

Introduction to Xilinx ISE 14.7

(Péter Szántó, BME MIT 2024-09-02)

This guide is intended to introduce the reader to the ISE 14.7 development system. It is only an introduction, because the system is presented through a few very simple examples, and many of its features are not mentioned.

First, we will get some practice in the development process through two very simple tasks, and then we will design a BCD counter, specifying the operation of the unit in a hardware description language. The simulator is used to check (verify) the correctness of the design. A simple device consisting of several functional units is then designed. The operation of the device is described hierarchically in Verilog. The Verilog code is synthesized and mapped to the FPGA circuit by ISE implementation steps. Finally, the generated configuration file is downloaded to the FPGA circuit of the measurement panel and the device is tested.

1. Parts of the ISE system

Xilinx, a major manufacturer of programmable logic devices (PLDs and FPGAs), has developed a computer-aided design system for using these devices. The company has also developed a simpler but functionally complete version of the ISE system, called WebPACK. The WebPACK system is made available free of charge by Xilinx (www.xilinx.com), and the user only needs to register to download the software. The WebPACK system, of course, only supports implementations with Xilinx company ICs (but does not support all ICs of all families, typically only those of lower complexity). With the free software, the company obviously wants to facilitate the distribution of its circuits.

The parts of the ISE system are illustrated in *Figure 1*, which is taken from the Programmable Logic Design Quick Start Handbook [2], available from www.xilinx.com/univ/.

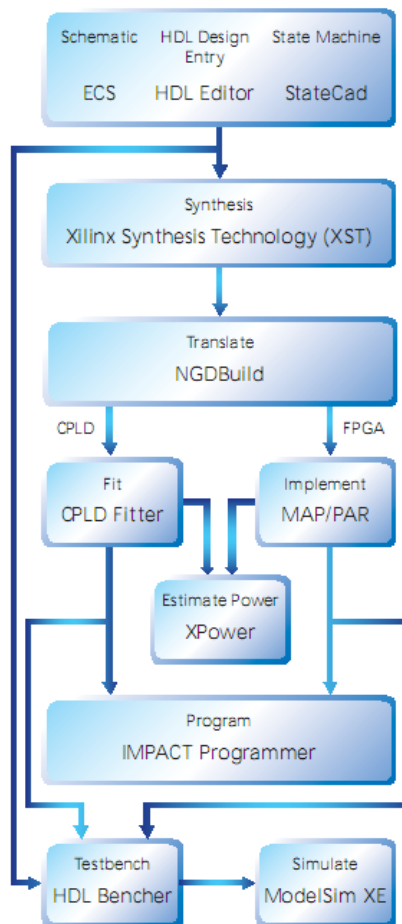


Figure 1 Design process with WebPACK

The operation of the sub-systems and parts of the ISE planning system is brought together by the Project Navigator software, the ISE framework.

Designers can enter their ideas and plans into the system in three different ways.

- You can enter it in the form of a schematic drawing, using Xilinx ECS, the schematic creator and input program. THE USE OF THIS SOFTWARE IS NOT RECOMMENDED.
- You can use hardware description language. This input is supported in the HDL editor section. The supported languages are Verilog and VHDL. The system also includes many sample codes in the form of templates.
- Other options are also available, such as using Xilinx designed modules (intellectual property – IP cores) or creating a processor-based system.

After the coding step is done, the next step is the verification of the design, where we check that the operation of the designed circuit matches the specification. Verification is usually done by simulation. The simulator for the WebPACK system is the Xilinx ISE Simulator.

For the simulation test, the model must be operated, "excited", i.e. the model inputs must be fed with appropriately varying signals. This is done by adding a series of so-called test vectors. The test vectors can be written in HDL by the designer as a test fixture - the graphical interface available in older ISE versions is no longer available.

Once the design is verified, the next step is synthesis, which is done by the Xilinx Synthesis Technology (XST) subsystem, which is also part of ISE (note: there are other programs for synthesis). The synthesizer generates from the HDL description a minimized and optimized netlist containing the hardware resources (for clarity, let's lie that they are logic gates and flip-flops) and their interconnections (note: actually Xilinx FPGAs contain LUTs and D FFs, but this is not very relevant now). This is followed by the Translate, Map and Place&Route phases (in short: implementation). Translate generates a file from the netlists and user constraints, Map maps this to the FPGA's primitive set, and finally Place&Route places the primitives in the device and establishes the physical connection between them.

The programming of the IC in the ISE after the generation of the programming file is controlled by the IMPACT subsystem, but since the boards used have their own drivers, we will use the LOGSYS GUI for the driver in the lab instead of this one.

2. Using the ISE system in the lab

2.1 The design process chosen

The ISE system is quite complex, which is also a consequence of the many services it provides. There is not enough time in the lab to get to know the system thoroughly. Of course, we cannot train an excellent FPGA designer in the basic lab, because it requires a lot of specialized knowledge and adequate practice. Interested students can get such training in the specialized courses and laboratories of the Faculty of Electrical Engineering.

The design procedure selected for the core lab is based on Verilog-based description and input of the design, i.e. it uses the HDL Design Entry subsystem.

After the Verilog code is created, the next step is functional verification by simulation.

After successful simulation, the implementation steps can be selected or started in the **Processes** window after selecting the top-level Verilog module.

For development, you need to specify the so-called User Constraints. In our case, this means specifying the Verilog port – FPGA pin mapping (other constraints may include timing parameters and the placement of each component on the chip).

Then the WebPACK synthesis (Synthesize) subsystem can be started, and the synthesized RTL description can be mapped to the given FPGA structure (Implement Design: Translate, Map, Place & Route).

Finally, the FPGA configuration file is created (Generate Programming File).

The design steps outlined above will be illustrated in the following with a sample tasks.

The design will follow the basic rules of correct design of digital networks, which is also expected when doing homework for the course.

2.2 Main requirements for correct design in this area

1. Design only synchronous sequential networks!

In the lab, we use a stricter constraint: only globally synchronous sequential networks are allowed to be designed, and the network can have only one clock domain. (This is necessary in the lab because students lack the necessary experience and thus have fewer problems processing the design.)

2. The primary (external) input signals of the synchronous network must be synchronized to avoid the occurrence of a metastable state in the system caused by the asynchronous nature of the input signal.

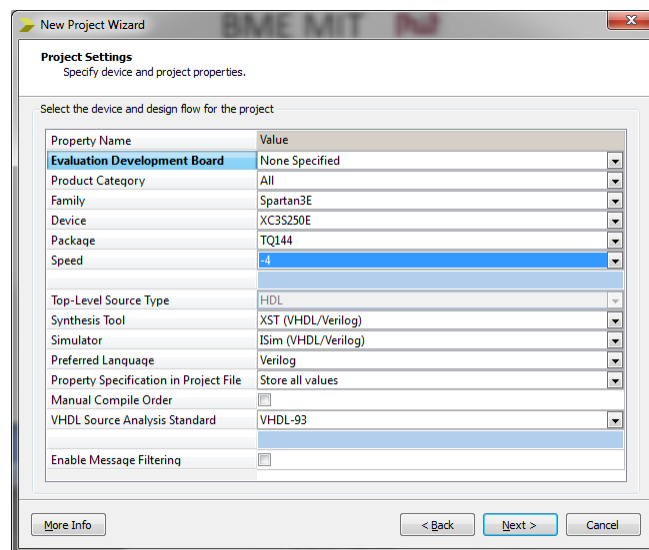
3. The design must be easy to verify. This topic is dealt with in a separate area of expertise, design for testability. For simpler networks, the most important requirement in this area is that **all storage elements and registers that define the internal state should have a reset signal.**

3. Creating a project

To start the ISE program, click on the ISE Project Navigator icon, or from the **Start** menu, **Xilinx Design Tools / 64-bit Project Navigator.**

To create a sample project, go to the **File** menu, select **New Project** and fill in the dialog box as shown below.

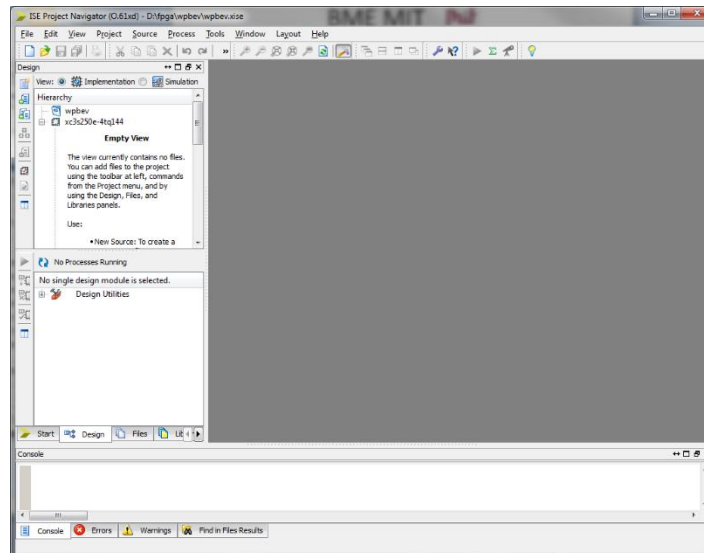
- The **Project Name** should be *wpbev*. The system will automatically create a folder with this name according to the path specified in the Project Location field. ISE will save the files for the project to this folder. Since ISE works with many files, the project is always stored locally, not on the server of the base lab!
- In the **Top-Level source type** field, select **HDL** from the drop-down box.



