Xilinx ChipScope ICON/VIO/ILA Tutorial

The Xilinx ChipScope tools package has several modules that you can add to your Verilog design to capture input and output directly from the FPGA hardware. These are:

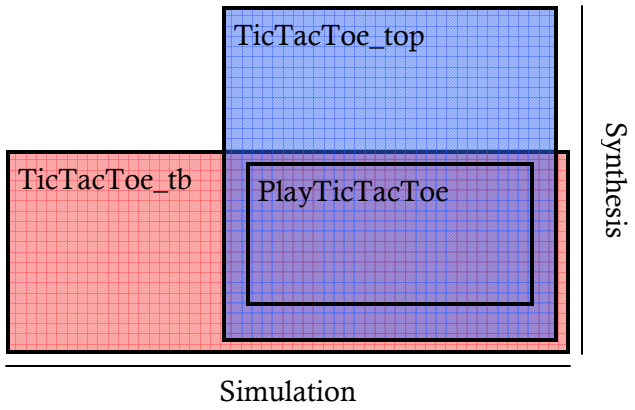
- ICON (Integrated CONtroller): A controller module that provides communication between the ChipScope host PC and ChipScope modules in the design (such as VIO and ILA).
- VIO (Virtual Input/Output): A module that can monitor and drive signals in your design in real-time. You can think of them as virtual push-buttons (for input) and LEDs (for output). These can be used for debugging purposes, or they can incorporated into your design as a permanent I/O interface.
- ILA (Integrated Logic Analyzer): A module that lets you view and trigger on signals in your hardware design. Think of it as a digital oscilloscope (like ModelSim's waveform viewer) that you can place in your design to aid in debugging.

These ChipScope modules are extremely useful because they allow you to view and manipulate signals directly from hardware during run-time. Since they are real Verilog modules and netlists, they get incorporated, synthesized, and implemented into your design just like any other Verilog code you would write.

Whether you know it or not, you've been using ChipScope modules in your designs for the past few weeks. Take a look at the top-level modules for all the previous labs we've finished—they all contain declarations and instantiations for ICON, VIO, and/or ILA modules. After working through this tutorial, you'll know how to add these modules to your design by yourself.

Simulation vs. ChipScope

Running a simulation involves compiling all your Verilog modules and running the testbench. All of this takes place on your PC—there is no actual hardware created in the process. This is convenient for several reasons: you can code your design without the physical FPGA, you can view every signal at any time in the simulation, and you can compile your Verilog code much faster than you would be able to synthesize it into hardware. For these reasons, we recommend that you do the vast majority of your debugging from within the simulation tool (ModelSim).

