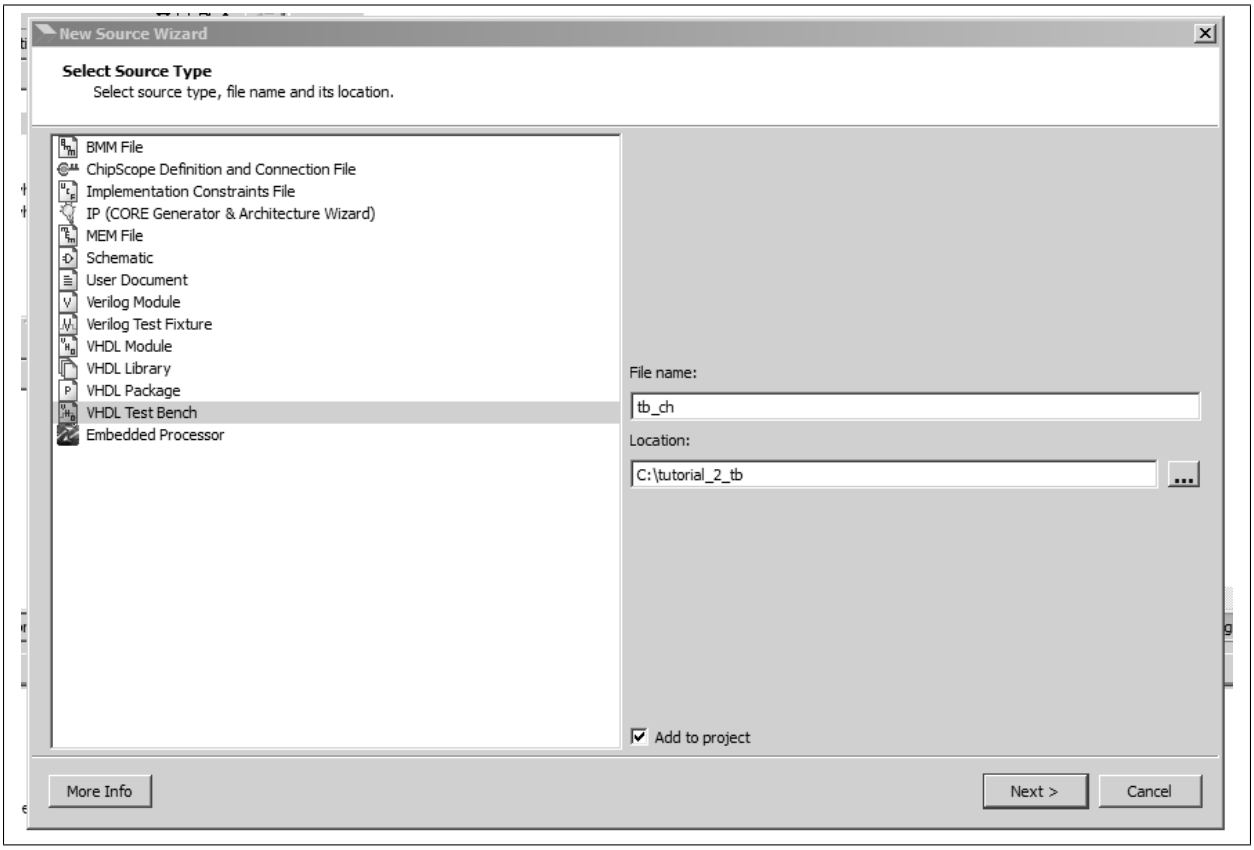
```
entity sigma_1 is
        port (x_in: in std_logic_vector(31 downto 0);
                      sigma_out: out std_logic_vector(31 downto 0));
end sigma_1;

architecture Behavioral of sigma_1 is
        signal tmp_0 : std_logic_vector(31 downto 0);
        signal tmp_1 : std_logic_vector(31 downto 0);
        signal tmp_2 : std_logic_vector(31 downto 0);
begin

        tmp_0 <= x_in(5 downto 0) & x_in(31 downto 6);
        tmp_1 <= x_in(10 downto 0) & x_in(31 downto 11);
        tmp_2 <= x_in(24 downto 0) & x_in(31 downto 25);

        sigma_out <= tmp_0 xor tmp_1 xor tmp_2;

end Behavioral;
```

