

ECE532S Digital Systems Design

Tutorial 4 - Debugging on Board Using the ILA

Last Updated: July, 2019

When designing hardware systems on FPGAs, it is not always possible to debug all issues using simulation. Sometimes we encounter issues that only present themselves once the bitstream is downloaded onto the FPGA, and sometimes we need to test things at a scale that would be intractably long to simulate. In these cases, we turn to using debug hardware cores that allow us to capture information about the actual hardware as it runs on the FPGA. In this tutorial, we'll be introducing one such core and applying it to some hardware design to see how it could be used to assist in debugging.

The *Integrated Logic Analyzer*, or *ILA* for short, is a Xilinx debug core that can capture the values of probed signals and then display those values on a waveform viewer within the Vivado environment. A "probed" signal is a wire that is connected to the ILA, without disturbing the existing connections to that signal; i.e. the ILA simply watches the values of that signal. Figure 1 shows the block diagram for the ILA, taken from the ILA product guide [1]. The core contains a number of probes, a clock signal at whose edges the data is captured, and some trigger signals. The probes are the signals to be captured, and the trigger signals determine when we start capturing a new set of data. Note, the probe signals themselves can be used to trigger the ILA, so we usually don't include the trigger signals in our instantiations of the ILA. More details will be covered later in the tutorial.

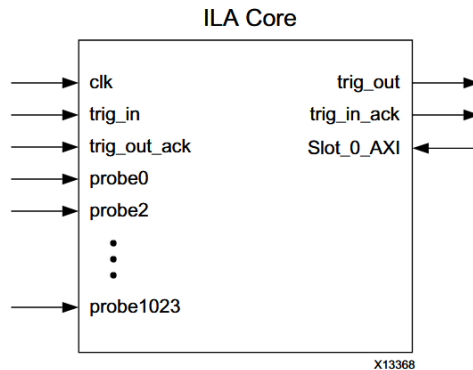


Figure 1: ILA block diagram

1 Marking Nets as Debug

There are two ways in which we can insert an ILA into a project. The first method is an automated insertion methodology used through the annotation of HDL source files, and the second is the actual instantiation of an ILA in IP Integrator view. The ILA Core instantiation will be covered in Section 4, and here we will discuss the debug flow available in Vivado for HDL source files.

Create a new Vivado project according to the steps in previous tutorial, making sure to add the *tutorial.v* and *tutorial.xdc* files as a source and constraint file respectively. The contents of *tutorial.v* are shown in Figure 2.

```
1  module tutorial (
2      input clk,
3      input reset,
4      input [7:0] swt,
5      output [7:0] led
6  );
7
8      wire direct_connect = swt[7];
9      wire [7:0] value_output;
10
11     //Counter
12     wire[26:0] count_limit = {swt[6:0],20'd0};
13     reg [26:0] counter;
14
15     always @(posedge clk) begin
16         if(reset) begin
17             counter <= 27'd0;
18         end else if(counter == count_limit) begin
19             counter <= 27'd0;
20         end else begin
21             counter <= counter + 1;
22         end
23     end
24
25     //Shift LED values
26     reg [6:0] led_shift;
27
28     always @(posedge clk) begin
29         if(reset) begin
30             led_shift <= 7'b0000001;
31         end else if(counter == count_limit) begin
32             led_shift <= {led_shift[5:0],led_shift[6]};
33         end
34     end
35
36     //Output
37     assign value_output = {direct_connect,led_shift};
38     assign led = value_output;
39
40 endmodule
```

Figure 2: Simple verilog module to shift output of LEDs with a counter

The *tutorial* module is the top level module of the project, and it's input and output signal names

should match those of the added constraints file. The module takes the switch values as input, and output some value to the LEDs. The most significant switch value is connected directly to the most significant LED, while the other switch are used to determine at which value some counter is to reset. Specifically, the lower 7 bits of the switches determine the number of 2^{20} cycles some counter should count to before resetting. Recall that our clock frequency is 100MHz, so a value of 100 on the switches should be approximately one second. Each time the counter resets, the values output to the lower LEDs is rotated to the left by one. In effect, the switches determine the rate at some pattern sifts through the LEDs. The pattern is fixed at 0000001 in binary. Note, the reset signal is active low.