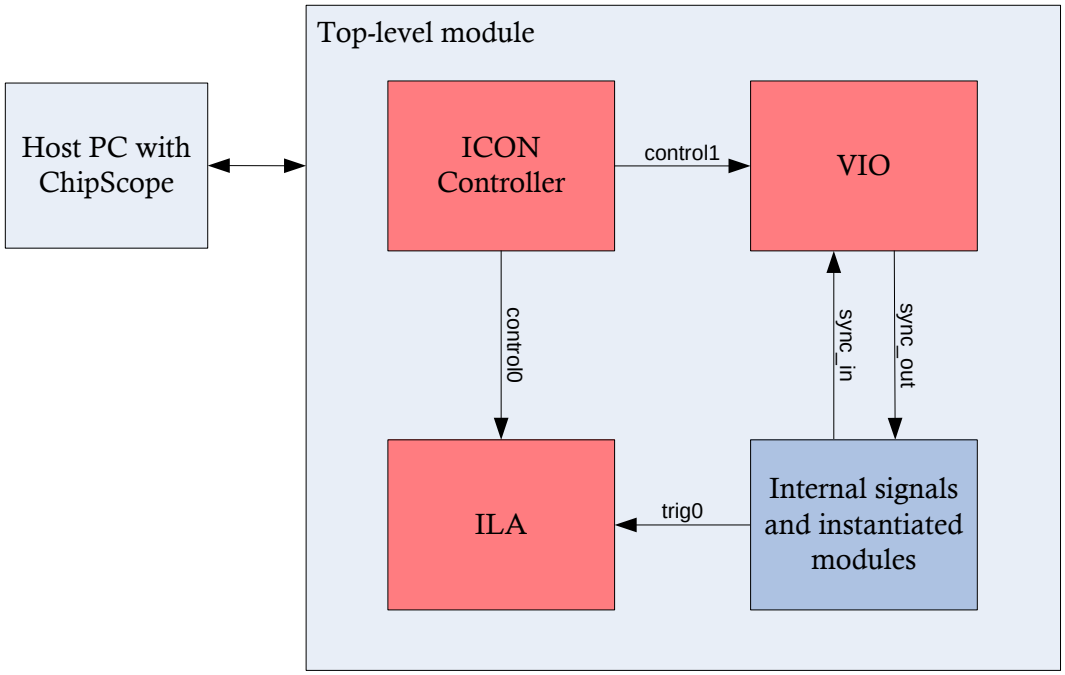


Version 1.0 – 10/31/2005 – Ben Nham

There is a pitfall to the simulation model, however. In simulation, you generally can't simulate the top level module, since that contains many system-level inputs and outputs (like the clock, vga/sound outputs, etc.) that the simulator has no simulation model for. Instead, you have to write a testbench that provides stimulus signals to replace these top-level signals to lower-level modules. In the example of the figure above, the testbench has no way of simulating the clock or vga outputs of the tictactoe_top module, since those modules live outside the PlayTicTacToe module that is being simulated. It is in these cases that ChipScope is extremely useful. Using ChipScope, you can capture almost any signal in your system, including top-level signals. If the problem with your design lies at the top-level or is fundamentally hardware-related, using ChipScope modules is probably the best way to debug them.

Remember: while simulation is a powerful tool, it takes place on a different axis than synthesis. Checking the design in simulation should catch most errors, but cannot catch errors that are made in the synthesizable-but-not-simulatable parts of your design (blue in the diagram above).

ChipScope Organization Details



Take a look at the ChipScope organization diagram above. To use ChipScope modules in your design, you must **always** generate and instantiate an ICON controller module. The ICON controller module communicates with the host PC and sends commands to other ChipScope modules via a control port. Your ICON controller module must be generated with the same number of control ports as there are other ChipScope modules in your design. For example, if you want to add an ILA module and a VIO module to your design, generate an ICON module with two control ports.

Once you've added the ICON module to your design, you can add as many ChipScope modules as you have control ports. VIO and ILA modules take the ICON control port as an input and then interact with the modules in your design through sync_in/sync_out and trigger ports respectively.

Incorporating ChipScope Modules into Your Design

Now that you've determined that you need ChipScope modules in your design, whether for debugging or as a permanent I/O interface, it's simple to add them to your design. You follow a four-step process:

1. **Generate** the ChipScope modules, using the ChipScope Core Generator.
2. **Incorporate and instantiate** the ChipScope modules into the top-level module in your design.
3. **Connect** the ChipScope modules to your design.
4. **Synthesize, implement, and run** the design on the FPGA**.**

Example Top-Level Module – A 16-bit Adder

Before we generate the ChipScope modules, find the top-level module you want to add the ChipScope modules to. That module might be named lab3_top, SOS_detector, etc.—it's the module that you normally click on to synthesize in Xilinx ISE.

For this tutorial, we'll use a very simple top-level module for design—a module that contains a 16-bit adder. The code for the top-level module and the code for the adder are shown below. You will also need a UCF file in the same directory to specify that the design's timing should be meet a 100 MHz clock constraint, and that the system clock is located at pin AJ15 on the board.

```
// counter_icon_test.v – top-level module
module counter_icon_test (input clk);

   // These wires will be hooked up to the vio later
   wire incr, rst;
   wire [15:0] count;

   counter #(16) countone(.clk(clk), .incr(incr), .rst(rst),
                          .count(count));
endmodule
```

