

Xilinx ISE tutorial

(Szántó Péter, 2009-09-23)

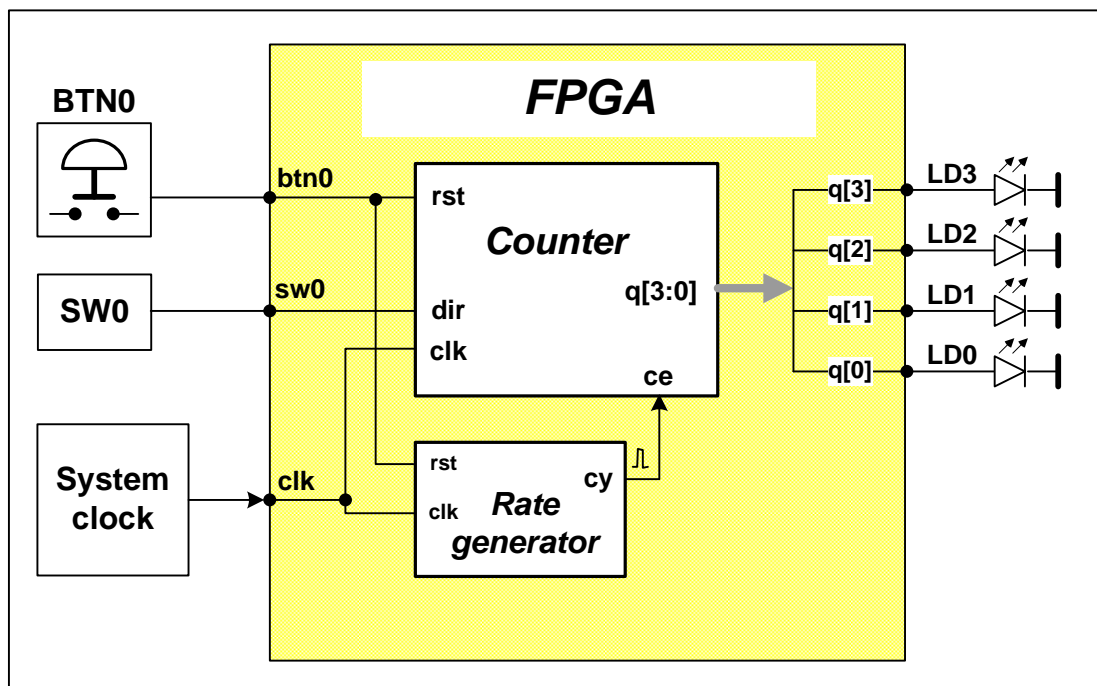
1. Tutorial project

For the Xilinx ISE tutorial the goal is to implement a one-digit BCD (binary coded decimal) second counter. That is, the counter counts from zero to nine and increments once every second. The actual value is displayed on 4 LEDs. Based on the state of the SW0 switch, the counter counts upwards or downwards.

During Measurement Laboratory 1., it is required to design synchronous systems, that is all flip-flops in the design should operate using the 50 MHz clock input available on the development board. If part of the design should operate less frequently (just like the counter in this tutorial), an enable signal should be generated. For example, for the second counter this enable signal has a frequency of 1 Hz – it takes value of “1” once in every second for exactly one system clock cycle long. That is, the enable signal is “0” for 49,999,999 clock cycles, then it goes to “1” for a single clock cycle.

To be able to set the system into a known state, an external reset signal should be present.

The schematic diagram of the system to be designed is shown below. The counter module is the BCD counter, while ratagen generates the necessary enable signal for the counter which was described above. These two submodules are connected together in the top-level module.



2. Creatimng the project

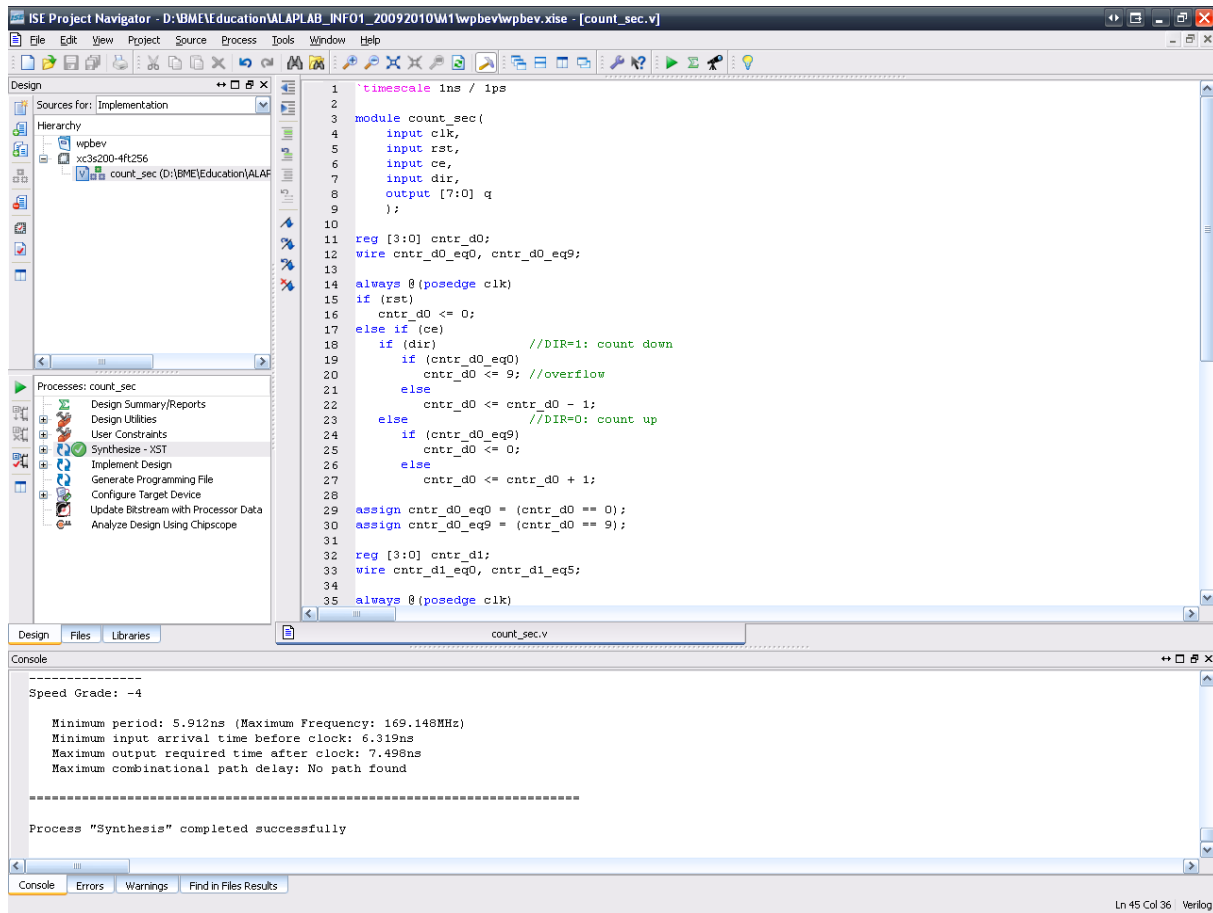
2.1 Starting Xilinx ISE

Click on the Xilinx Project Navigator icon on the desktop (or use the start menu item: **Programs / Xilinx 11.2 / ISE / Project Navigator**).