Simulation

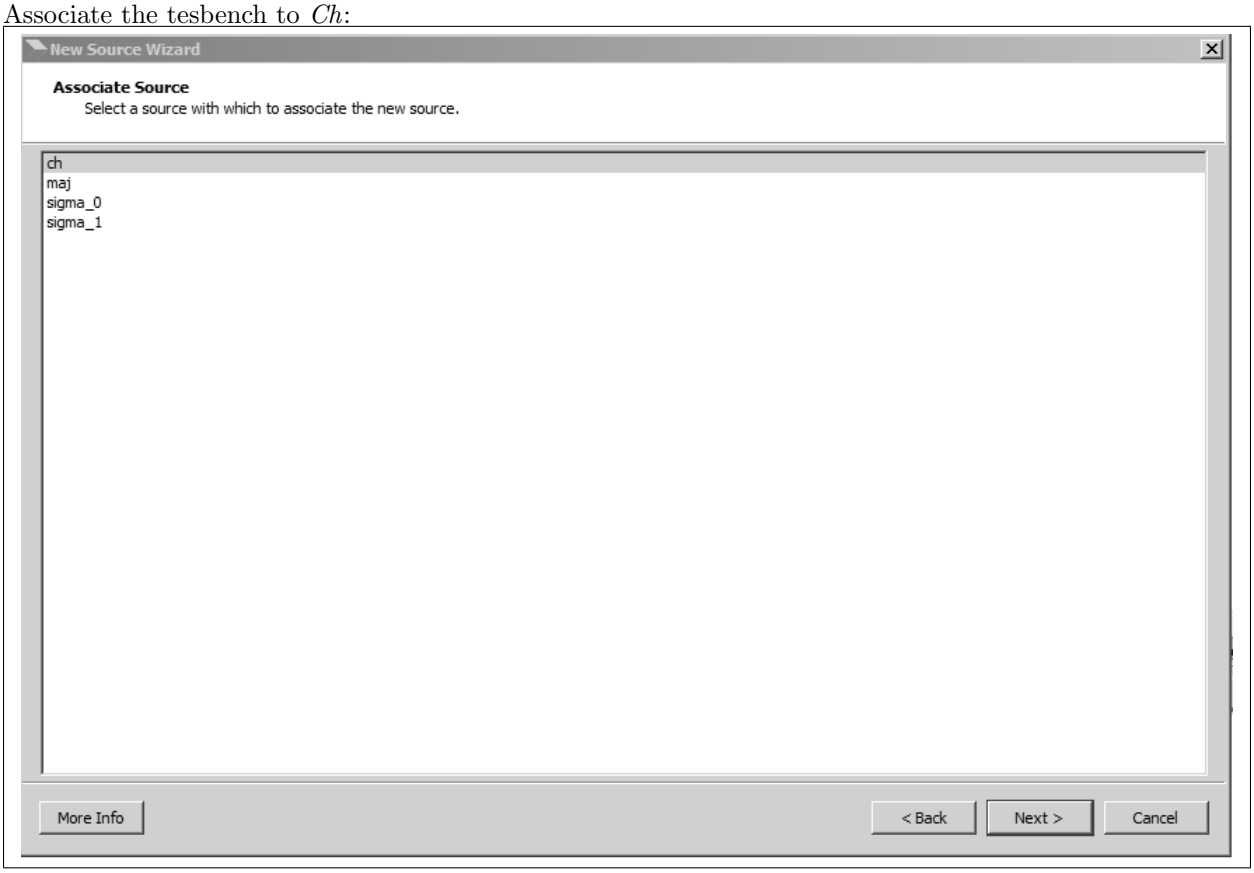
In this section, we will create a testbench for each module. This tesbench imports a certain module and send signals to its ports in order to verify its correctness. In order to create test vectors, it is useful to have a C implementation of each function. For instance:

```
/**
 * C implementation of some of the SHA-256 functions
 * for generating test vectors.
 */

#include <stdint.h>
#include <stdio.h>

uint32_t rotr(uint32_t value, uint32_t shift) {
    return (value >> shift) | (value << (sizeof(value) * 8 - shift));
}

uint32_t ch(uint32_t x, uint32_t y, uint32_t z) {
    return (x & y) ^ (~x & z);
}

uint32_t maj(uint32_t x, uint32_t y, uint32_t z) {
    return (x & y) ^ (x & z) ^ (y & z);
}

uint32_t sigma_0(uint32_t x) {
    return rotr(x, 2) ^ rotr(x, 13) ^ rotr(x, 22);
}

uint32_t sigma_1(uint32_t x) {
    return rotr(x, 6) ^ rotr(x, 11) ^ rotr(x, 25);
}

int main(int argc, char **argv)
{

    uint32_t x = 0x38203523, y = 0x68382835, z = 0x92835234;

    printf("%x\n", ch(x, y, z));

}
```