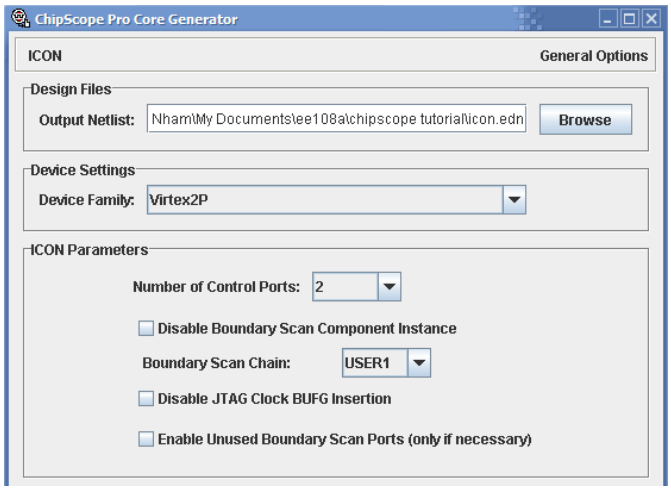
```
// counter.v
module counter(clk, incr, rst, count);
   parameter n = 5;

   input clk, incr, rst;
   output [n-1:0] count;

   wire [n-1:0] count;
   wire [n-1:0] next_count = incr ? count + 1 : count;

   dffre #(n) count_reg(.clk(clk), .en(1'b1), .r(rst),
                        .d(next_count), .q(count));
endmodule
```

```
// counter_icon_test.ucf
NET "clk" LOC = "AJ15";
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 10 ns HIGH 50 %;
```

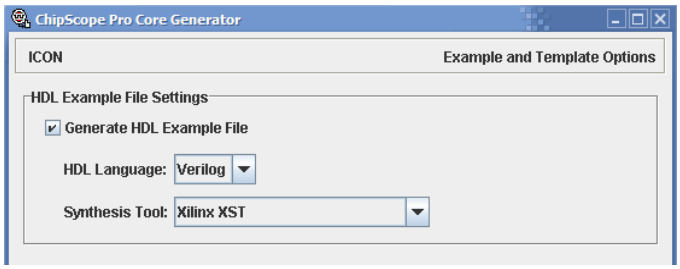
Create these files and save them to a new directory.

Generating ChipScope Modules

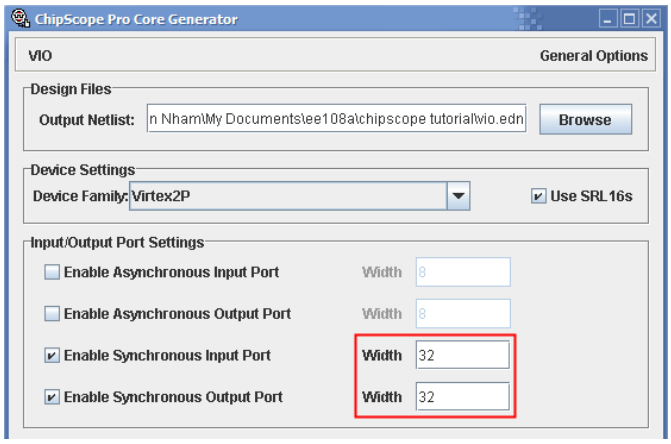
To generate the ChipScope modules, launch the ChipScope Core Generator from the ChipScope Start Menu Programs folder. To start, generate the ICON controller core. For device family, make sure **Virtex2P** is selected—that's the type of FPGA we're using. For the **Number of Control Ports**, select the number of other ChipScope modules you'll be adding to your design. In our case, we'll be adding one VIO module and one ILA module, which means we need two control ports (one for each). Make sure the output directory is where your design files are (in this case, where you saved the files for the adder).



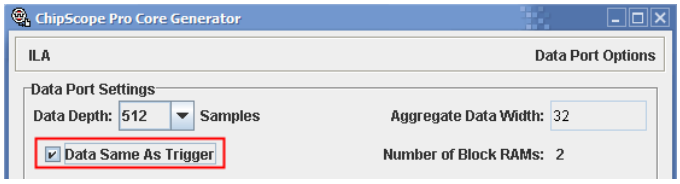
Hit next. Now you're free to generate the ICON core. Make sure that the generated HDL language is **Verilog**.

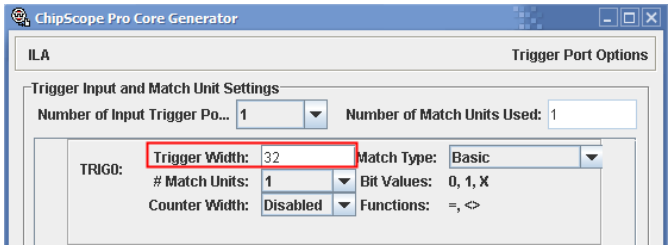


You'll generate the VIO and ILA cores in a similar fashion. Launch the Core Generator again (or say yes when it asks you if you want to generate more cores), and select VIO or ILA. We'll generate a VIO core first. Enable the **synchronous input and output** ports, and size them so that you have enough bits to send and receive control signals to and from the module you want to control. We'll arbitrarily chose 32 bits since it contains enough bits to send and receive signals from the 16-bit adder. Go ahead and generate the VIO core into the same directory as the ICON core.