In the **Device Properties** box that appears after clicking **Next**, select the following values from the drop-down lists in the **Value** column (-- which appears when you click on the values --):

Device Family: Spartan3E
Device: xc3s250E
Package: tq144
Speed Grade: -4
Synthesis Tool: XST (VHDL/Verilog)
Simulator: ISim (VHDL/Verilog)

Clicking **Next** and then **Finish** will create the empty project.



There are 4 windows on the main Project Navigator screen:

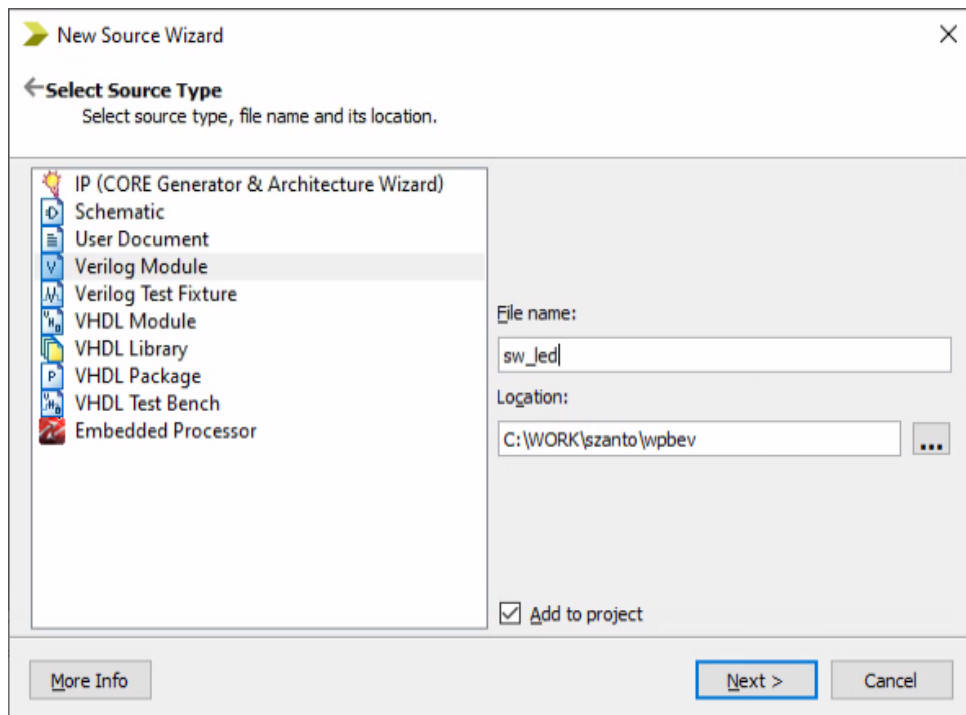
- At the top left is the **Hierarchy** (project sources) window, which shows the design files for the project and their relationship to each other. Here you can also select the type of files you want to see: the **Implementation** option shows only the files of the modules for implementation, while **Simulation** shows the testbench files for simulation.
- **Processes** window, which shows which processes can be executed on the design file selected in the **Hierarchy** window above it.
- To the right of the above is the **Editor** window, where you can view and edit the various design files.
- At the bottom is the **Console** (message) window, which contains the log files of the currently running or running design process. In this window, you can choose between four different views with the tabs. The **Console** view shows the complete messages. Very useful are the **Warnings** and **Errors** views, which extract warnings and error messages from the long log files.

3. Displaying the status of switches on the LEDs

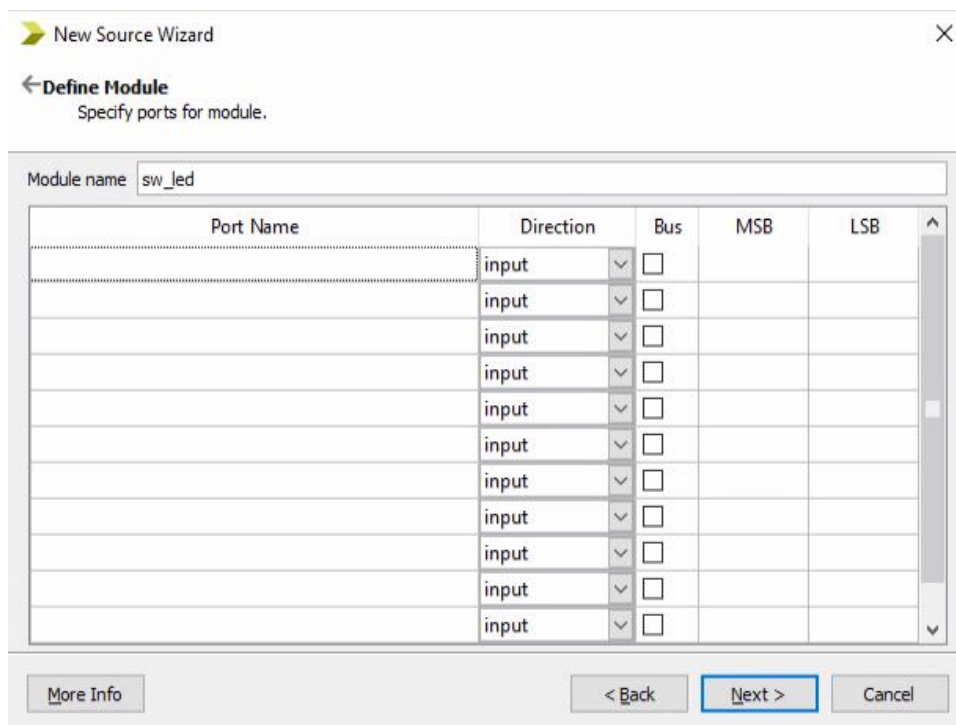
In our simplest example, the status of the 8 switches on the development board is displayed on the 8 LEDs. A given LED will light up when the associated switch is turned on, so using the FPGA we are essentially creating a quite expensive wire.

3.1 Verilog source code

To create a new module, from the **Project** menu, click **New Source** and select **Verilog Module** as the source. The name of the module to be created should be `sw_led`, which should be entered in the **File Name** window, and select **Add to Project**. After clicking **Next**, the **Define Verilog Source** wizard will prompt you to edit the port list, which we won't use now, so **Next**.



The port list is not filled out in this step, it will be written directly to the source code.



In the port list of the module there is one 8-bit input (switch - sw) and one 8-bit output (LEDs - led). The module is very simple: we assign the value of the input sw to the output led.

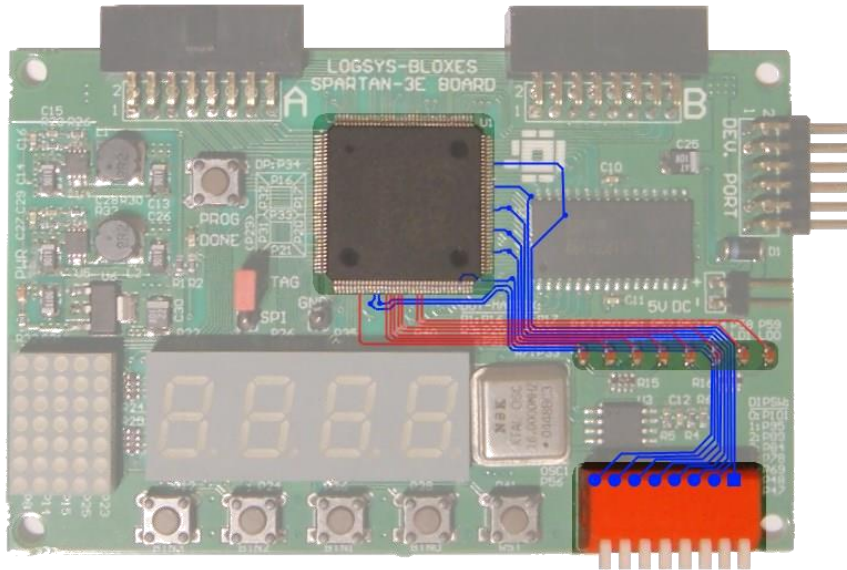
```

module sw_led(
    input [7:0] bw,
    output [7:0] led
);
assign led = sw;
end modules

```

3.2 Verilog port - FPGA pin assignment

In addition to the Verilog code, we also need to tell the development environment which ports in the Verilog code should be connected to which pin of the FPGA. This depends on the design of the printed circuit board (PCB), so it should be read from the documentation of the board; or in our case it is written onto the PCB. To illustrate the current design:



To perform the pin assignment, a constraint file is added to the project. Select the **Project / New Source** menu item, and in the window that pops up, select **Implementation Constraint File** and choose pins as the name.