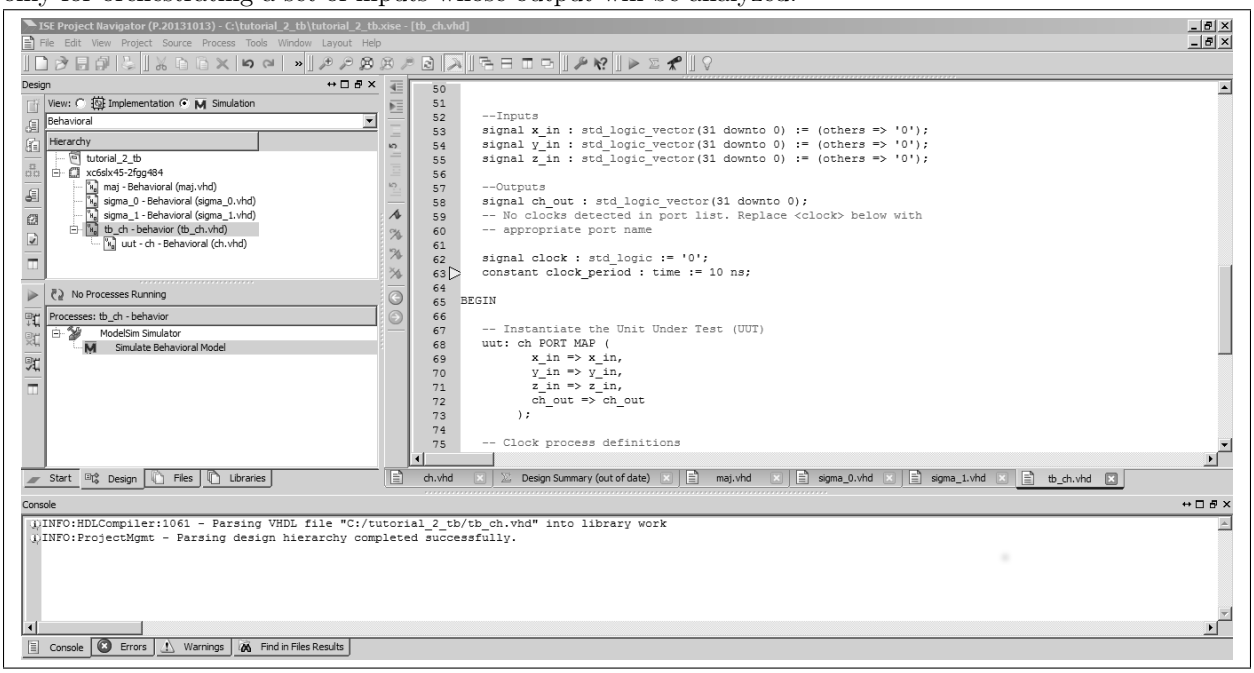
We'll start by creating a testbench for the *Ch* function. First, select *Ch* as Top Block again. Then, select your project in the Implementation view, right-lick and select New Source. This time, you should select VHDL Test Bench.
Generally, as a convention the name of a testbench starts by the *tb_* prefix. Therefore, you can choose the following file name:



Associate the tesbench to *Ch*:



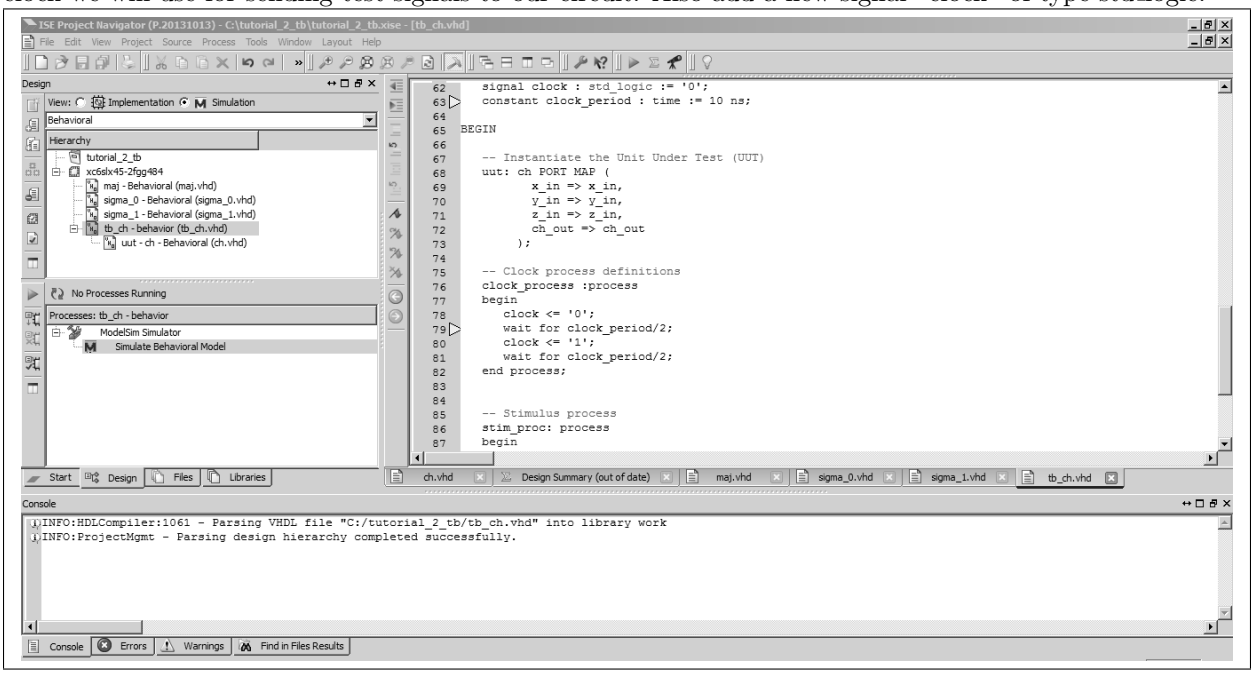
Now, go to the Simulation view and open the *tb_ch.vhd* file. You will see that the circuit that you are testing appears as a component (lines 42-49). Around lines 67-72 it has been instantiated via the port map statement.
Since our circuit (categorized as Unit Under Test (UUT)) is not sequential and it does not have a clock, Xilinx does not define one. We have to fix this. We need a clock that could be used to feed our circuit but only for orchestrating a set of inputs whose output will be analyzed.



