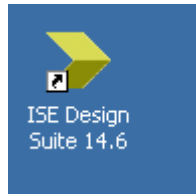


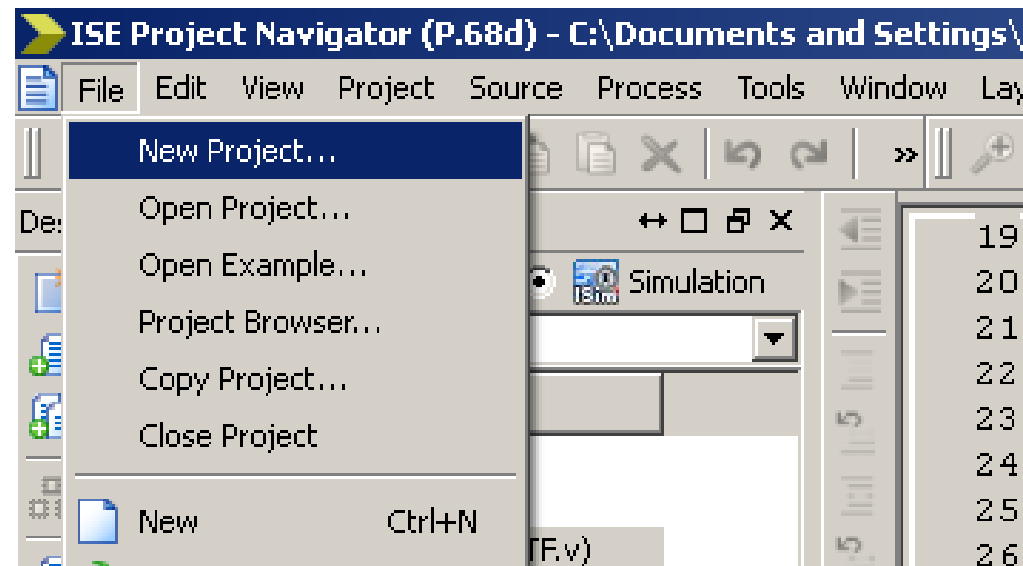
# Digital design laboratory 6

# Preparations

- Launch the ISE Design Suite



- Create new project:  
File -> New Project...



# Preparations

- Name: DigLab6
- Location: D:\DigLab6
- Working directory:  
D:\DigLab6
- Press Next

**New Project Wizard**

**Create New Project**  
Specify project location and type.

Enter a name, locations, and comment for the project

Name: DigLab6

Location: D:\DigLab6

Working Directory: D:\DigLab6

Description:

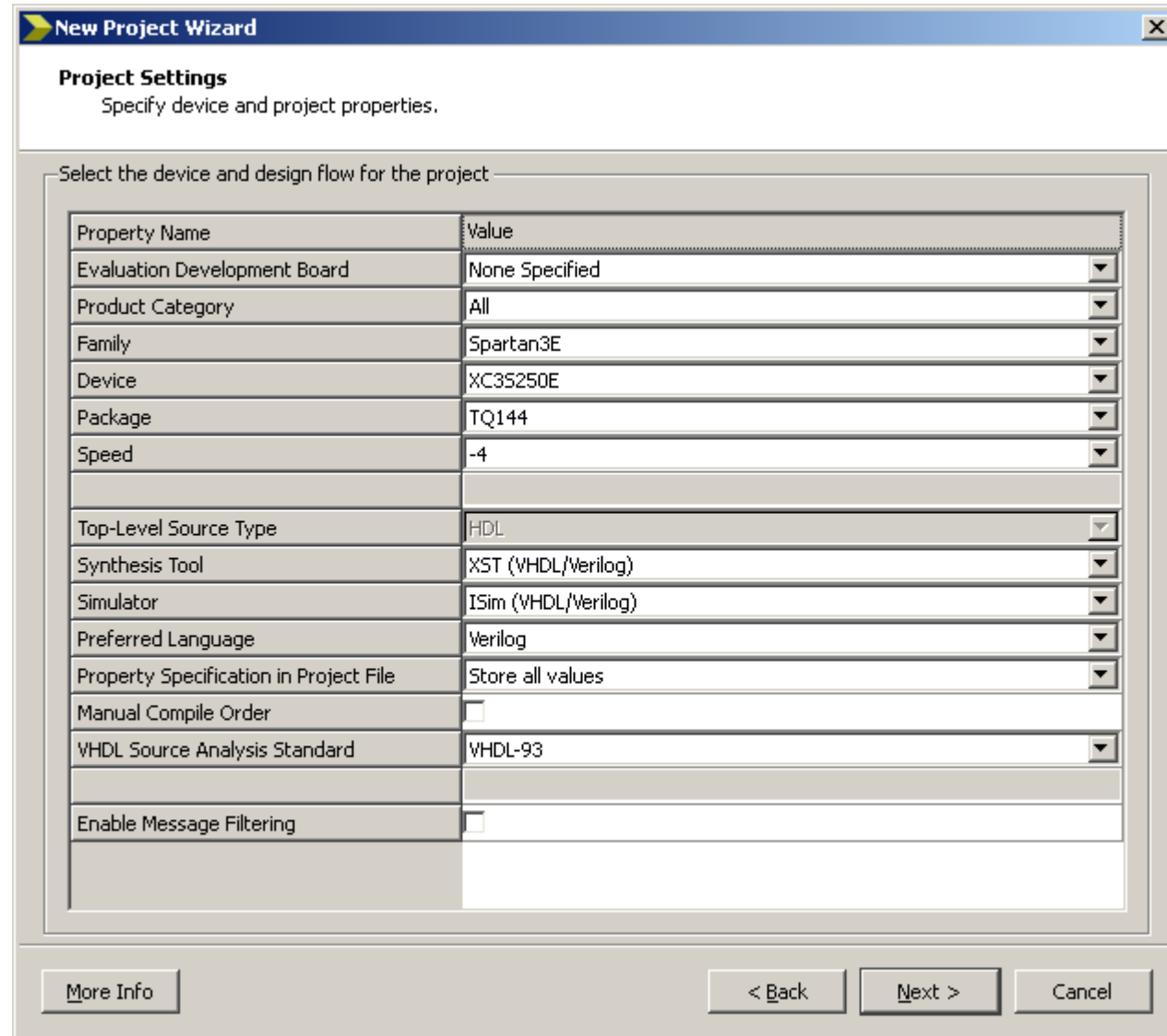
Select the type of top-level source for the project