The main Window of the Project navigator consists of 4 windows:

- Top left: **Sources** window: sources of the current project. In the drop-down menu you can select to show source files for Implementation or for Simulation.
- **Processes** window shows the allowed processes to be ran using the source file selected in the Sources window.
- The **Editor** window is the Xilinx text editor to edit different kinds of source files.

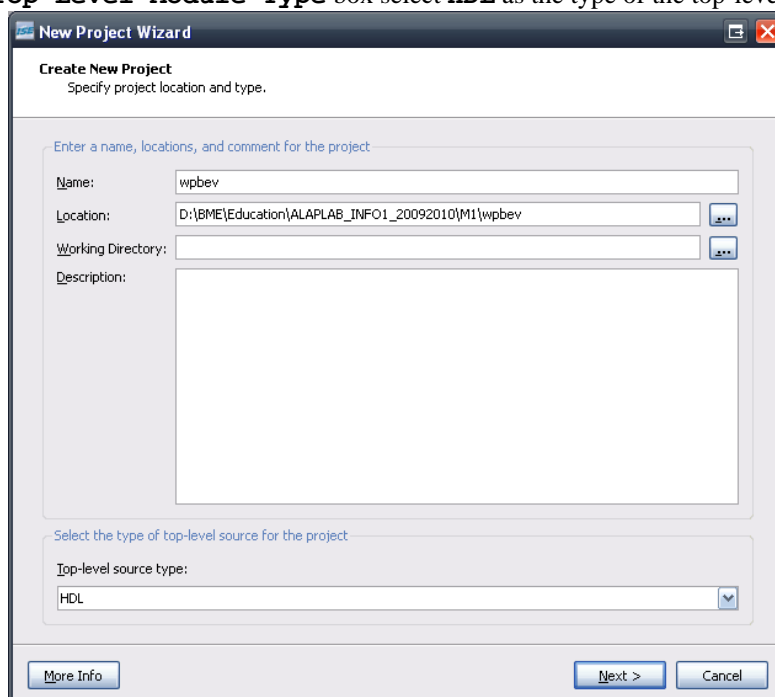
- At the bottom, the **Transcript** window shows messages from the tool. The **Console** window lists all messages, while **Warnings** and **Errors** lists only the selected messages..



2.2 Creating a new project

In **File** menu, select **New Project** option.

- Set **Project Name** to `wpbev`. Create a folder on drive D for the project (do not use space or any special character in the folder name, otherwise ISE does not work correctly!).
- In the **Top-Level Module Type** box select **HDL** as the type of the top-level module.



After pressing the **Next** button, set the FPGA type:

Device Family: Spartan3

Device: xc3s200

Package: ft256

Speed Grade: -4

Synthesis Tool: XST (VHDL/Verilog)

Simulator: ISim (VHDL/Verilog)

New Project Wizard

Device Properties
Specify device and project properties.

Select the device and design flow for the project

Property Name	Value
Product Category	All
Family	Spartan3
Device	XC3S200
Package	FT256
Speed	-4
Top-Level Source Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	VHDL
Property Specification in Project File	Store non-default values only
Manual Compile Order	<input type="checkbox"/>
Enable Enhanced Design Summary	<input checked="" type="checkbox"/>
Enable Message Filtering	<input type="checkbox"/>
Display Incremental Messages	<input type="checkbox"/>

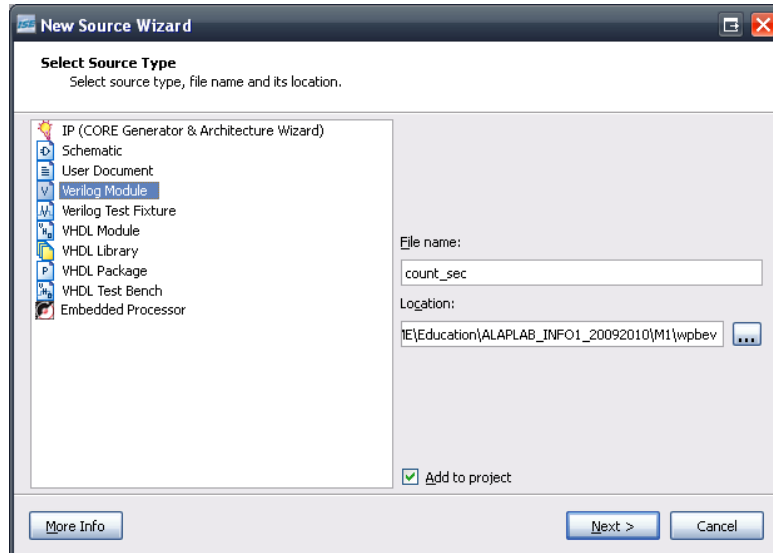
[More Info](#) [< Back](#) [Next >](#) [Cancel](#)

After press **Next** button, ISE offers the creation of new source files, but at the moment this option is not used. Also skip the **Add Existing Sources** option, and press **Next** then **Finish** to finalize the creation of the empty project..

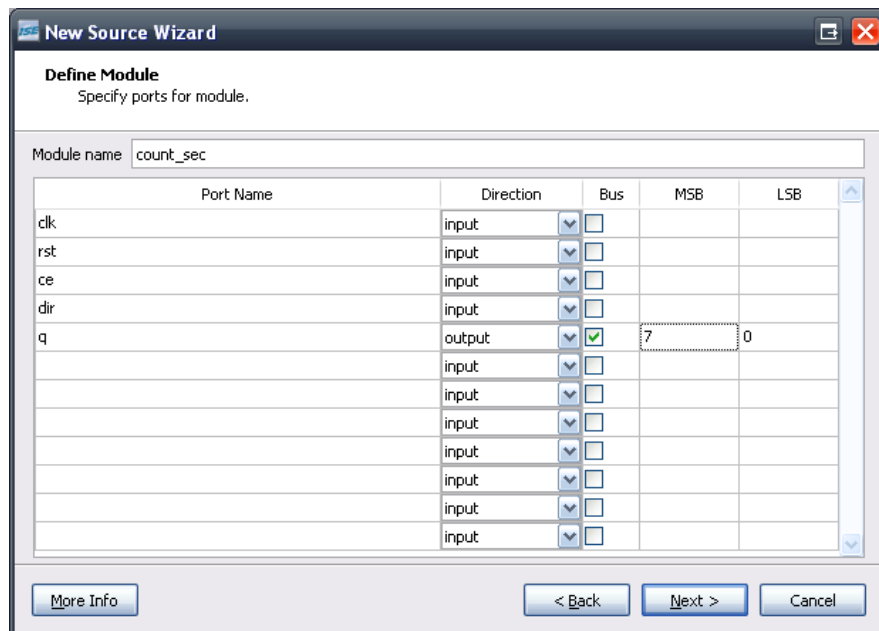
3. Creating the Verilog codes

3.1 Creating the counter

To create a new module select **Project/New Source**. In the menu select **Verilog Module**. The name of the module should be *count_sec*, (**File Name** window), and click the **Add to Project** option. After pressing the **Next** button you can define the ports of the module.