To monitor some of these signals after we download the system onto the FPGA, we need to mark the signals in question *as debug*. There are a number of ways to mark a signal as debug, and we'll cover a few of them. First, we can annotate the HDL itself to mark a wire or register output as debug. For verilog, the notation to mark the signal *direct_connect* as debug for example is the following:

```
(* mark_debug = "true" *) wire direct_connect = swt[7];
```

Use this methodology and **mark** both the signal *direct_connect* and *value_output* as debug. Note, marking a signal as debug also prevents Vivado from optimizing a signal away, which can often happen in some of the optimization stages of Synthesis. Now run **Synthesis** for the project but do not proceed to *Implementation* when *Synthesis* completes. Instead, select **Synthesis** → **Open Synthesized Design** from the *Flow Navigator* pane on the left side of the Vivado interface. From the drop down menu in the top right of the Vivado interface, select **Debug** to switch to the debug view (see Figure 3 for reference).

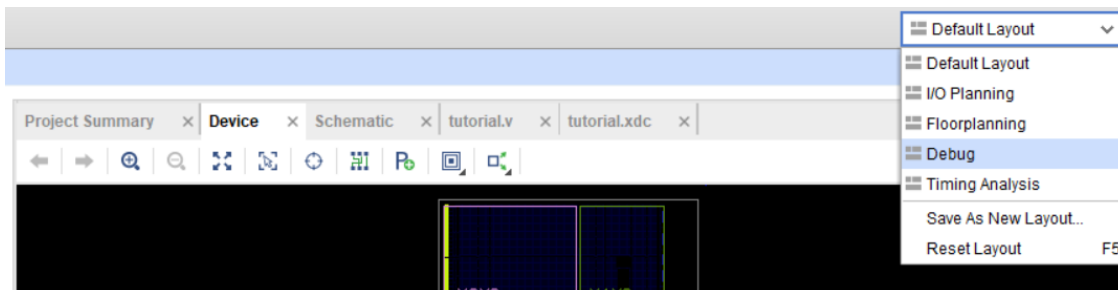


Figure 3: Selecting the Debug view from the open Synthesized Design

You should see the *Debug* tab in the bottom window. Here are listed all nets that have been marked as debug in your design; check to make sure both of the signals we've marked for debug are listed here. See Figure 4 for reference. Note, the nets are listed under the *Unassigned Debug Nets* heading, since these debug nets have not yet been connected to an *ILA*.

We can also add more nets to the debug list from the *Open Synthesized Design* view. In the **Netlist** window on the left, select the **Netlist** tab to see a list of all of the nets in your design. Note, this will not include any nets that have been optimized away during the recently executed *Synthesis*. You can **right click** any of the nets listed here and select **Make Debug** to add the net to the list of debug nets. Find the net corresponding to the *counter* signal and mark it as debug (see Figure 5). We should now have a total of three nets listed under the *Unassigned Debug Nets* header in the *Debug* tab. You can also mark a net *as debug* using the TCL Console, though we won't be demonstrating that in this tutorial.

Name	Driver Cell	Driver Pin	Probe Type
Unassigned Debug Nets (9)			
out_value (8)	Multiple	Multiple	
out_value[0]	FDSE	Q	
out_value[1]	FDRE	Q	
out_value[2]	FDRE	Q	
out_value[3]	FDRE	Q	
out_value[4]	FDRE	Q	
out_value[5]	FDRE	Q	
out_value[6]	FDRE	Q	
out_value[7]	LUT1	O	
direct_connect	IBUF	O	