Top-level source type:  
HDL

More Info Next > Cancel

# Preparations

- Check FPGA settings
- If OK, press Next
- Then press Finish

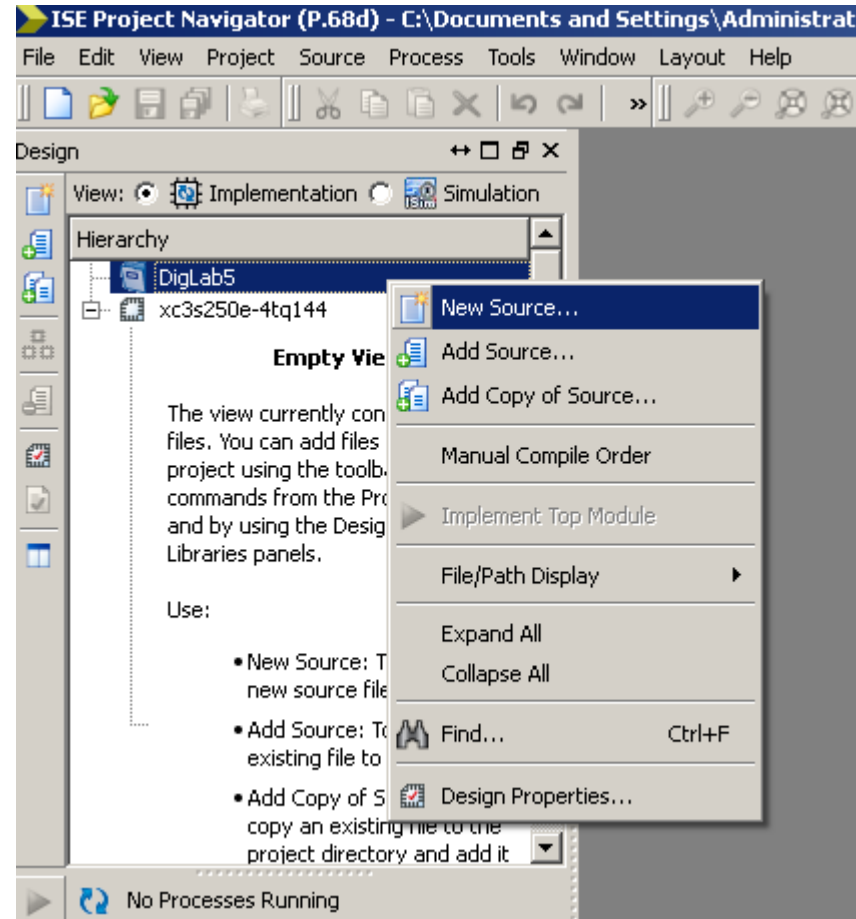


The screenshot shows the 'New Project Wizard' dialog box, specifically the 'Project Settings' step. The title bar reads 'New Project Wizard' with a close button. Below the title bar, the text 'Project Settings' is followed by the instruction 'Specify device and project properties.' The main area is titled 'Select the device and design flow for the project' and contains a table of settings. At the bottom, there are three buttons: 'More Info', '< Back', and 'Next >', along with a 'Cancel' button.

Property Name	Value
Evaluation Development Board	None Specified
Product Category	All
Family	Spartan3E
Device	XC3S250E
Package	TQ144
Speed	-4
Top-Level Source Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	Verilog
Property Specification in Project File	Store all values
Manual Compile Order	<input type="checkbox"/>
VHDL Source Analysis Standard	VHDL-93
Enable Message Filtering	<input type="checkbox"/>

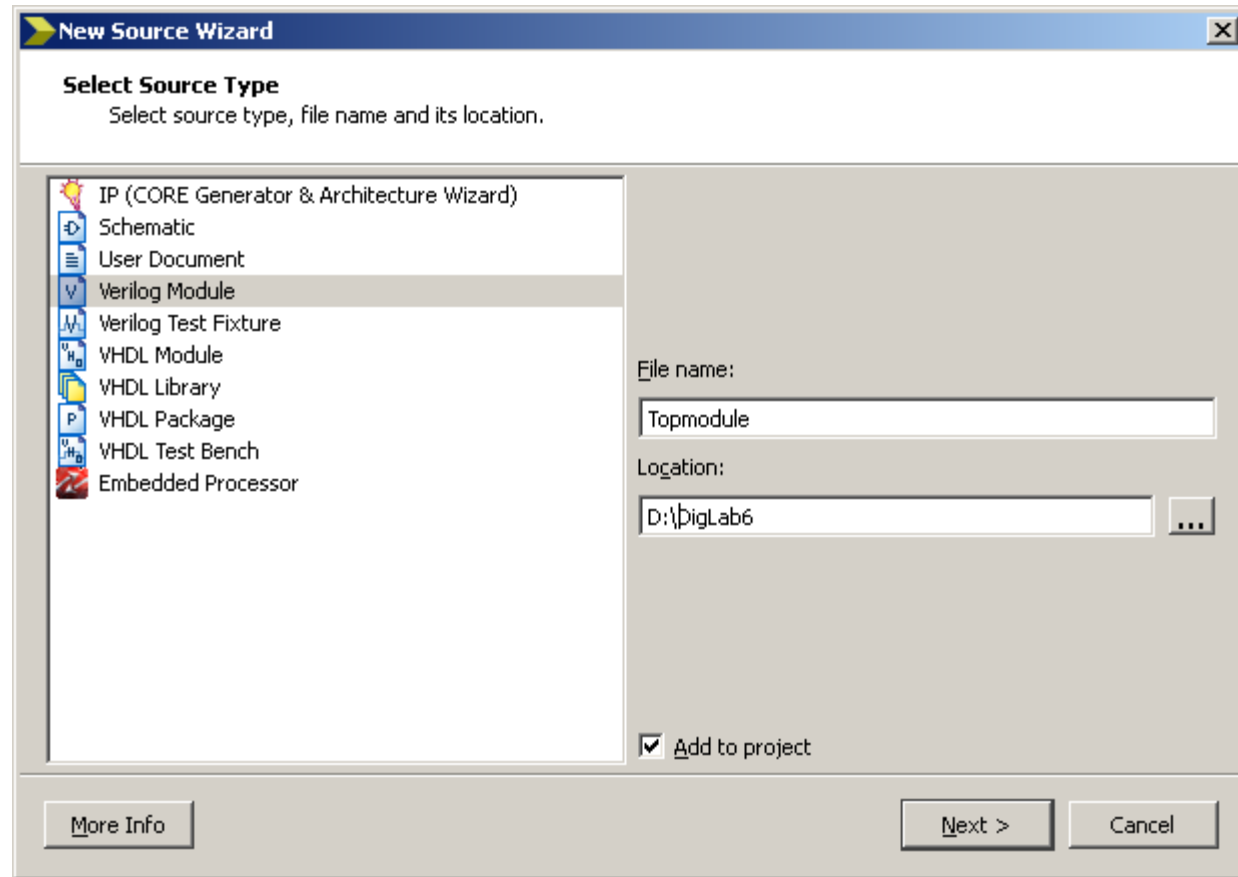
# Add module to the project

- In the top left corner, right click on the project (DigLab6)
- Select New Source...