Using the schematic diagram above, create the necessary ports for the counter.



Press **Next**, **Finish** to see the module declaration in the editor window:

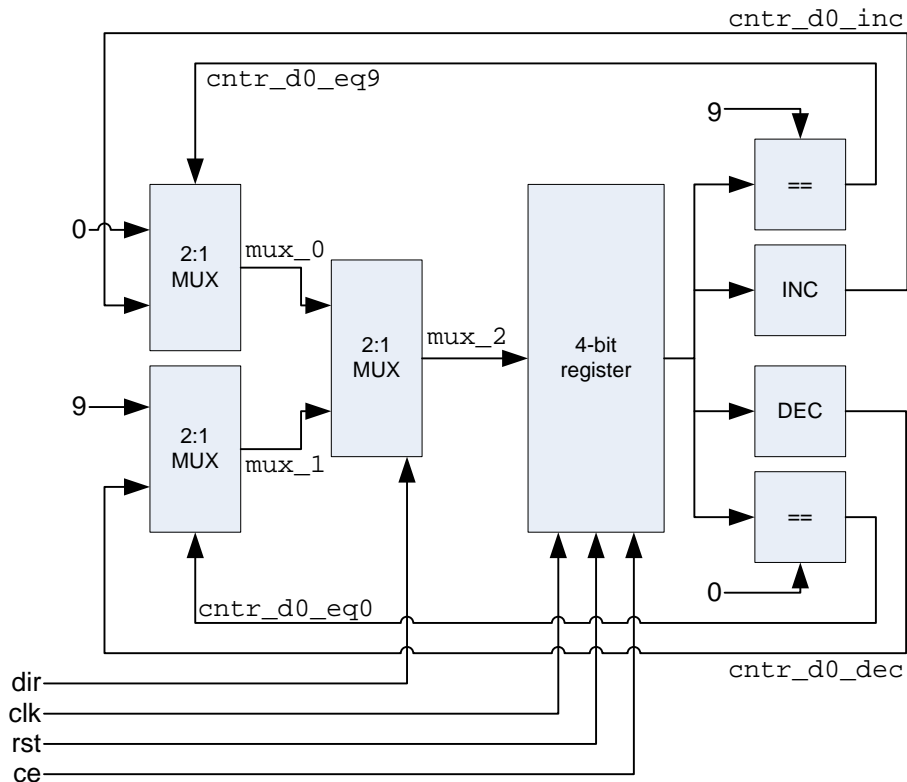
```
module count_sec(  
    input clk,  
    input rst,  
    input ce,  
    input dir,  
    output [3:0] q  
);  
  
endmodule
```

3.2 A sablon szerinti leírás módosítása

The counter is realized with a 4-bit register (D FF), which has reset and enable inputs. The INC and DEC blocks generates the incremented and decremented values based on the register output. The content of the register is update only when the enable input (ce) is '1'. Based on the actual value, the next value should be::

- counting upwards (*dir* = '1')
 - if the counter reached its end value (9): 0
 - otherwise actual value + 1
- counting downwards (*dir* = '0')
 - if the counter reached its end value (0): 9
 - otherwise actual value - 1

To detect the end-value, two combinatorial comparators are used, which compare the output of the register with 9 and 0. Based on these, the schematic diagram of the counter looks as follows:



In Verilog it is possible to describe the exact same structure (for the register reg typed variables are declared, for the combinatorial part wire variables are used).

```
reg [3:0] cntr_d0;
wire [3:0] cntr_inc, cntr_dec;
wire [3:0] mux_0, mux_1, mux_2;
wire cntr_d0_eq0, cntr_d0_eq9;

always @(posedge clk)
if (rst)
    cntr_d0 <= 0;
else if (ce)
    cntr_d0 <= mux_2;

assign cntr_d0_inc = cntr_d0 + 1;
assign cntr_d0_dec = cntr_d0 - 1;
assign cntr_d0_eq0 = (cntr_d0 == 0);
assign cntr_d0_eq9 = (cntr_d0 == 9);
assign mux_0 = (cntr_d0_eq9) ? 0 : cntr_d0_inc;
```

```
assign mux_1 = (cntr_d0_eq0) ? 9 : cntr_d0_dec;
assign mux_2 = (dir) ? mux_0 : mux_1;
```

The only *always* block contains the rising edge of the clock signal in its event list, therefore after implementation D FFs will be generated. Updating the register content is allowed only if the enable signal (ce) is '1'.

Verilog is a relatively high level language, therefore it is possible to create a functionally equivalent, but more user-friendly code.

```
reg [3:0] cntr_d0;
wire cntr_d0_eq0, cntr_d0_eq9;

always @(posedge clk)
if (rst)
    cntr_d0 <= 0;
else if (ce)
    if (dir)                //DIR=1: count up
        if (cntr_d0_eq9)
            cntr_d0 <= 0; //overflow
        else
            cntr_d0 <= cntr_d0 + 1;
    else                    //DIR=0: count down
        if (cntr_d0_eq0)
            cntr_d0 <= 9;
        else
            cntr_d0 <= cntr_d0 - 1;

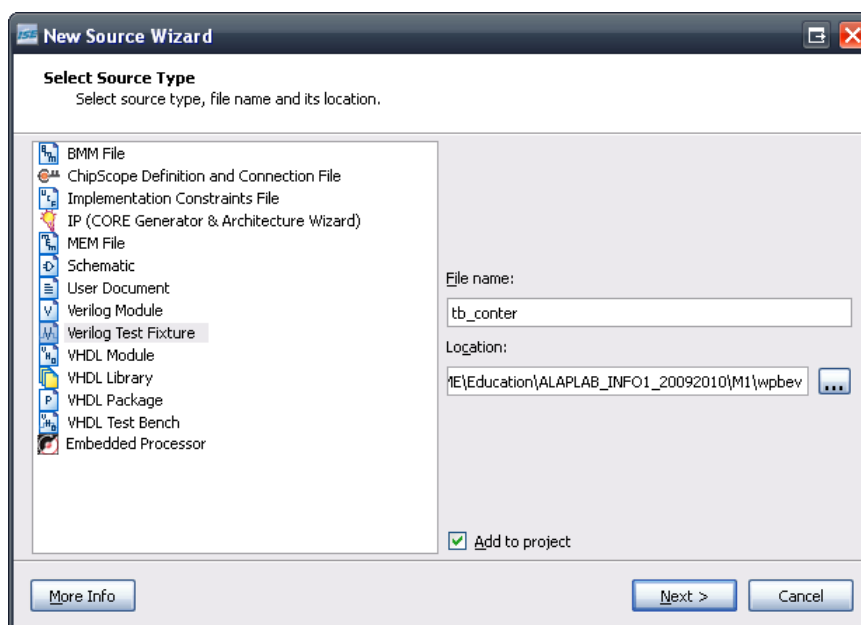
assign cntr_d0_eq0 = (cntr_d0 == 0);
assign cntr_d0_eq9 = (cntr_d0 == 9);
```

After creating the Verilog code, select the count_sec module in the Sources window. In the Processes window double-click on the **Synthesize-XST / Check Syntax** which runs syntax checking. If there are errors, correct them before moving on.