Figure 4: List of nets marked for debug

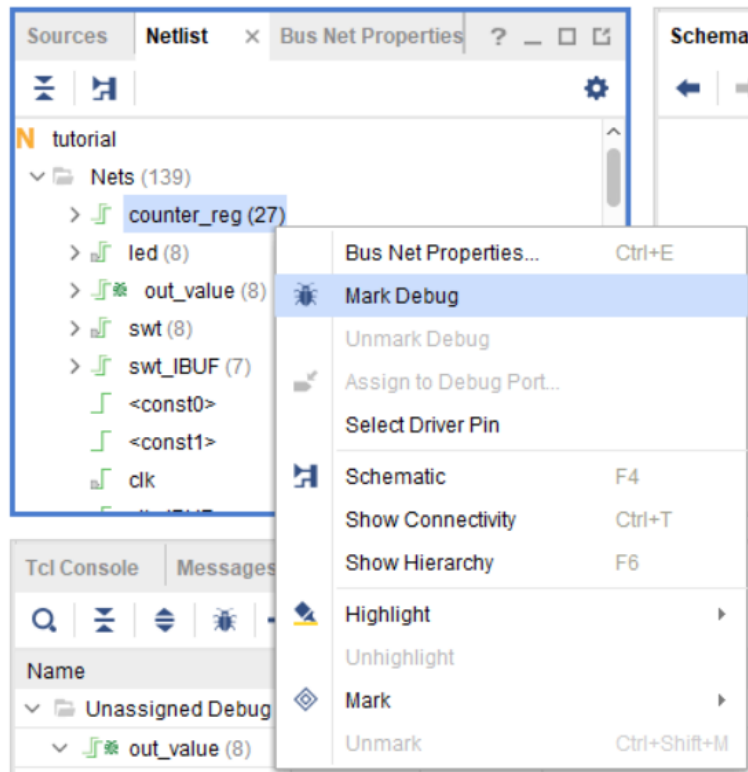


Figure 5: Marking the counter register as debug

2 Running the Debug Wizard

In order to connect the nets we've marked as debug, we need to run the *Debug Wizard*. To open the *Debug Wizard*, select **Tools** → **Set Up Debug** from the Vivado window tool bar. Click **Next** on the first window to bring up the list of nets we've marked as debug (see Figure 6). Each net is also listed with the associated clock, which is the clock used to sample the net. Some of the signals we've marked as debug are not synchronous signals, since they are combinational in nature, and as such have no clock selected by default. Our design has only one clock, so it should be used for all signals. **Right click** each signal without an associated clock, choose **Select Clock Domain...**, and select the only clock listed.