# Add module to the project

- Select Verilog module
- Name: Topmodule
- Press Next button



# Add module to the project

- 4 inputs: clk, rst, bt, sw
- 1 output: ld
- bt, sw and ld: bus!
- Set the MSB values!
- Press Next
- Then press Finish

**New Source Wizard**

**Define Module**  
Specify ports for module.

Module name:

Port Name	Direction	Bus	MSB	LSB
clk	input	<input type="checkbox"/>		
rst	input	<input type="checkbox"/>		
bt	input	<input checked="" type="checkbox"/>	3	0
sw	input	<input checked="" type="checkbox"/>	7	0
ld	output	<input checked="" type="checkbox"/>	7	0
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		

# Task 1: Full adder implementation

- Add another module to the project using the previous steps.
- Right click on DigLab6 in the top left corner, then select New Source...
- Verilog module, File Name: Full\_Adder
- Press Next



# Task 1: Full adder implementation

- Inputs: a, b, ci
- Outputs: s, co
- Press Next
- Press Finish

**New Source Wizard**

**Define Module**  
Specify ports for module.

Module name: Full\_Adder

Port Name	Direction	Bus	MSB	LSB
a	input	<input type="checkbox"/>		
b	input	<input type="checkbox"/>		
ci	input	<input type="checkbox"/>		
s	output	<input type="checkbox"/>		
co	output	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		
	input	<input type="checkbox"/>		

More Info    < Back    Next >    Cancel

# Task 1: Full adder implementation

- You should see this:

```
1  `timescale 1ns / 1ps
2  ////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date:    08:24:37 10/11/2018
7  // Design Name:
8  // Module Name:    Full_Adder
9  // Project Name:
10 // Target Devices:
11 // Tool versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ////////////////////////////////////////////////////////////////////
21 module Full_Adder(
22     input a,
23     input b,
24     input ci,
25     output s,
26     output co
27 );
28
29
30 endmodule
31
```

# Task 1: Full adder implementation

- Reminder: full adder truth table and equations

Inputs			Outputs	
a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$co = a'bc + ab'c + abc' + abc$$

$$co = a'bc + abc + ab'c + abc + abc' + abc$$

$$co = (a'+a)bc + (b'+b)ac + (c'+c)ab$$

$$co = bc + ac + ab$$

$$s = a'b'c + a'bc' + ab'c' + abc$$

$$s = a'(b'c + bc') + a(b'c' + bc)$$

$$s = a'(b \text{ xor } c)' + a(b \text{ xor } c)$$

$$s = a \text{ xor } b \text{ xor } c$$

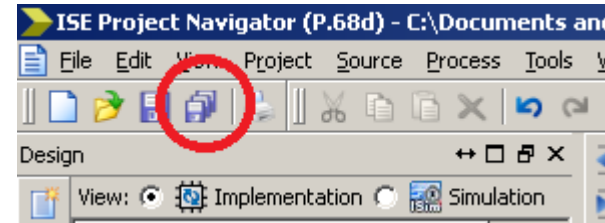
# Task 1: Full adder implementation

- Implement the above equations in Verilog.
- None of the outputs are registers, so their values have to be set by the **assign** statement.

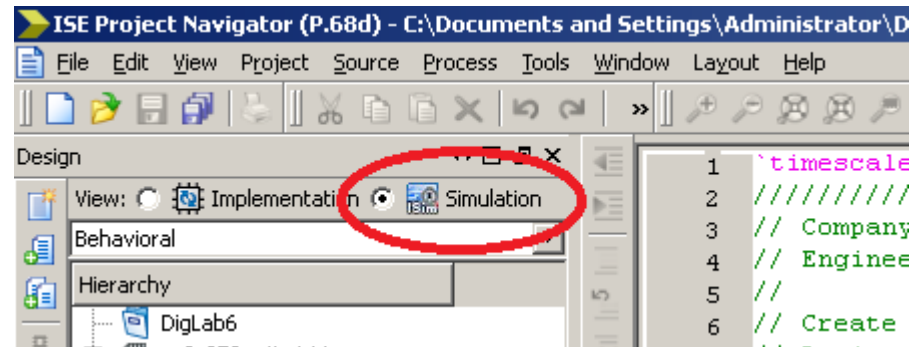
```
20 ///////////////////////////////////////////////////  
21 module Full_Adder(  
22     input a,  
23     input b,  
24     input ci,  
25     output s,  
26     output co  
27 );  
28  
29     assign co = b&ci | a&ci | a&b;  
30     // XOR operator: ^  
31     assign s = a^b^ci;  
32  
33 endmodule  
34
```