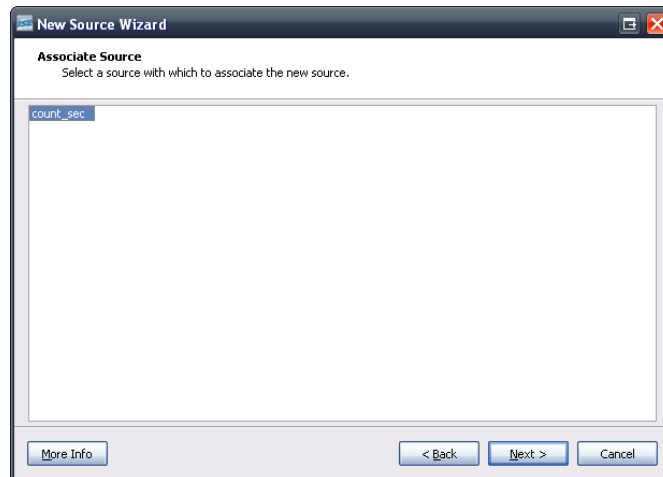
4. Simulation

4.1 Creating the testbench

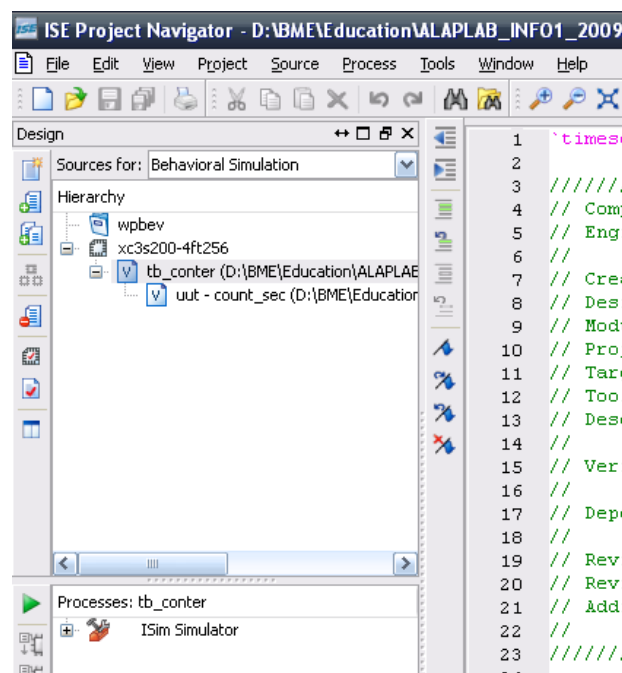
To be able to functionally verify the module you have just created you have to generate input signals for the module. This is done in the so-called testbench. First, create a Verilog Test Fixture using **Project / New Source** and selecting **Verilog Test Fixture** as the type of the source.



In the next window select the module you want to create a testbench for. In our case we have only one module (count_sec), so select it.



Press **Next**, **Finish** to have the file generated. In the top-left drop-down menu select **Behavioral Simulation** to see the source which are only valid for simulation. As you can see, the module to be tested (count_sec) is a submodule for the testbench (tb_counter).



4.2 Creating input signals

The automatically generated testbench file contains the following:

- instantiation of the module to be tested with the name UUT (unit under test)
- reg type variables for the input signals
- wire type variables for the output signals
- all inputs are set to 0 at the beginning of the simulation (initial block)

```
`timescale 1ns / 1ps
module tb_counter;

// Inputs
reg clk;
reg rst;
reg ce;
reg dir;

// Outputs
```

```

wire [3:0] q;

// Instantiate the Unit Under Test (UUT)
count_sec uut (
    .clk(clk),
    .rst(rst),
    .ce(ce),
    .dir(dir),
    .q(q)
);

initial begin
    // Initialize Inputs
    clk = 0;
    rst = 0;
    ce = 0;
    dir = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here

end

endmodule

```

As the counter contains sequential logic, a clock signal should be generated:

```

always #5
    clk <= ~clk;

```

For the other inputs generate the following waveforms.

- set *rst* to '1' during 7 – 27 ns
- set *ce* to '1' after 107 ns
- set *dir* to '1' after 1007 ns

These can be generated with the following code:

```

initial
begin
    #7 rst <= 1;
    #20 rst <= 0;
end

initial
    #107 ce <= 1;

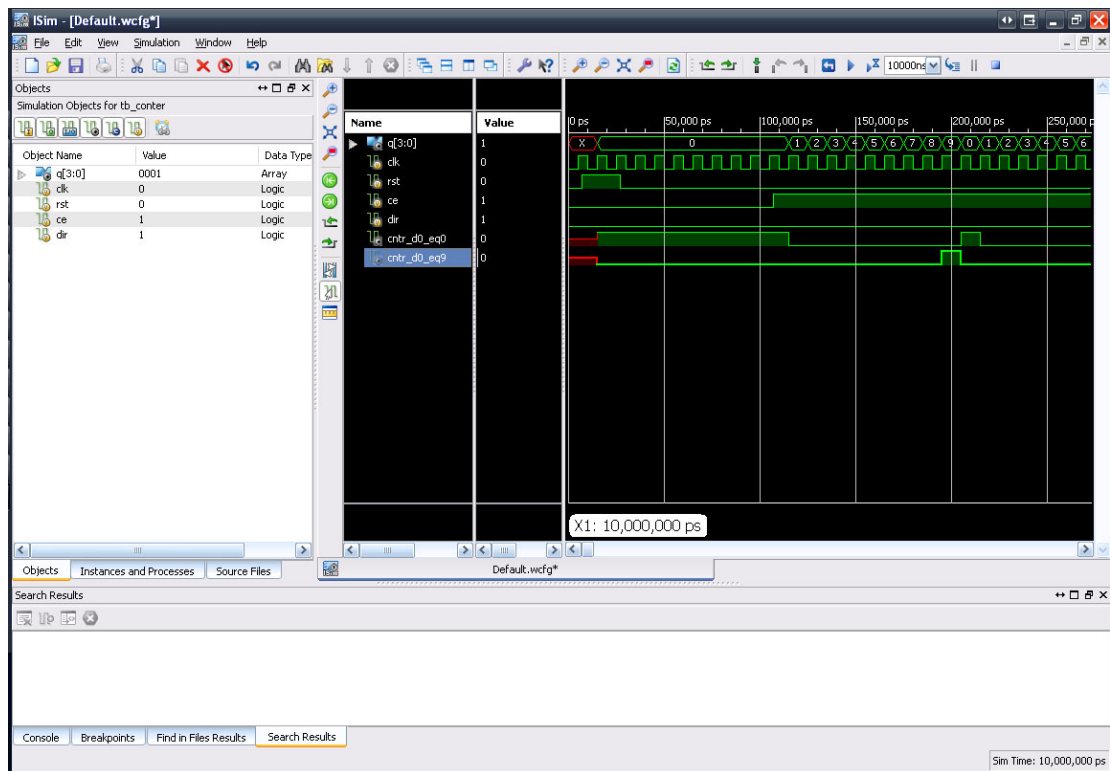
initial
    #1007 dir <= 1;