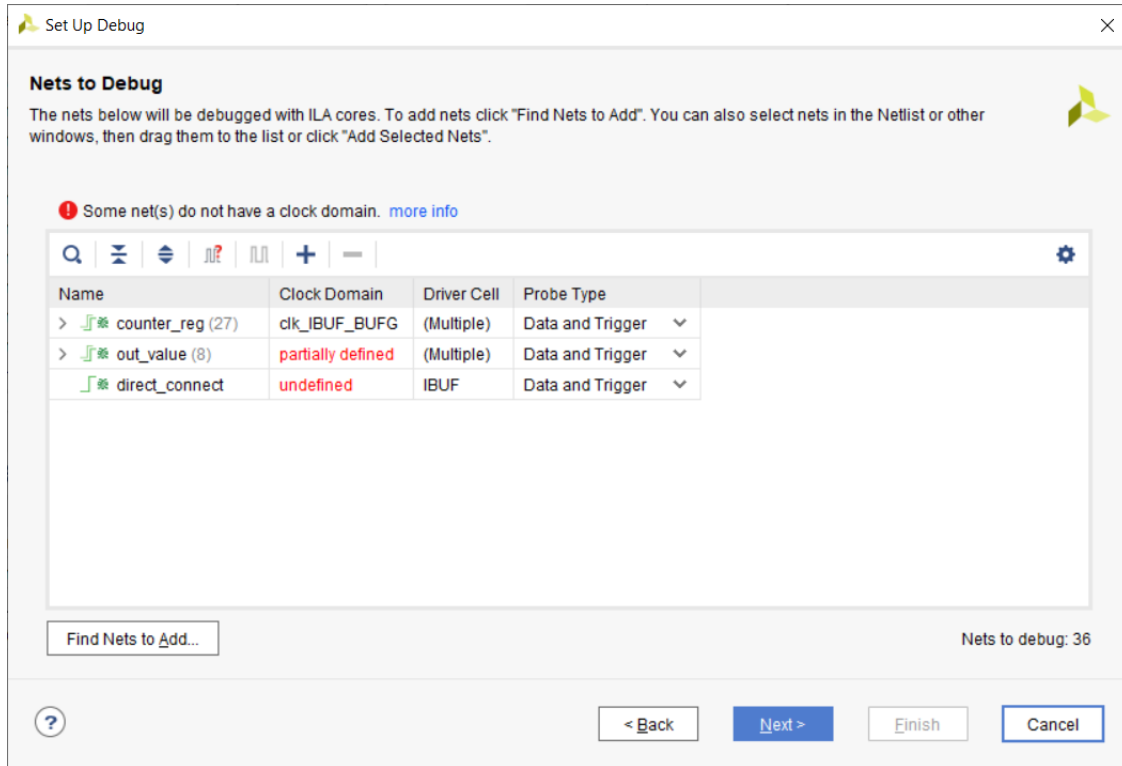


Figure 6: List of nets in the Debug Wizard

Click **Next** to advance to the *ILA Core Options* menu. In this window, we can select the depth of our *ILA* core, i.e. the number of samples of the probe signals that the *ILA* can store at a time. Also, the *Input pipe stages* setting is important if the *ILA* is causing timing failures, as increase pipeline registers can ease timing closure for the probed signals. Leave the *Sample of data depth* at **1024** and click **Next** and then **Finish** to complete the *Debug Wizard*. Check the *Debug* tab again to see that an *ILA* core has been added (named *u_ila_0*, or something along those lines) that contains all of our debug nets. The *Unassigned Debug Nets* listing should contain no nets at this time. See Figure 7 for reference.

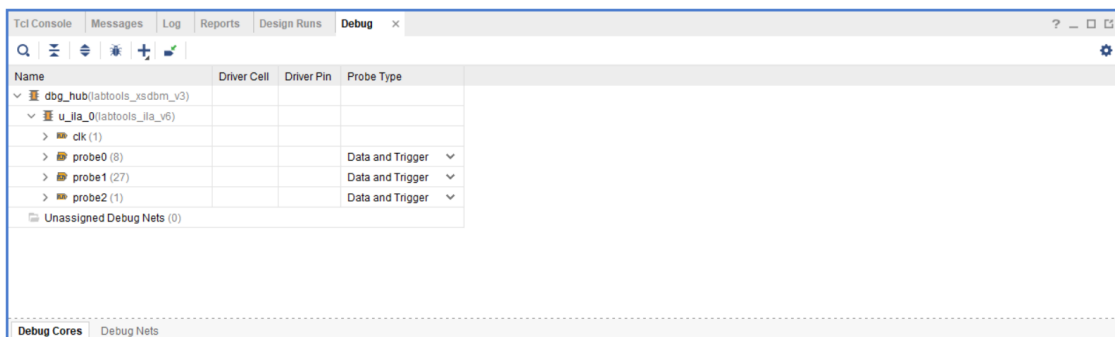


Figure 7: List of nets marked for debug, after assignment to ILA cores

3 Using the ILA in Hardware Manager

Now that we've added the *ILAs*, we can save the project, close the *Synthesized Design*, and rerun **Synthesis**, **Implementation**, and **Generate Bitstream** to generate the final bitstream that includes the *ILAs*. Open **Hardware Manager** and download the bitstream to the FPGA. Note, when selecting a bitstream to program the FPGA, you are also prompted to select a *Debug probes file*. This file contains information about all of the debug cores in the design. This field should be populated automatically to the *probes file* created when you generated the bitstream. See Figure 8 for reference.