# Task 1: Full adder simulation

- First of all: Press Save All button

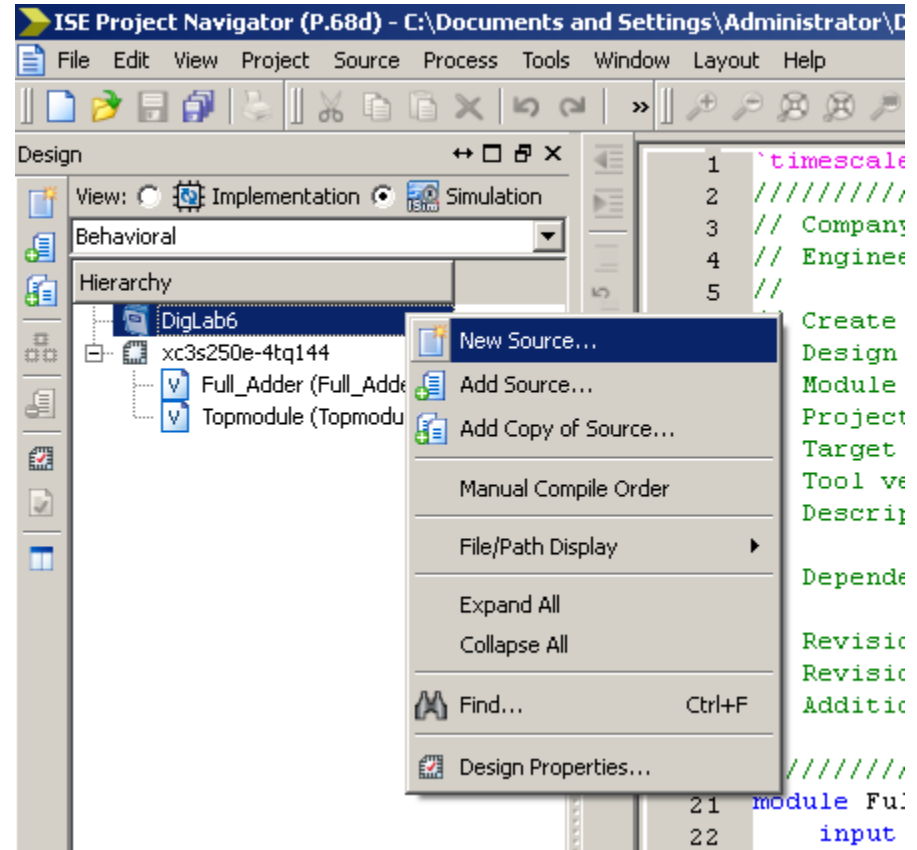


- Switch to Simulation mode



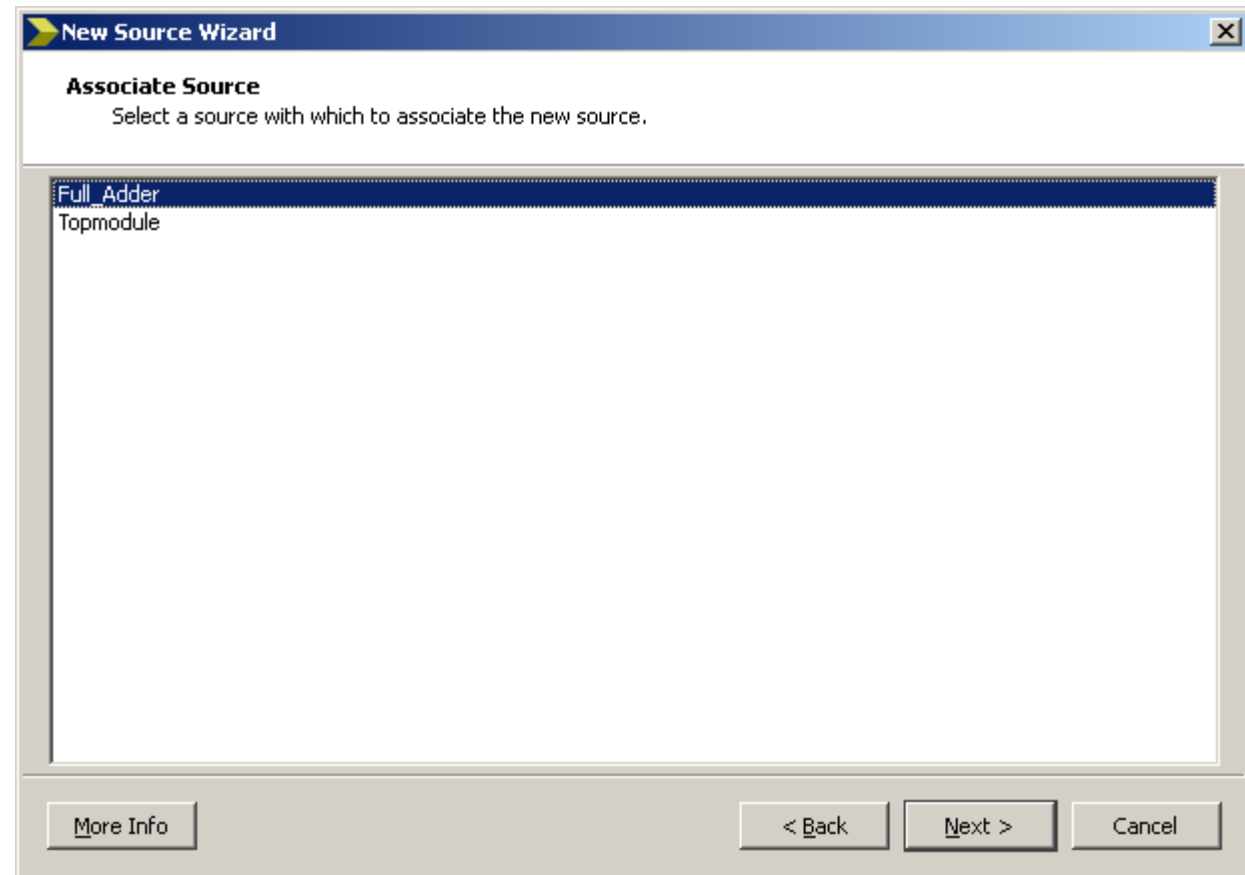
# Task 1: Full adder simulation

- Right Click on DigLab6, select New Source
- Select Verilog Test Fixture
- File Name: Full\_Adder\_TF
- Press Next



# Task 1: Full adder simulation

- On the next screen, select the Full\_Adder module
- If selected, press Next
- Then press Finish



# Task 1: Full adder simulation

- Now we are going to test the Full\_Adder for every possible input combination
- A possible way is to type in all combinations:
- `#100 a=1'b0; b=1'b0; ci=1'b0;`  
`#100 a=1'b0; b=1'b0; ci=1'b1;`  
...
- This would be quite annoying. Instead, we are going to use a **for loop**



# Task 1: Full adder simulation

- Add an integer i variable to the Test Fixture file **ABOVE** the initial begin part:

```
36 // Instantiate the Unit Under Test (UUT)
37 Full_Adder uut (
38     .a(a),
39     .b(b),
40     .ci(ci),
41     .s(s),
42     .co(co)
43 );
44
45 integer i;
46
47 initial begin
48     // Initialize Inputs
49     a = 0;
50     b = 0;
51     ci = 0;
```

# Task 1: Full adder simulation

- Add the for loop after the Add stimulus here part:

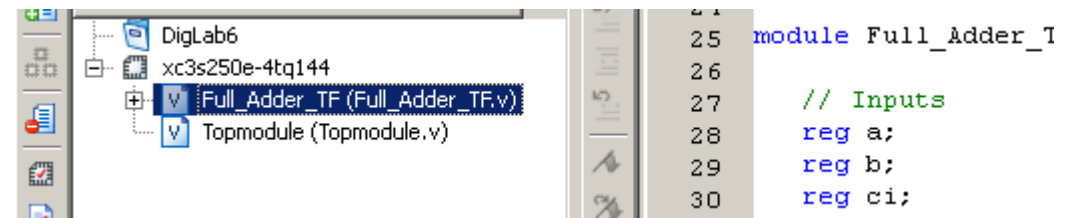
```
45     integer i;
46
47     initial begin
48         // Initialize Inputs
49         a = 0;
50         b = 0;
51         ci = 0;
52
53         // Wait 100 ns for global reset to finish
54         #100;
55
56         // Add stimulus here
57         for (i=0; i<8; i=i+1)
58         begin
59             {a,b,ci}=i;
60             #100;
61         end
62
63     end
64
65 endmodule
66
```

# Task 1: Full adder simulation

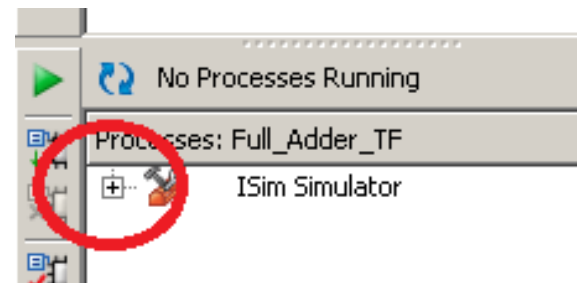
- Press the Save All button
- The `{a,b,ci}` concatenates the three 1-bit variables (a, b and ci) into one 3-bit variable.
- The `i` variable is a 32-bit integer. The `{a,b,ci}=i;` command copies the last 3 bits of the `i` variable into the concatenated `{a,b,ci}`.
- In other words: in every iteration of the for loop, the value of `a` is set to the value of the third bit of `i`, the value of `b` is set to the value of the second bit of `i`, and finally the value of `ci` is set to the first bit of `i`:
- `a=i[2], b=i[1], ci=i[0]`.

# Task 1: Full adder simulation

- Check if every modification is saved
- If so, select the test fixture file (left click on it):

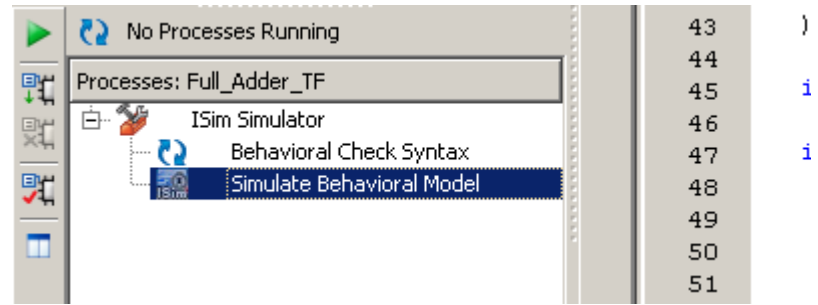


- Then press the plus button on the left of the ISim Simulator on the middle-left part of the screen



