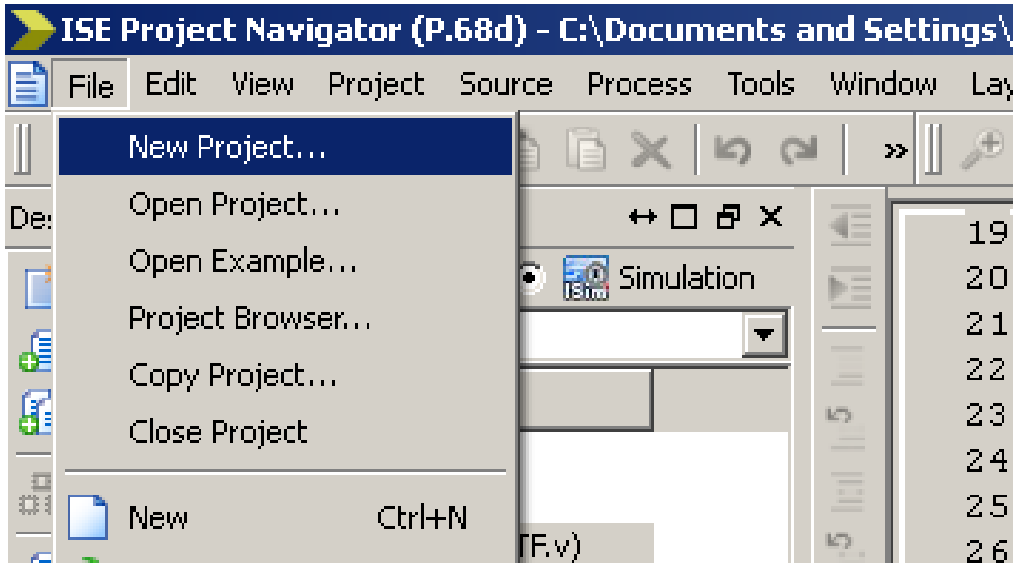
Digital design laboratory 5

Preparations

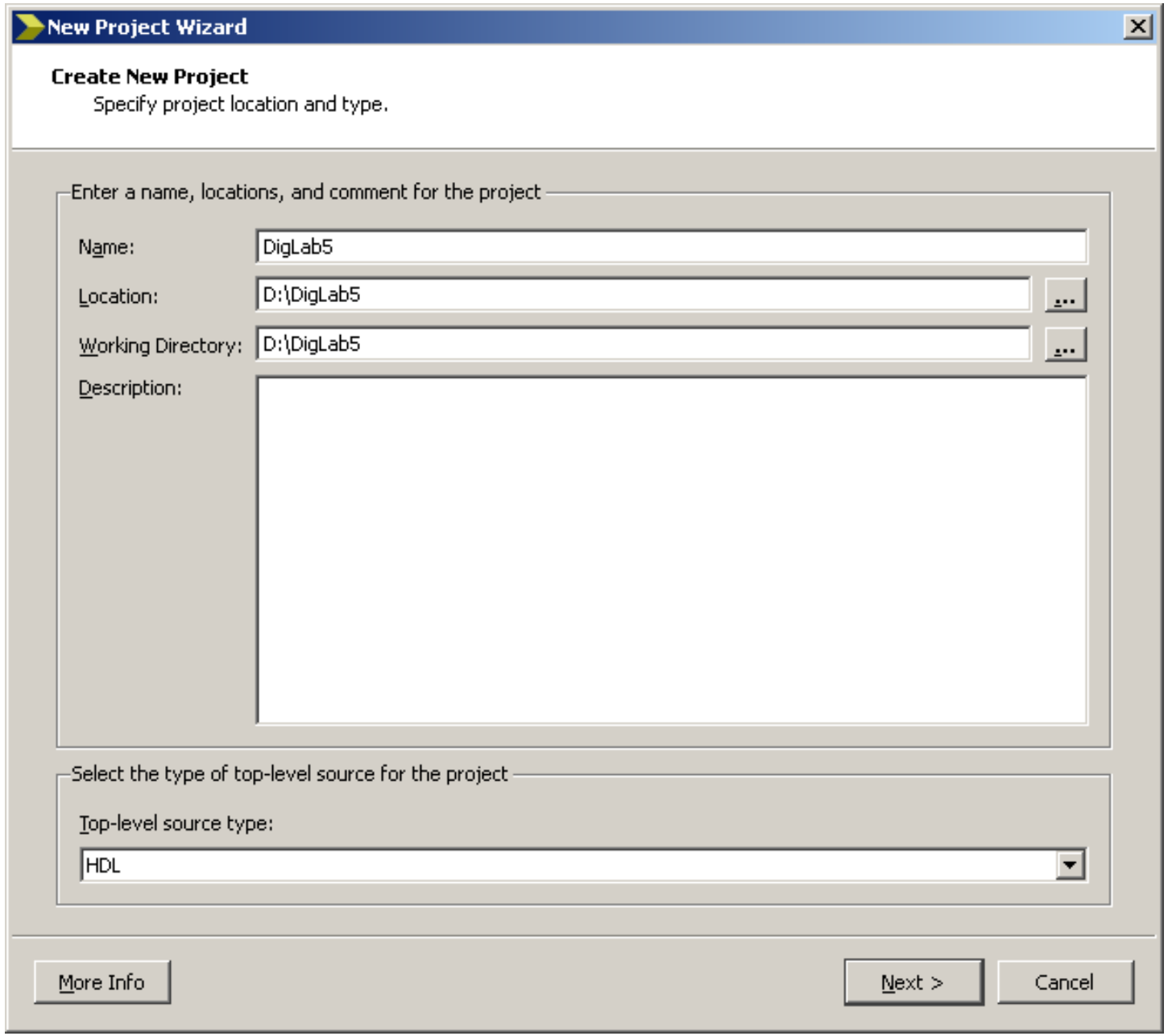
- Launch the ISE Design Suite

- Create new project:
  File -> New Project…



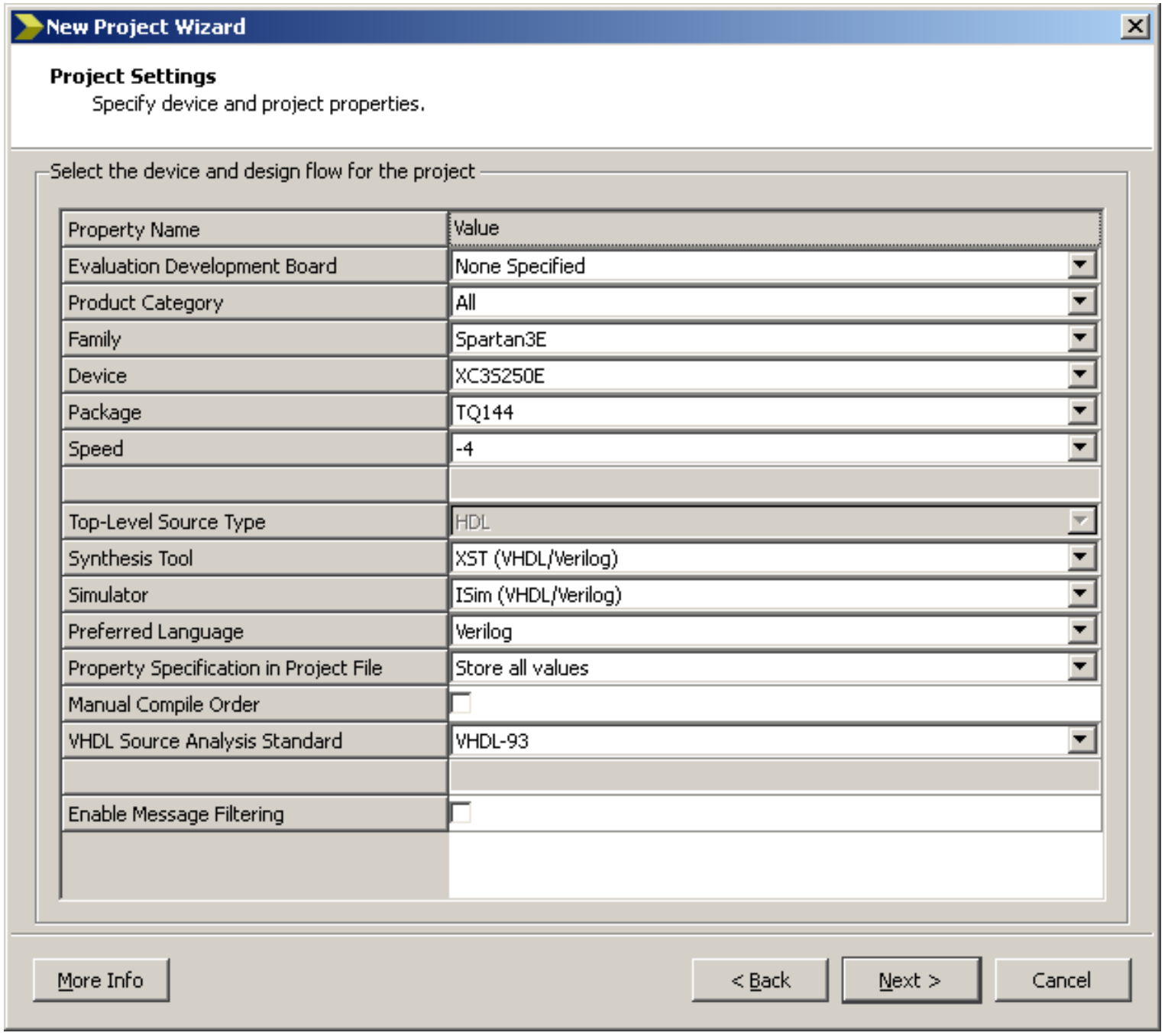
Preparations

- Name: DigLab5
- Location: D drive! D:\DigLab5
- Working directory: The same as Location
- Click Next



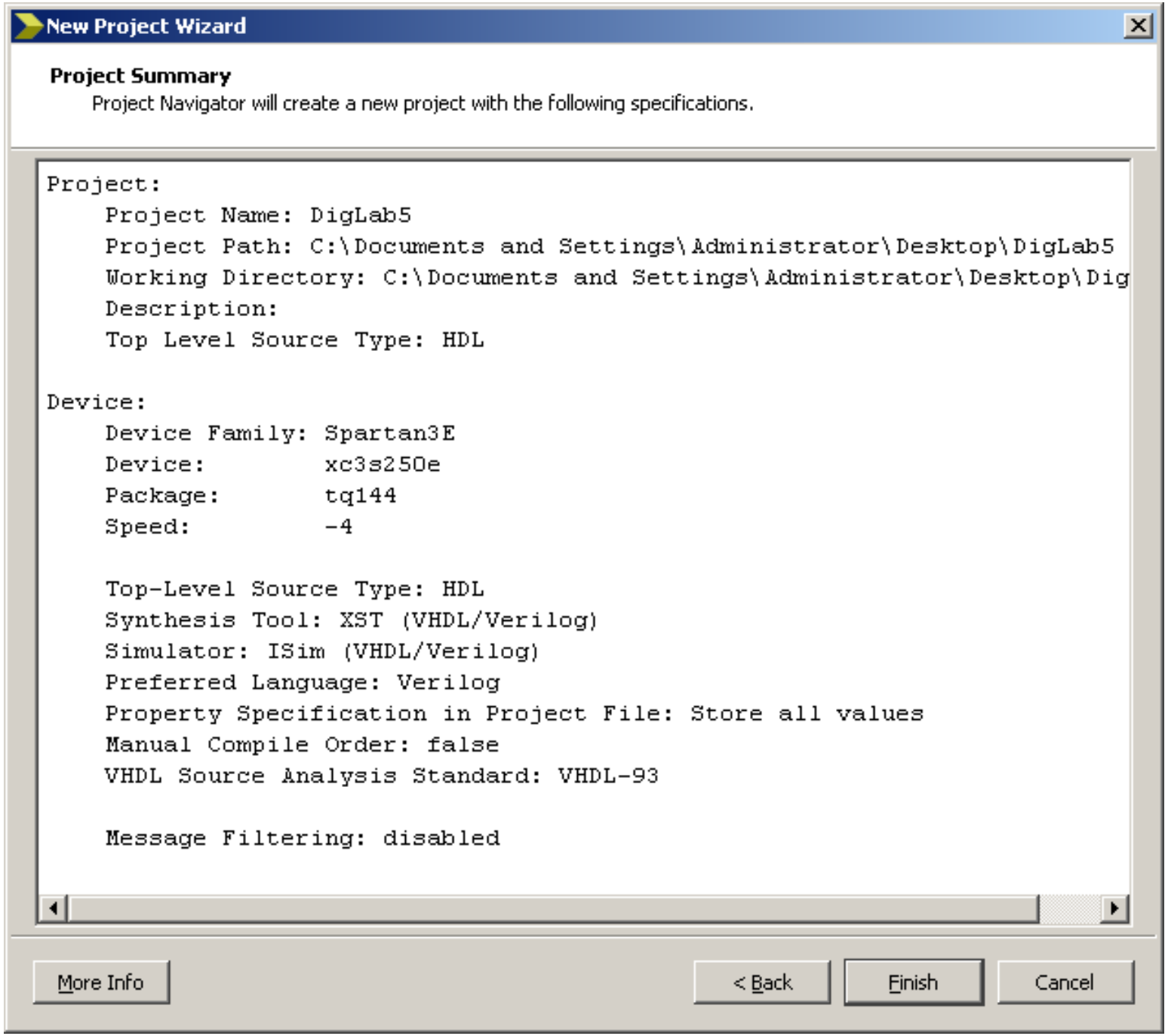
Preparations

- Check the following:
  Board
  Device
  Package
  Speed
- If OK, Click Next



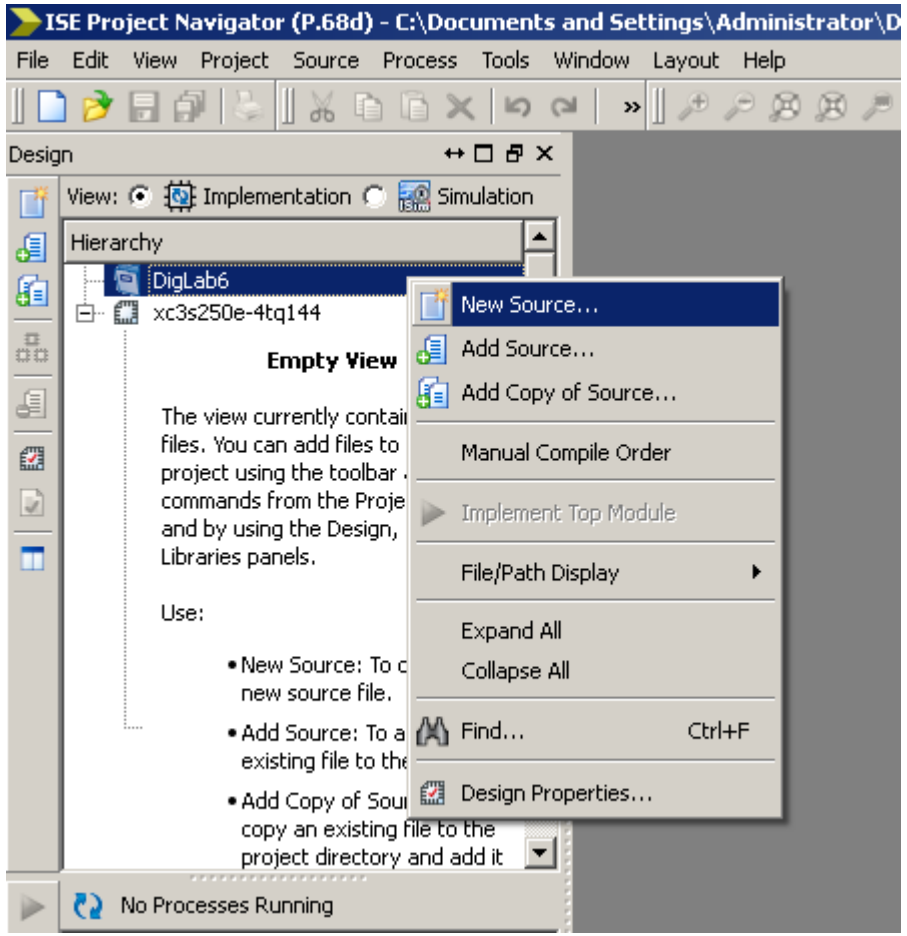
Preparations

- Click Finish



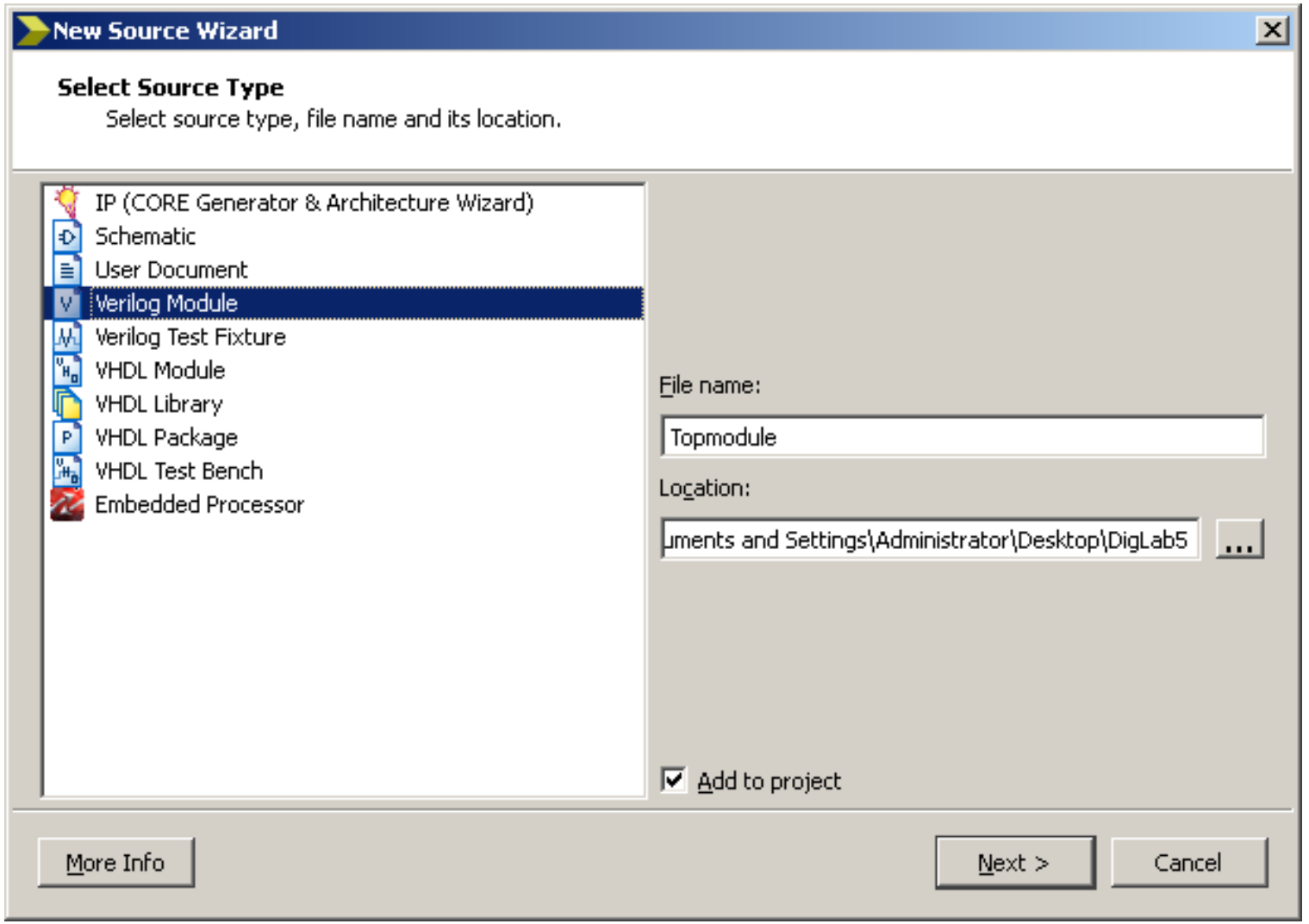
Add module to the project

- In the top left corner, right click on the project (DigLab5)
- Select New Source…