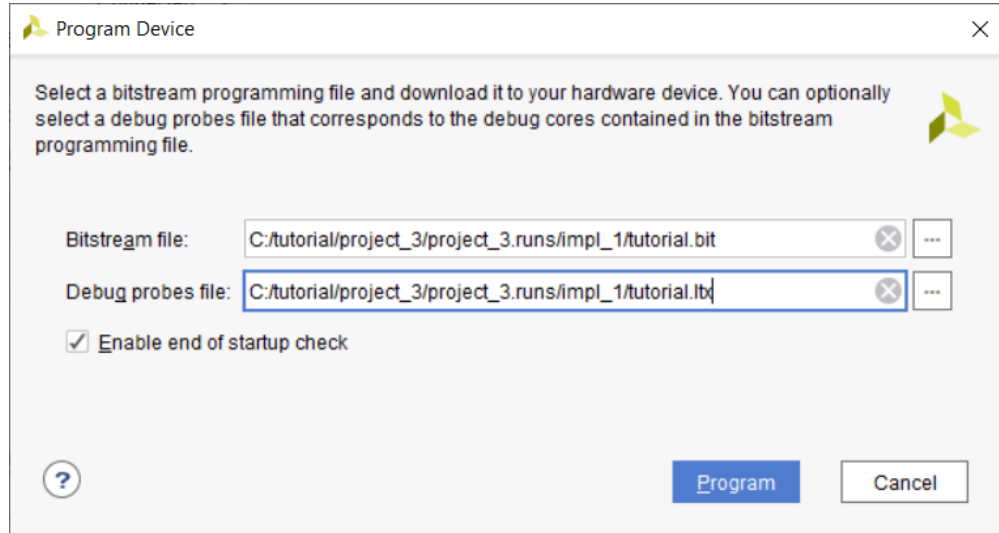


Figure 8: Selecting a bitstream and a probes file when programming the FPGA

Once the FPGA has been programmed, all debug cores should be listed under the FPGA in the *Hardware* window (see Figure 9). To open the ILA window, **double click** on the listed ILA, *hw_ila_1* in our case. This should open the ILA interface in a new tab of the editor pane (though it may have been open already by default). The ILA interface is how we control the ILA and view the probed results. It is made up of three panes, depicted in Figure 10. The top window contains the waveform viewer, where the values of our probed signals are displayed. The bottom left window contains two tabs, with the *ILA settings* and the *ILA status*. And the bottom right window contains two tabs, for modifying the *trigger* and *capture* settings. The ILA we've implemented doesn't support capture settings, so only the trigger settings are of note for us.

To setup the *ILA*, first we'll modify the settings, **open** the settings tab. The main setting of note for our purposes is the *Trigger position in window*. This setting determines at which position within the captured data the trigger event occurs. Earlier in the tutorial we mentioned that the *ILA* starts capturing data when the trigger is encountered. In reality, the *ILA* continuously saves the probed data in a circular buffer fashion, i.e. the newest probed data saves over the oldest data already stored. When a trigger is encountered, the *ILA* stops saving probed data once enough data has been stored such that the trigger event appears in the position indicated by the *Trigger position in window* setting. Leave this value at the default setting of **511**.