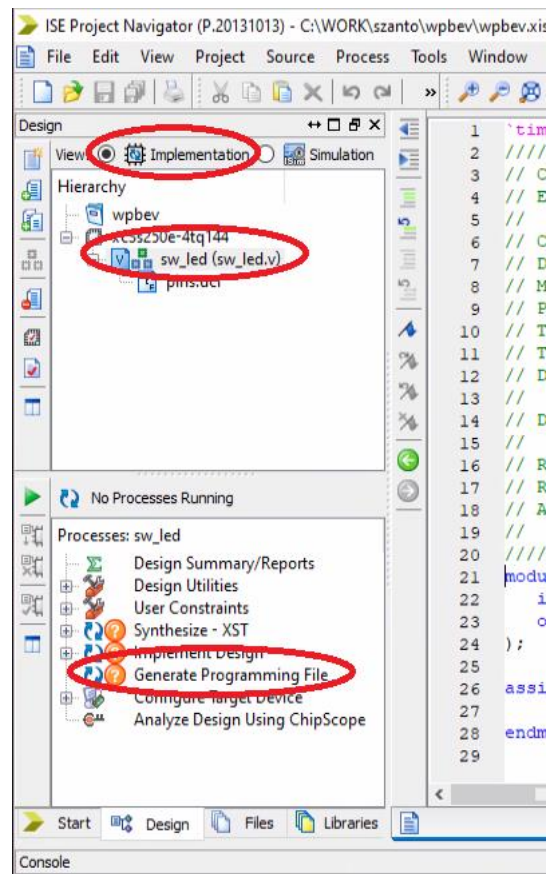
After pressing the appropriate amount of **Next/Finish** buttons, the *pins.ucf* file will appear in the **Sources** window. Here, we need to specify the corresponding pin for each port bit (i.e. all eight bits in the case of our 8-bit signal). If you have successfully found the names of the pins used from the panel, you will get a *ucf* file quite similar to the one below.

```
NET "sw[0]" LOC="p101";
NET "sw[1]" LOC="p95";
NET "sw[2]" LOC="p89";
NET "sw[3]" LOC="p84";
NET "sw[4]" LOC="p78";
NET "sw[5]" LOC="p69";
NET "sw[6]" LOC="p48";
NET "sw[7]" LOC="p47";

NET "led[0]" LOC="p59";
NET "led[1]" LOC="p58";
NET "led[2]" LOC="p54";
NET "led[3]" LOC="p53";
NET "led[4]" LOC="p52";
NET "led[5]" LOC="p51";
NET "led[6]" LOC="p50";
NET "led[7]" LOC="p43";
```

3.3 Implementation

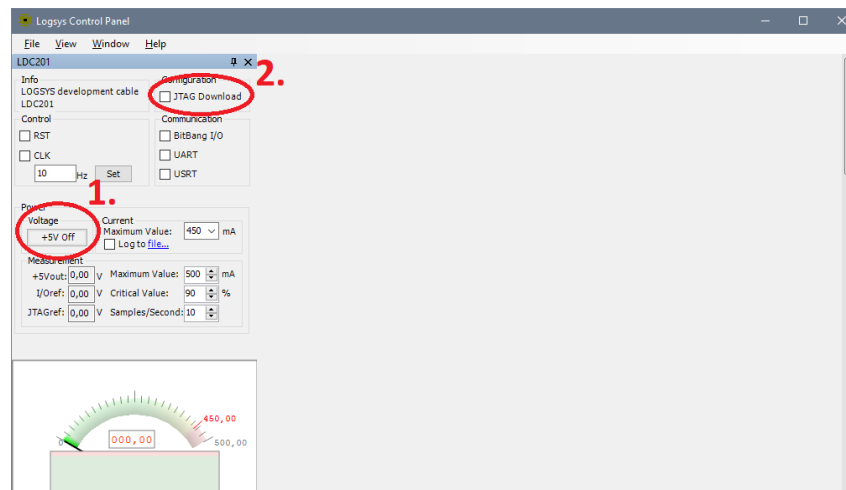
To realize this, the HDL description must first be synthesized. In the **Hierarchy** window, select the top-level module of the Verilog module hierarchy (we currently have only one module, so that's it). Then, from the **Processes** window, start the **Synthesize** procedure. After the synthesis is finished, you need to map the synthesized design to the FPGA structure (**Implement Design**), and finally you need to generate the file needed to program the FPGA IC (**Generate Programming File**). Just click on **Generate Programming File**, if the previous implementation steps are missing, they will be automatically executed by ISE.



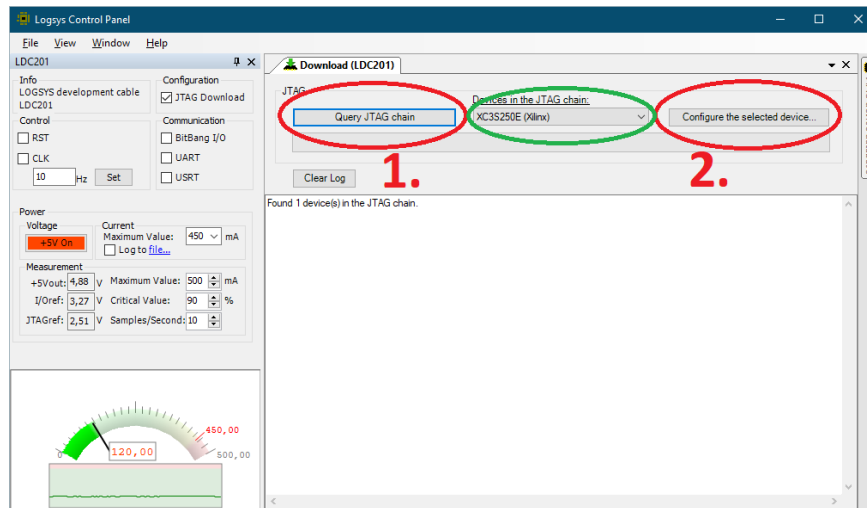
3.4 FPGA configuration

Once the programming file is generated, the FPGA should be programmed with the hardware structure generated during the translation of the design. For this step we use the Logsys GUI application.

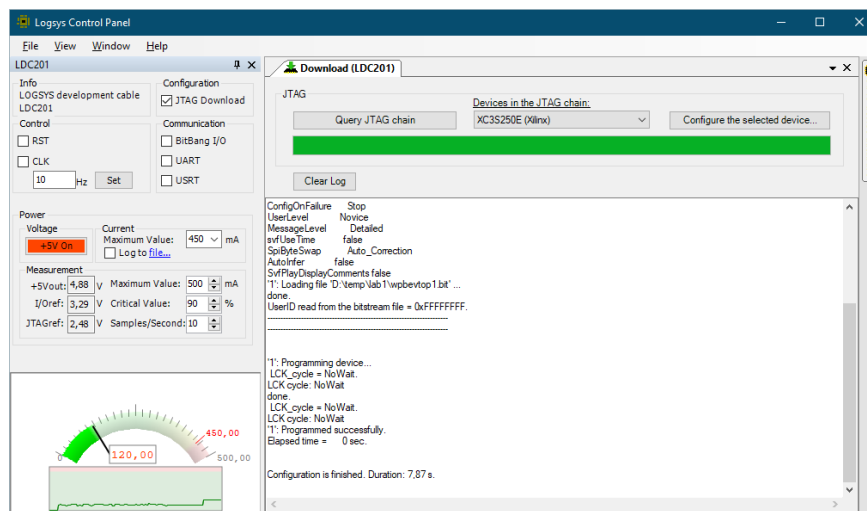
If the FPGA board is connected to the USB port of the PC, the first thing to do after booting is to turn on the power supply and then select the "JTAG Download" option.



The configuration window will then appear, where you must first click on the "Query JTAG Chain" button. The devices are then searched for and the "Devices in the JTAG Chain" menu will display the found XC3S250E type FPGA. After successful detection, click on "Configure the selected device..." and select the configuration (.bit) file generated by ISE.



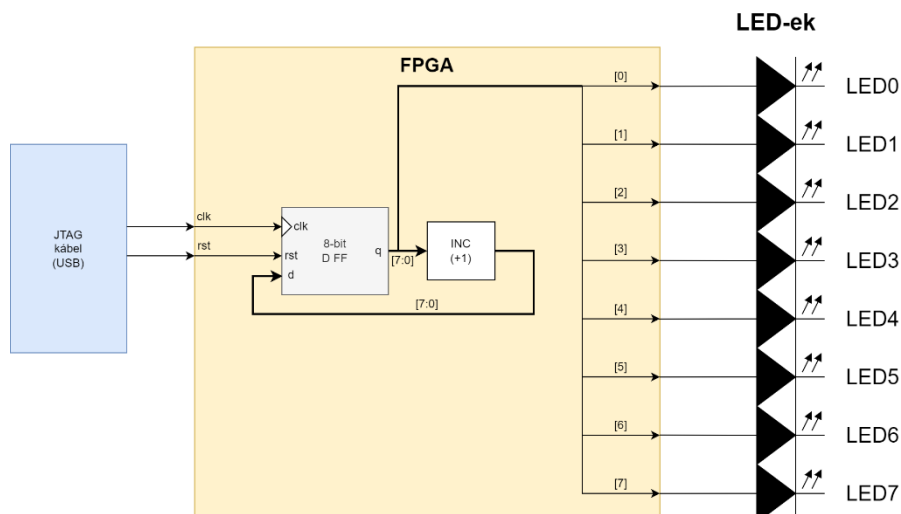
The successful configuration is reported in the Log window.



After programming, check the operation of the device, i.e. check that the LEDs turn on/off when the switches are adjusted.