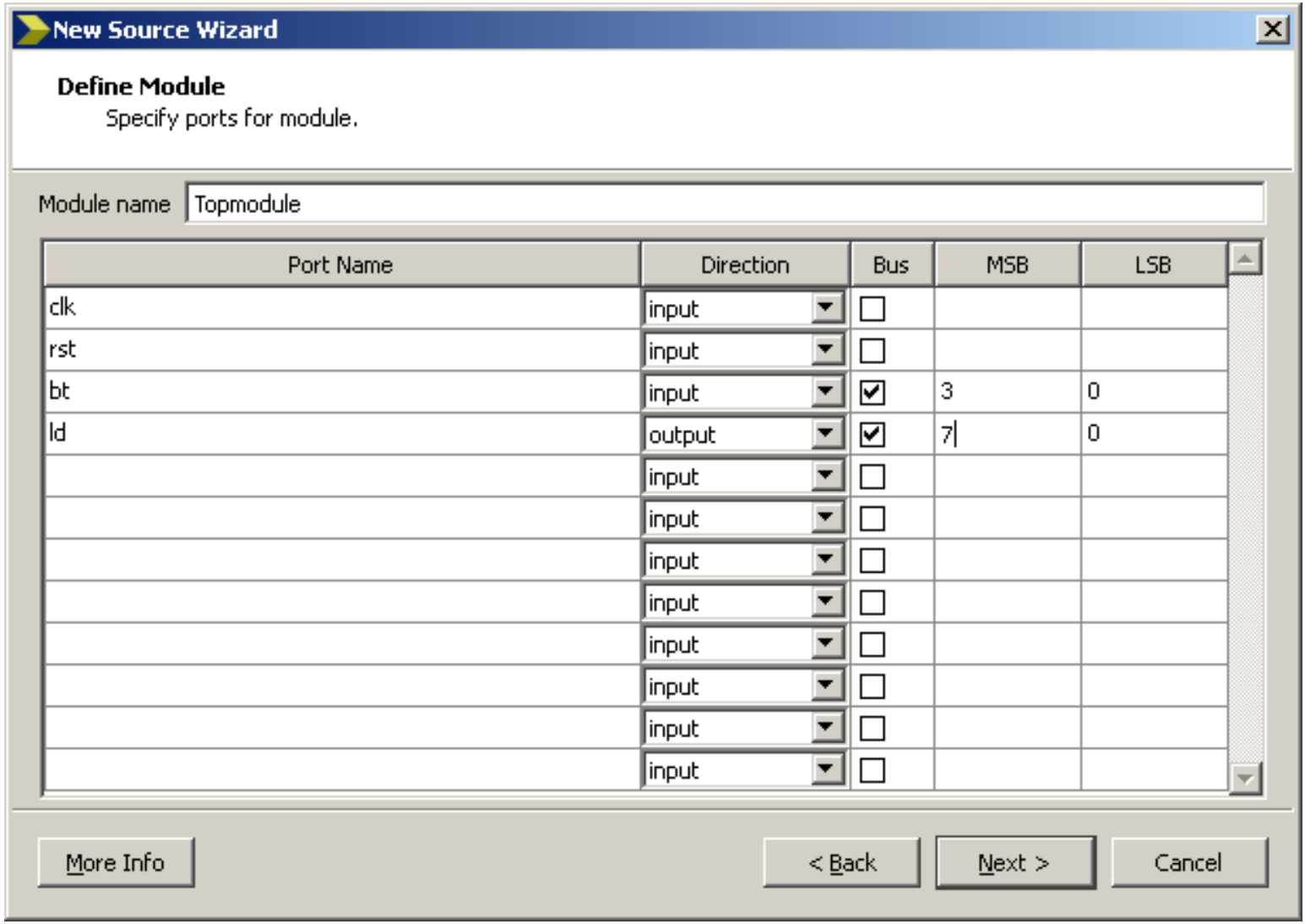
Add module to the project

- Select Verilog Module
- Name: Topmodule
- Do NOT modify the Location!
- Click Next



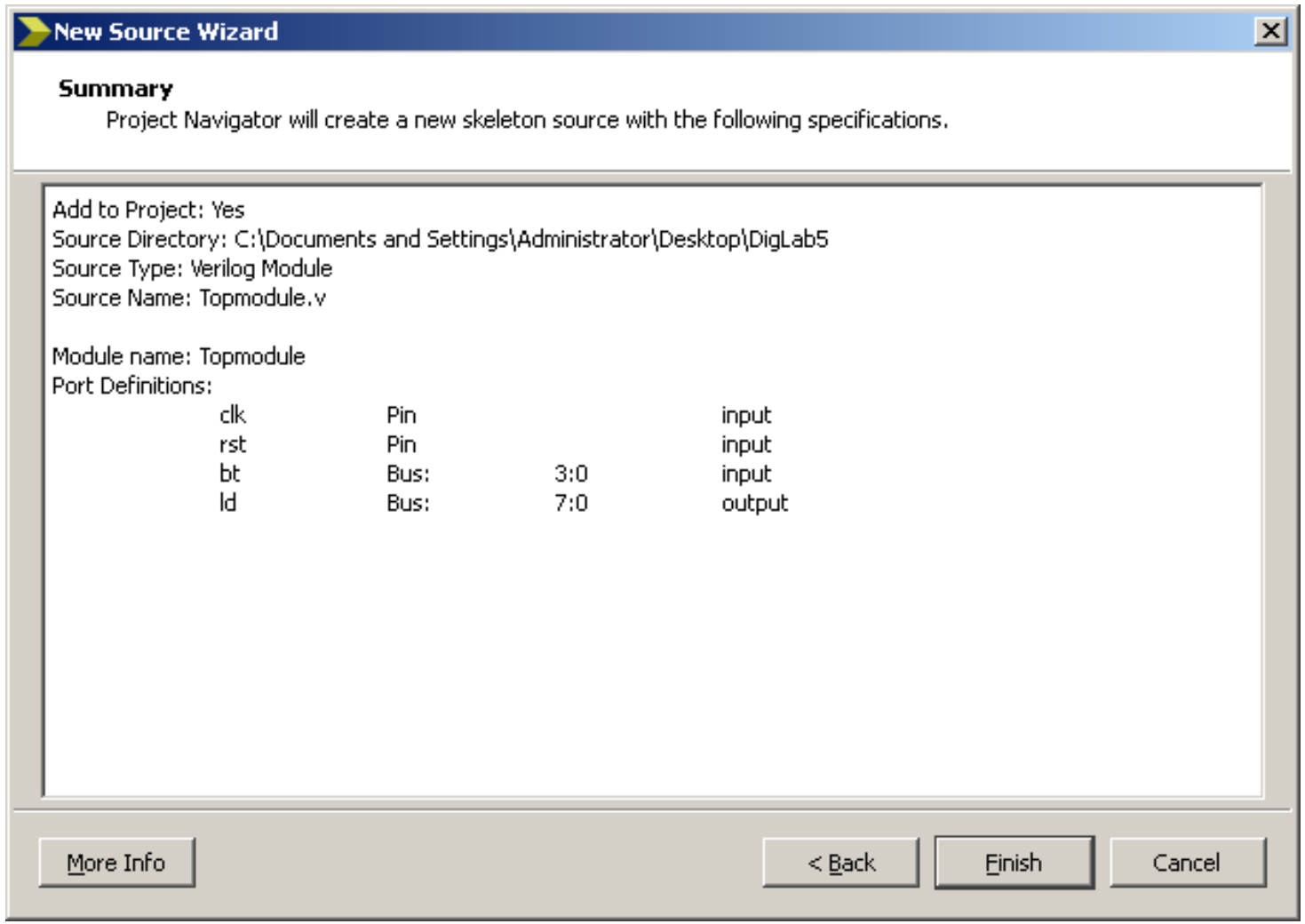
Add module to the project

- Add the following ports:
- rst, clk, bt, ld
- rst, clk and bt are inputs
- ld is an output
- bt and ld are buses!
- Click Next



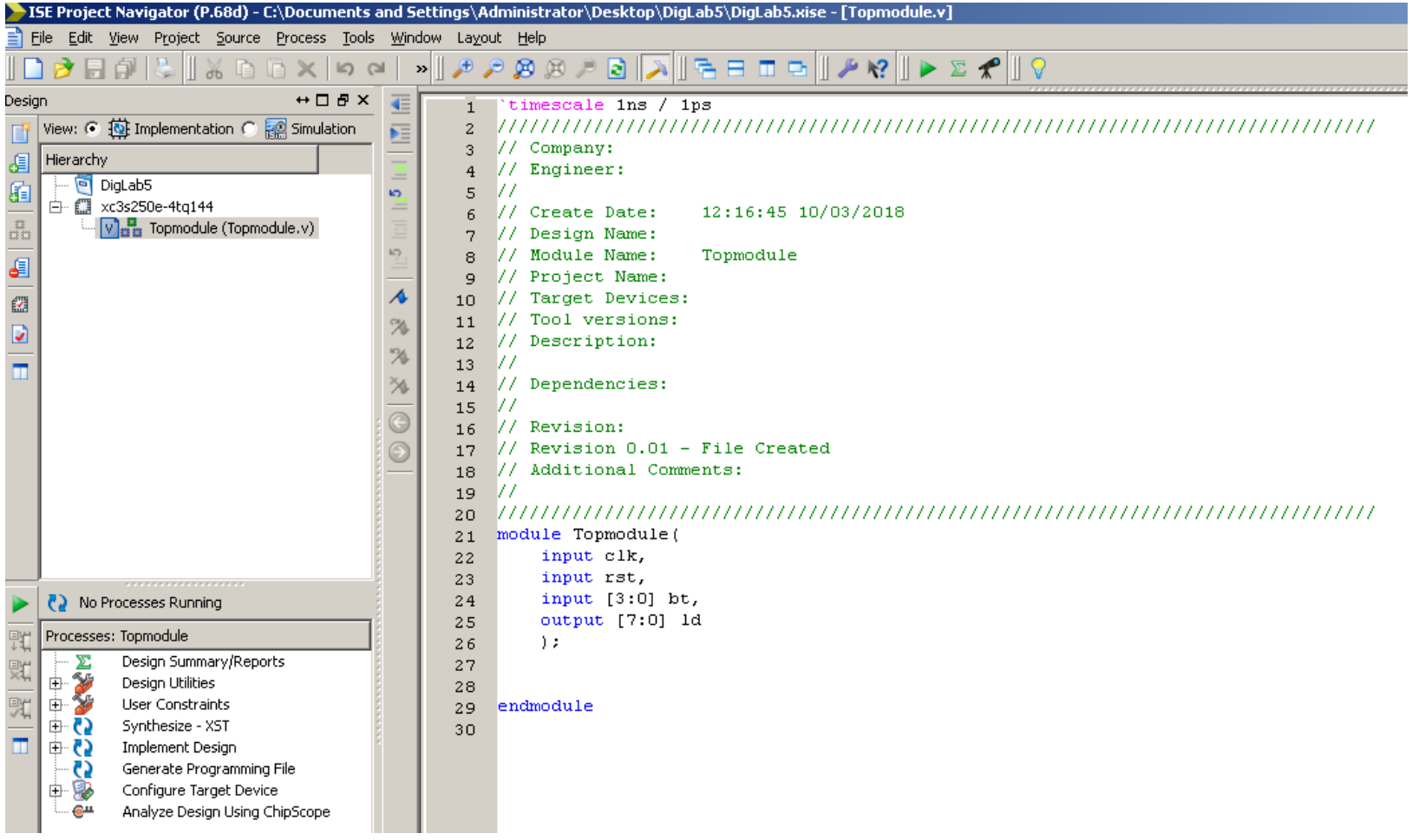
Add module to the project

- Click Finish



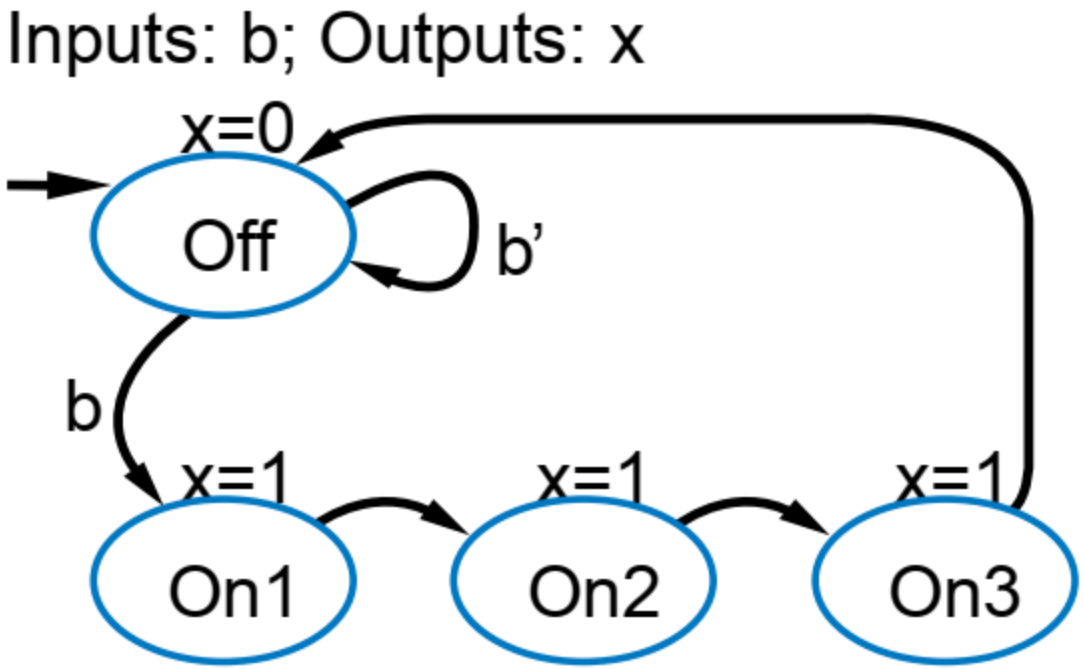
Adding a module to the project

- You should see this:



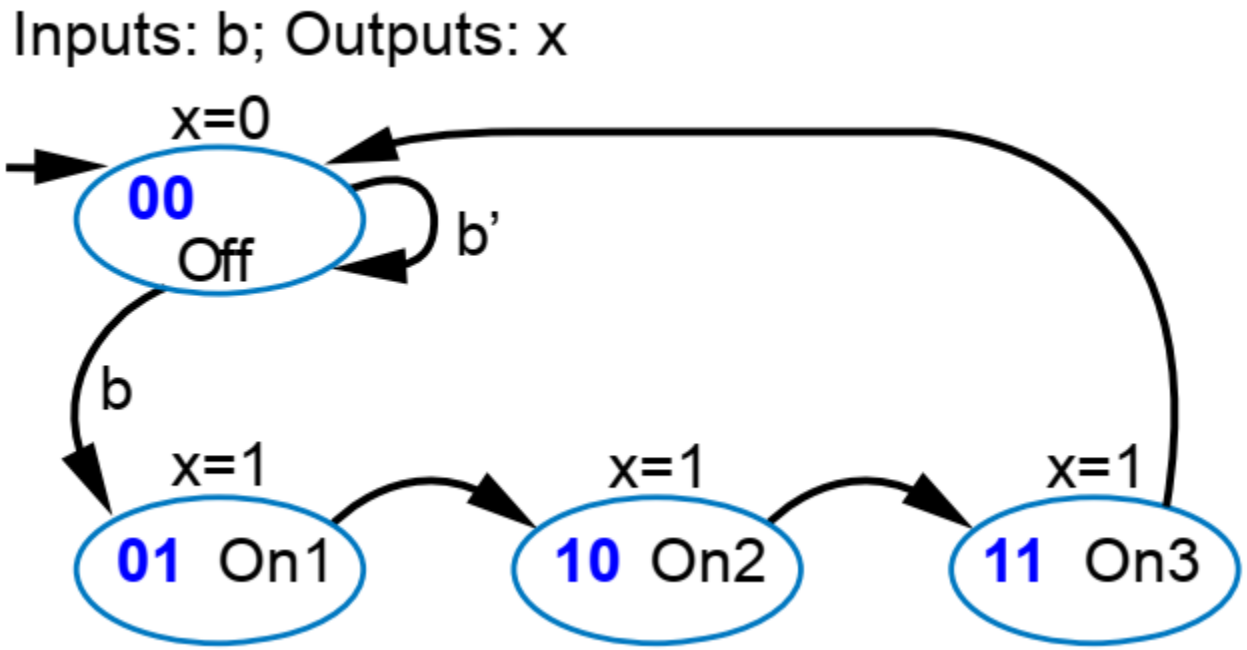
Task 1 - Design

- We are going to implement the 1 cycle low, 3 cycles high system first
- The state diagram was the following:



Task 1 - Design

- After encoding the states, the graph was the following:



Task 1 - Design

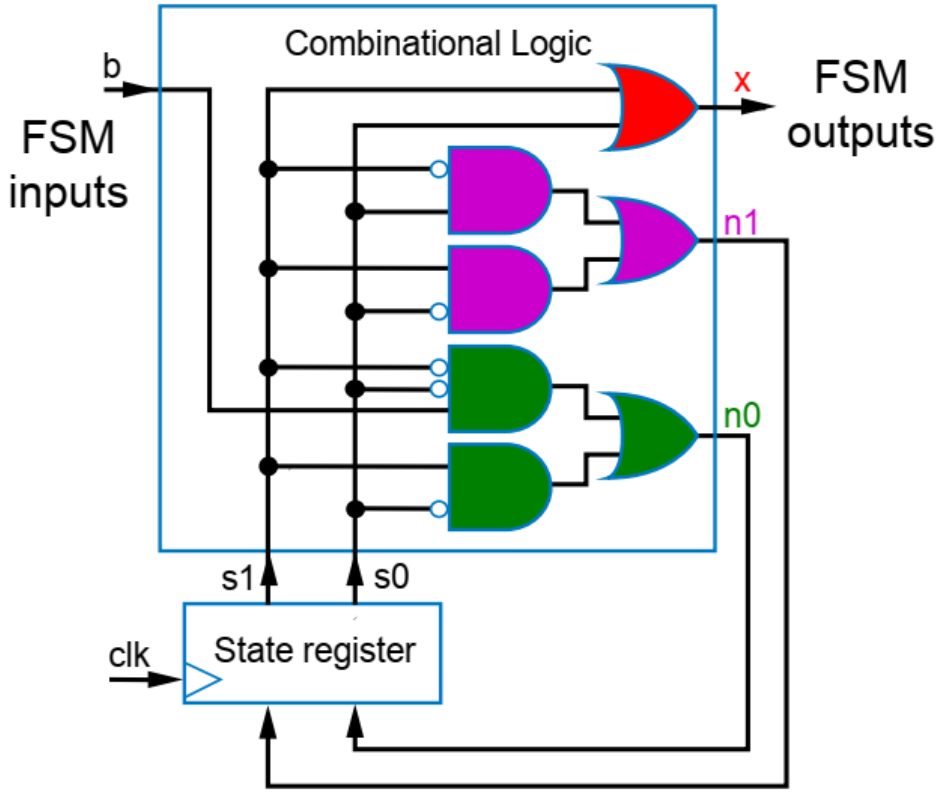
- State table, Boolean equations and the circuit:

	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0

$x = s1 + s0$

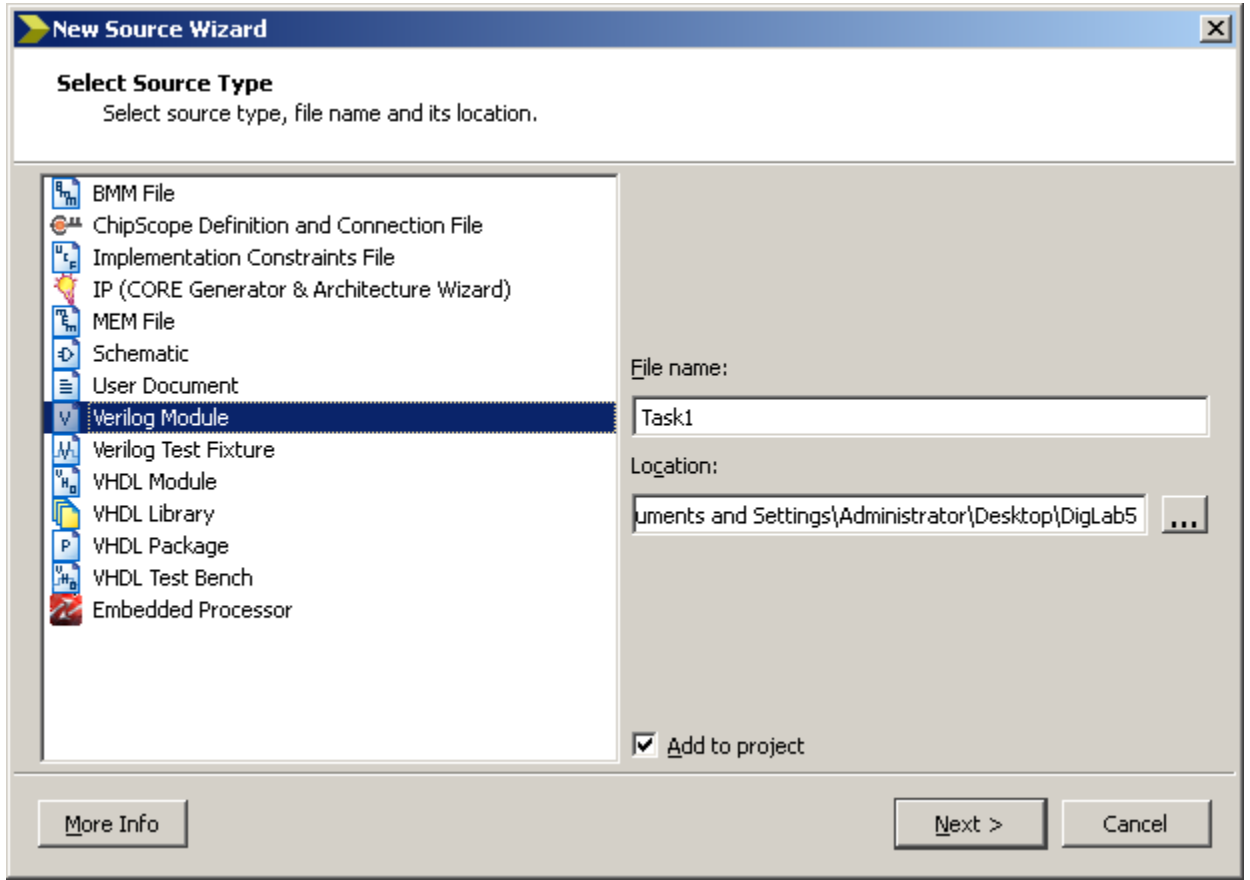
$n1 = s1's0 + s1s0'$

$n0 = s1's0'b + s1s0'$



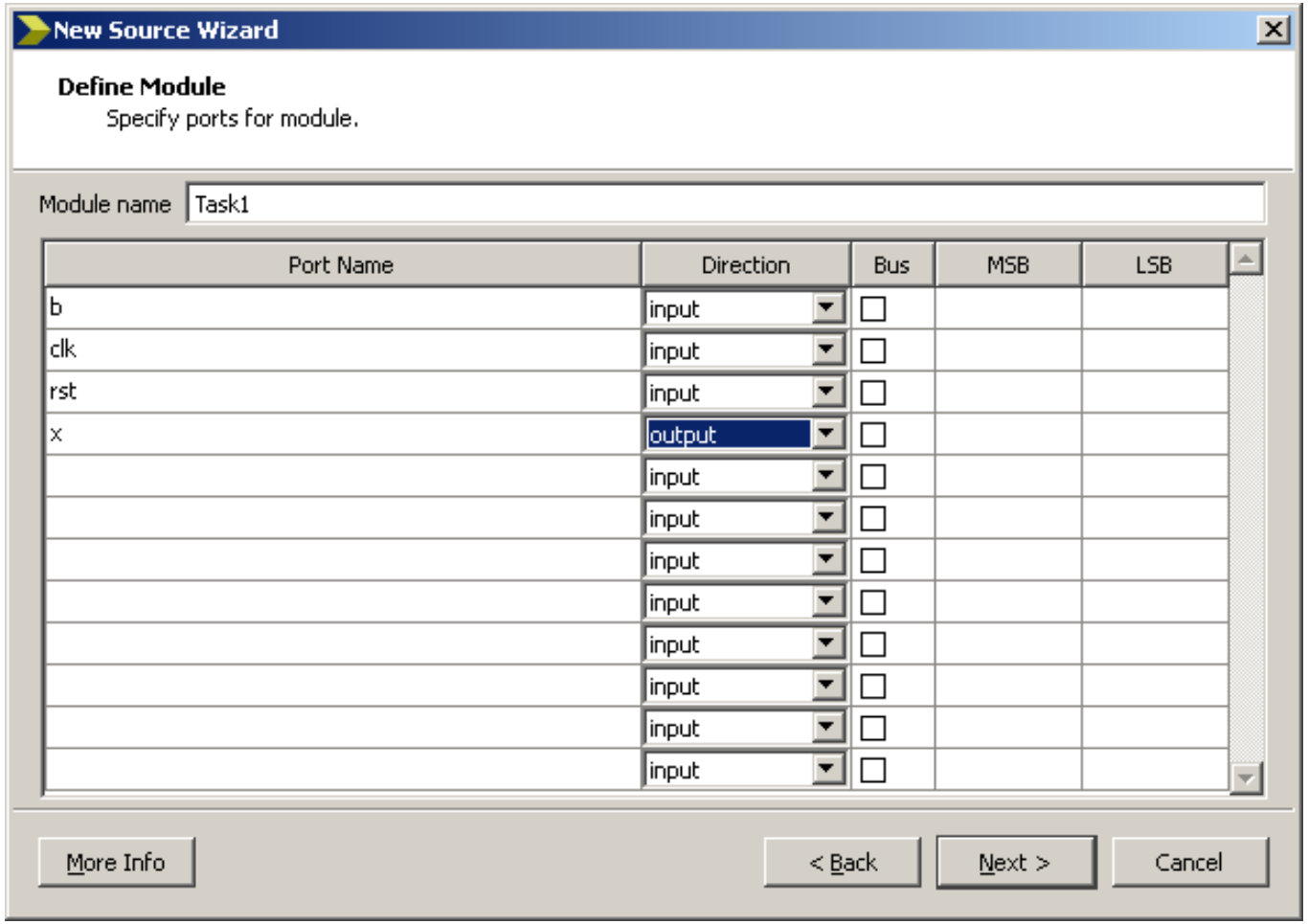
Task 1 - Implementation

- Add another module to the project following the previous steps.
  Name: Task1



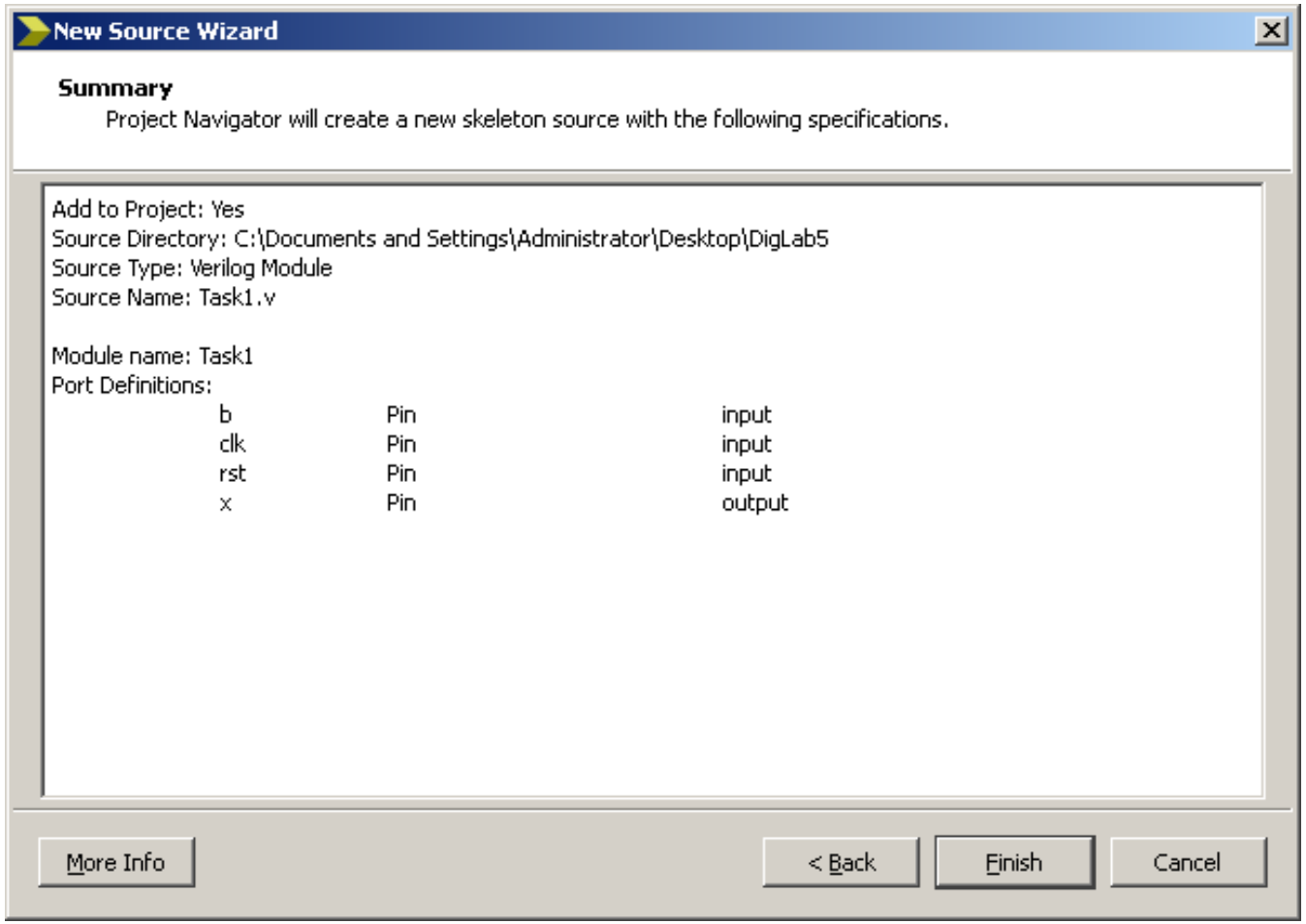
Task 1 - Implementation

- Add the following inputs and output, then click Next:



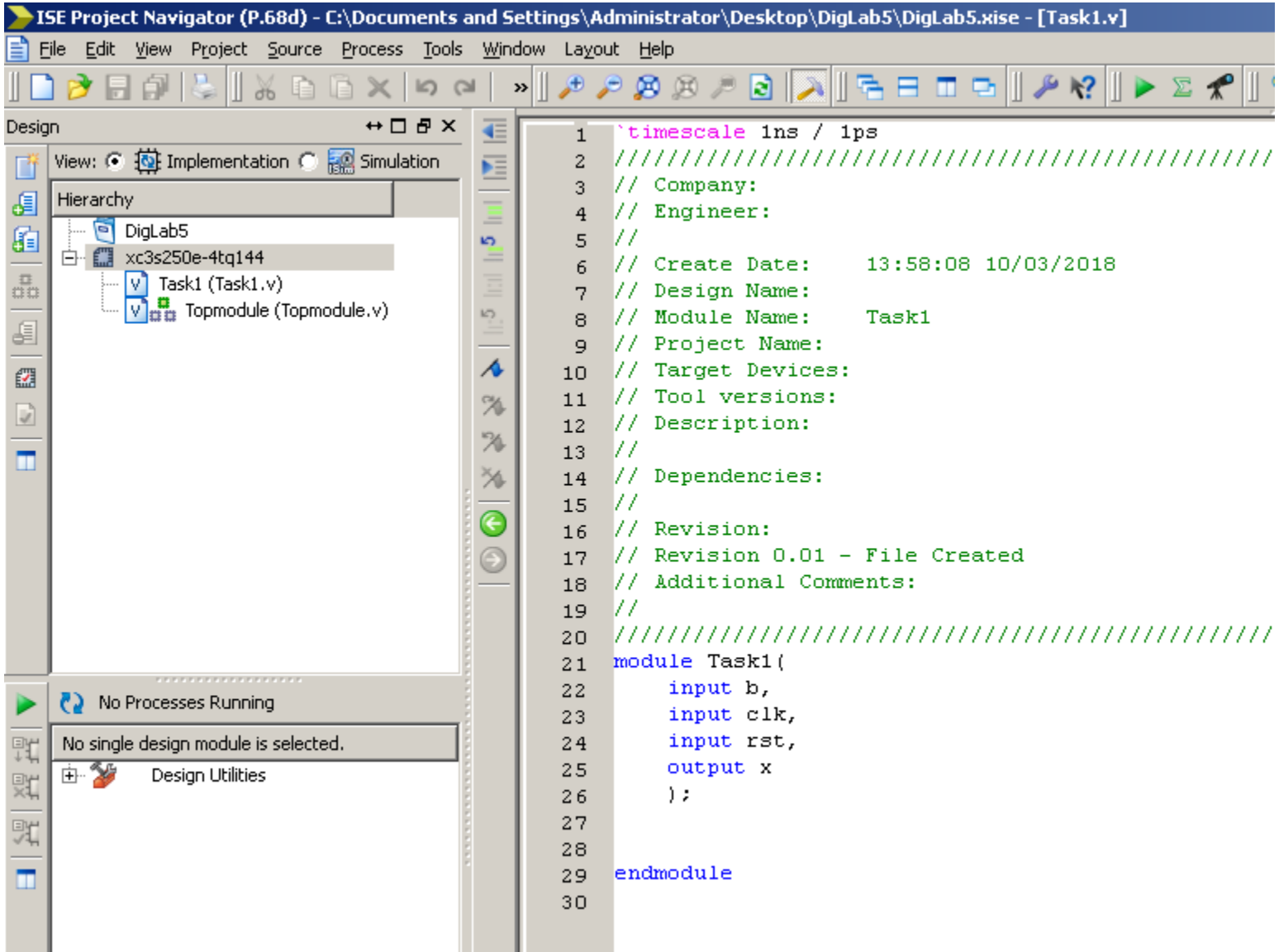
Task 1 - Implementation

- Click Finish



Task 1 - Implementation

- You should see this:



Task 1 – Implementation

- We start the implementation with the state register. The reset signal will be added to the implementation: after reset, we force the circuit to go into the Off (encoded 00) state.
- State register implementation:
- state and next_state are 2 bit registers
- State is the same as s1s0
- Next state is the same as n1n0
- If the reset input is high at the rising edge of the clock, we go to the Off (00) state. Else we go into the next state.

```
reg [1:0] state;
wire [1:0] next_state;

always @ (posedge clk)
if (rst) state <= 2'b00;
else state <= next_state;
```