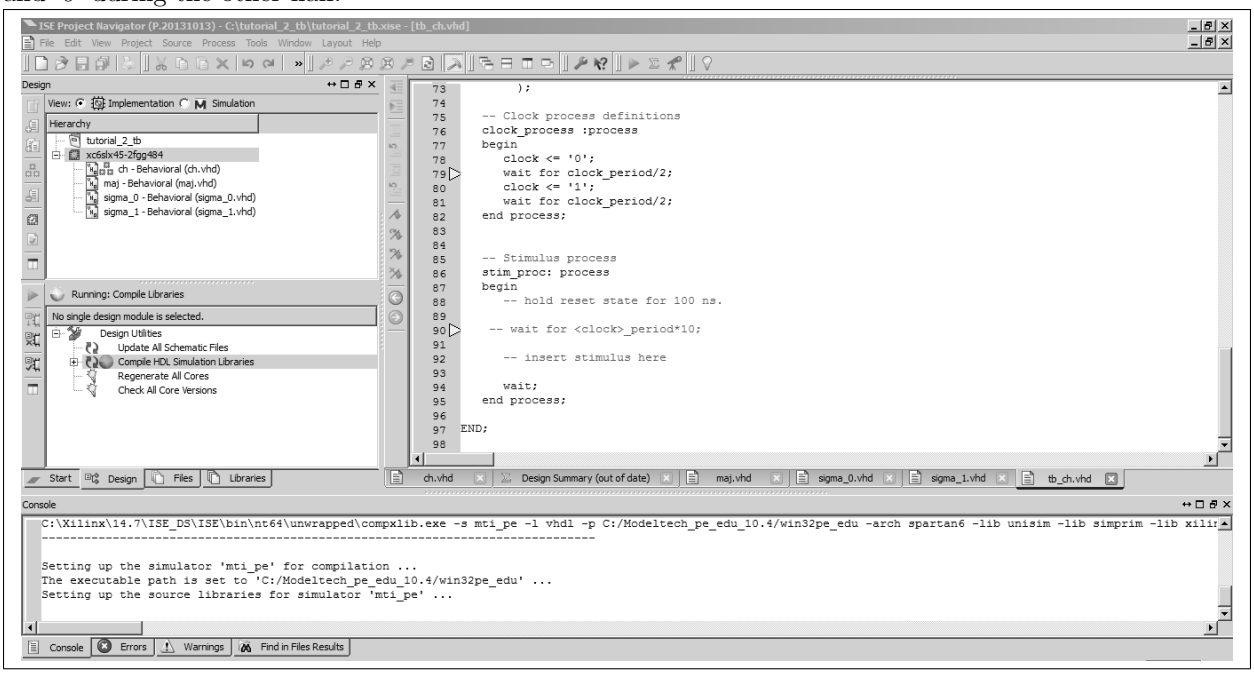
Go to line 62 and rename the <clock>_period signal as clock_period i.e.

```
constant clock_period : time := 10 ns;
```

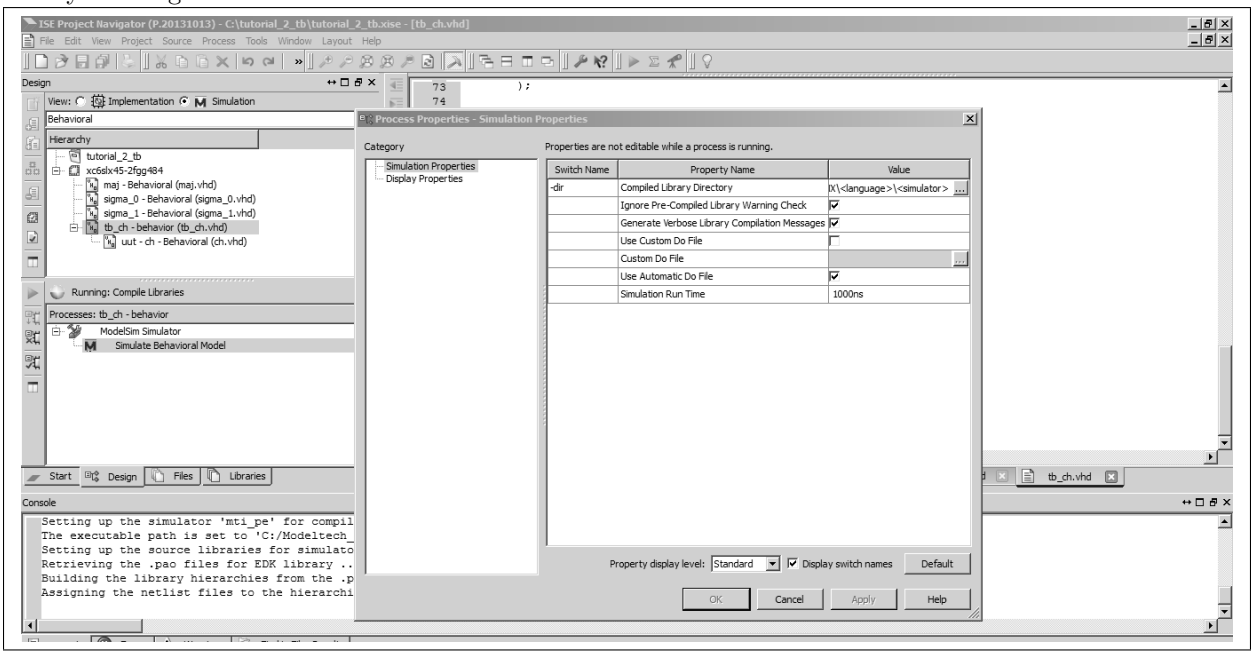
This constant of type *time* allow time magnitudes such as nanoseconds (ns) and defines the period of the clock we will use for sending test signals to our circuit. Also add a new signal "clock" of type std_logic.