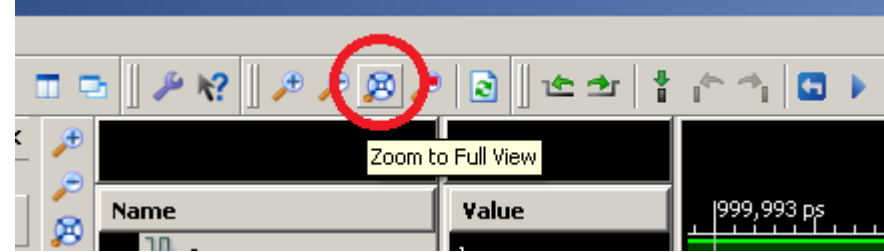
# Task 1: Full adder simulation

- Then run (by double left click) Simulate Behavioral Model



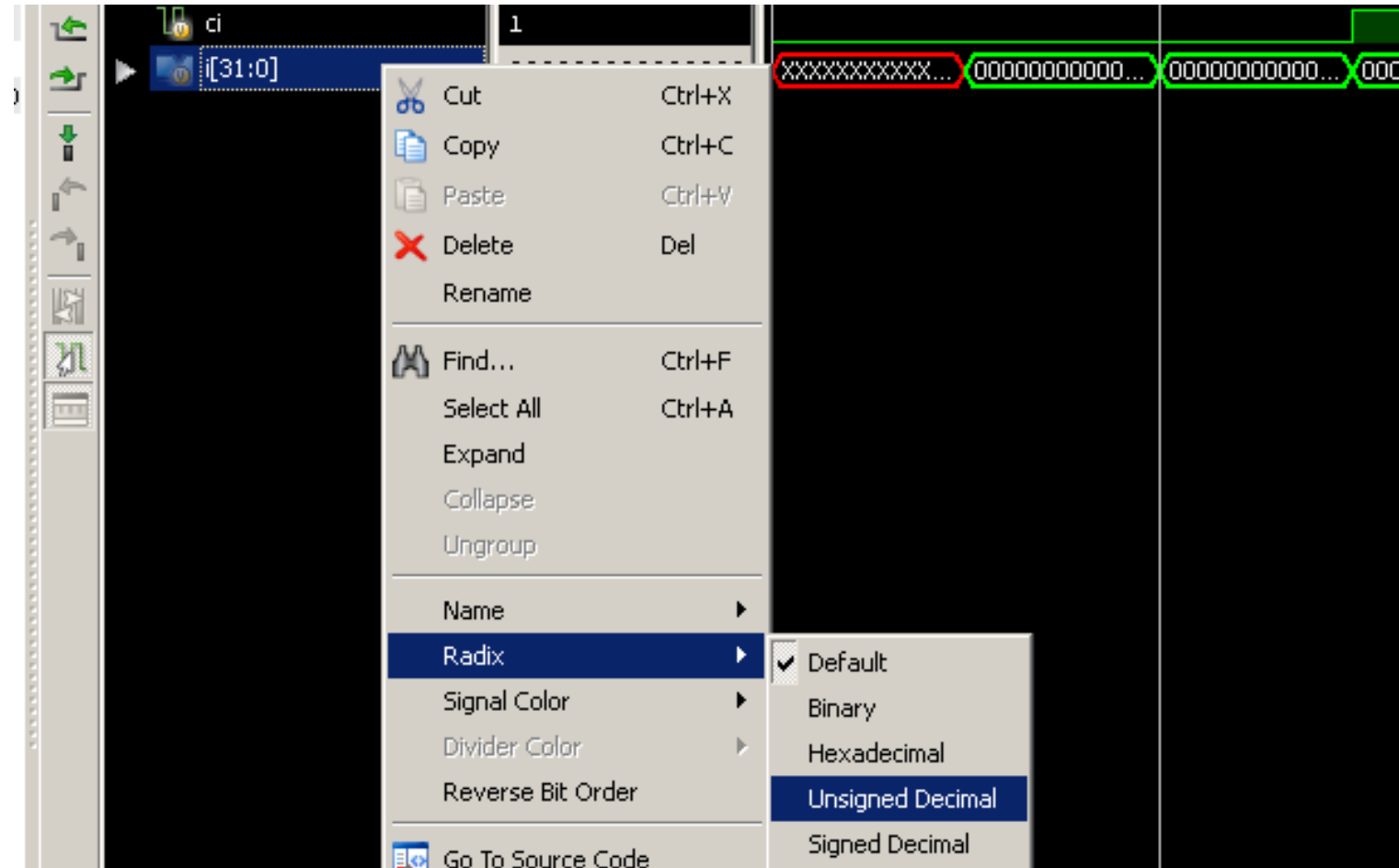
- The simulator application launches

- Press the Zoom to full view button:



- You can see the values of the s, co, a, b, ci wires and the i variable.

- Right click on `i[31:0]`,  
select Radix ->  
Unsigned decimal



# Task 1: Full adder simulation

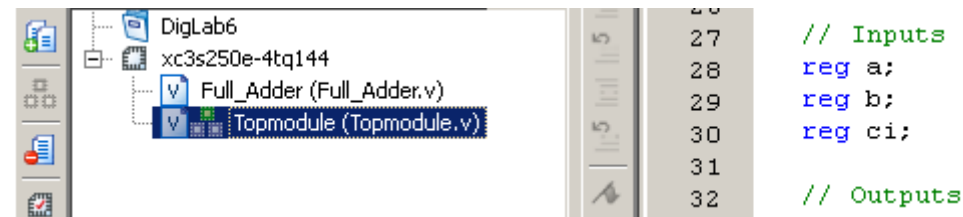
- Verify the correct behavior of the full adder module (s and co).
- Also, check the connection between a, b, ci and the bits of i.
- When you have finished, close the simulator. A popup window will appear asking whether save the changes or not. Select No.

# Task 1: Full adder implementation

- Now you are going to implement a 4 bit adder in the Topmodule.
- Switch back to implementation mode:



- Select Topmodule.v





# Task 1: Full adder implementation

- In this implementation, we are going to add two 4-bit numbers using the 1-bit full adder modules.
- The values of the four bit inputs (A and B) will be set on the switches.
- The result will be displayed on the leds (in a binary form).
- Add the following wires to the Topmodule:

```
20 ////////////////////////////////////////////////////
21 module Topmodule(
22     input clk,
23     input rst,
24     input [3:0] bt,
25     input [7:0] sw,
26     output [7:0] ld
27 );
28
29     wire [3:0] A, B, S; // S = A + B;
30     wire [4:0] C; // C: carry
31
32
33
34 endmodule
35
```

# Task 1: Full adder implementation

- Now we are going to connect the A, B and S variables to the switches and the leds:

```
21 module Topmodule(  
22     input clk,  
23     input rst,  
24     input [3:0] bt,  
25     input [7:0] sw,  
26     output [7:0] ld  
27 );  
28  
29     wire [3:0] A, B, S; // S = A + B;  
30     wire [4:0] C; // C: carry  
31  
32     assign A = sw[3:0]; // first operand  
33     assign B = sw[7:4]; // second operand  
34     assign ld[3:0] = S;  
35     assign ld[4] = C[4]; // Carry out of the last full adder  
36     assign ld[7:5] = 3'd0; // Unused bits  
37  
38  
39 endmodule  
40
```

# Task 1: Full adder implementation

- Note: you can set the value of `ld` with 1 command using concatenation {}:

```
31
32     assign A = sw[3:0]; // first operand
33     assign B = sw[7:4]; // second operand
34     //assign ld[3:0] = S;
35     //assign ld[4] = C[4]; // Carry out of the last full adder
36     //assign ld[7:5] = 3'd0; // Unused bits
37
38     assign ld = {3'd0, C[4], S}; // Concatenation
39
40
```