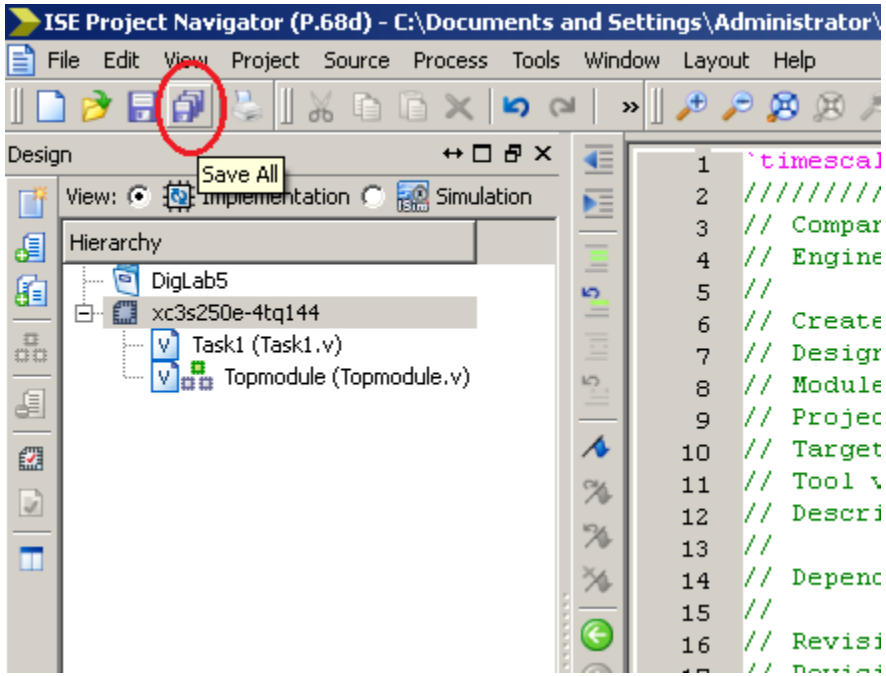
Task 1 – Implementation

- Next we implement the combinational logic . The combinational logic provides the output signal and the bits of next_state.
- Add the highlighted lines:
- Remember!
  Verilog operators:
  AND: &
  OR: |
  NOT: ~

```
28    reg [1:0] state;
29    wire [1:0] next_state;
30
31    always @ (posedge clk)
32    if (rst) state <= 2'b00;
33    else state <= next_state;
34
35    assign x = state[0] | state[1];
36    assign next_state[0] = ~state[1]&~state[0]&b | state[1]&~state[0];
37    assign next_state[1] = ~state[1]&state[0] | state[1]&~state[0];
38
39  endmodule
```

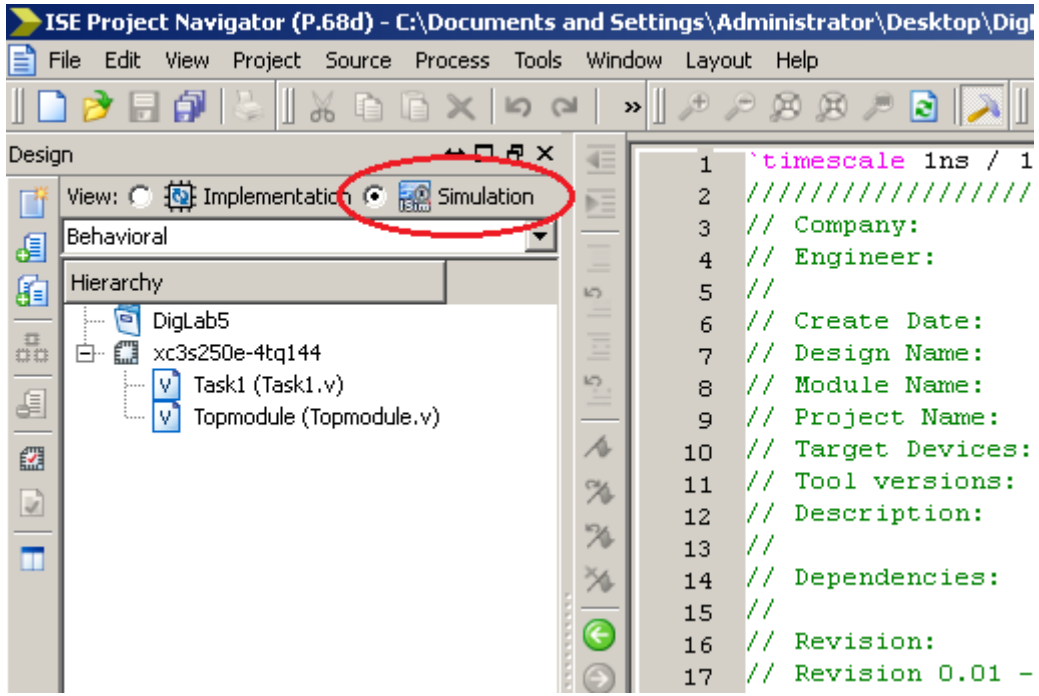
Task 1 - Simulation

- Verify the circuit with simulation.
- First of all: Click Save All



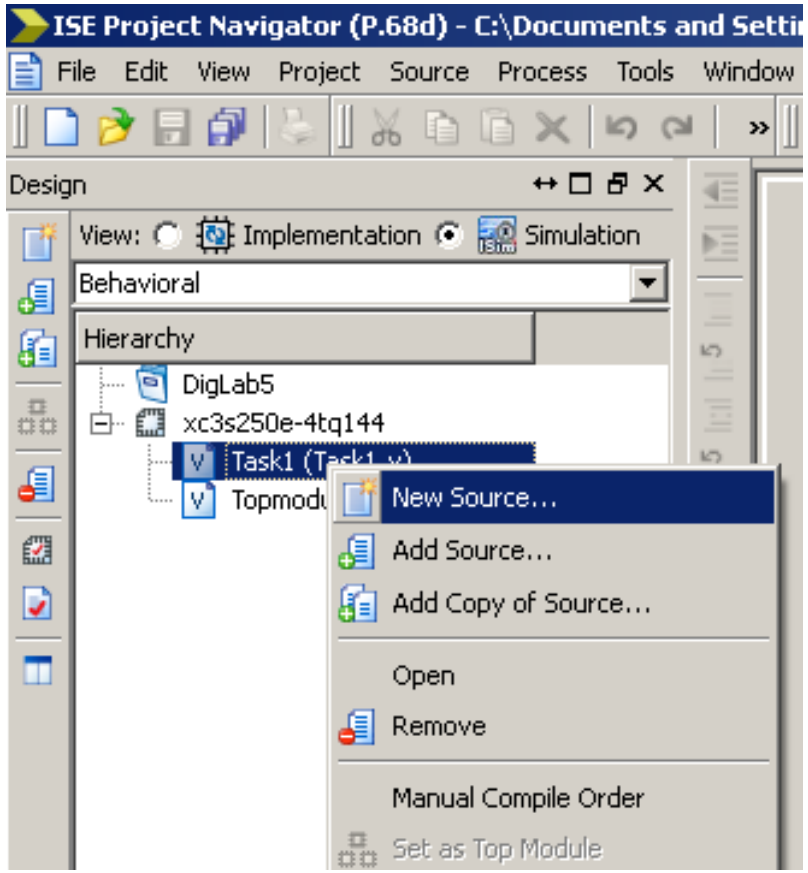
Task 1 - Simulation

- After saving, in the top left corner, switch to Simulation mode from Implementation



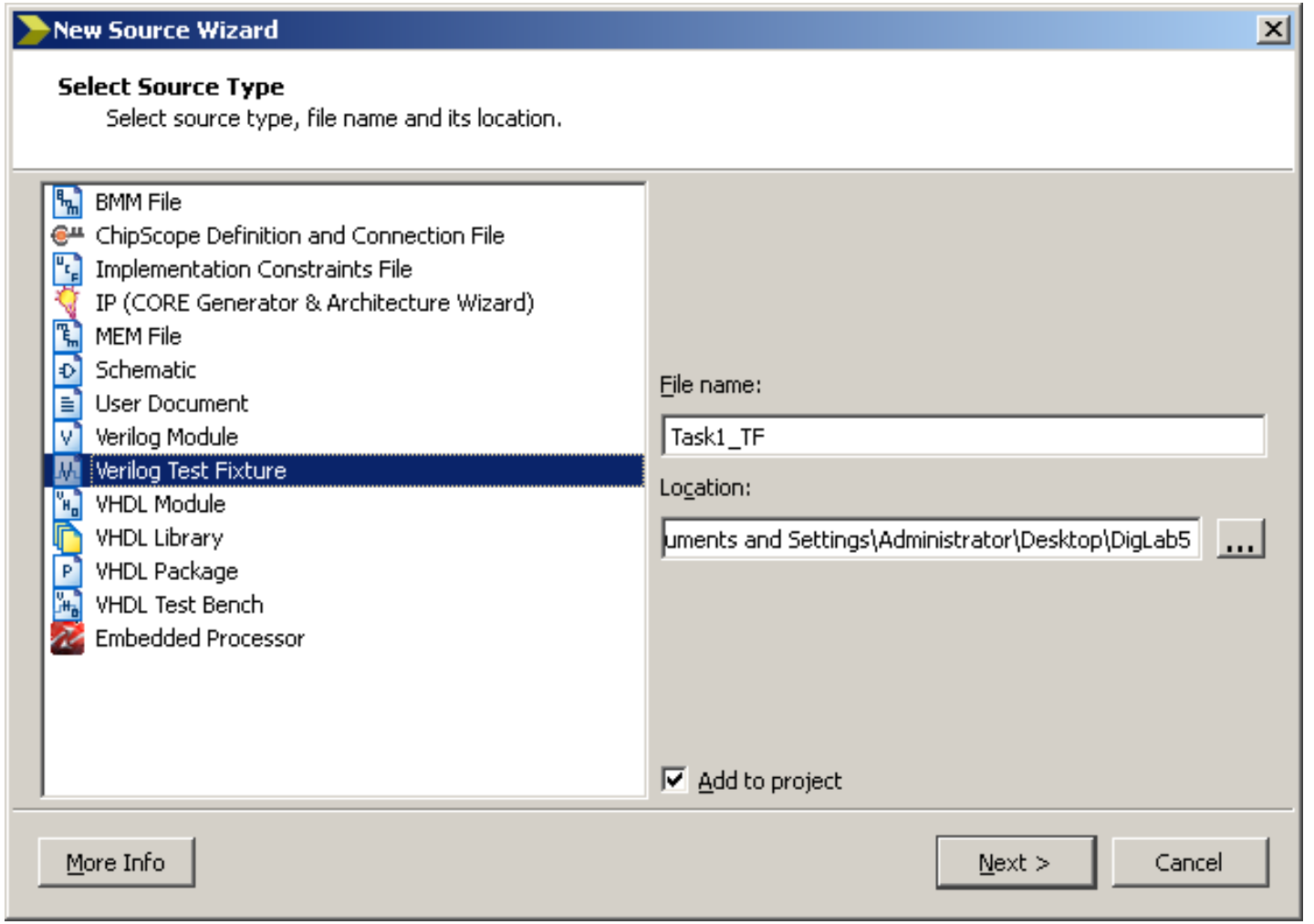
Task 1 - Simulation

- Right click on the Task1 module, and select New Source…



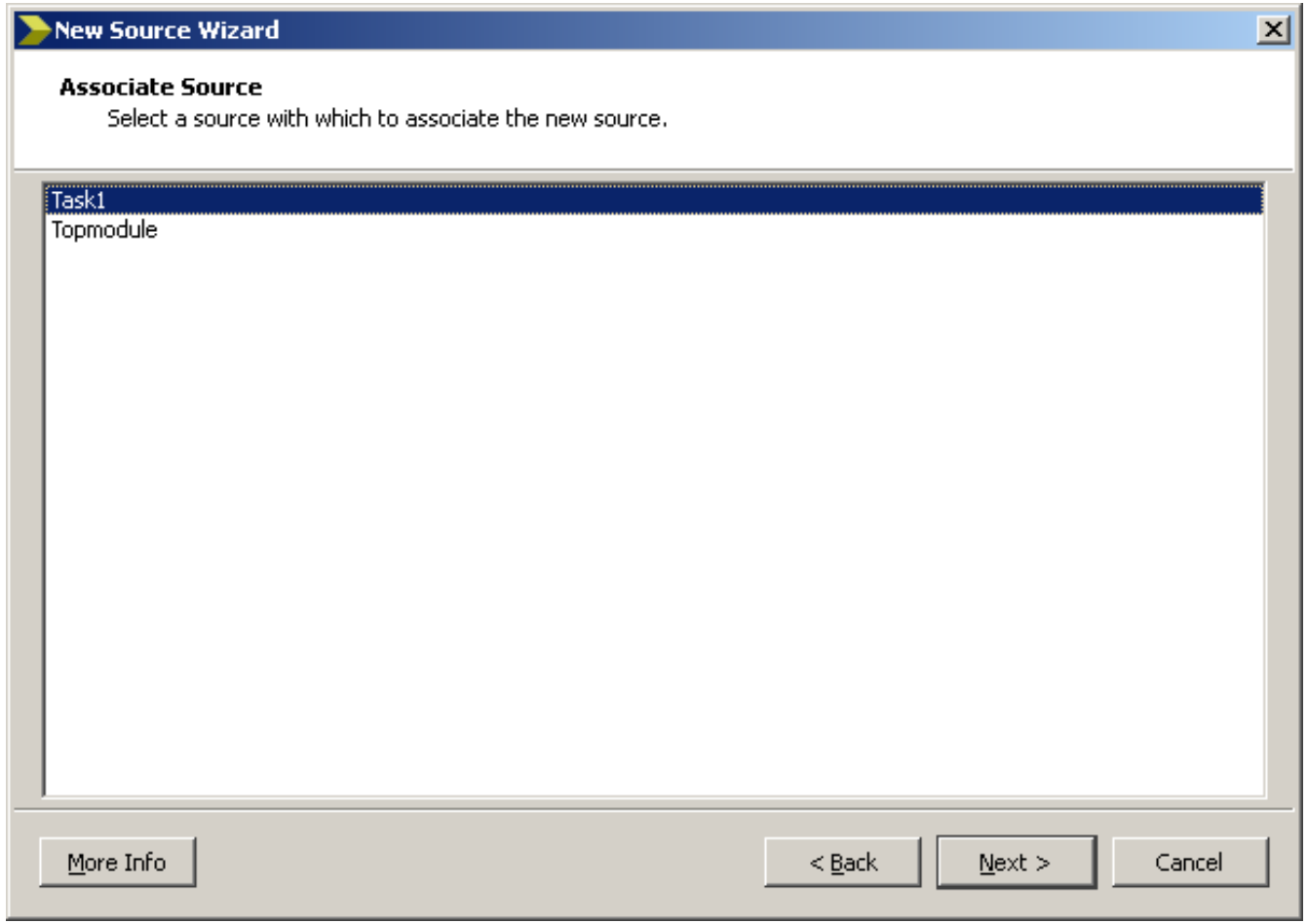
Task 1 - Simulation

- Select Verilog Test Fixture
- File Name: Task1_TF
- Don’t modify the Location!
- Click Next



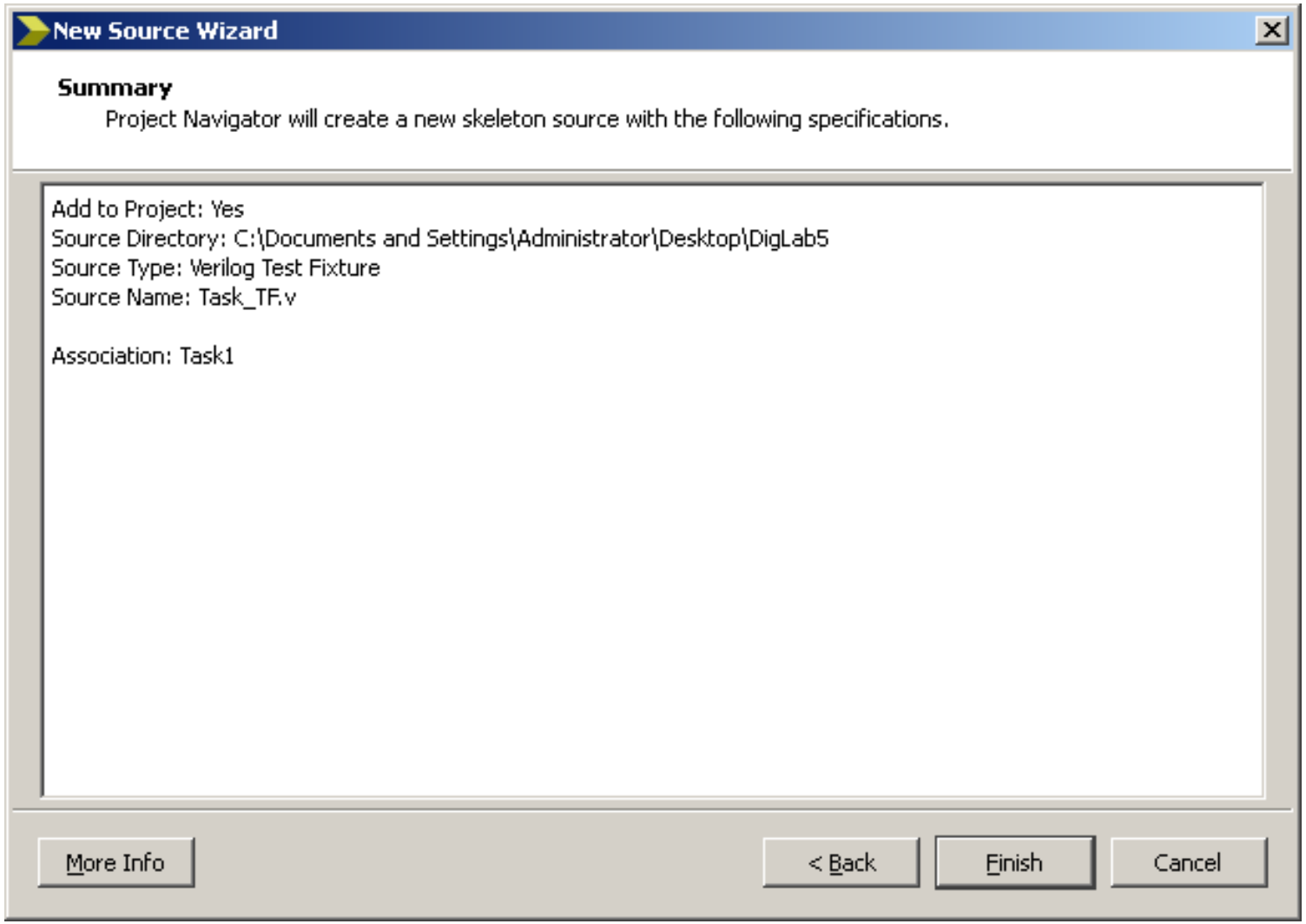
Task 1 - Simulation

- Make sure that the Task1 module is selected in the next window, then click Next