```

In the Sources window select the testbench, then in the processes window double click on the **Simulate Behavioral Model** option to start the simulator.

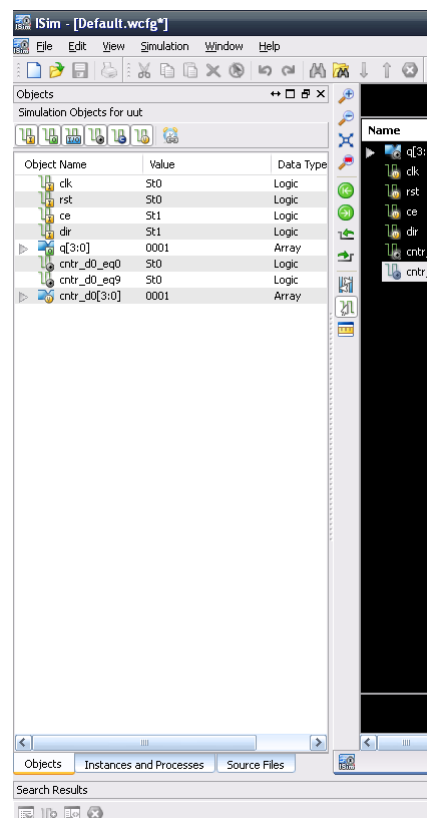
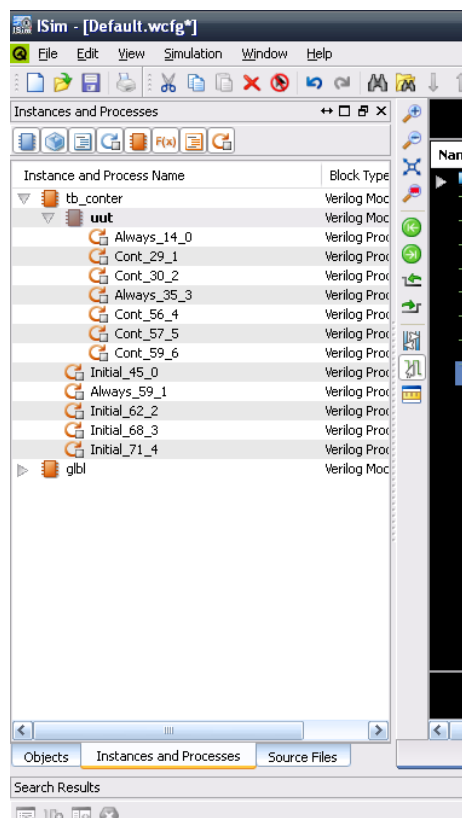
4.3 Using ISE simulator

Simulation results are presented in a new window. Note, that before the reset, registers contain undefined values (red X).



By right clicking on the signals you can change the display format – set q[3:0] to hexadecimal.

The simulator only shows the inputs and output automatically. However, often it is necessary to inspect internal signals of the system. Internal signals can be added to the simulator waveform without modifying the Verilog code. Select **Instances and Processes** tab in the left window to see the testbench hierarchy. In the hierarchy select the counter instance (UUT), then select the Objects tab. Here you can see all internal signals of the counter, which can be drag-and-dropped into the waveform window. After restarting the simulation (restart/play icons at the top) you will see the values of the newly added signals.

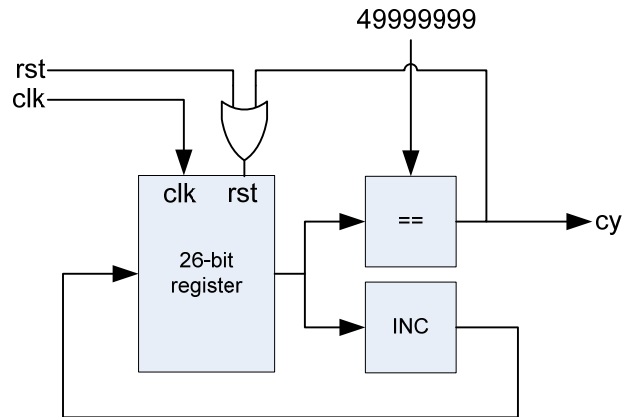


4.4. Rategen module

Create a new module with a name rategen (**Project / New Source**).

The rate genertaror which generates the enable signal for the counter is basically a counter which counts from 0 to 49,999,999 to be able to have 50,000,000 different states. To be able to do that, a 26 bit counter is required. The end-value of this counter will be the enable signal for the BCD counter.

The schematic diagram is:



The corresponding Verilog code:

```
module rategen(
    input clk,rst,
    output cy
);
    //Generate 1 clock wide pulse on output CY
    reg [25:0] Q;

    always @(posedge clk)
    begin
        if (rst | cy)
            Q <= 0;
        else
            Q <= Q + 1;
        end

    assign cy = (Q == 49999999);
    //assign cy = (Q == 4);
endmodule
```