4. Binary counter display on LEDs

In our second project, we are implementing a sequential network. To do this, we first remove the file `sw_led.v` from the project (right-click on the file, then "Remove"). Block diagram of the project:



4.1 Verilog source code

Then, as in the previous point, in the **Project** menu, click on **New Source** and select **Verilog** Module to create an empty Verilog file named `bin_cntr` (binary counter).

Since we have 8 LEDs, we will create an 8-bit counter. In addition to the 8-bit output, since it is a sequential network, we will need a clock (`clk`) and a reset (`rst`) input. So, our port list is:

```
module bin_cntr(  
    input clk,  
    input rst,  
    output [7:0] led  
);
```

To describe the serial network, we need a variable of type `reg`, which is set in a clock-sensitive always block. Its function is simple: it is reset to zero on reset, otherwise it counts up (when the final value - 255 - is reached, the counter overflows and the next value is 0).

```
reg [7:0] cntr;  
always @(posedge clk)  
    if (rst)  
        cntr <= 0;  
    else  
        cntr <= cntr + 1;  
  
assign led = cntr;
```

4.2 Port-pin assignment

Compared to the previous design, the switch input no longer exists, and we have two new ports: clock and reset. On the development board used, the latter can come from the JTAG development cable, or we can use the reset and clock sources on the board. To be able to see the counter values changing with our eyes, we need a low frequency clock signal, which can be supplied by the developer cable. Accordingly, the port assignment:

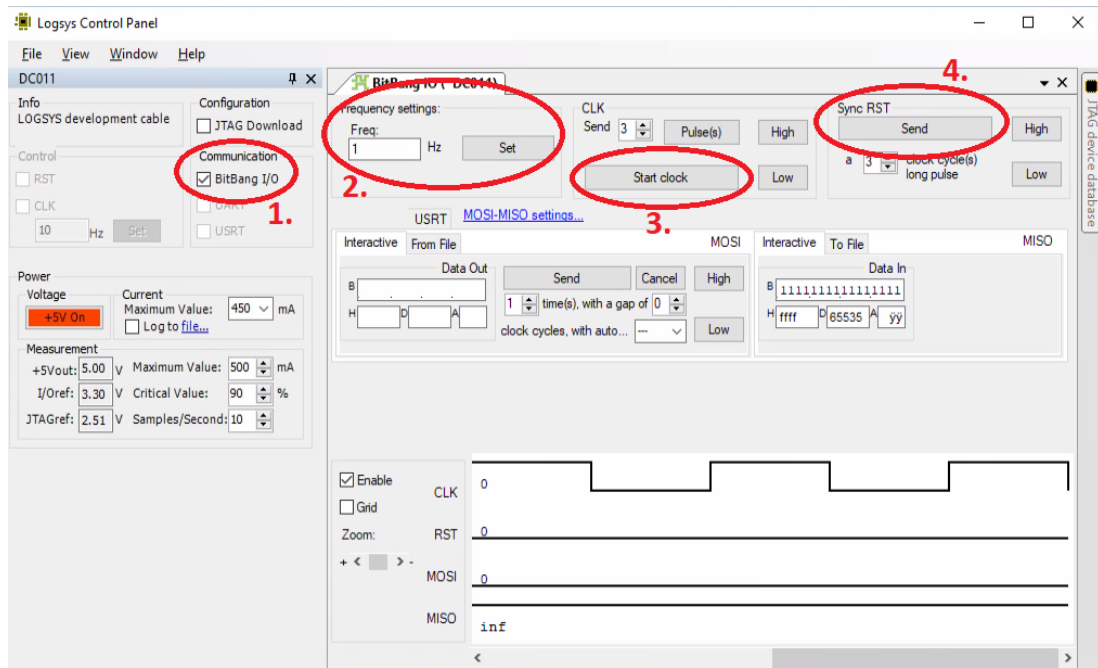
```
NET "clk" LOC="p129" | PULLDOWN;  
NET "rst" LOC="p119" | PULLDOWN;  
  
NET "led[0]" LOC="p59";  
NET "led[1]" LOC="p58";  
NET "led[2]" LOC="p54";  
NET "led[3]" LOC="p53";  
NET "led[4]" LOC="p52";  
NET "led[5]" LOC="p51";  
NET "led[6]" LOC="p50";  
NET "led[7]" LOC="p43";
```

4.3 Implementation, programming, testing

Implement the design in ISE by clicking on the **Generate Programming File** option.

Then program the FPGA with the .bit file you just created.

After programming, select the "BitBang I/O" option in the Logsys GUI, then in the window that appears, set a sufficiently low frequency (1.5 Hz), click on the "Set" and "Start clock" buttons. This will generate a continuous clock signal from the download cable. If you want to reset the counter, click on the "Send" button under Sync RST.



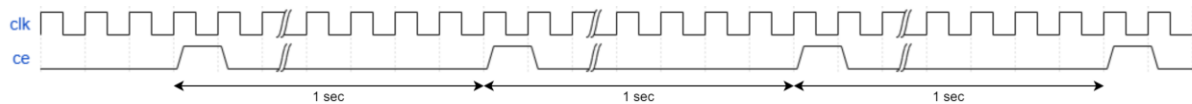
Check that the LEDs on the counter are working properly.

5. BCD counter and enabling signal generator

The third task is to implement a one-digit second counter. The counter displays the current second value in BCD encoding on the lower 4 LEDs of the meter panel, hence these four LEDs can take values 0...9. The counter counts up or down depending on the position of the SW0 switch.

The device will be implemented with a *strictly synchronous* sequential network. This means that all the sequential units of the device will have the same clock, which is called the system clock in the block diagram. Unlike the previous task, now this will be the 16 MHz oscillator on the board.

To make the counter only operate every second, we generate a one clock-period long pulse every second and connect it to the clock enable input of the counter (*ce*). This enable pulse is generated by a rate generator.



Note: In principle, one could also choose to generate a signal with a frequency of 1/second from the 16 MHz external clock signal, which is then actually used as a clock signal (i.e., connected to the clock inputs of the flip-flops). However, this solution is not recommended in FPGAs for reasons not discussed here. In Lab 1, it is a requirement that all flip-flops must operate from the external clock signal! Less common events can be implemented with enable signals as shown here.

The network is of course also equipped with a reset signal (*rst*), which is assigned to one of the push buttons (BTN0).

In the design, we will first design the functional units (modules) of the device, and then connect the modules in the top-level Verilog module, thus creating a hierarchical design.

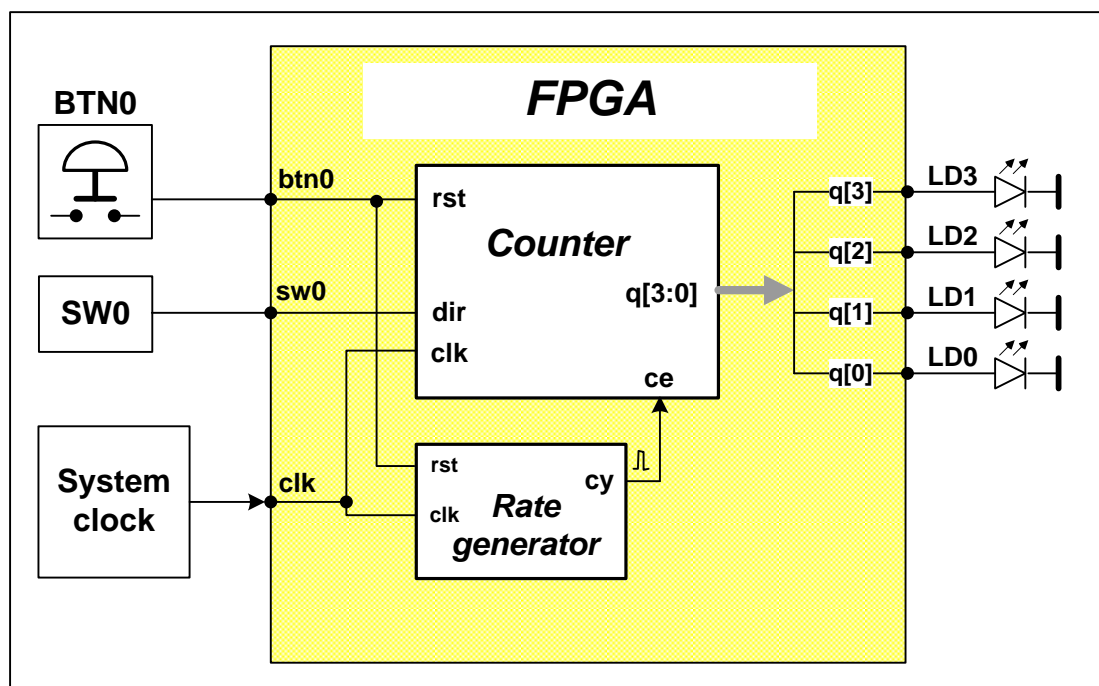


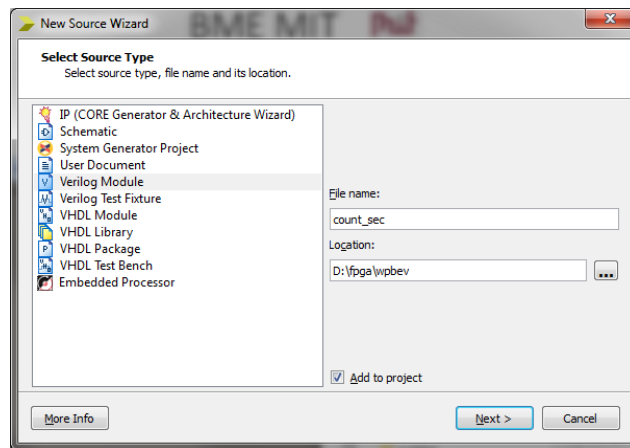
Figure 2 Functional schematic of the device to be designed

Remove from the project the Verilog and UCF files used in the previous point: right-click on the file and then "Remove")!