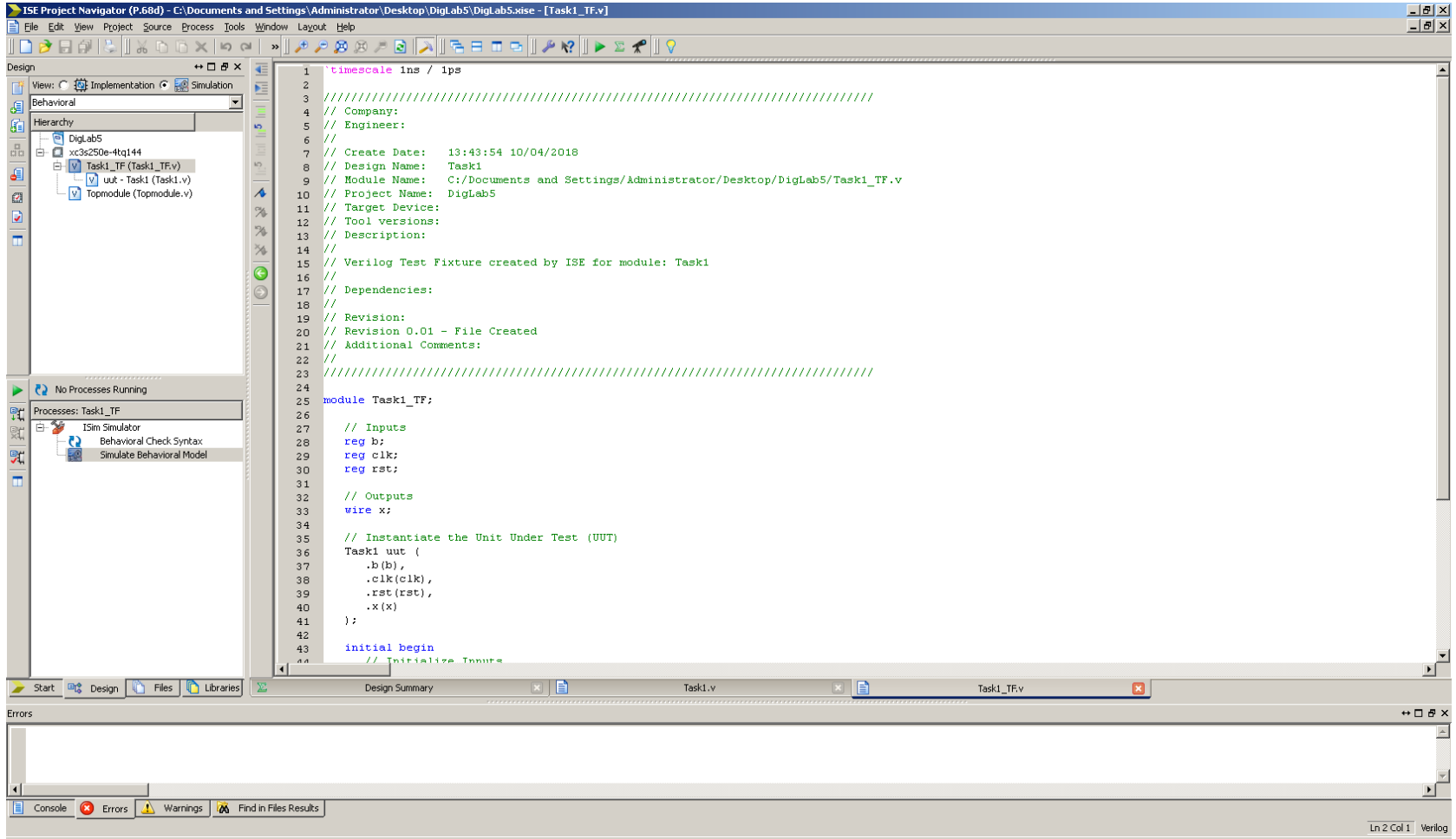
Task 1 – Simulation

- Click Finish



Task 1 - Simulation

- The software creates the Test Fixture file. You should see this:



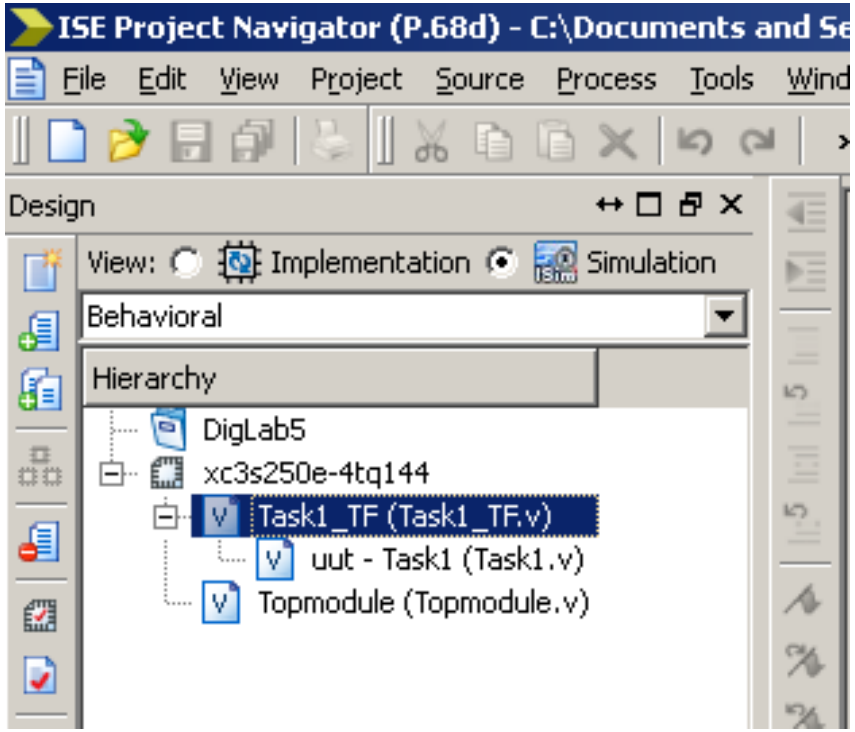
Task 1 - Simulation

- Go to the bottom of the code. We will add a reset input, then after the reset, we will set the input to b=1, and we will provide a CLK also for the circuit. Go to the “Add stimulus here” part. Insert the always #50 part AFTER the end statement!

```
49        // Wait 100 ns for global reset
50        #100;
51
52        // Add stimulus here
53        #100 rst = 1;
54        #100 rst = 0;
55        #100 b = 1;
56        #100 b = 0;
57
58    end
59
60        always #50 clk = ~clk;
61
```

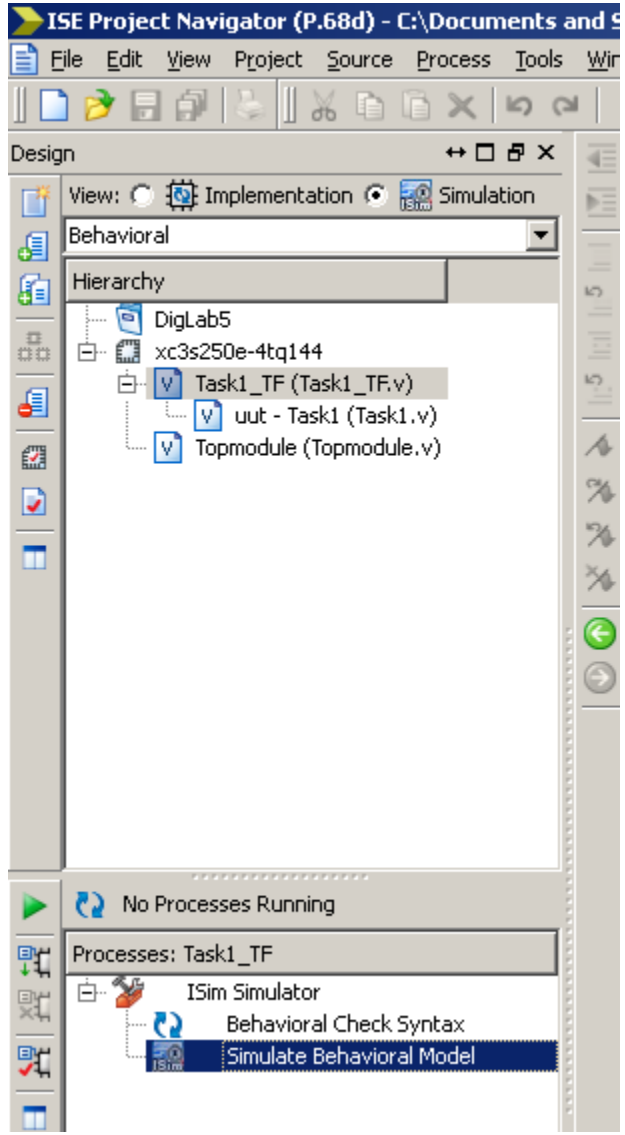
Task 1 - Simulation

- Save All again
- Then make sure that the Task1_TF file is selected in the top left corner:



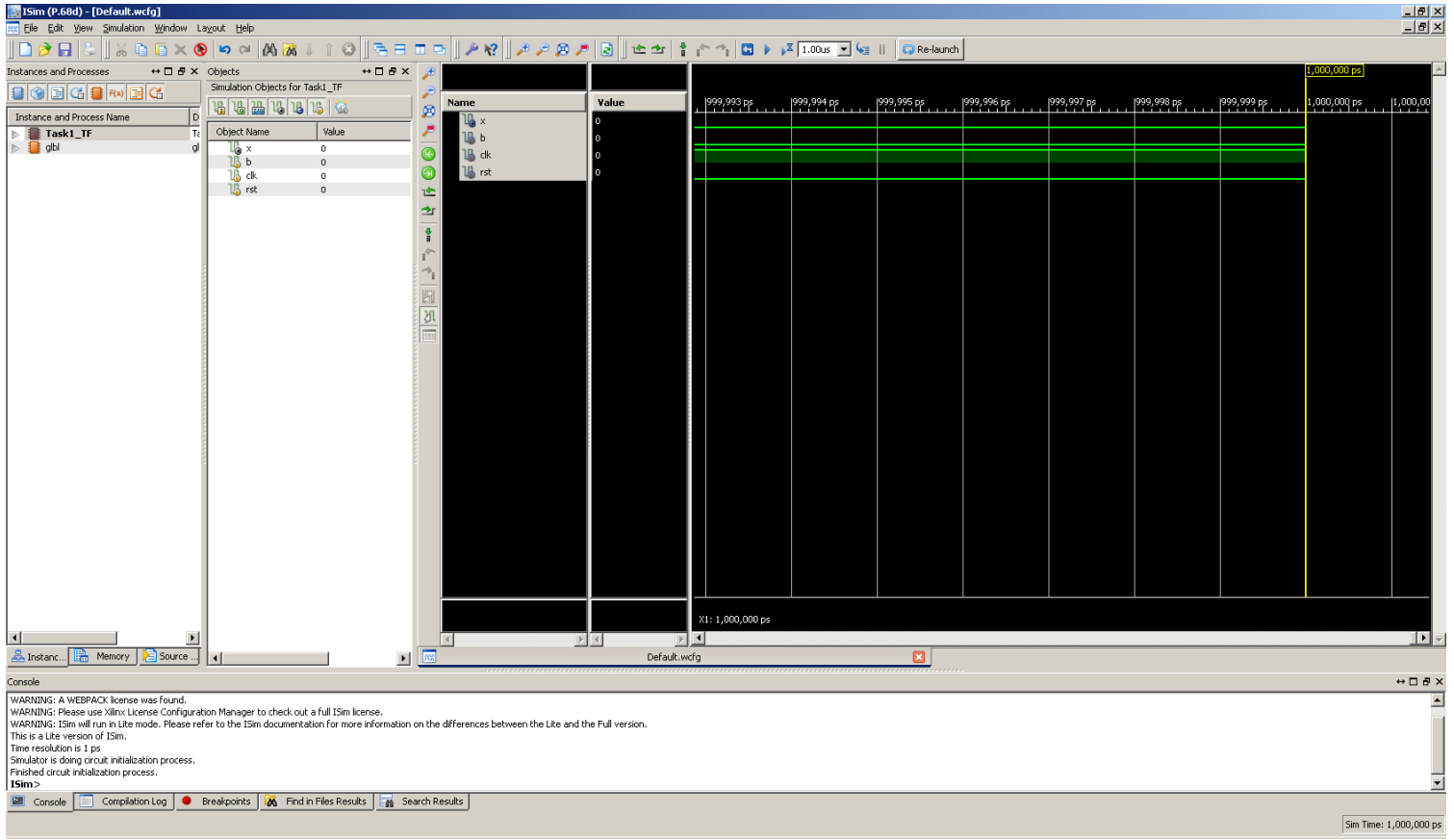
Task 1 - Simulation

- Go down to “Processes: Task_TF”
- Click on the plus sign on the left of ISIM Simulator
- Then double click on Simulate Behavioral Model (Or right click on Simulate Behavioral Model, and select Run from the Menu)



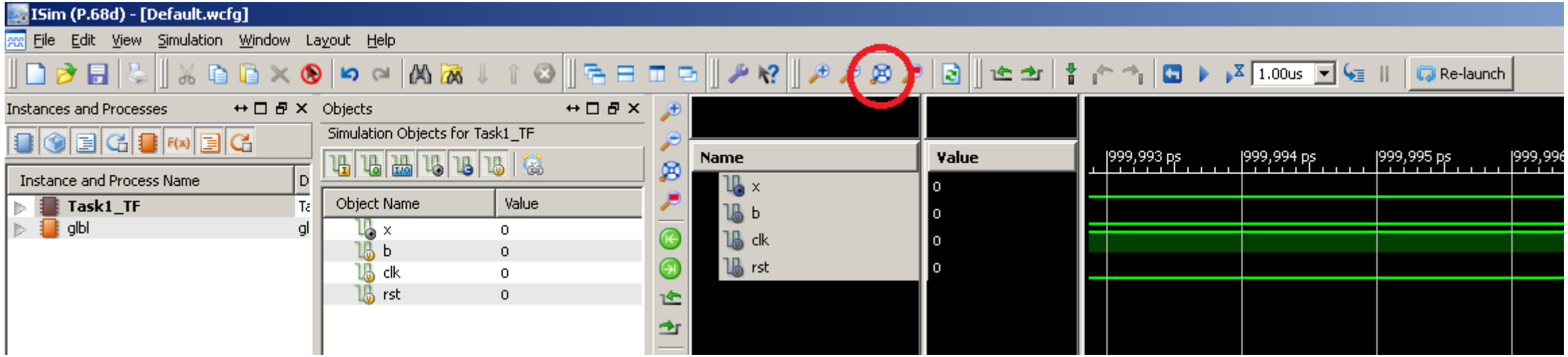
Task 1 - Simulation

- The simulator launches. You should see this:



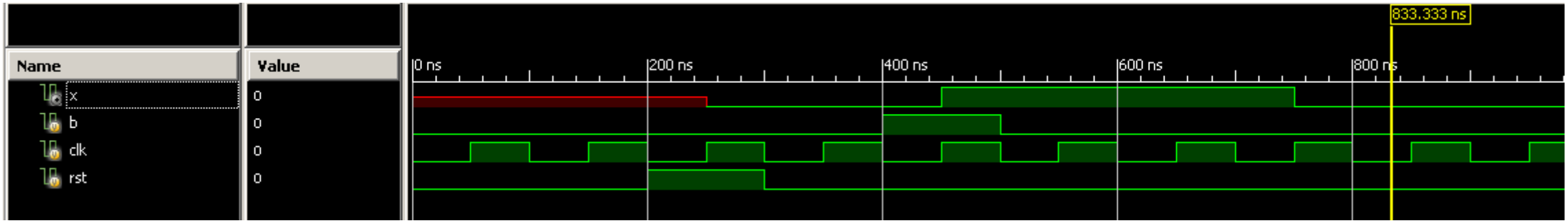
Task 1 - Simulation

- On the top menu, select the third magnifier (Zoom to full View)



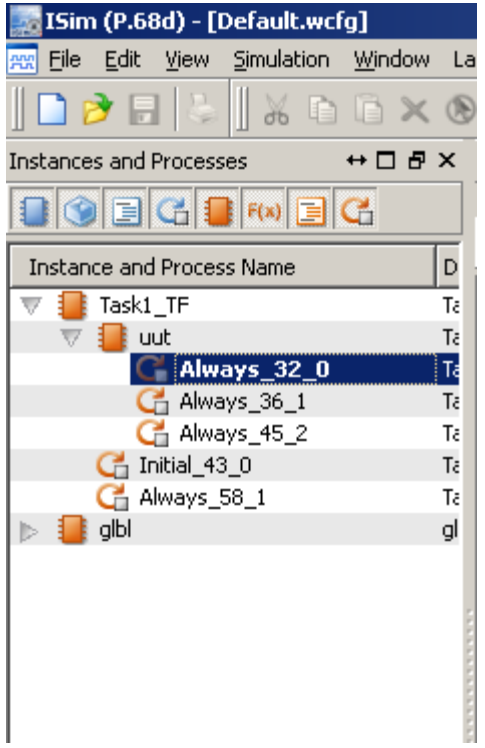
Task 1 - Simulation

- First the value of the x output is undefined (red line). After rst=1 and the first rising edge of the CLK, the output is set to 0 (because we go into state 00=OFF). Then, after the first rising edge when b=1, the x output will be 1 for exactly 3 clock cycles.