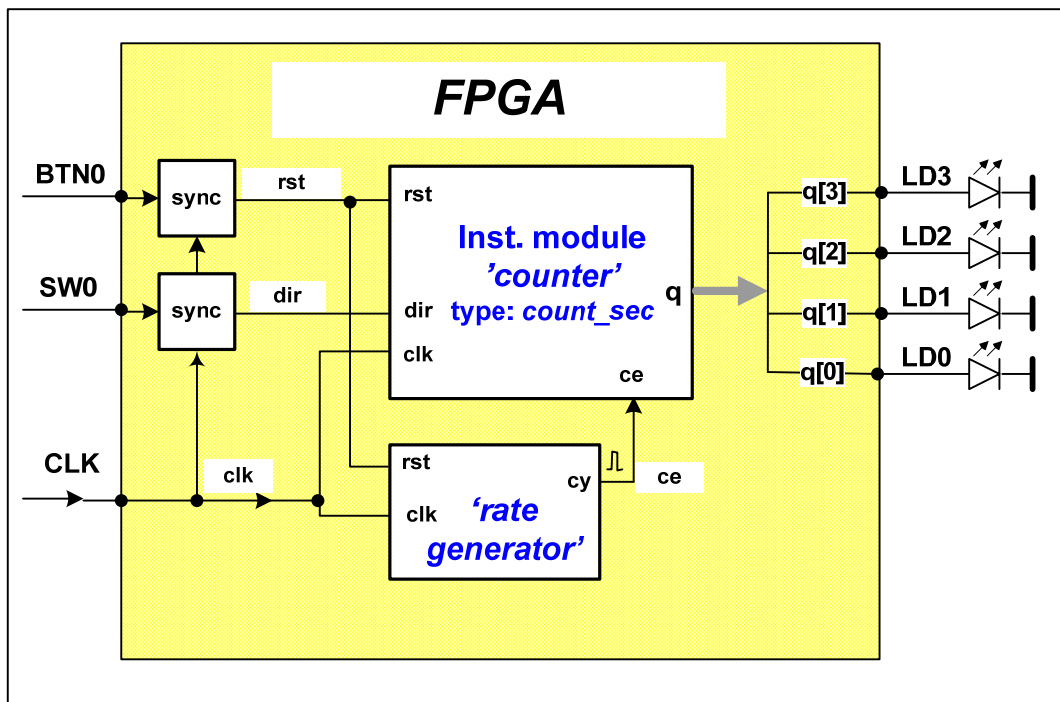
To make simulation faster you can use a more frequent enable signal which is shown commented in the code above.

4.5. Top-level module

The top-level module describes the connection between its sub-modules and connection to the outer world (that is, connection to the FPGA pins).

Create a new Verilog source named wpbevtop1 (**Project / New Source / Verilog module**).

To define its ports, take a look at the schematic below.



The top level declaration therefore should look like the code below:

```
module wpbevtop1(
    input clk, btn0, sw0,
    output [3:0] q
);
```

The external asynchronous signals (button, switch) are synchronized to the system clock by sampling them into registers:

```
reg rst, dir;
always @(posedge clk)
//Synchronize inputs
begin
    rst <= btn0;
    dir <= sw0;
end
```

The next task is to instantiate the submodules, namely create one instance from rategen and one instance from count_sec.

Instantiation of rategen:

```
wire ce;
rategen rategenerator(
    .clk(clk),
    .rst(rst),
    .cy(ce)
);
```

Instantiation of the counter:

```
count_sec counter(
    .clk(clk),
    .rst(rst),
    .ce(ce),
    .dir(dir),
    .q(q)
);
```

So, with the input synchronization and all submodule instantiation the top-level module code:

```

module wpbevtop1(
    input clk, btn0, sw0,
    output [3:0] q
);

reg rst, dir;
always @(posedge clk)
//Synchronize inputs
begin
    rst <= btn0;
    dir <= sw0;
end

wire ce;
rategen rategenerator(
    .clk(clk),
    .rst(rst),
    .cy(ce)
);

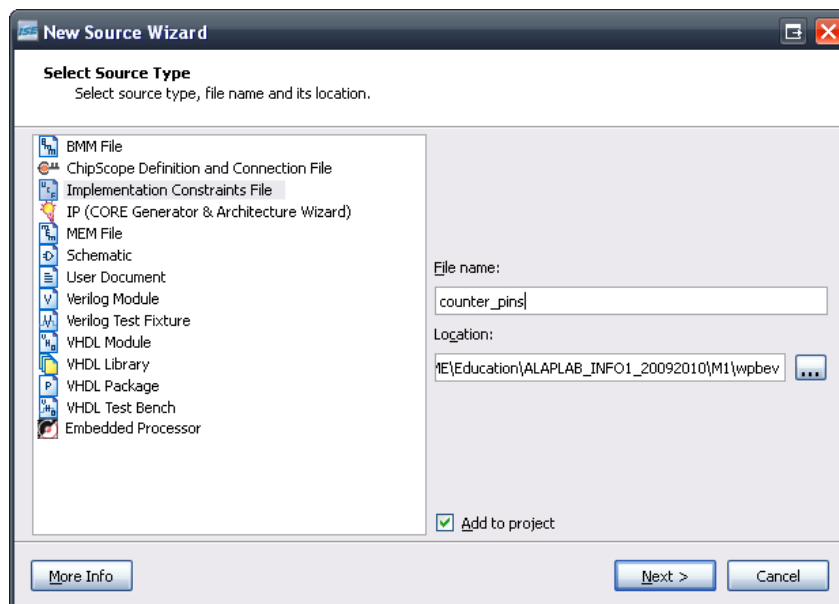
count_sec counter(
    .clk(clk),
    .rst(rst),
    .ce(ce),
    .dir(dir),
    .q(q)
);

endmodule

```

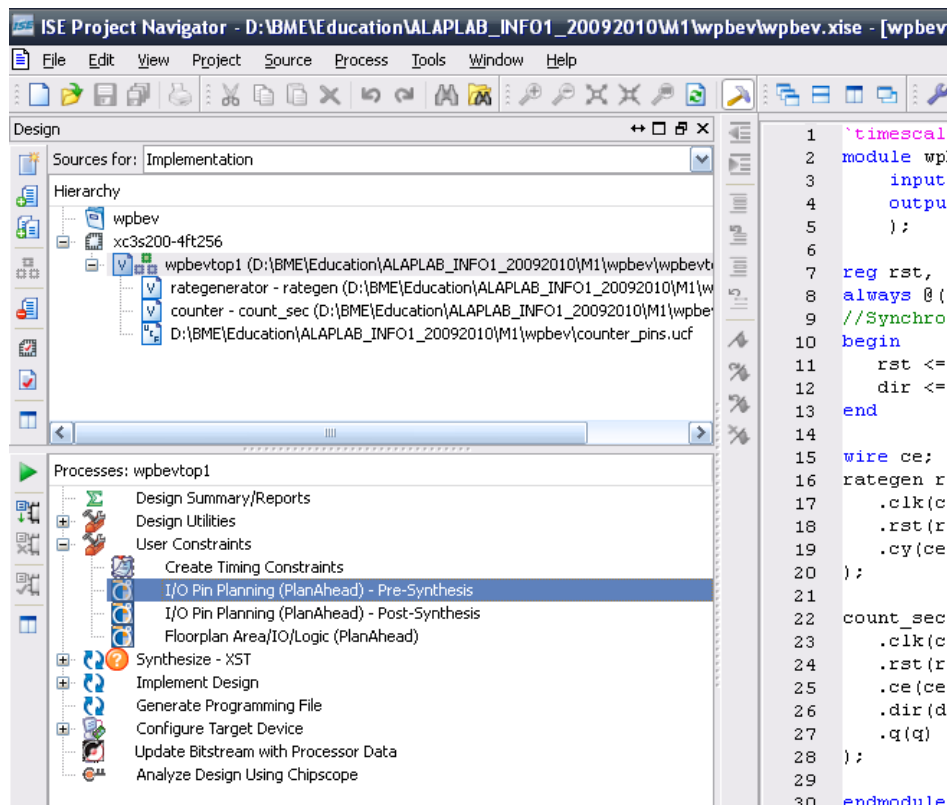
The ports of the top-level module are connected to the pins of the FPGA. The port – pin mapping can be done with a Xilinx tool, called PlanAhead.