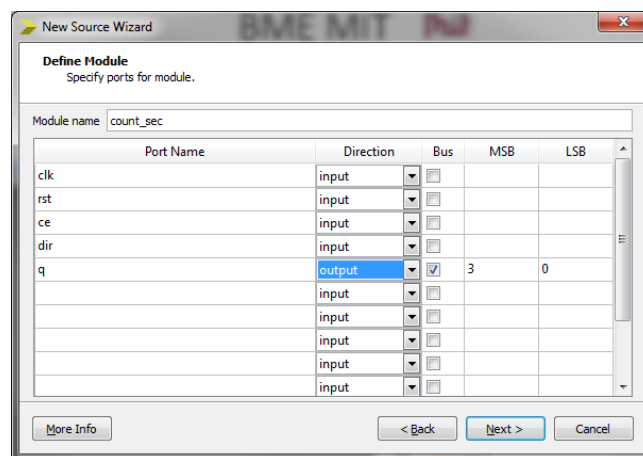
5.1 Preparation of the counter module description

5.1.1 Creating the counter module framework

Create a new Verilog module: in the **Project** menu, click on **New Source** and select **Verilog Module** as the source. The name of the module to be created should be `count_sec`, which should be entered in the **File Name** window, and select **Add to Project**. After clicking on **Next**, the **Define Verilog Source** wizard will offer to edit the port list, which we will use for now.



Based on the block diagram and type in the inputs and then the outputs. The q signal is a bus output, so click on the appropriate row in the **Direction** column and select the *output* direction from the drop-down list, and in the MSB window type 3 to control the four LEDs.



After clicking **Next** and **Finish**, the editor window will display the frame of the module with the port list and the signal declarations:

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

5.1.2 Creating the module functional description using a template

So far, we have started with an empty Verilog file, but it is worth noting that the ISE development system contains an HDL description skeleton for many functional elements, here called a template. Let's create the functional description of the counter using such a template.

In the **Edit** menu, select **Language Templates**, and in the window that appears, select the file **w_CE_and_Sync_Active_High_Reset** (with Count Enable and ...) from the **Verilog \ Synthesis Constructs \ Coding Examples \ Counters \ Binary \ Up/Down Counters** folder, and copy the description template that appears in the editor window to the *count_sec.v* file opened in the HDL editor, under the module header.

The template does not contain real, but function-specific generic signal names, such as <reg_name>, <up_down>. These must of course be rewritten to match the real signal names in the port list: <reg_name> in place of q , <up_down> in place of dir , and so on. Then the module description will be as follows.

```
module count_sec (
    input clk,
```

```

    input rst,
    input ce,
    input dir,
    output [3:0] q);

reg [3:0] cntr;
always @(posedge clk)
if (rst)
    cntr <= 0;
else if (ce)
    if (dir)
        cntr <= cntr + 1;
    else
        cntr <= cntr - 1;

assign q = cntr;

end modules

```

Note: The **Synthesis Constructs** folder is actually an example repository, so it's worth a look.

5.1.3 Modifying the template description

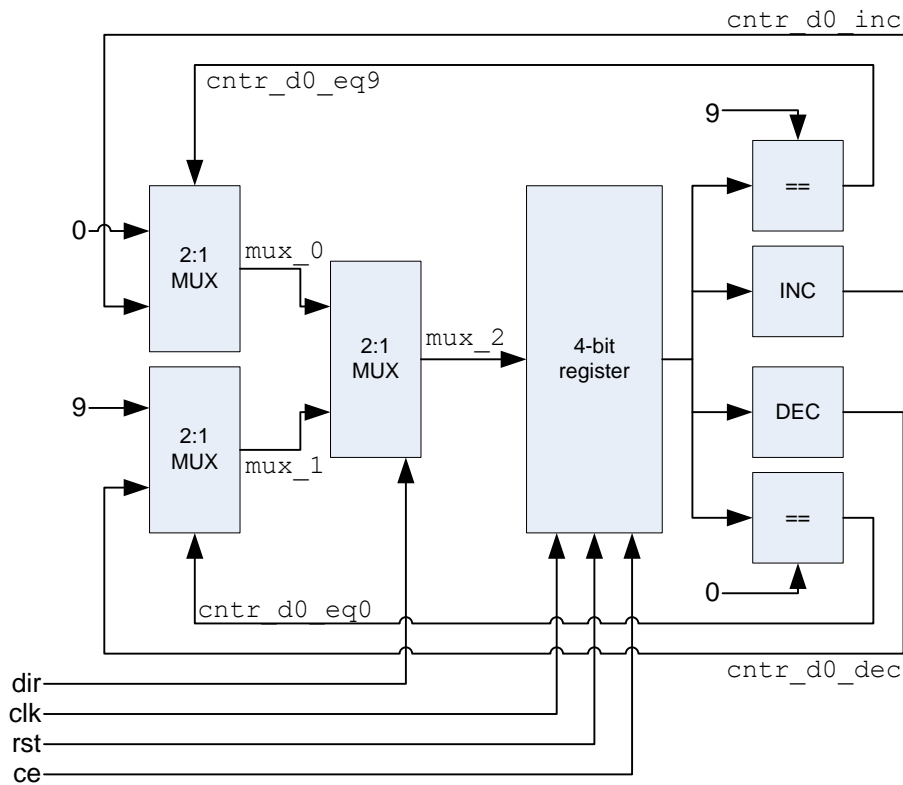
The template is not exactly how we want it to work, so we will modify it. Let's think about what functionality we need, starting with the implementation of the lower local value (and consequently renaming the `cntr` variable in the above code ("`cntr_d0`") to allow for further functional extension).

The up/down counting is implemented by a 4-bit register (4 D flip-flops, with reset and clock enable inputs) and combinational logic using the register output, which produces the incremented and decremented values of the register output (INC and DEC blocks, respectively). The contents of the register, which stores the current value of the counter, are updated every clock cycle when the external enable signal (`ce`) is set to '1' and the external reset is inactive (`rst` = '0').

The register input can be set to the following values based on the current control signals:

- count up (`dir == 1`)
 - the counter has reached its final value (9): 0
 - has not reached its final value: current value + 1
- count down (`dir == 0`)
 - the counter has reached its final value (0): 9
 - has not reached its final value: current value - 1

To detect the final values, two equality comparators can be used, which compare the output of the counter register with 9 and 0 respectively. These considerations result in the following block diagram:



In our first Verilog description, let's stick to a structure that perfectly matches the block diagram. To use the internal signals, we need to declare them: the variable implementing the sequential network (register) should be of type *reg*, while the variables implementing combinational logic should be of type *wire*.

```
reg [3:0] cntr_d0;
wire [3:0] cntr_d0_inc, cntr_d0_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 signal in the sensitivity list of our *always* block is the rising edge of the clock signal, so the values are only evaluated at this time, i.e. the variable written here actually produces D FFs. However, the content of the resulting D FFs is only modified if the external *ce* signal is '1'. The other variables are of *wire* type, so they can only be assigned a value by the *assign* statement, and after implementation they result in combinational logic.

However, Verilog is a relatively high-level language, so it is possible to write code that is functionally identical to the above description, but is easier to understand and more concise:

```
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==9)
        cntr_d0 <= 0; //overflow
    else
        cntr_d0 <= cntr_d0 + 1;
else //DIR=0: count down
    if (cntr_d0==0)
        cntr_d0 <= 9;
    else
        cntr_d0 <= cntr_d0 - 1;

```

5.1.4 Syntax check of the module description

After carefully checking the source file (e.g. for missing semicolons at the end of statements), we can perform a syntactic check with the development system. Select the module you want to check in the **Project** window (and if you have more than one module, set it as top module: right-click and **Set as Top Module**), then double-click in the **Processes** window to launch **Synthesize-XST / Check Syntax**. The result of the check will be displayed in the bottom message window, where you can view the Errors and Warnings separately by clicking on the corresponding tabs.

Otherwise, the ISE subsystems start all further processing by checking the initial file, so that a check is automatically performed when the simulation is started as described in the next section.

5.2 Verification of the module by simulation

5.2.1 Creating the test environment

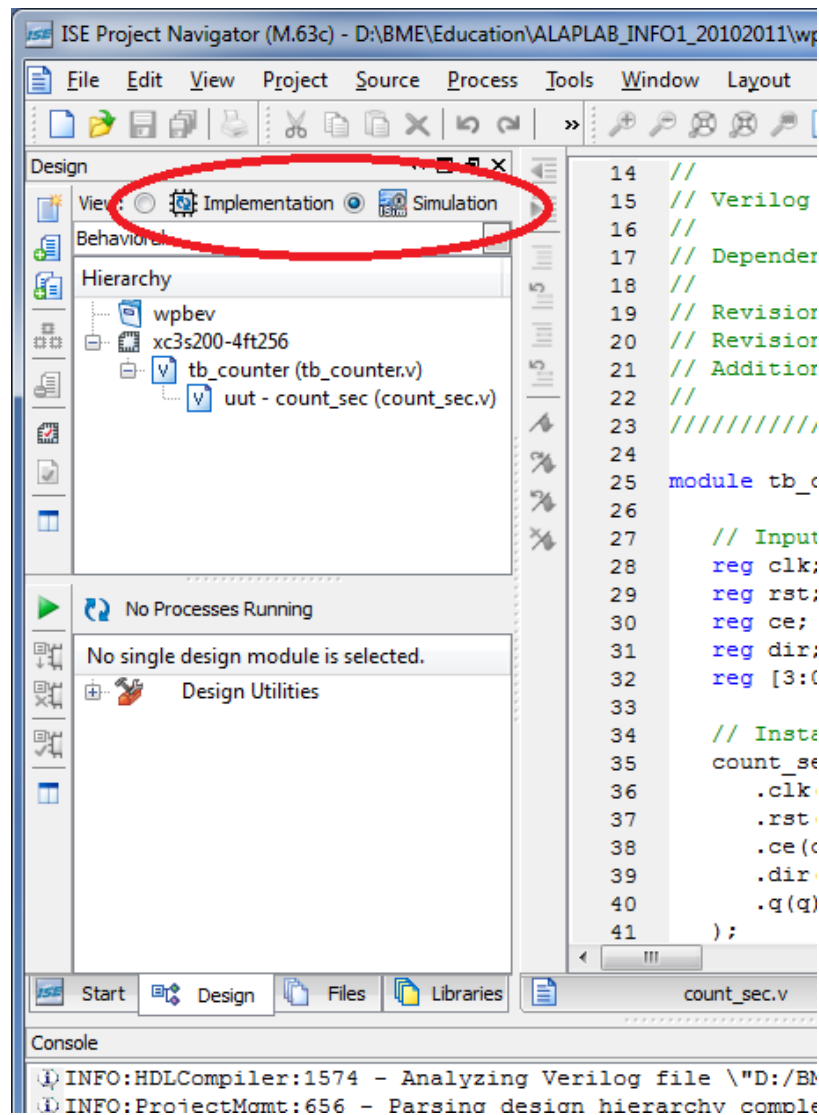
The test environment (test fixture) is created by specifying the waveforms (excitation signals) that drive the unit.

To be able to test the design for implementation, we need to create an environment that properly drives the inputs of the system under test (test fixture). In older versions of ISE, it was possible to specify the excitation signals in a graphical interface, but in newer versions this feature has been removed, so the only option is to write them in HDL.

First, add a new source to the project: in the **Project / New Source** window, select **Verilog Test Fixture**. The file name should be **tb_counter**!

In the next window, select the module for which you want to generate the testbench - in this case we have only one module - count_sec - so this is the only choice.

Click **Next, Finish** to approve the file generation and return to the ISE main window. By selecting **Simulation** in the top left **View** section, the **Hierarchy** window will also display the testbench file, which is a submodule of the module to be tested (the hierarchy can be expanded by a + sign in front of the module name).



5.2.2 Creating excitation signals

The automatically generated Verilog Test Fixture file contains the following:

- replication of the module to be tested
- declaring variables of type *reg* for input signals
- declaration of *wire* type variables for output signals
- set all input variables to 0

The Verilog code generated (as explained above, we leave it to the reader to interpret):

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

// Inputs
reg clk;
reg rst;
reg ce;
go ahead;

// Outputs
wire [3:0] q;

// Instantiate the Unit Under Test (UUT)
count_sec new (
```



```

        .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

end modules

```

As with any serial network, the counter under test needs to generate a clock signal. At time 0 of the simulation, the variable *clk* is set to 0 (see *initial* block above). The clock signal is nothing more than a square wave with a 50% fill factor; it can be generated by inverting the current value per unit of time. A possible Verilog code (note: the *always* block is outside the *initial* blocks above!):

```

always #5
    clk <= ~clk;

```

Since the set time unit in the *timescale* directive is ns, the low and high states of the generated square wave are both 5 ns long, so the period time is $2 \times 5 = 10$ ns (100 MHz frequency). This is not important for behavioural simulation, but for post-implementation simulation that takes into account the FPGA's internal timings, it is important to use a clock signal with the right frequency.

The control signals *rst*, *ce* and *dir* are generated as follows:

- *rst* should be set to '1' during simulation 7 - 27 ns
- *ce* should be '1' after 107 ns
- *dir* change to '1' after 1007 ns

This can be generated with the following Verilog code

```

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

initial
    #107 ce <= 1;

initial
    #1007 dir <= 1;

```

Comments:

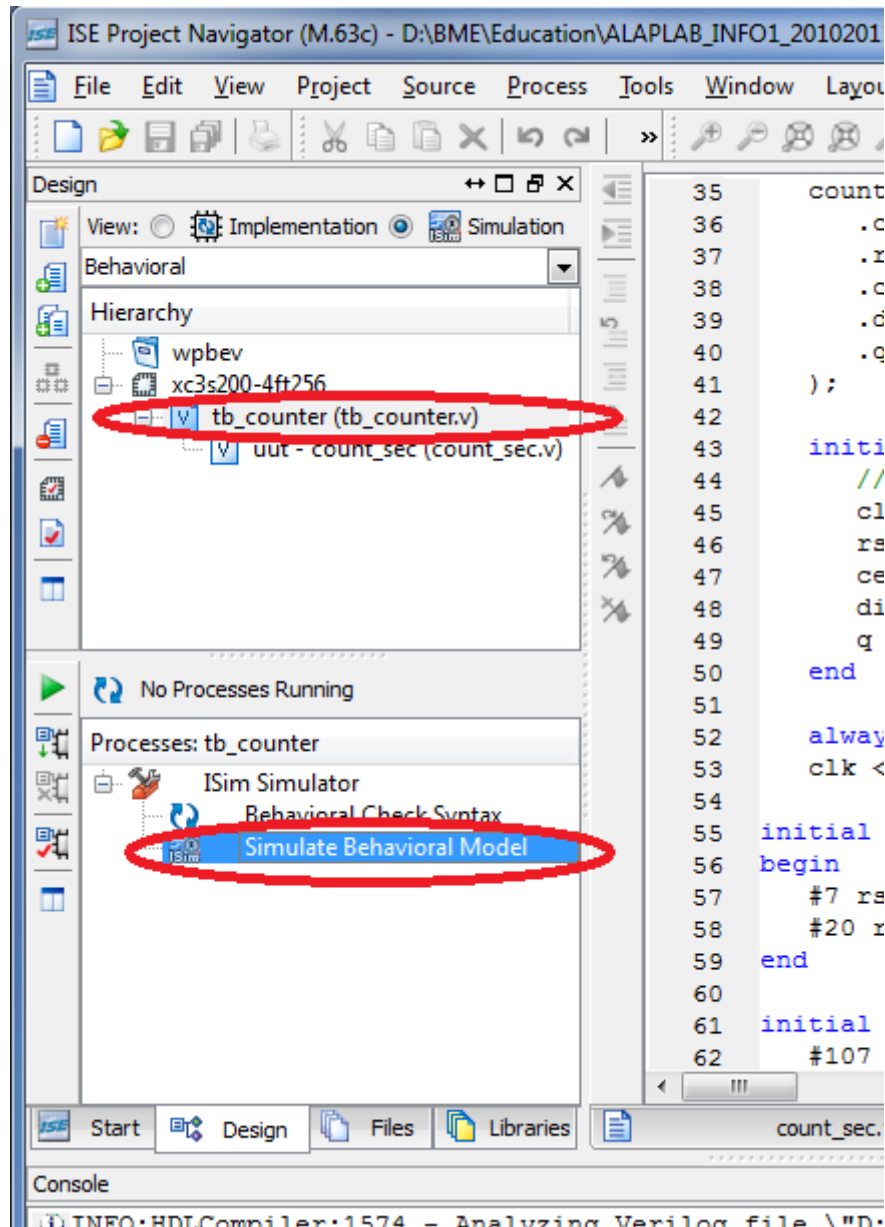
- the delays within an *initial* block add up
- *initial* blocks run in parallel, each starting at time 0

5.2.3 Functional simulation using ISE Simulator

The simulation at this design step is a functional check. The ISE Simulator can also take into account the real hardware's delays, but at this level the design is not yet mapped to a real IC type, so we cannot calculate real delay values.

Select **Simulation** from the **View** options in the **Project Navigator** program, then select the testbench file (*tb_counter*) in the **Hierarchy** window. In the **Processes** window, start the **ISim Simulator** / **Simulate Behavioral Model** program.

ATTENTION: For simulation, always select the testbench file in the **Hierarchy** window!!! The simulator can be started even if the module for implementation is selected, but in this case the input signals will not be driven by anything, i.e. the module will not work (in this case, in the simulation waveform window, all input signals are high impedance (blue), while the output signals are undefined (red)).



The simulation result will be displayed in a new window (Figure 5). It is worth noticing that before the *rst* signal is activated, the value of the storage elements (registers) is - correctly - considered by the simulator as unknown, indicated by the value X (red color).