Finally, generate the ILA core. Again, make the **trigger width** wide enough so that you can view the signal that you're interested in viewing (in this case, we're probably at the very least interested in viewing the 16-bit result of the adder, so make it at least 16 bits wide). Just to be safe, we'll again make the trigger 32 bits wide. On the next property sheet, make sure **Data Same as Trigger** is checked. You may also want to change the **Data Depth** setting for more resolution (at the cost of using more BRAMs). In our case, we chose a data depth of 512 samples, so we can view a maximum of 512 clock cycles' worth of signals around the trigger we choose. (This





is a key difference between using ChipScope and ModelSim: in ModelSim, you can view all signals at all points and time; in ChipScope, you can only view a certain window of signals.)

Incorporating and Instantiating the ChipScope Modules

Now that you've generated the ChipScope modules, add them to your design. This basically amounts to cutting and pasting from the example Verilog files that the Core Generator provides you to the top-level module you're working on.

For example, after generating the ICON module, you'll see a file called **icon_xst_example.v** (if you see the VHDL file instead, go back and regenerate the core making sure you chose Verilog for the last step). It looks something like this:

```
module icon_xst_example();

  //------------------------------------------------------------
  //  ICON core wire declarations
  //------------------------------------------------------------
  wire [35:0] control0;
  wire [35:0] control1;

  //------------------------------------------------------------
  //  ICON core instance
  //------------------------------------------------------------
  icon i_icon (.control0(control0), .control1(control1));

endmodule


//------------------------------------------------------------
//  ICON core module declaration
//------------------------------------------------------------
module icon (control0, control1);
  output [35:0] control0;
  output [35:0] control1;
endmodule
```

You'll see similarly named **vio_xst_example.v** and **ila_xst_example.v** files. For each file, take the wire and core instantiations and paste them into your top level module. Then take the module declaration and paste it to the end of the file that contains your top-level module (after the endmodule statement of the top-level module). Remember to change the control inputs into the modules so they match the control wires that are actually coming out of the ICON core (i.e. control0 or control 1 goes into the ILA instead of the default control). Afterwards, you should end up with a top-level module that looks like this. Notice that we simply copied-and-pasted, except for modifying the control inputs into the ILA and VIO.

```
module counter_icon_test (input clk);
   // These wires are hooked up to vio below
   wire incr, rst;
   wire [15:0] count;

   counter #(16) countone(.clk(clk), .incr(incr), .rst(rst),
                          .count(count));

   //------------------------------------------------------------
   //  ICON core wire declarations
   //------------------------------------------------------------
   wire [35:0] control0;
   wire [35:0] control1;
```

```
   //-----------------------------------------------------------
   //  VIO Core wire declarations
   //-----------------------------------------------------------
   wire [31:0] sync_in;
   wire [31:0] sync_out;

   // Next 3 lines are explained in next section
   assign rst = sync_out[0];
   assign incr = sync_out[1];
   assign sync_in  = {16'b0, count};

   //-----------------------------------------------------------
   //  ILA Core wire declarations
   //-----------------------------------------------------------
   wire [31:0] trig0;

   // Next line is explained in next section
   assign trig0 = {14'b0, count, incr, rst};

   //-----------------------------------------------------------
   //  ICON core instance (from icon_xst_example.v)
   //-----------------------------------------------------------
   icon i_icon (.control0(control0), .control1(control1));

   //-----------------------------------------------------------
   //  VIO core instance (from vio_xst_example.v)
   //-----------------------------------------------------------
   vio i_vio (.control(control0), .clk(clk), .sync_in(sync_in),
              .sync_out(sync_out));

   //-----------------------------------------------------------
   //  ILA core instance (from ila_xst_example.v)
   //-----------------------------------------------------------
   ila i_ila (.control(control1), .clk(clk), .trig0(trig0));

endmodule


//-----------------------------------------------------------
//  ICON core module declaration (from icon_xst_example.v)
//-----------------------------------------------------------
module icon (control0, control1);
  output [35:0] control0;
  output [35:0] control1;
endmodule

//-----------------------------------------------------------
//  VIO core module declaration (from vio_xst_example.v)
//-----------------------------------------------------------
module vio (control, clk, sync_in, sync_out);
  input  [35:0] control;
  input  clk;
  input  [31:0] sync_in;
  output [31:0] sync_out;
endmodule

//-----------------------------------------------------------
//  ILA core module declaration (from ila_xst_example.v)
//-----------------------------------------------------------
module ila (control, clk, trig0);
  input [35:0] control;
  input clk;
  input [31:0] trig0;
endmodule
```