First, create an Implementation Constraint File (.ucf) with **Project / New Source** menu, and select **Implementation Constraint File**. Name it to *counter_pins*.

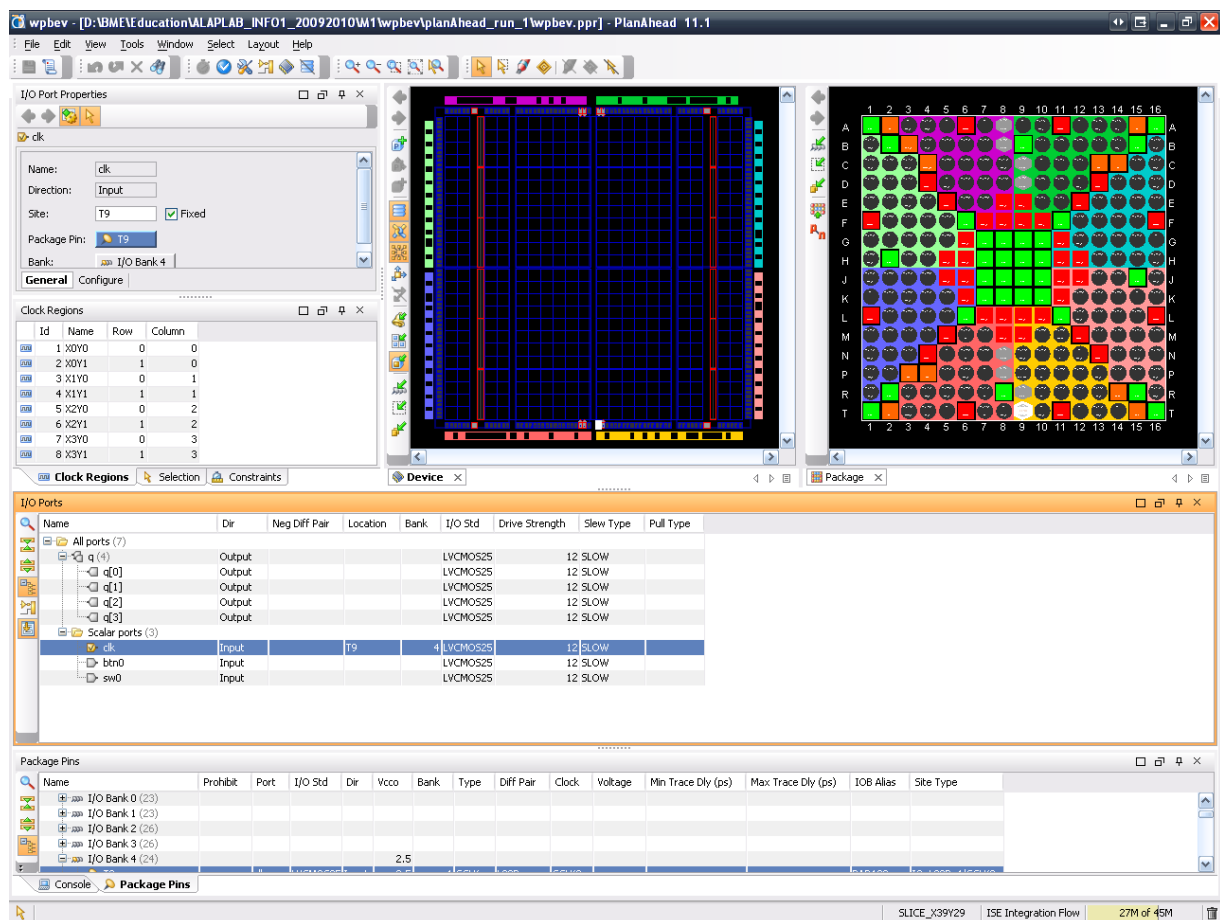


Click **Next/Finish** to finish the file generation. In the top-left drop-down menu select Implementation to verify that the *counter_pins.ucf* file is added to the project.

Select the top-level module in the sources Window. In the Processes window click on **User Constraints / I/O Pin Planning (PlanAhead) – Pre-Synthesis**.



This starts PlanAhead.



In the I/O Ports window you can see all ports of *wpbevtop1* module. Pin names should be typed into the LOC column (or alternatively, select the port and type in the pin name in the top-left window called I/O Properties).

The peripheral – pin name associations can be read from the development board (e.g. clock is connected to pin T9).

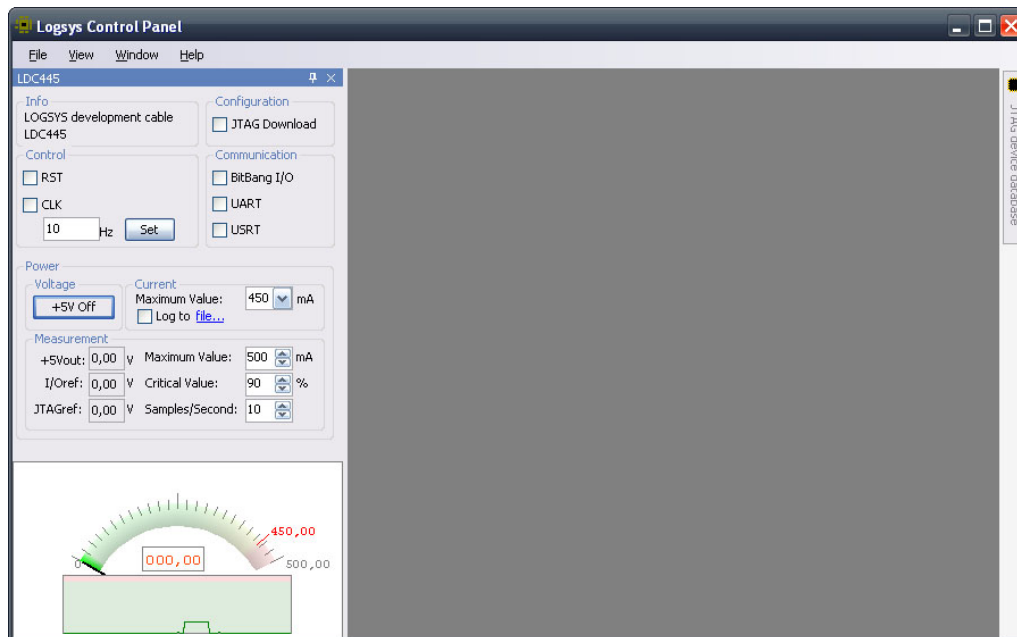
After typing in all pi names, save the PlanAhead project and quit. In ISE **Sources** window select the *counter_pins.ucf* file, then in the **Processes** menu click on **Edit Constraints (Text)** option to see the syntax of the UCF file.

```
NET "clk" LOC="t9" ;
NET "btn0" LOC="m13" ;
NET "sw0" LOC="f12" ;
NET "q<3>" LOC="n14" ;
NET "q<2>" LOC="l12" ;
NET "q<1>" LOC="p14" ;
NET "q<0>" LOC="k12" ;
```