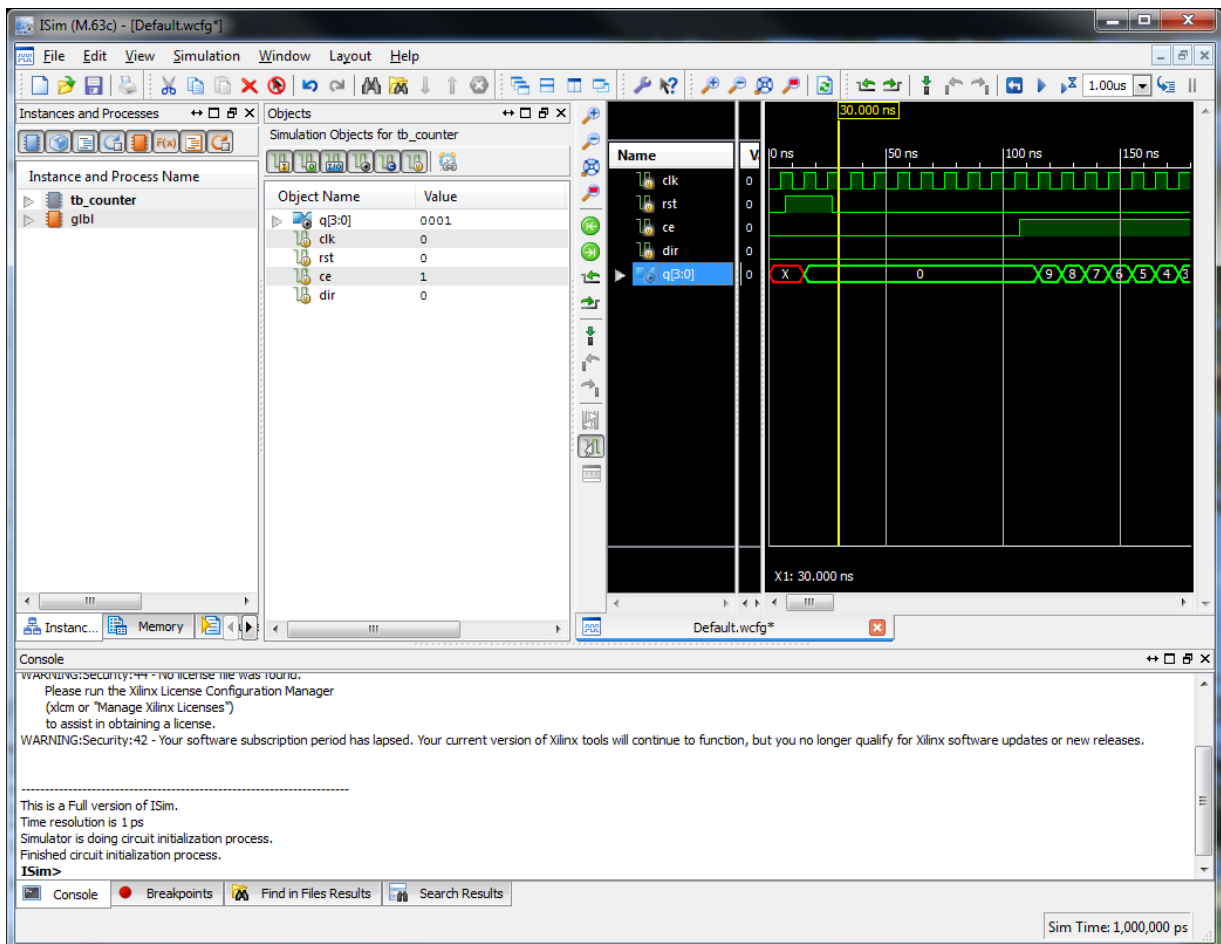


Figure 5 Simulation result in the Xilinx ISE Simulator waveform window

Click on each signal name to change the format of the display using the popup menu's Radix options. Setting this to hexadecimal for the q[3:0] output makes it easier to check that it is working correctly.

It's often very useful to examine the internal signals of modules - to do this, you don't need to export these signals from the module, you can simply add them to the **Wave** window in the simulator. Clicking on the **Instances and Processes** tab in the left-hand window displays the simulation hierarchy, which includes the instantiated *count_sec* module (called UUT - unit under test). Selecting this module and then clicking on the **Objects** tab shows all the signals in the module, of which the signal you want to test can be added to the **Wave** window using a simple drag-and-drop method (Figure 6). After that, all that is needed is to restart the simulation to monitor the values of the signals (Blue enter-like button in the top menu, followed by the sandbox play-like button).

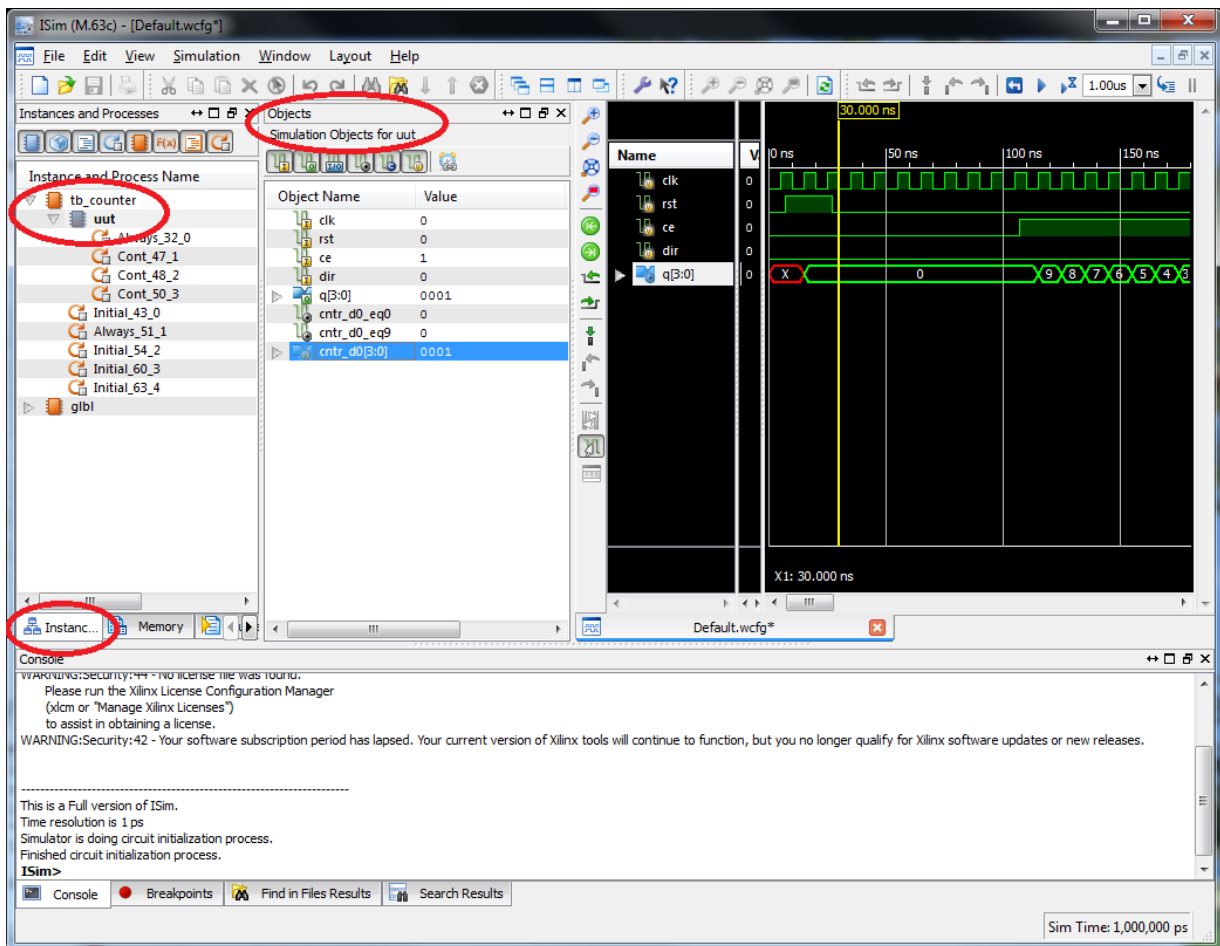


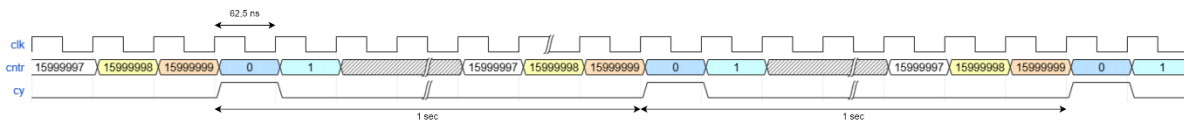
Figure 6 Simulation with internal signals

5.3. Additional functional units

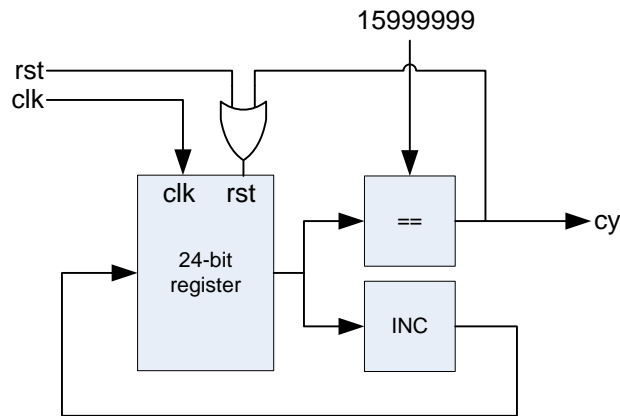
5.3.1 The rate generator module

The module should be called "rategen" for short. Using the **Project / New Source** command, generate a Verilog Module type file named rategen, then press **Next** and **Finish**, leaving the portlist table empty.