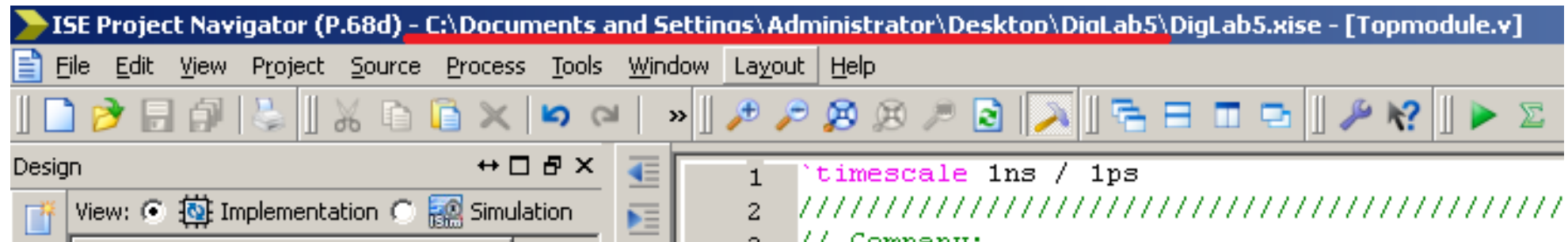
# Task 1: Full adder implementation

- The next step is the instantiation of the full adder modules:
- Don't forget to set the first carry in input (C[0]) to 0.

```
31
32     assign A = sw[3:0]; // first operand
33     assign B = sw[7:4]; // second operand
34     //assign ld[3:0] = S;
35     //assign ld[4] = C[4]; // Carry out of the last full adder
36     //assign ld[7:5] = 3'd0; // Unused bits
37
38     assign ld = {3'd0, C[4], S}; // Concatenation
39
40     assign C[0] = 1'b0; // Setting the first carry in to constant 0
41     Full_Adder F0(.a(A[0]), .b(B[0]), .ci(C[0]), .s(S[0]), .co(C[1]));
42     Full_Adder F1(.a(A[1]), .b(B[1]), .ci(C[1]), .s(S[1]), .co(C[2]));
43     Full_Adder F2(.a(A[2]), .b(B[2]), .ci(C[2]), .s(S[2]), .co(C[3]));
44     Full_Adder F3(.a(A[3]), .b(B[3]), .ci(C[3]), .s(S[3]), .co(C[4]));
45
46
47
48     endmodule
49
```

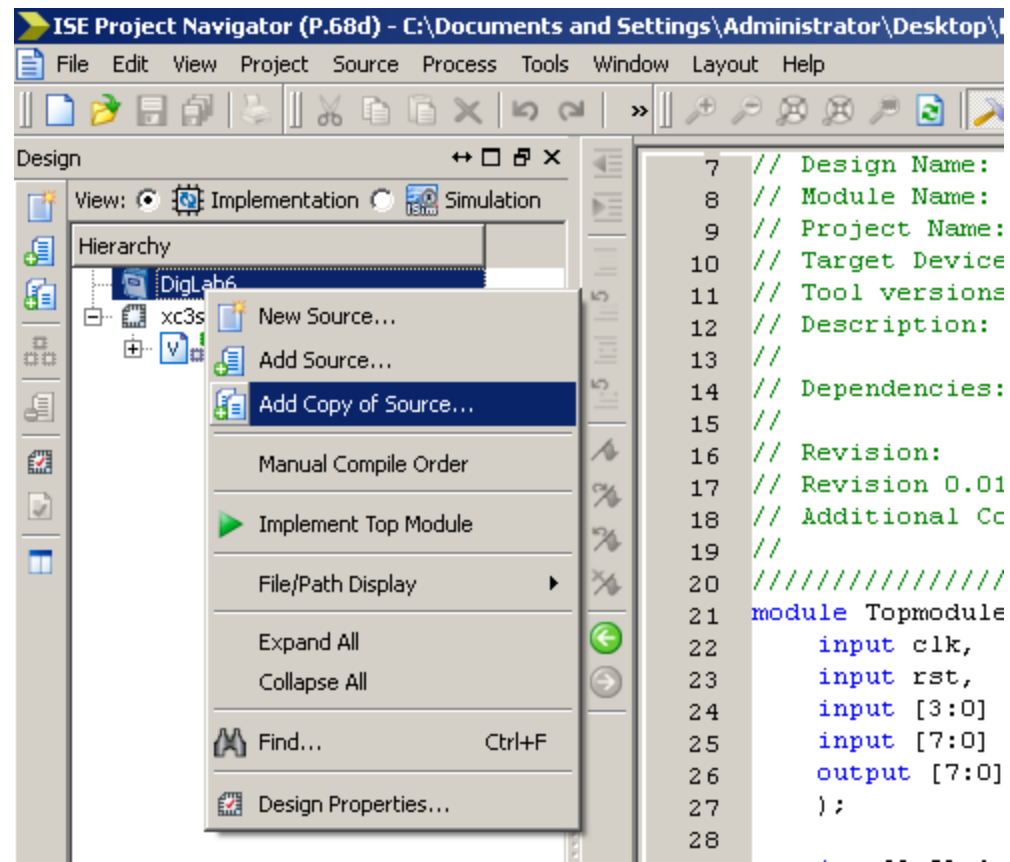
# Task 1: Full adder implementation

- 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:



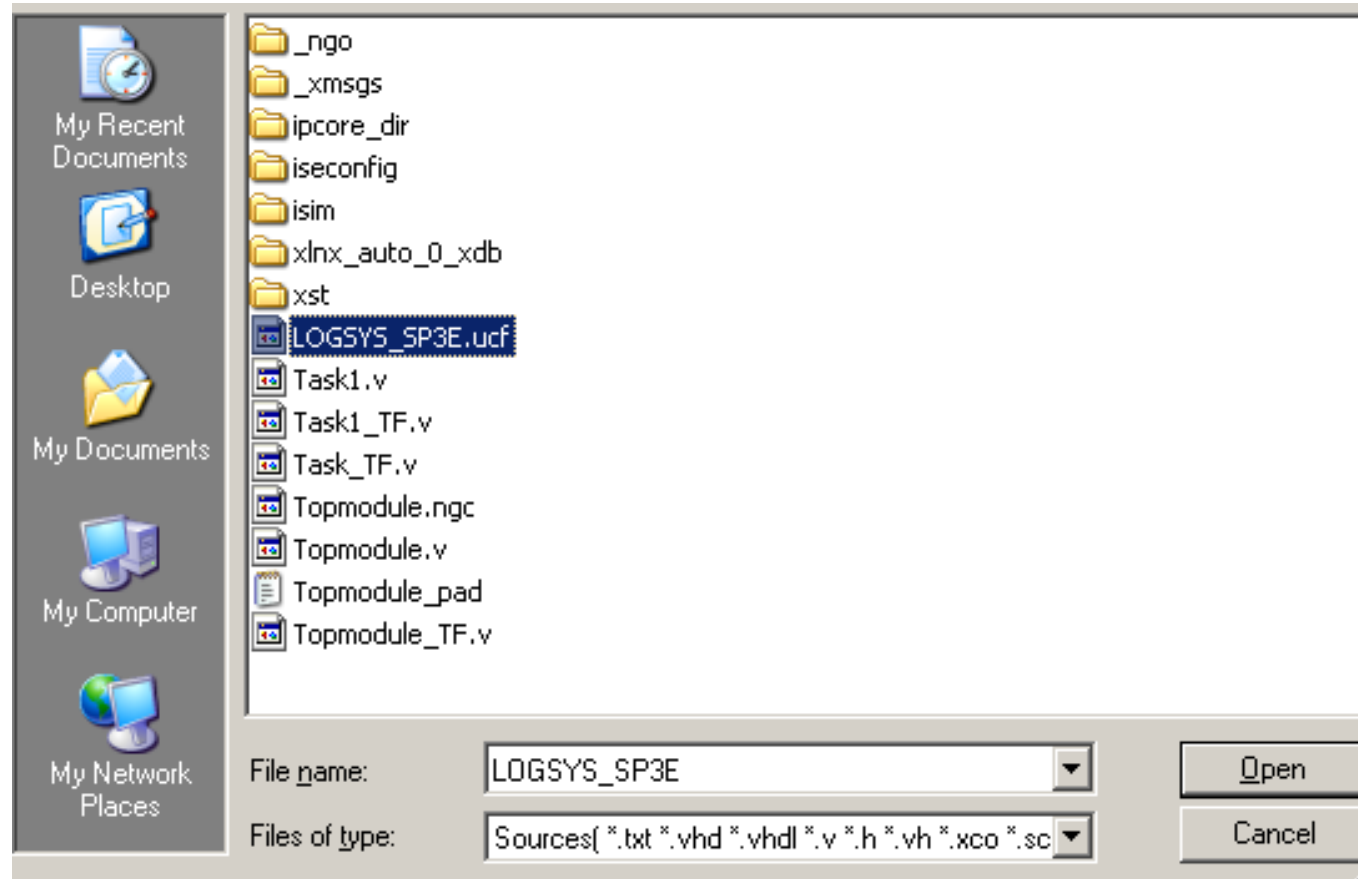
# Task 1: Full adder implementation

- Save All changes
- Right click on the project and select Add Copy of Source...



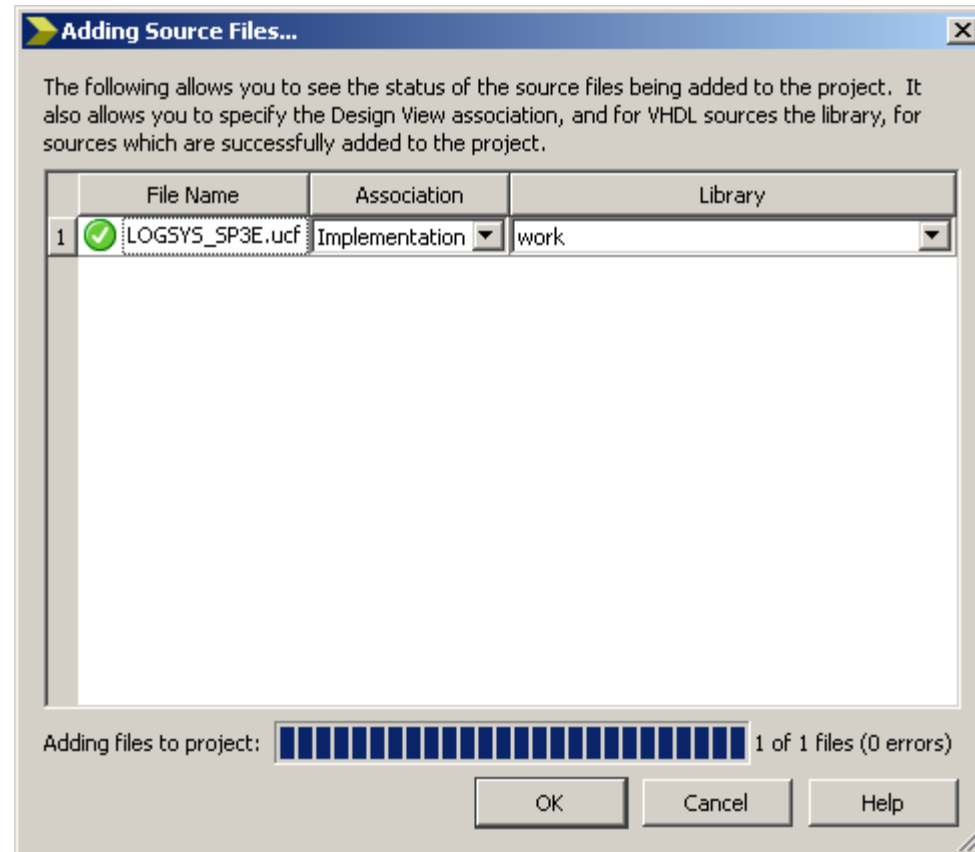
# Task 1: Full adder implementation

- Select the .ucf file, then press Open



# Task 1: Full adder implementation

- The following window appears. Press OK.





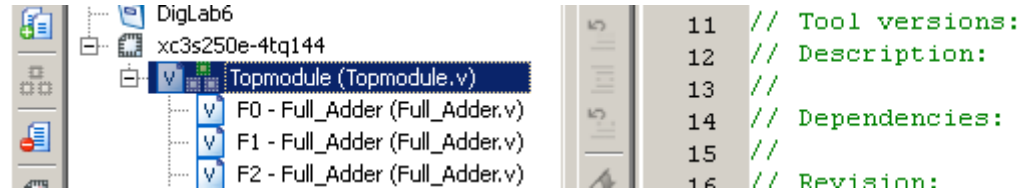
# Task 1: Full adder implementation

- Open the ucf file in the editor
- Uncomment the following lines: clk, rst, bt, sw, ld
- Then Save All

```
13 NET "clk"      LOC = "P129" | PULLDOWN;
14 NET "rst"     LOC = "P119" | PULLDOWN;
15
16 # 4 darab aktiv 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";
41
```

# Task 1: Full adder implementation

- Select the Topmodule.v file



- Under Processes: Topmodule (middle-left of the screen), run **Generate Programming File**
- You will see warnings, that's because we did not use every input. Its not a problem.
- If generated successfully, you can upload the .bit file, and try the 4 bit adder.