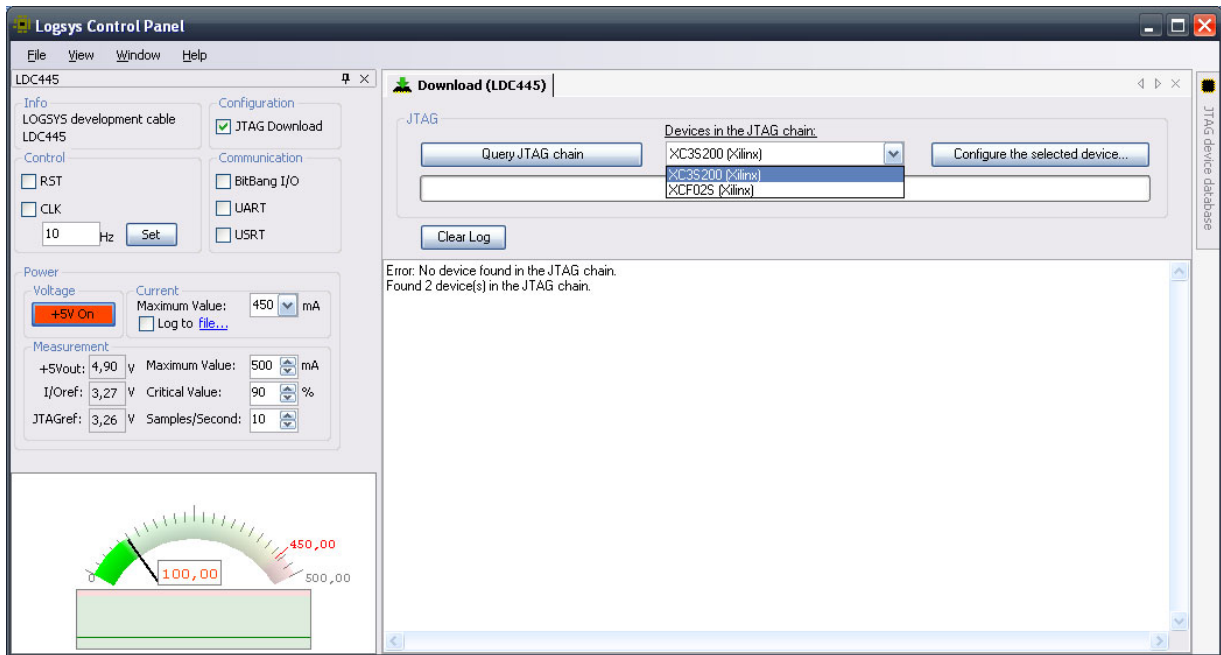
5. Implemting the design

Select the top-level Verilog module in the Sources window. Then, in the Processes window click on the Generate Programming File option to implement the design and generate the configuration file.

To download the generated BIT file into the FPGA start LOGSYS GUI from the desktop.



- Power up the board by clocking on the **+5V Off** button.
- Check the box next to the **JTAG Download** text to have the configuration panel.



Click **Query JTAG chain**. You will see two devices: XC3S200 FPGA and XCF02S flash memory. In the drop-down menu select the FPGA and then click on the **Configure the selected device** button. Select the bit file from your own ISE project directory.

Verify the operation of the design.

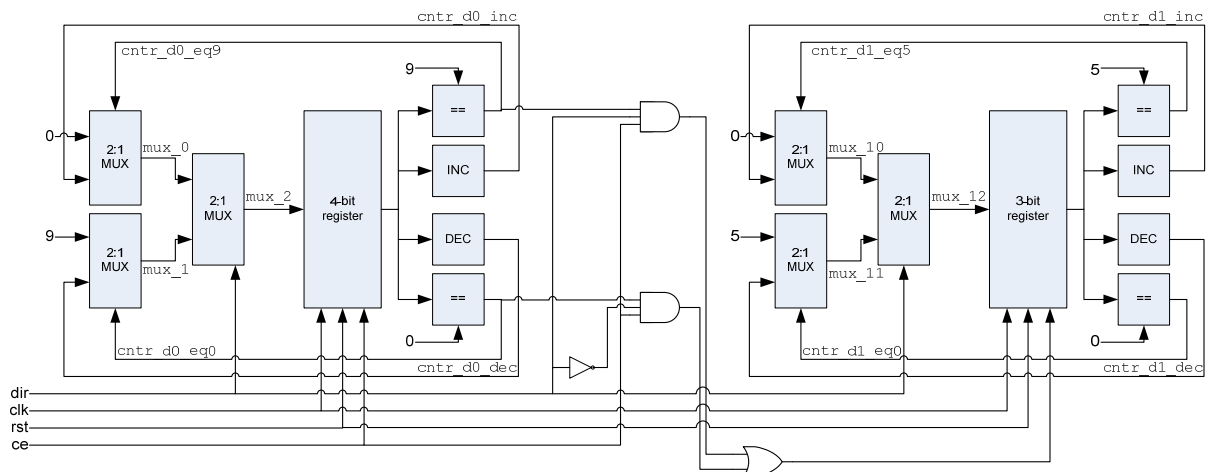
6. Modifying the counter

At this point you should have a correctly operating one-digit BCD counter. The next step is to modify the source code of the counter to have a real second counter which is able to count from 0 to 59, still in BCD format. For this, two BCD digits are required: the least significant counts from 0 to 9, while the most significant one counts from 0 to 5. Till now, you have created the LS digit.

Let's think about the operation of the MS digit. Unlike the LS digit, which counts whenever its enable signal is '1', the MS digit is only enabled when the LS digit reached its final value: 9 in the case of up-counting and 0 in the case of down-counting. Thus, the MS digit should be only changed:

- counting upwards ($dir='1'$), LS digit is 9 ($cntr_d0_eq9='1'$) and the external enable signal is active ($ce='1'$) OR
- counting downwards ($dir='0'$), LS digit is 0 ($cntr_d0_eq0='1'$) and the external enable signal is active ($ce='1'$)

Based on these, the resulting schematic diagram looks as follows:



Modify the Verilog sources to implement the 2-digit counter:

- In the count_sec module declare the necessary variables and describe the functionality of the MS digit IN A SEPARATE always block.
- Increase the output port width in the counter and in the top-module.
- Connect the new output bits to LEDs 6 – 4

Simulate the modified count_sec module.

Implement the design and configure the FPGA.