Connecting the ChipScope Modules

You're almost done! Now you just need to add a few lines to actually hook up the ChipScope modules that you plopped down in your design with the signals you're interested in monitoring or driving. In the sample code above, you'll notice that we've added these lines already.

For the VIO module, you probably want it to be able to send rst and incr signals to your adder and for it to see the count output. So after the VIO Core wire instantiations, we added these lines to hook up the signals:

```
assign rst = sync_out[0];
assign incr = sync_out[1];
assign sync_in  = {16'b0, count};
```

This lets the sync_out[0] output of the VIO drive the rst input of the adder, the sync_out[1] output drive the incr output, and assigns the bottom 16 bits of the VIO sync_in input to the count output of the adder.

We also want hook up signals we want to monitor to the ILA trigger input. When debugging sequential logic, you usually want to view the control signals (like resets and enables) and the states (like count in this case). So we assign count, incr, and rst to the bottom 3 bits of the trigger input of the logic analyzer.

```
assign trig0 = {14'b0, count, incr, rst};
```

Synthesizing, Implementing, and Running Your Design

Now you can finally re-simulate and re-implement your design. Before starting, make sure all the auxiliary files that the Core Generator created (like icon.edn, ila.arg, etc.) are in the directory where your Xilinx ISE project is. Do **not** include the example Verilog files (like icon_xst_example.v) with your project—the module declarations inside your top-level file are sufficient.

Synthesize, implement, and generate the bitstream.

Using ChipScope Analyzer

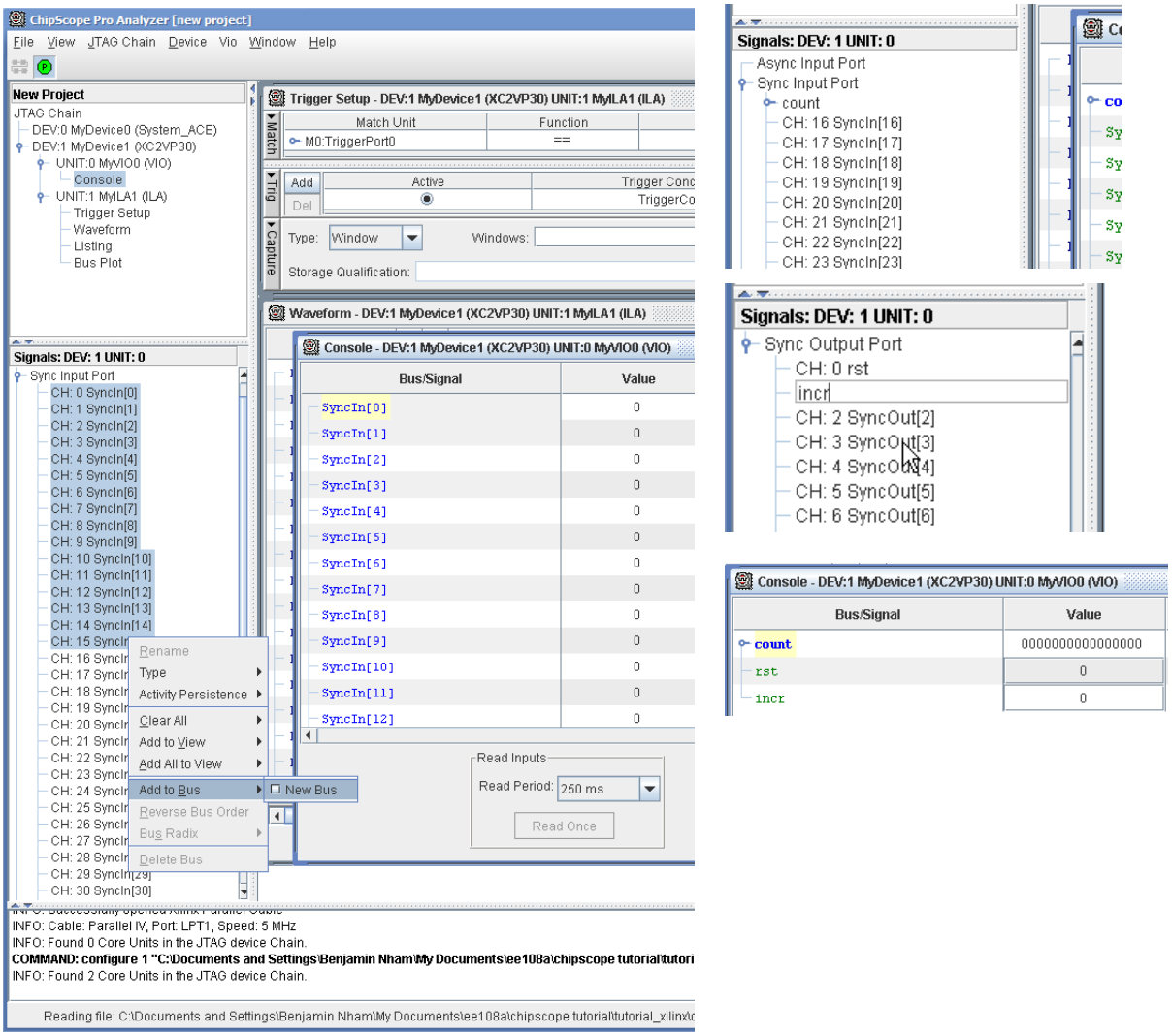
Finally, you can use the analyzer to interact with your ChipScope modules. You've already had practice doing this in lab already, so this will be a brief review.