# 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!

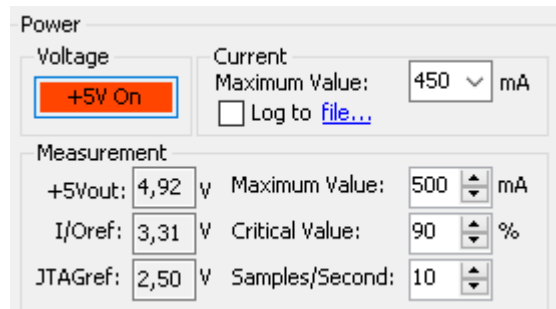


# Task I – Generate programming file

- Launch the Logsys GUI application

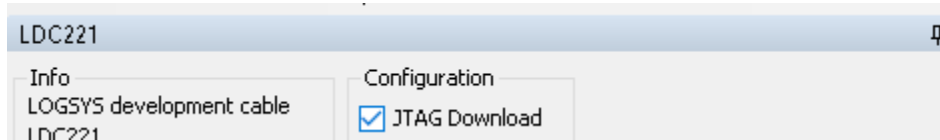


- Press the +5V button to turn the board on

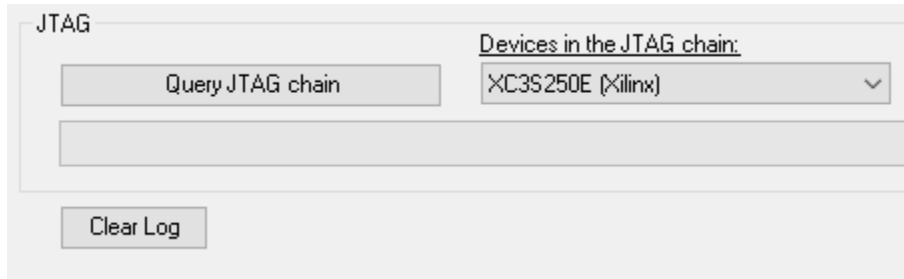


# Task I – Generate programming file

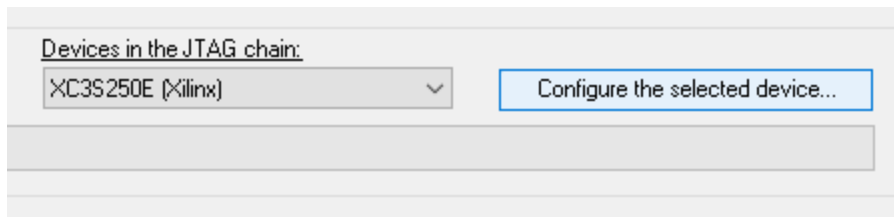
- On the right side of the screen, select JTAG download:



- Press the „Query JTAG chain” button

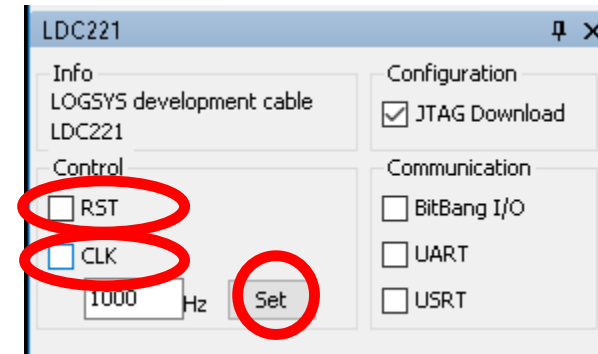


- Then press „Configure the Selected Device”



# 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 2: Multifunction register

- We are going to implement a register with the following functionality:
- Shift to the left, shift to the right, load, set
- bt[0]: shift to the left
- bt[1]: shift to the right
- bt[2]: load content from switches
- bt[3]: set every bit to 1
- If no button is pressed, just maintain the present value
- The clear operation is done by the reset input

# Task 2: Multifunction register

- Go back to the ISE Project Navigator, select Topmodule.v
- Comment or delete the previously added lines, except for the module declaration and endmodule

```
21 module Topmodule(  
22     input clk,  
23     input rst,  
24     input [3:0] bt,  
25     input [7:0] sw,  
26     output [7:0] ld  
27 );  
28  
29 // wire [3:0] A, B, S; // S = A + B;  
30 // wire [4:0] C; // C: carry, S = A + B;  
31 //  
32 // assign A = sw[3:0]; // first operand  
33 // assign B = sw[7:4]; // second operand  
34 // assign ld[3:0] = S;  
35 // assign ld[4] = C[4]; // Carry out of the last full adder  
36 // assign ld[7:5] = 3'd0; // Unused bits  
37 //  
38 // assign ld = {3'd0, C[4], S}; // Concatenation  
39 //  
40 // assign C[0] = 1'b0; // Setting the first carry in to constant 0  
41 // Full_Adder F0(.a(A[0]), .b(B[0]), .ci(C[0]), .s(S[0]), .co(C[1]));  
42 // Full_Adder F1(.a(A[1]), .b(B[1]), .ci(C[1]), .s(S[1]), .co(C[2]));  
43 // Full_Adder F2(.a(A[2]), .b(B[2]), .ci(C[2]), .s(S[2]), .co(C[3]));  
44 // Full_Adder F3(.a(A[3]), .b(B[3]), .ci(C[3]), .s(S[3]), .co(C[4]));  
45  
46  
47 endmodule
```



## Task 2: Multifunction register

- Add the following register variables: `reg [7:0] state, next_state;`
- Implement the state register:

```
43 // Full_Adder F2 (.a(A[4]), .b(B[4]), .ci(C[4]), .s  
44 // Full_Adder F3 (.a(A[3]), .b(B[3]), .ci(C[3]), .s  
45  
46     reg [7:0] state, next_state;  
47  
48     always @ (posedge clk)  
49     if (rst) state = 8'd0;  
50     else state = next_state;  
51  
52 endmodule  
53
```

## Task 2: Multifunction register

- Next we implement the next\_state logic.
- Add an always (\*) block to the code with begin and end

```
45
46     reg [7:0] state, next_state;
47
48     always @ (posedge clk)
49     if (rst) state = 8'd0;
50     else state = next_state;
51
52     always @ (*)
53     begin
54
55     end
56
```

# Task 2: Multifunction register

- Add the following skeleton code to cover each case

```
45
46     reg [7:0] state, next_state;
47
48     always @ (posedge clk)
49     if (rst) state = 8'd0;
50     else state = next_state;
51
52     always @ (*)
53     begin
54         if (bt[0]) // shift to the left
55         else if (bt[1]) // shift to the right
56         else if (bt[2]) // load content
57         else if (bt[3]) // set every bit to 1
58         else // maintain value
59     end
60
61 endmodule
```

## Task 2: Multifunction register

- Implement the functionality by adding the following lines to the skeleton:

```
always @ (*)
begin
    if (bt[0]) // shift to the left
        next_state <= {state[6:0], 1'b0};
    else if (bt[1]) // shift to the right
        next_state <= {1'b0, state[7:1]};
    else if (bt[2]) // load content
        next_state <= sw;
    else if (bt[3]) // set every bit to 1
        next_state <= 8'd255;
    else // maintain value
        next_state <= state;
end
```

- Notice that bt[0] has priority over the other buttons, bt[1] has priority over bt[2] and bt[3], and bt[2] has priority over bt[3].
- So the priorities (in descending order): bt[0], bt[1], bt[2], bt[3].

# Task 2: Multifunction register

- Don't forget to connect the state register to the leds:

```
always @ (*)
begin
    if (bt[0]) // shift to the left
        next_state <= {state[7:1], 1'b0};
    else if (bt[1]) // shift to the right
        next_state <= {1'b0, state[6:0]};
    else if (bt[2]) // load content
        next_state <= sw;
    else if (bt[3]) // set every bit to 1
        next_state <= 8'd255;
    else // maintain value
        next_state <= state;
end

assign ld = state;
```

# Task 2: Multifunction register

- Save all changes, and generate the programming file.
- Download it to the FPGA.
- Check functionality:
  - Add clock (5 Hz)
  - Reset
  - Load a value
  - Shift to the left/right
  - Set every bit to 1
  - Reset again
  - Etc.

# Task 3: Multifunction register

- Try to modify the code on your own:
- Instead of simply shifting to the left and to the right, rotate the value of the state register in both directions.
- When no button is pressed, invert every bit of the register content instead of maintaining its value.