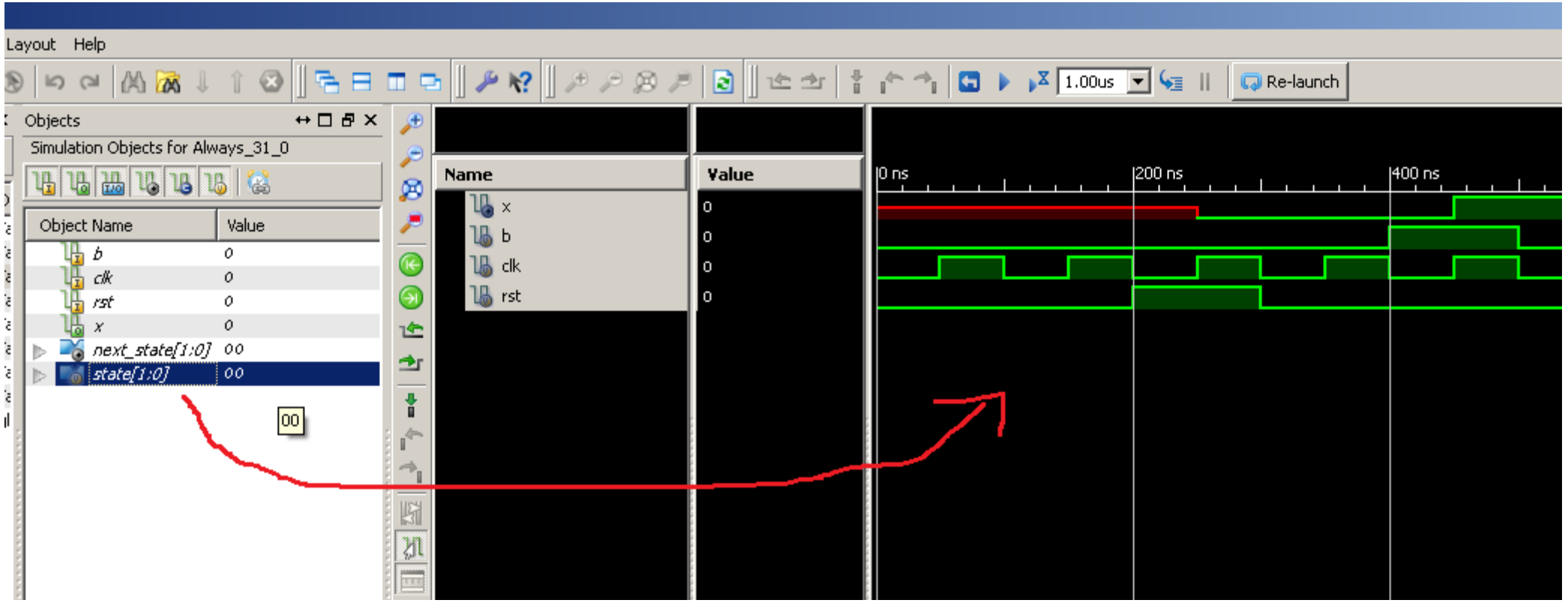
Task 1 - Simulation

- You can plot the waveform of the state register also.
- On the left menu (Instance and Process Name) click on the triangle left to Task_TF, then click on the triangle left to uut. Then select Always_32_0
  (or maybe it's called Always_31_0, etc).



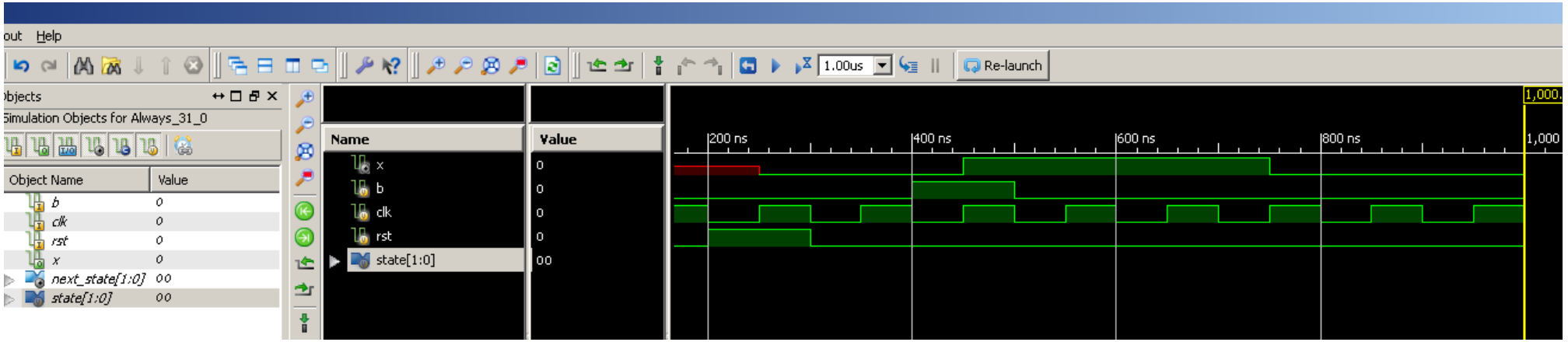
Task 1 - Simulation

- The state and next_state variables appear in the right menu.
- Click on state[1:0], simply drag it and drop it into the waveform's area.



Task 1 - Simulation

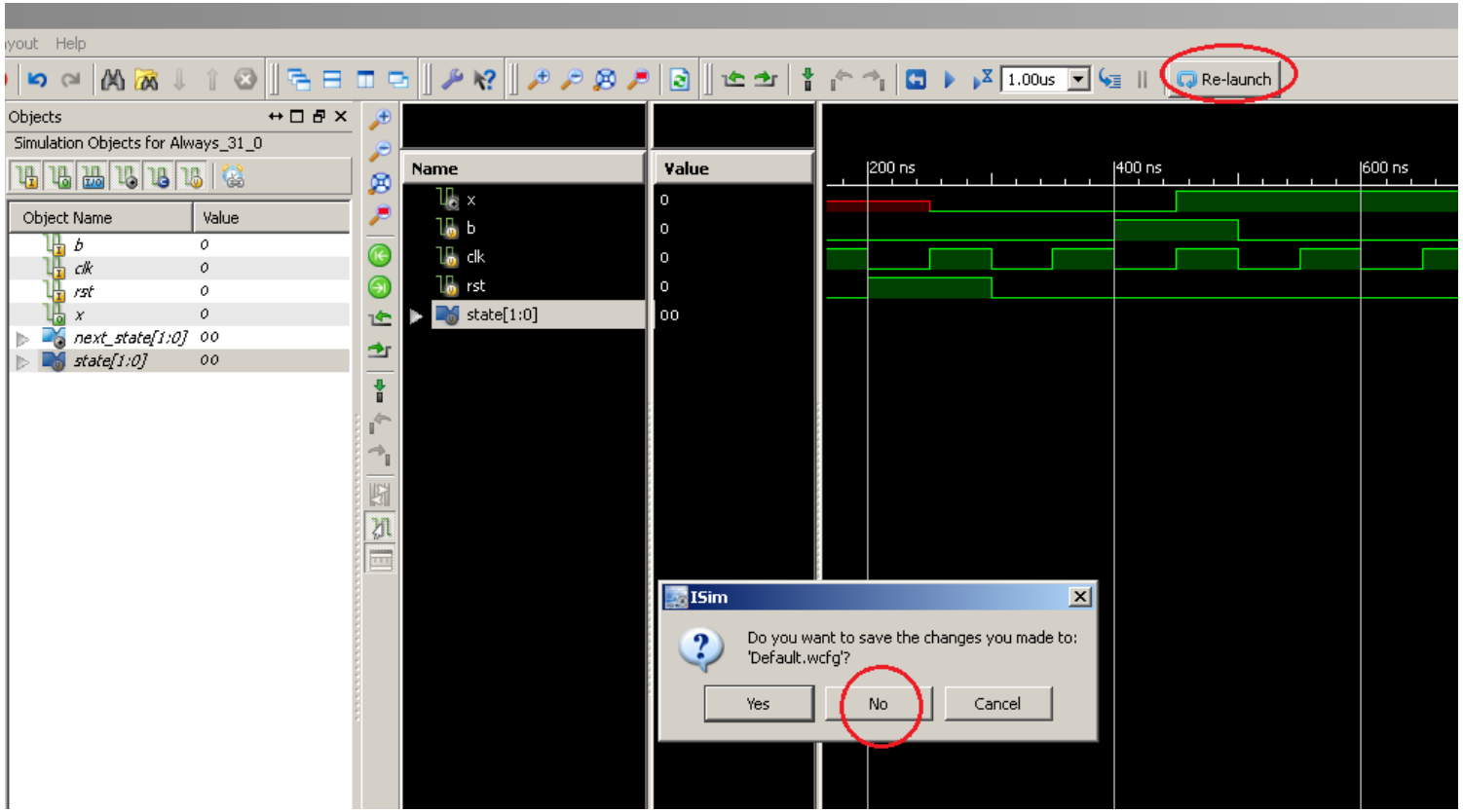
- After dropping it, it should appear among the other signals:



- However, the waveform is missing

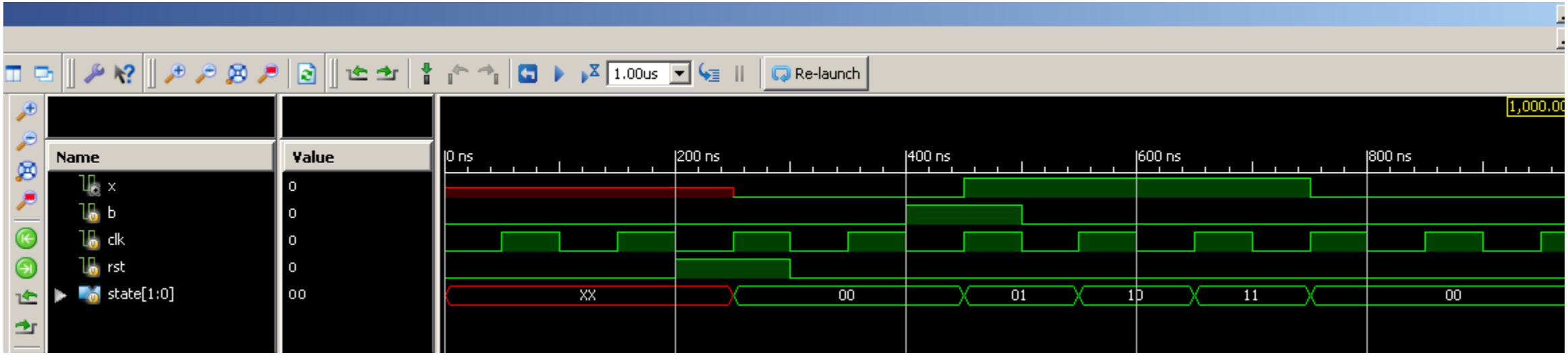
Task 1 - Simulation

- Press the Re-launch button. A popup window asks whether save changes or not. Select No.



Task 1 - Simulation

- Press the Zoom to full view button again (3rd magnifier button on the top). Now you can see the states also. Notice that every change is synchronized to the rising edge of the clock. Study the output, check the order of the states. Is the behavior of the FSM correct?

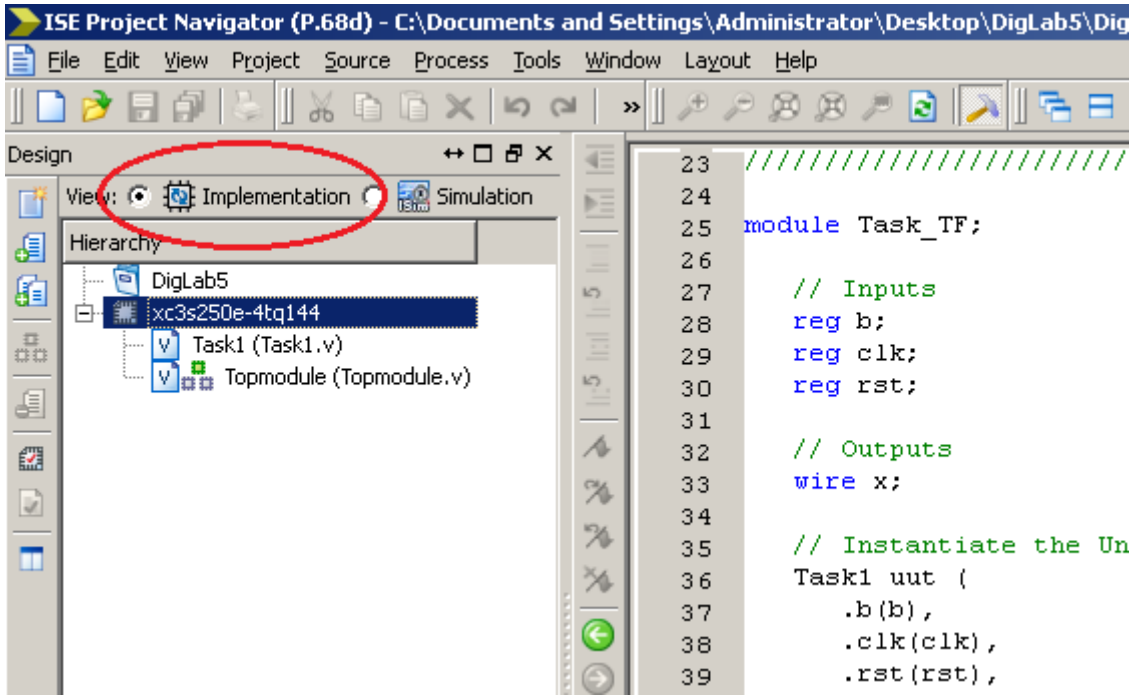


Task 1 – Update implementation

- We defined the behavior of the combinational circuit using Boolean equations. That’s fine.
- However, the readability of the code can be improved by the application of the case command.
- First, close the ISim simulator (File -> Exit). A popup window might appear. Don’t save the changes.

Task 1 – Update implementation

- In the top left corner of the ISE Project Navigator, switch back to Implementation mode.



Task 1 – Update implementation

- Select the Task1.v file.
- Delete the written code, except for the following lines:

```
20  //////////////////////////////////////////////////////////////////////////////////
21  module Task1(
22      input b,
23      input clk,
24      input rst,
25      output x
26      );
27
28     reg [1:0] state;
29
30     always @ (posedge clk)
31     if (rst) state <= 2'b00;
32     else state <= next_state;
33
34  endmodule
35
```

Task 1 – Update implementation

- Add/modify the highlighted lines:

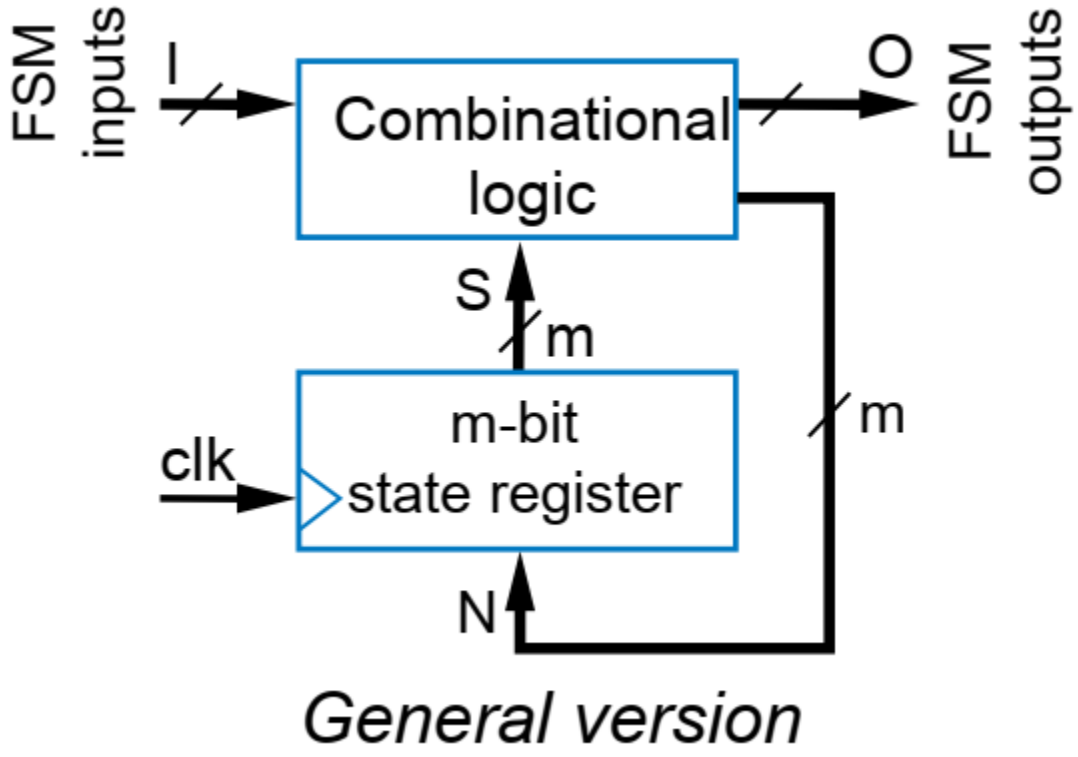
```
20  //////////////////////////////////
21  module Task1(
22      input b,
23      input clk,
24      input rst,
25      output reg x
26      );
27
28      reg [1:0] state;
29      reg [1:0] next state;
30
31      always @ (posedge clk)
32      if (rst) state <= 2'b00;
33      else state <= next state;
```

Task 1 – Update implementation

- Values of **reg** variables have to be set inside an always block.
- To set the value of a **reg** type variable, **use <= instead of =** (see later).
- The commands inside an always block are evaluated once a specified event occurs.
- The event is specified after the @ part in the brackets, this is the so-called sensitivity list: always @ (... Sensitivity list …)
- Example 1: always @ (posedge clk)
  In this case the block is evaluated after the rising edge of the clk. **This is used to describe the state register.**
- Example 2: always @ (*)
  The always block is evaluated if ANY of the inputs changes. **This is used to describe combinational logic.**

Task 1 – Update implementation

- Remember the general design of FSM implementations:
- The combinational logic part is described with an always @ (*) block
- The state register is described with an always @ (posedge clk) block



General version

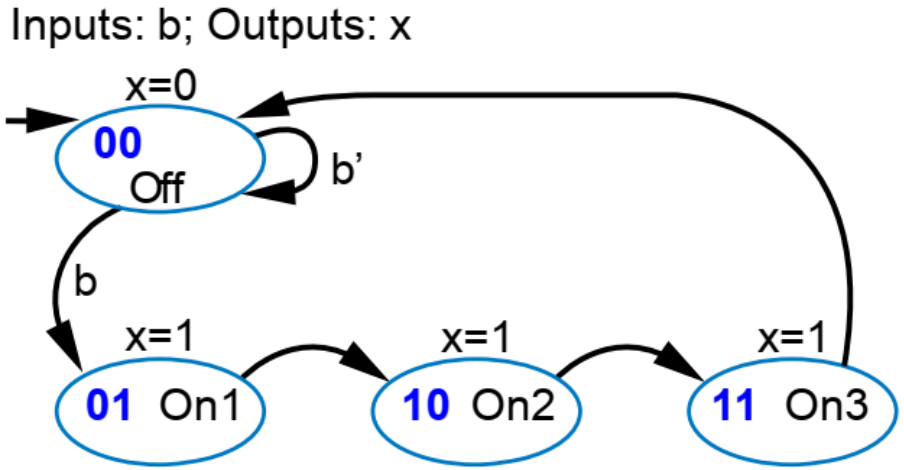
Task 1 – Update implementation

- The combinational logic provides the next_state signal and the ouput of the FSM (check previous figure).
- First we implement the generation of the next_state signal.
- We can use the **case … endcase** commands inside the always block.
- This improves the readability of the code.