Next, we focus on the *Trigger* settings. Here, we setup what condition triggers a capture event. Press the **+** button to add a new probe to our triggers, and select *direct_connect* from the list. Next, we decide what specific condition on our trigger signal indicates a trigger event. We set the *Operator* field to **==** and set the *Value* field to **1 (logical one)**, which will setup a trigger event

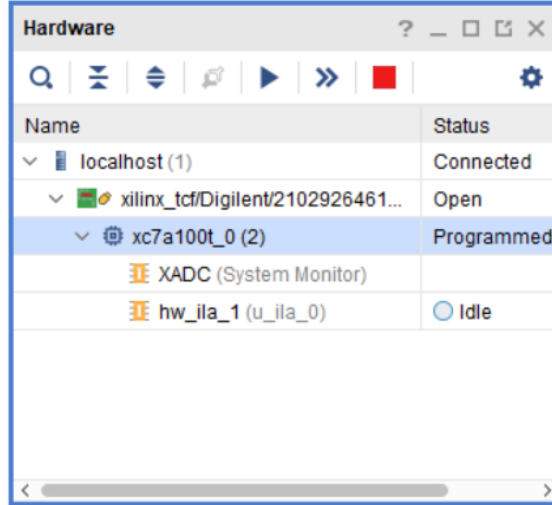


Figure 9: The FPGA device with a single ILA core listed

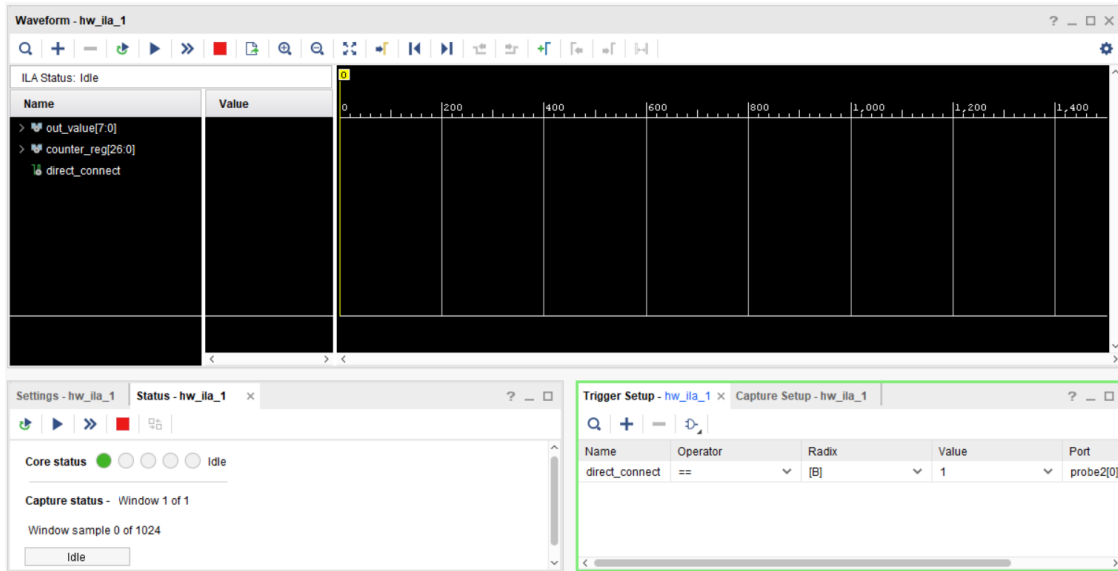



Figure 10: The ILA interface within Hardware Manager

when *direct_connect*, a.k.a. `swt[7]`, equals logical one. Note, we could also set the *Value* to *R* if you want a rising edge to trigger an event. If we have multiple trigger signals, the  button determines how to combine the different trigger event values; select the **global AND** option, which will trigger when all of the conditions are met (whereas the *global OR* will trigger when any of the conditions are met). For our single trigger condition, either option will work equally well.


Now that we've set our trigger, we need to run the *ILA*. Open the *status* tab to observe that the current status is *Idle*. **Push** all the switches on your board to the zero position, and then press the  button to *Run* the *ILA*, where it waits for the trigger event. The status bar in the lower part of the *status* tab should show that the *ILA* is 49% full, since we set our trigger position in the middle of our window. **Push** `swt[7]` up to trigger the event. The status should change quickly to *Full* and then *Idle* again. The waveform view should show the captured values now, as shown in

Figure 11. The marker labelled with a T shows where the trigger event occurred. If you zoom into the waveform, you should notice that the output values change every cycle, since we set the switches to zero. So even though we can't see the LEDs changing, they are in fact changing values. Try different values of the switches and re-Run the *ILA* to verify that the counter works as expected.