The clock generator is essentially a counter that divides the 16 MHz clock of the FPGA board by 16 million (i.e., counts from 0 to 15999999) and generates a one clock-cycle long pulse (cy) once per count period. Such a division requires a binary counter of at least 24 bits. The counter name should again be cntr, declared as a register (`reg [23:0] cntr;`). Timing diagram:



The block diagram of the unit to be implemented is therefore:



The Verilog code that implements this:

```

module rategen(
    input clk, rst,
    output cy
);
    //Generate 1 clock wide pulse on output CY
    reg [23:0] cntr;

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

    assign cy = (cntr == 15999999);
    //assign cy = (cntr == 4);

end modules
  
```

Another split ratio is currently "commented out" in the module description. This division ratio can be used to check the design by simulation. With the "operational" division ratio, the output of the module would only change every 16 millionth simulation step, leading to very long simulation runtimes.

5.3.2 Creating the top-level module and user constraints

The top-level module describes the connections between functional units and the whole device to the outside world. The top module usually contains many inputs and outputs and quite a few functional modules. The resulting large number of signal names must be carefully specified.

Use **Project / New Source / Verilog module** to create a *wpbevtop1* module frame.

To describe the top module, let's take again the functional block diagram of the device (*Figure 2*), on the basis of which we started the design. The top module essentially describes this block diagram.

For clarity and ease of understanding, this outline has been slightly redrawn (*Figure 7*), taking into account the aspects of the module description.

In fact, you can name the top module's inputs and outputs anything you like, but the naming must be consistent with the naming in the user "pin" bindings (what signals should be connected to each pin of the FPGA). All I/O pins of the development board used in the lab are defined in the corresponding UCF file, which can be found on page 15 of the documentation. In this case, however, do not use it, but create your own ucf file! Next to each element (quartz, buttons, switches) on the board, you will find which pin of the FPGA it is connected to.

Take the block diagram and copy the signal names in sequence into the input/output list.

```

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

As already mentioned in the introduction, in a correct design the primary (external) inputs of the synchronous network should be synchronized to avoid the occurrence of a metastable state in the system caused by the asynchronous nature of the input signal. In this exercise, the synchronizing network should not be described as a separate module, because it does not improve the overview. A simple single-stage synchronization network is chosen and described by an *always* statement in the top module.

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

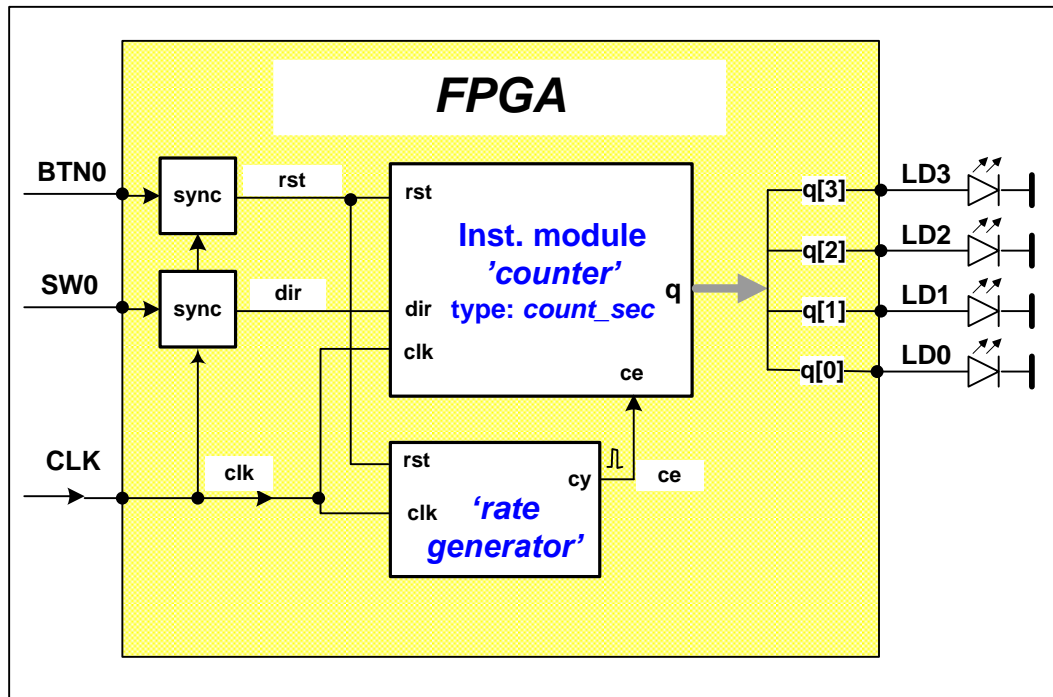


Figure 7 Schematic of the top module

Next, the modules must be instantiated (inserted). The instantiation must specify the signals which are connected to the ports of the instantiated module. If a module is instantiated only once (or, in the case of multiple instantiations, for one instance), the instance name may be the module name. This will be used several times in our sample example. **Note that Verilog considers undeclared signals to be of type 1-bit wire (this is the case even if you misspell the variable name). Since the signal is declared automatically, in many cases you will not get an error message, only a malfunction. In Warnings, however, the error can be detected.** If the current signal names of any ports of two modules are the same, the two ports are connected. This also applies to other elements of the top module.

If you want to connect two signals with different signal names, you can use the *assign* statement.

We start the instantiation with the *rategen* module that generates our 1-second enable signal. Let's start from the declaration of the module:

```
module rategen(
    input clk, rst,
    output cy
);
```

First, delete the module keyword, then enter the instance name after the module type. Finally, connect the corresponding signals of the top module to the ports of the instance.

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

```

        .rst(rst),
        .cy(ce)
    );

```

So, with the input synchronisation and the submodules inserted, our complete top-level module looks like this.

```

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)
);

end modules

```

To perform the pin assignments, a constraint file is added to the project. Select **Project / New Source**, and in the window that pops up, go to **Implementation Constraint File** and choose `counter_pins` as the name.

After pressing the **Next/Finish** button the **Sources** window will display the `counter_pins.ucf` file, set the `counter_pins.ucf` as shown below (or read from the board). Note that although the clock (clk) port name is the same as in the previous task, it is now connected to a different pin, as we use the onboard oscillator instead of the development cable since the development cable provided clock.

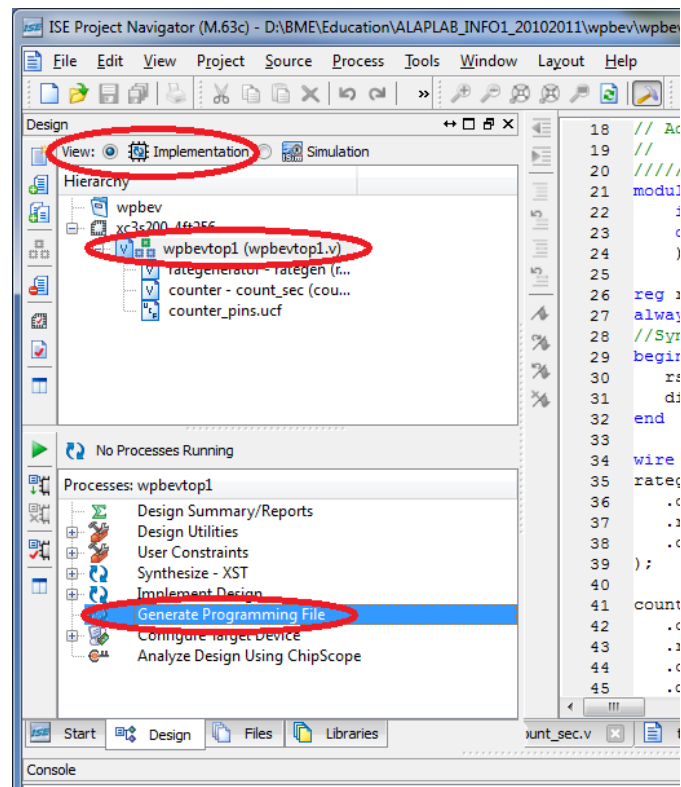
```

NET "clk" LOC="p56";
NET "btn0" LOC="p38";
NET "sw0" LOC="p101";
NET "q[3]" LOC="p53";
NET "q[2]" LOC="p54";
NET "q[1]" LOC="p58";
NET "q[0]" LOC="p59";

```

5.4. Implementation, FPGA configuration

Compile the design and generate the FPGA configuration file using the **"Generate Programming File"** option. Make sure that the top-level module is selected in the Hierarchy window.



Once the configuration file is ready, configure the FPGA in LOGSYS GUI and check the operation.