After loading the bitstream, you'll see a console pop up. We'll interact with the VIO module first. Double click the VIO heading in the top-left of the analyzer to bring up the VIO console.

Now, you'll have to rename the signals in the console so they make some sense to you. Recall that we assigned count to the lower 16 bits of sync_in. Now, we'll rename the lower 16 bits of sync_in to reflect that fact. Select the lower 16 bits of sync_in in the lower left portion of the ChipScope window (select SyncIn[0], hold down shift, and select SyncIn[15]), right-click, and select Add to Bus. Rename the new bus count.

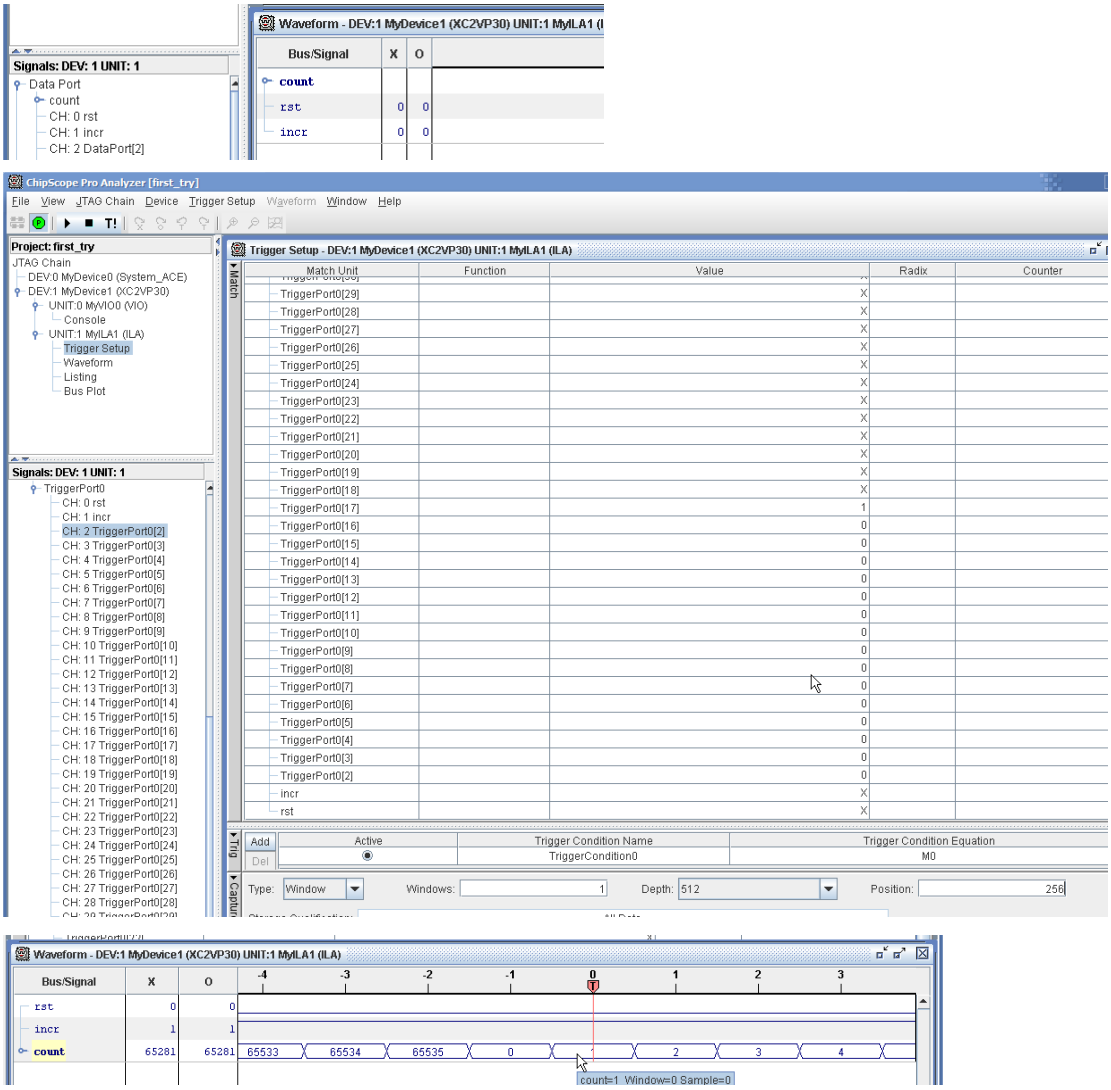
Also recall that we drove the reset and incr inputs of our counter using the lower two bits of sync_in. Rename the lower two bits of sync_in to reflect this fact (right click on the signal in the lower-left corner of ChipScope Analyzer and click "Rename"). Finally, drag the rst, incr, and count signals from the Signals window to the Console window. Turn rst and incr into single-pulse inputs (right click and choose single pulse). This means that when you click on

rst and incr, they will send a 1-cycle high signal to the counter to the respective input pins. Every time you click incr, you should see the count increment by one, as expected.



Now, onto the logic analyzer. Double click on the ILA unit in the top-left hand corner of ChipScope Analyzer to bring up the Waveform and Trigger windows. For the waveform, you'll want to do the same thing as we did above—rename the signals so they mean something useful. Select the 0^{th} bit of the data port and rename it reset. Select the 1^{st} bit of the data port and rename it incr. Select bits 3-18, add them to a bus, and rename the bus count.

To actually start viewing signals, you'll have to set up a trigger condition. To check whether the ILA is actually working, you can select Trigger Immediate from the toolbar (the T! button), and the waveform should show the current state of the circuit. You can also set a specific trigger. For example, you can add the trigger count = 1, and the next time the counter gets to a state where the count = 1, a waveform will be captured for you to view. To add a trigger, just set the signal that you want to trigger on in the Trigger window (you may need to expand it first) and set the bits/buses that you want to trigger on. In this case, count = 1 is equivalent to trig0[2] = 1, so we simply trigger on that. You can do an equality trigger on any signal that you fed into your ILA (in this case, count, rst, or incr). The following screenshots show the result of a trigger on count = 1.



Conclusion

Congratulations—you've just added ChipScope modules to your design. While this example was very simple, you can see how easy and useful adding ChipScope modules to your design is. By using these modules, you can capture signals directly off the hardware while it is running—including top-level signals that you might not be able to capture using ModelSim. You can also use it to simulate virtual push-buttons and LEDs by using the VIO module, which can be easier and more precise than using the actual LEDs and push buttons on the board. All in all, learning how to use the ChipScope modules is a very useful skill—one that will probably come in handy at least once while you're completing your final project.