Task 1 – Update implementation

- Add the following code. Notice that it implements the next states in the correct order, and the conditions (b and b') are also evaluated!
- Note that we use **<=** instead of **=** !

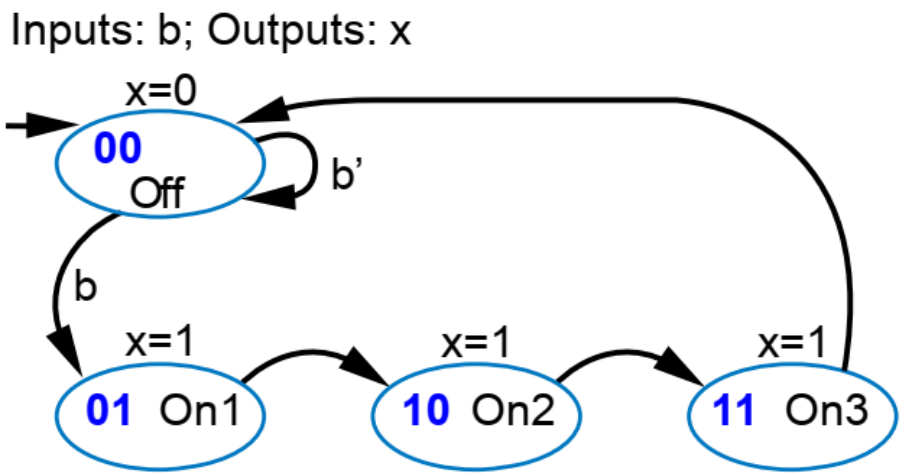
```
31
32    always @ (posedge clk)
33    if (rst) state <= 2'b00;
34    else state <= next_state;
35
36    always @ (*)
37    case (state)
38    2'b00:    if (b) next_state <= 2'b01;
39              else next_state <= 2'b00;
40    2'b01:    next_state <= 2'b10;
41    2'b10:    next_state <= 2'b11;
42    2'b11:    next_state <= 2'b00;
43    endcase
44
45  endmodule
46
```



Task 1 – Update Implementation

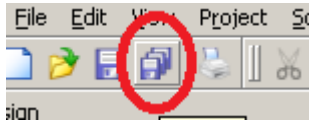
- Now implement the output function, using **case…endcase**.

```
35
36      always @ (*)
37      case (state)
38      2'b00:    if (b) next_state <= 2'b01;
39                else next_state <= 2'b00;
40      2'b01:    next_state <= 2'b10;
41      2'b10:    next_state <= 2'b11;
42      2'b11:    next_state <= 2'b00;
43      endcase
44
45      always @ (*)
46      case (state)
47      2'b00:    x <= 1'b0;
48      2'b01:    x <= 1'b1;
49      2'b10:    x <= 1'b1;
50      2'b11:    x <= 1'b1;
51      endcase
52
53  endmodule
```

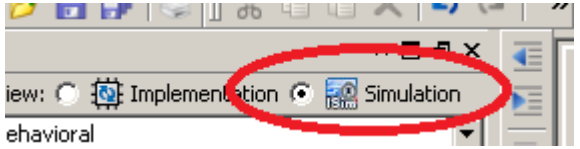


Task 1 – Simulation II

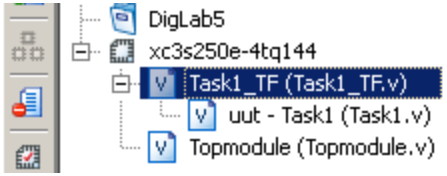
- Simulate the behavior of the modified circuit.
- Save all changes.



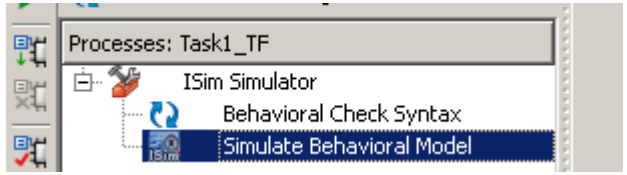
- Switch to Simulation mode on the top left corner.



- Select the Task1_TF file.



- Run the simulation: “Simulate Behavioral Model”



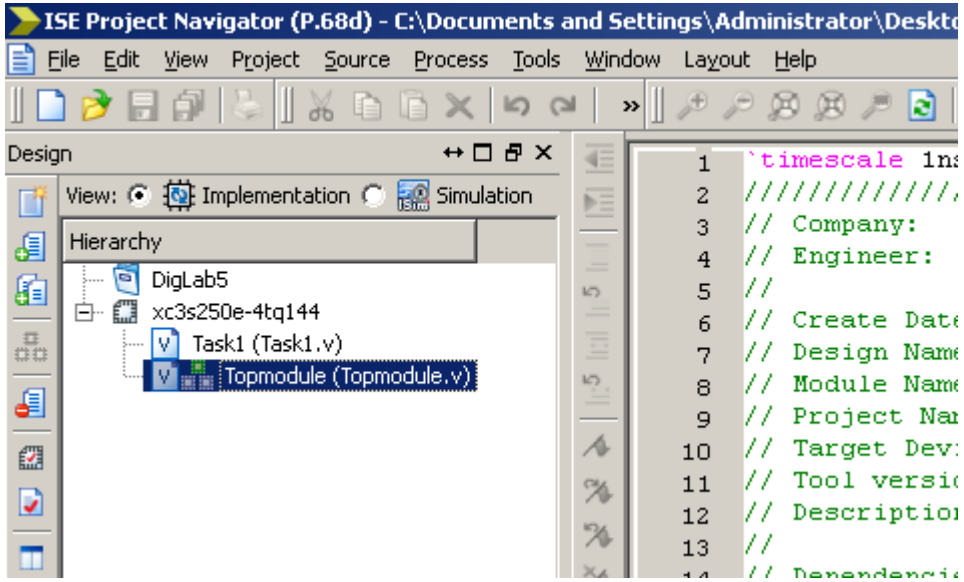
- Add the state register’s value to the waveforms (slides 33-37)

Task 1 – Simulation II

- Study the waveforms. Did the behavior of the FSM change after the modifications?

Task I – Generating programming file

- Now the FSM will be uploaded to the FPGA.
- First it has to be added to the top module. This is the so-called instantiation.
- Close the Isim simulator.
- Go back to implementation mode.
- Now select the top module file.



Task I – Generating programming file

- Instantiation means that we create an instance of the module, and connect its input(s) and output(s) to some wires of the top module.
- Add the following line to the topmodule:

```
21  module Topmodule(
22      input clk,
23      input rst,
24      input [3:0] bt,
25      output [7:0] ld
26      );
27
28      Task1 FSM1(.b(bt[0]), .clk(clk), .rst(rst), .x(ld[0]));
29
30  endmodule
31
```

Task I – Generating programming file

- The line can be interpreted as the following:
- Task1 FSM1(…): A Task1 type module is placed in the top module, and its name is FSM1
- .b(bt[0]): The bt[0] input of the Topmodule is connected to the b input of the FSM1 module.
- .clk(clk): The clk input of the Topmodule is connected to the clk input of the FSM1 module (two different wires, but with the same name!)
- .rst(rst): Similar to clk
- .x(ld[0]): The ld[0] output of the Topmodule is connected to the b output of the FSM1 module.

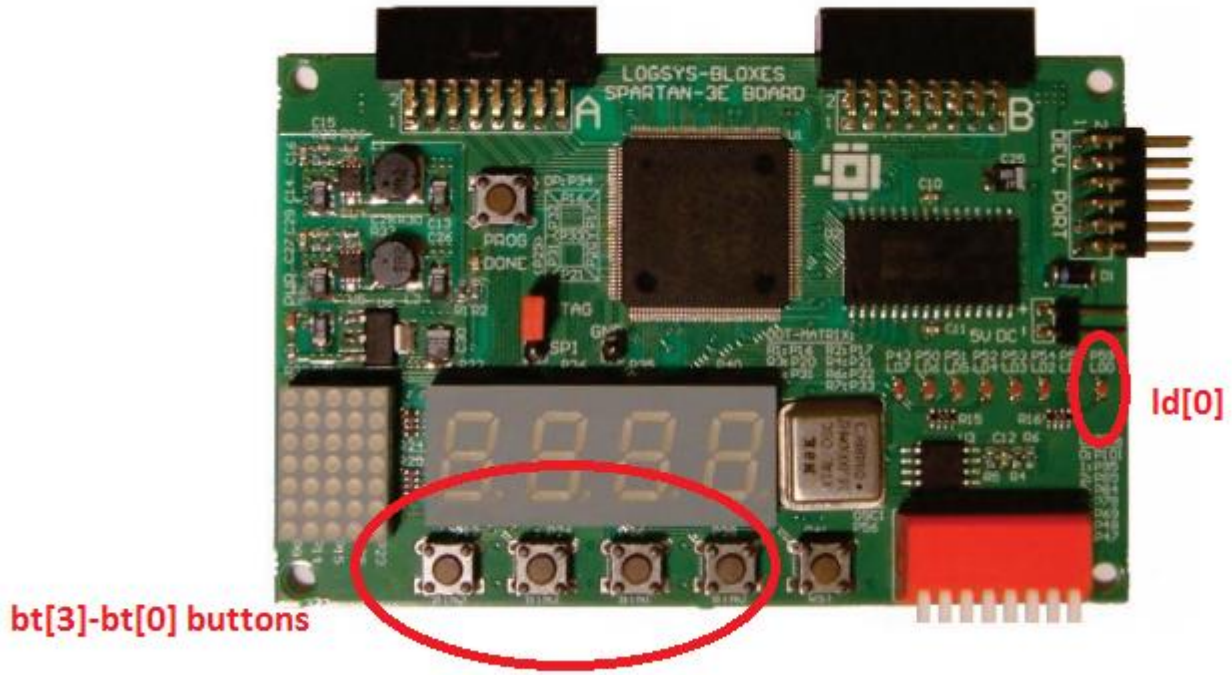
Task I – Generating programming file

- The bt and ld wires are multi-bit buses. However, we use only 1 bit of the input, and we do not define the value of the ld[7:1] outputs. This might lead to errors and warnings.
- To avoid this, modify the code in the following way:

```
21  module Topmodule(
22      input clk,
23      input rst,
24      input [3:0] bt,
25      output [7:0] ld
26      );
27
28      wire b;
29      assign b = bt[0]|bt[1]|bt[2]|bt[3];
30      assign ld [7:1] = 7'b0000000;
31
32     Task1 FSM1(.b(b)  .clk(clk), .rst(rst), .x(ld[0]));
33
34  endmodule
```

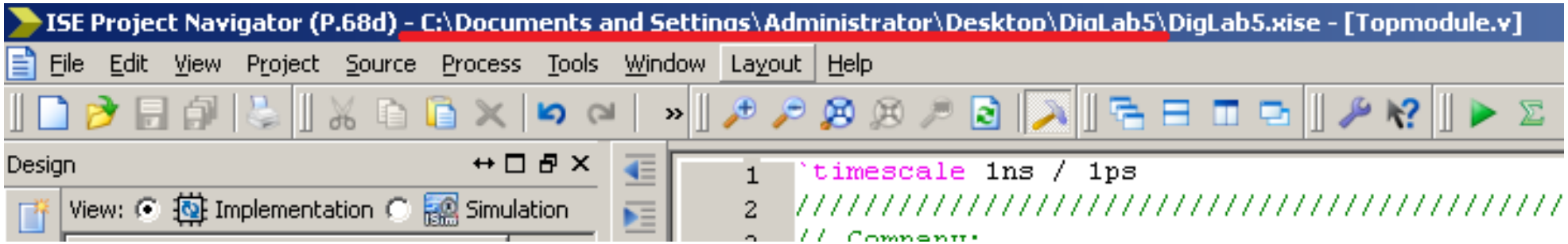
Task I – Generating programming file

- Now the ld[0] led will turn on if any of the buttons are pressed.



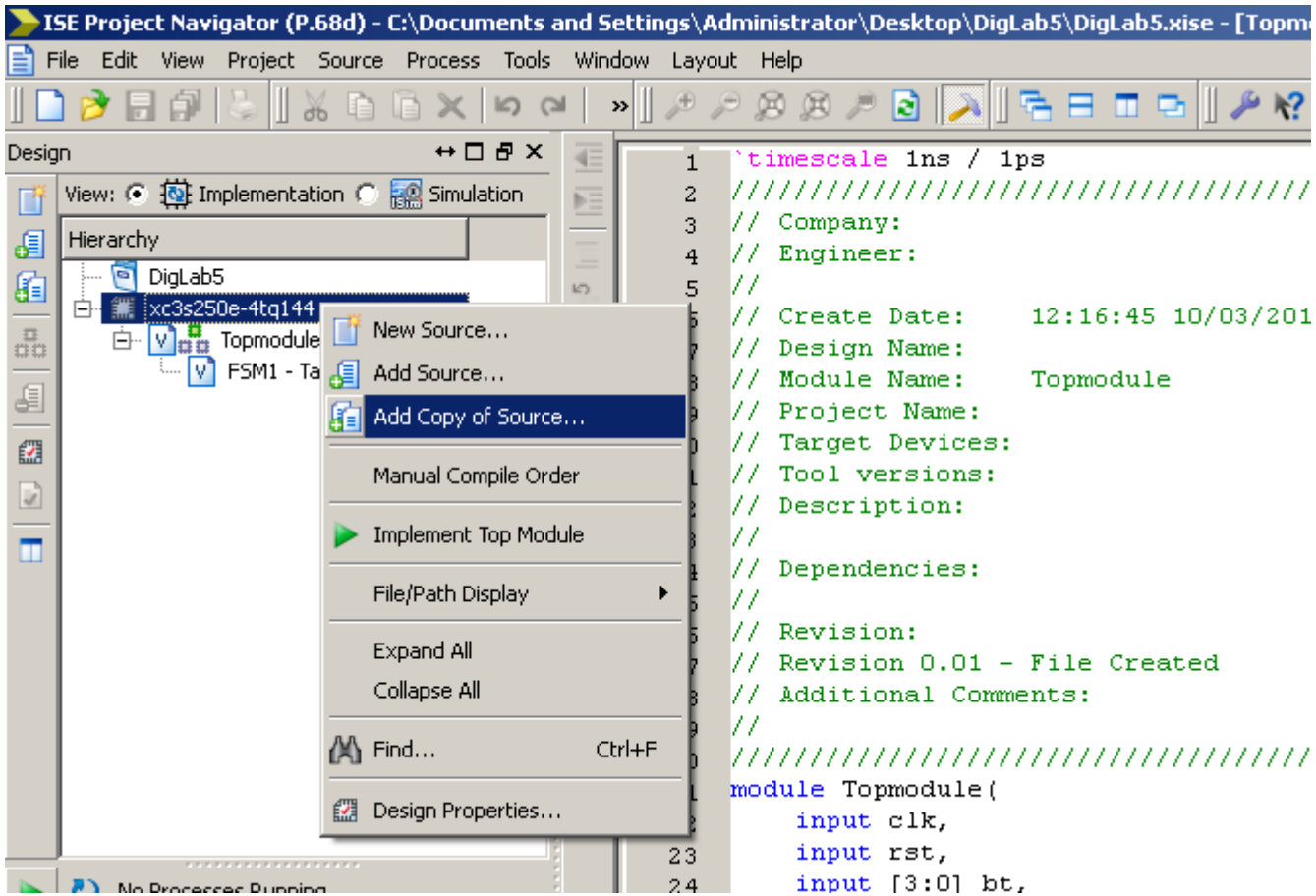
Task I – Generate programming file

- Before generating the programming file, you have to add the .UCF file to the project.
- Download it from this link.
- Unzip it into your working directory.
- Your working directory appears in the title bar of the project navigator.
- Example:



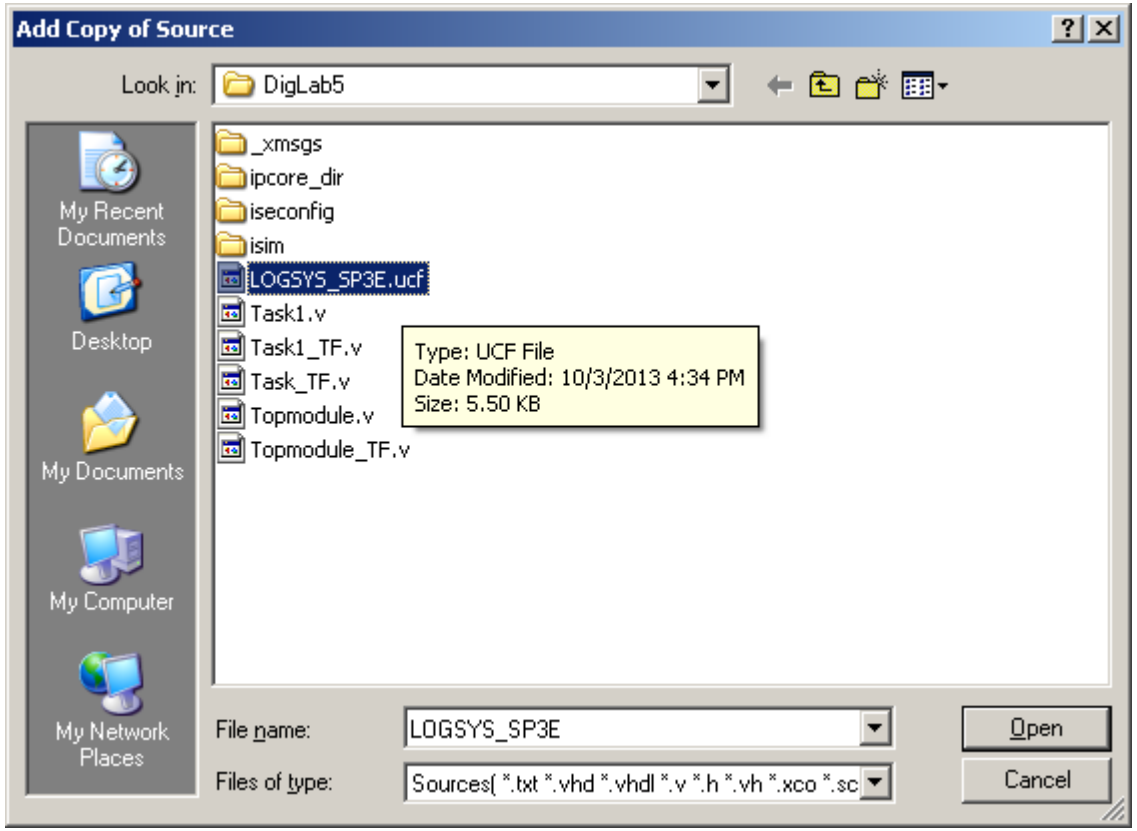
Task I – Generate programming file

- Right click on the project and select Add Copy of Source…



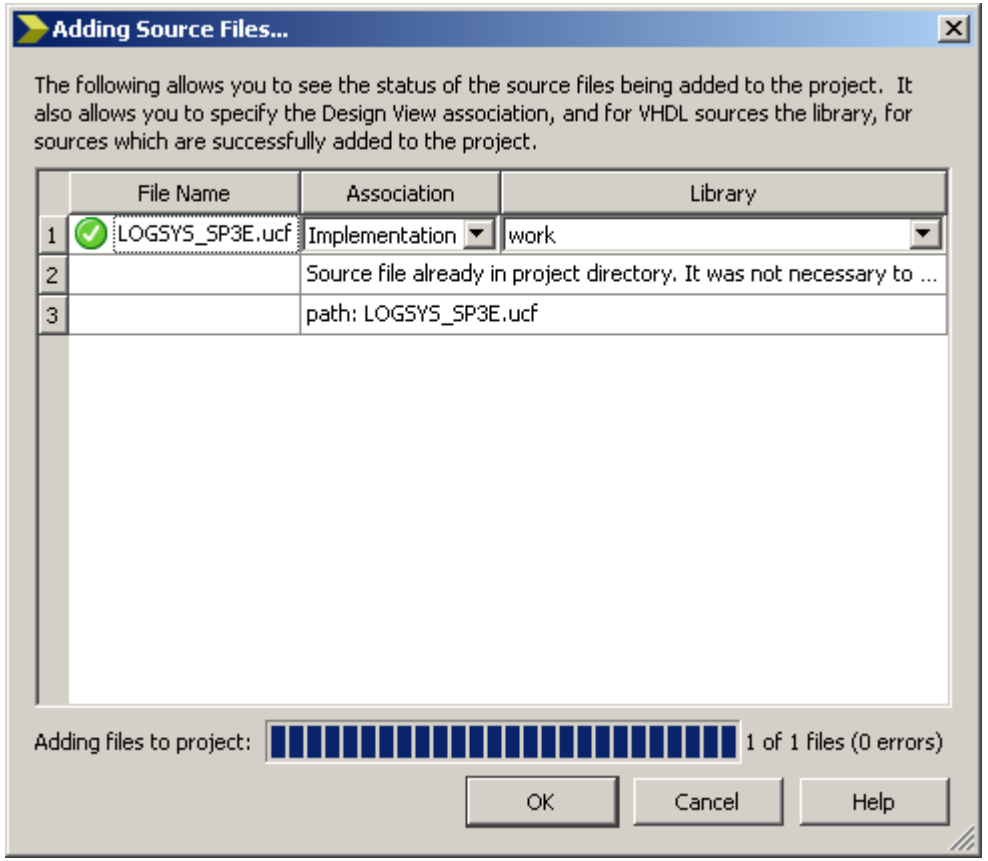
Task I – Generate programming file

- Select the LOGSYS_SP3E.ucf file



Task I – Generate programming file

- The following window appers. Select OK.



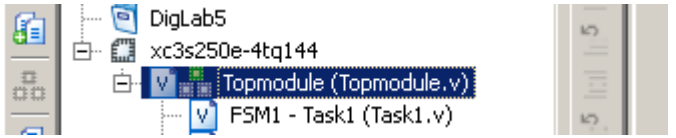
Task I – Generate programming file

- Open the LOGSYS_SP3E file and uncomment (delete the # character from the beginning) the following lines: clk, rst, bt<3>…bt<0>, ld<7>…ld<0>

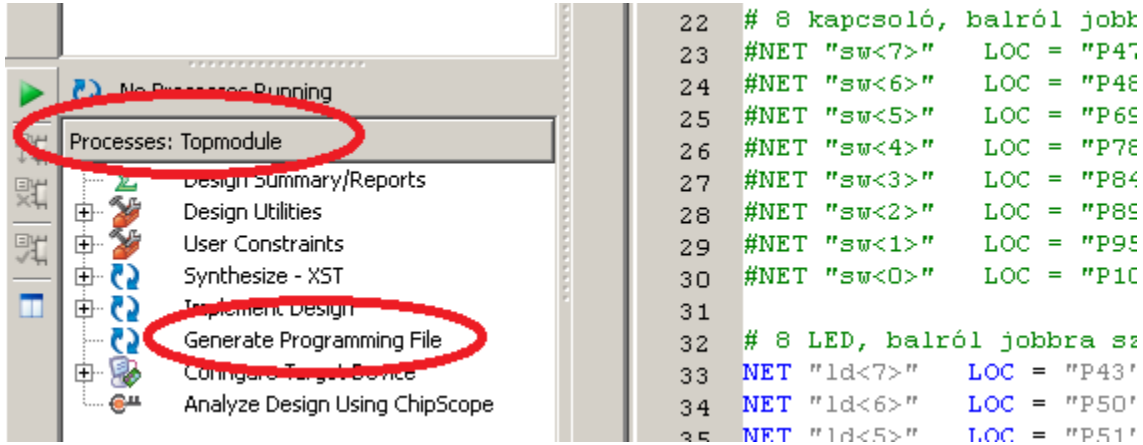
```
13  NET "clk"     LOC = "P129" | PULLDOWN;
14  NET "rst"     LOC = "P119" | PULLDOWN;
15
16  # 4 darab aktív magas nyomógomb, balról jobbra számozva
17  NET "bt<3>"   LOC = "P12";
18  NET "bt<2>"   LOC = "P24";
19  NET "bt<1>"   LOC = "P36";
20  NET "bt<0>"   LOC = "P38";
21
22  # 8 kapcsoló, balról jobbra számozva
23  #NET "sw<7>"   LOC = "P47";
24  #NET "sw<6>"   LOC = "P48";
25  #NET "sw<5>"   LOC = "P69";
26  #NET "sw<4>"   LOC = "P78";
27  #NET "sw<3>"   LOC = "P84";
28  #NET "sw<2>"   LOC = "P89";
29  #NET "sw<1>"   LOC = "P95";
30  #NET "sw<0>"   LOC = "P101";
31
32  # 8 LED, balról jobbra számozva
33  NET "ld<7>"   LOC = "P43";
34  NET "ld<6>"   LOC = "P50";
35  NET "ld<5>"   LOC = "P51";
36  NET "ld<4>"   LOC = "P52";
37  NET "ld<3>"   LOC = "P53";
38  NET "ld<2>"   LOC = "P54";
39  NET "ld<1>"   LOC = "P58";
40  NET "ld<0>"   LOC = "P59";
```

Task I – Generate programming file

- Press Save All
- Select the Topmodule file

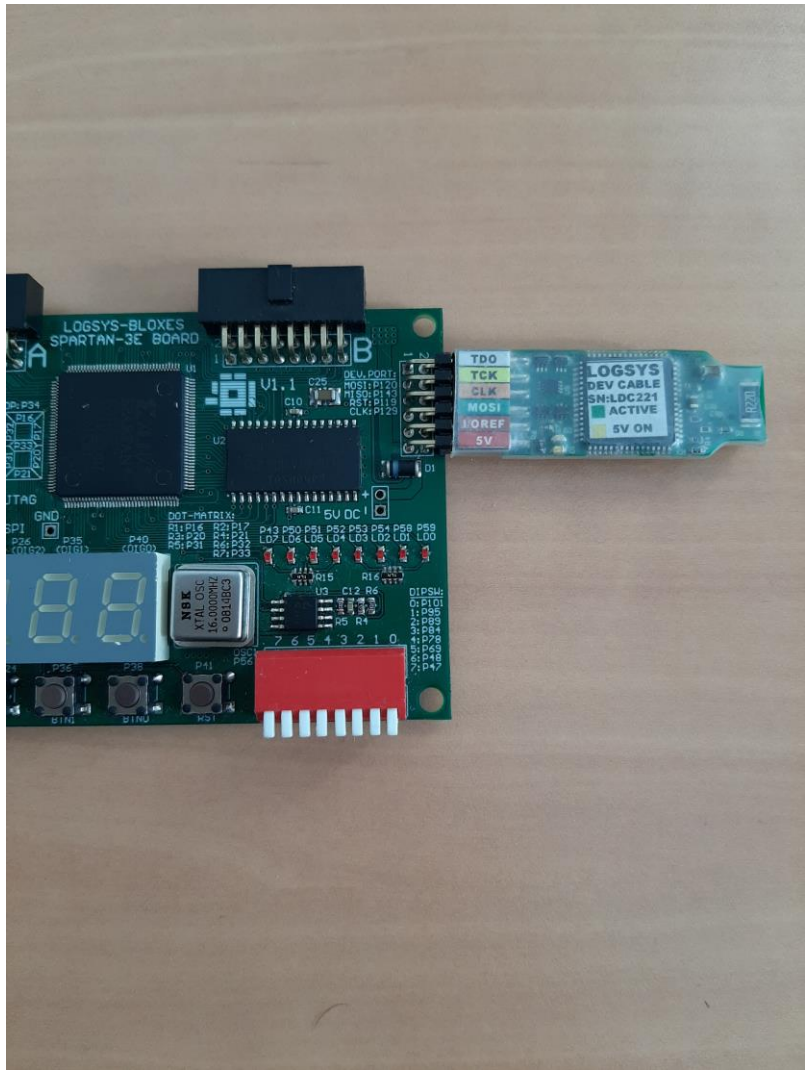


- In Processes: Topmodule (on the mid-left of the screen), double click on Generate Programming File



Task I – Generate programming file

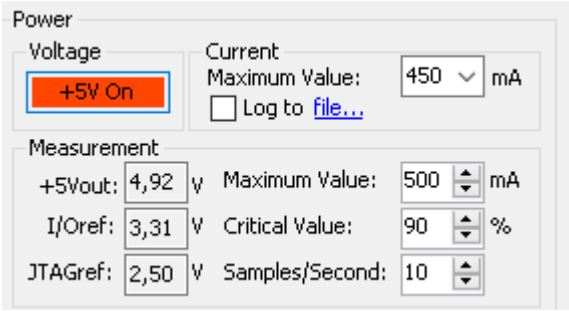
- If the program file was generated successfully, you can connect the FPGA board to the PC
- Mind the orientation of the JTAG connector!



Task I – Generate programming file

- Launch the Logsys GUI application

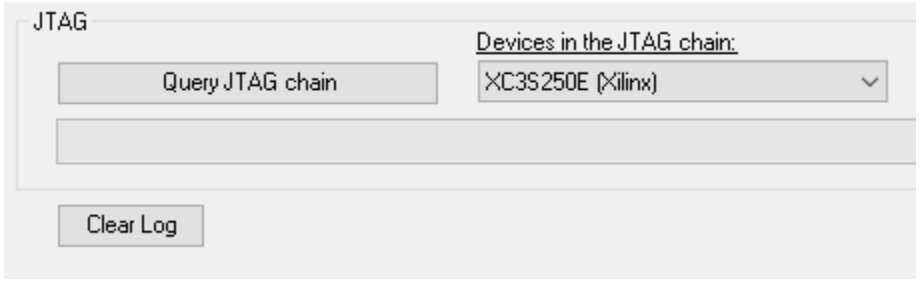
- Press the +5V button to turn the board on



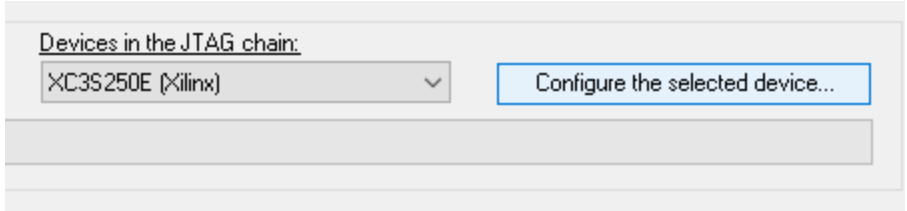
Task I – Generate programming file

- On the right side of the screen, select JTAG download:

- Press the „Query JTAG chain” button

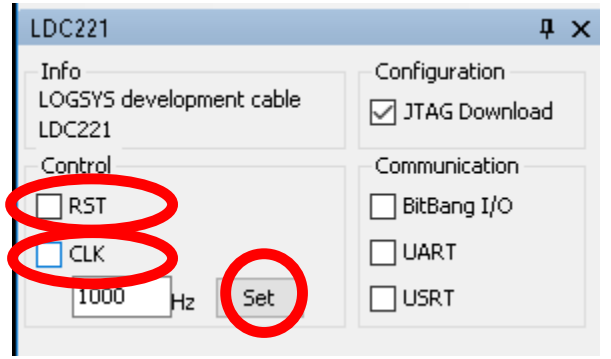


- Then press „Configure the Selected Device”



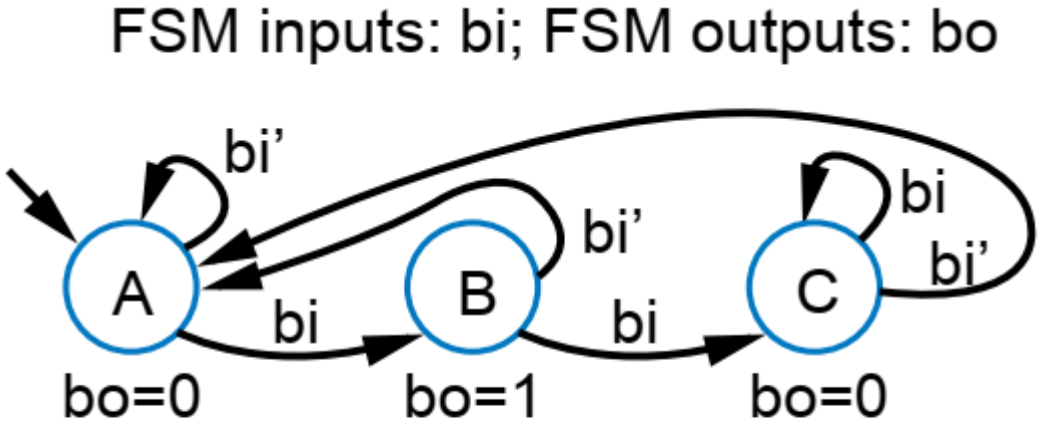
Task I – Generate programming file

- Browse the generated file in your working directory
- Press Open
- The circuit needs a system clock input and a reset input
- First set the value of the clock frequency to 1000
- Press the Set button
- Click into the CLK checkbox
- Note: on Windows 10 the tick might not appear, but it should be fine
- You can reset the circuit by pressing the RST checkbox



Task II

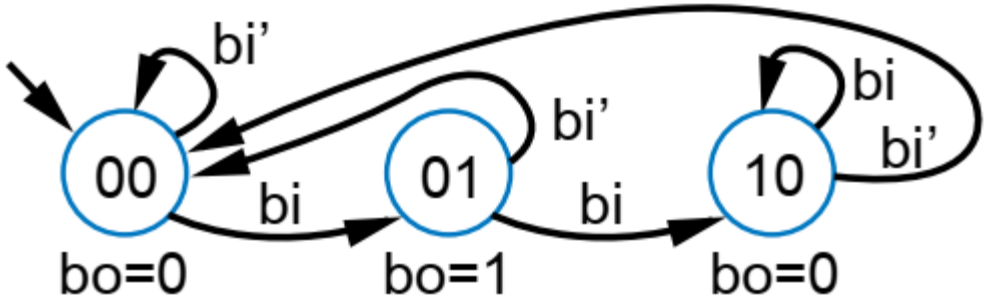
- If you have time, try to implement the following FSM on your own:
- FSM: Button press synchronizer – no matter how long the bi button is pressed, the output high is 1 clock cycle wide on the bo signal



Step 1: FSM

Task II

- Button press synchronizes: Encoding states and State table



Step 3: Encode states

	Combinational logic					
	Inputs			Outputs		
	s1	s0	bi	n1	n0	bo
A	0	0	0	0	0	0
	0	0	1	0	1	0
B	0	1	0	0	0	1
	0	1	1	1	0	1
C	1	0	0	0	0	0
	1	0	1	1	0	0
unused	1	1	0	0	0	0
	1	1	1	0	0	0

Step 4: State table

Task II

- Boolean equations of the combinational circuit:

$$n1 = s1's0bi + s1s0bi$$
$$n0 = s1's0'bi$$
$$bo = s1's0bi' + s1's0bi = s1s0$$

Task II

- Main steps:
  1. Add new module (e.g. Task2)
  2. 3 inputs (clk, rst, bi – single wires), 1 output (bo, single wire)
  3. Modify bo from output to output reg in the generated .v file
  4. Add the state and next_state registers.
  5. Implement the state register using always @ (posedge clk)
  6. Implement the next_state logic using always(*) and case…endcase
  7. Implement the bo logic using using always(*) and case…endcase
  8. Verify with simulations