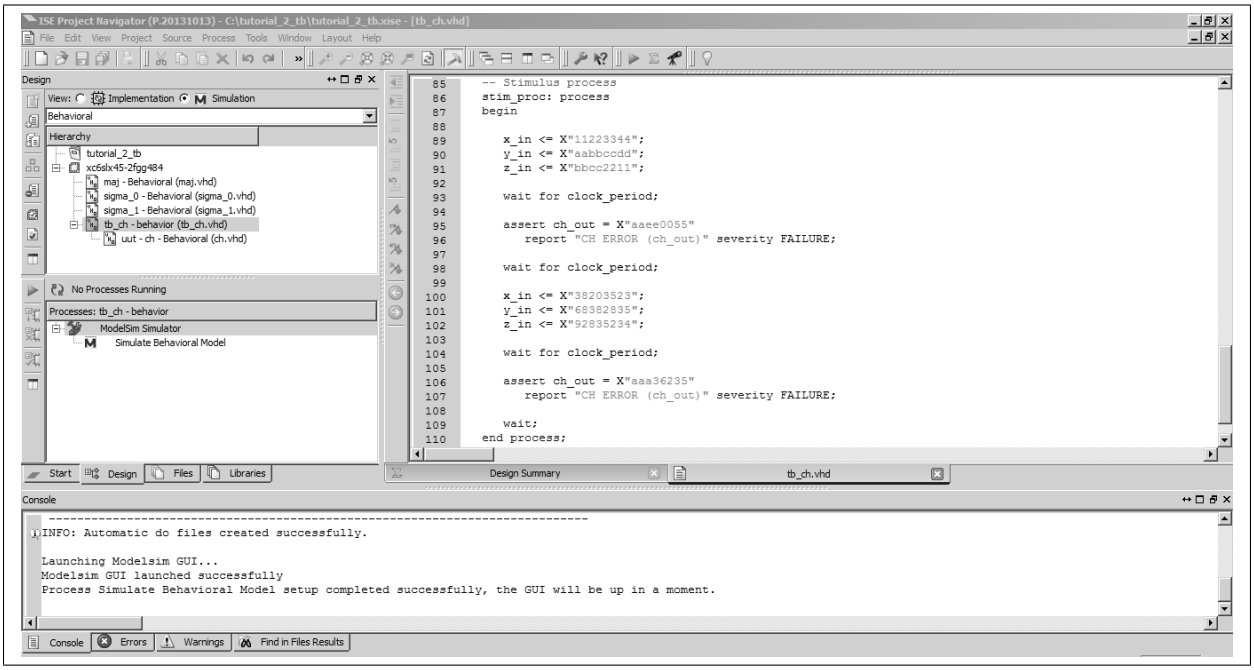
In line 74, rename the <> characters from the process label (clock_process) and inside, correct the name of the signal to clock. This process creates a clock by setting this signal to '1' during the first half of the period and '0' during the other half.



At this point, we are almost ready to simulate *Ch*. First, come back to the Implementation view, select your project and double-click Compile HDL Simulation libraries. Then, go to the Simulation view, select *tb_ch*, right-click on Simulate Behavioural module, select Process Properties and click on Ignore Pre-compiled library warning check:



Then, using the C implementation we will generate a couple of test vectors. We will edit the process in the tesbench with label *stim_proc*. Remove everything and feed the *x_in, y_in and z_in* signals with the test vectors as seen below:



We will add two assertions that verify the result. Using assertions is useful when one has different signals in a circuit that wants to test and want to avoid looking at a wave diagram every time a module is simulated. Our assertion will check that the *ch_out* signal corresponds to each test vector we generated before. Now:

- Simulate the tesbench and check that you do not receive errors and the assertions are accepted. Model Sim stops the simulation as soon as one of the assertion is violated and you will receive a message in the console.
- Following this procedure, create testbenches for each function and a set of corresponding test vectors using a C implementation and add assertions that validate the functionality of *Maj, sigma_0, sigma_1*.
- Depending on your project, implement the basic functions of the round as combinational circuits and create testbenches that validate its implementation with a set of test vectors.