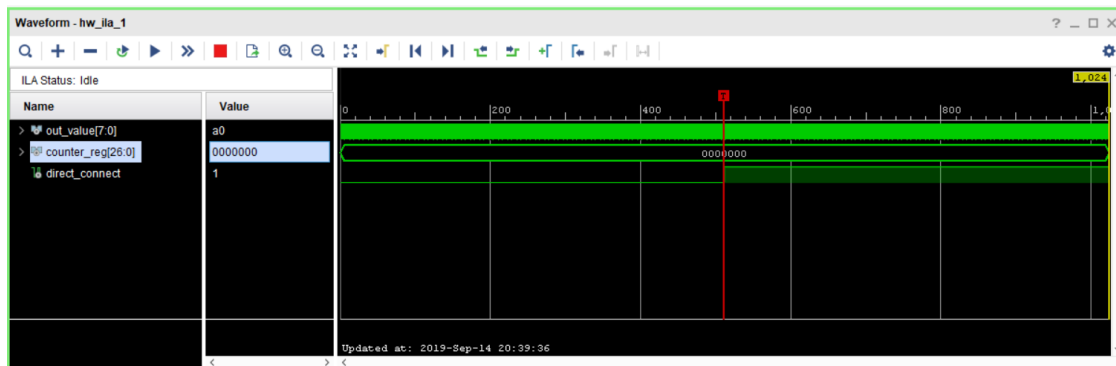


Figure 11: The ILA waveform output

4 ILA Core Instantiation in IP Integrator

Previously in the tutorial, we covered the automated *ILA* insertion flow using the *Debug Wizard*. We can also insert an *ILA* core manually into our design. This later methodology tends to be the easier approach for signals within the IP Integrator environment. For this tutorial, we'll be inserting an *ILA* into our DDR-based MicroBlaze system of tutorial 3. **Open** the project from tutorial 3, or alternatively create a new project and recreate the system we designed there. **Open** the block diagram. Note, we can also mark a signal as debug in this view and use the flow present in Section 1. To see how this works, **right click** on the *interrupt* output of the *AXI Uartlite* and select **Debug**. If we save the block diagram now and close it, we can run the steps of Section 1 to automatically insert the *ILA*. We won't be using that method in this part of the tutorial. **Right click** the signal you just marked as debug and select **Clear Debug** to remove the signal from the Debug Nets list.

Instead, we will be manually inserting an *ILA*. **Right click** anywhere in the block diagram editor and select **ADD IP**. Search for and select *ILA* to add a new *ILA* to the project. **Double click** on the newly added *ILA* to bring up its configuration window. From here, we can select between *Native* and *AXI* mode. The native mode has simple probes of some specified width, while the *AXI* mode include full *AXI* interfaces, which is a series of signals that implement the *AXI* protocol. For this section of the tutorial, we will be demonstrating the use of the *AXI*-based *ILA*, so make sure **AXI** is selected for the *Monitor Type*. Next, we can select how many probes to include. If we're using *AXI* mode, we can only have a single probe (since *AXI* has multiple signals itself), but we can select up to 1024 individual signals in native mode. Finally, we can select the depth of the *ILA*; we'll leave the *Sample Data Depth* at **1024** for this tutorial. See Figure 12 for reference. The next tab in the configuration allows us to set the parameters for each probe. The parameters for the *AXI* probes are deduced automatically from the design, though in native mode here is where you would select the bitwidth of each of your probes.

Close the configuration window. **Connect** the *SLOT_0_AXI* signal of the *ILA* core to the *AXI* input to the *MIG DDR Controller*. We also need to connect the *clk* signal of the *ILA* to match the clock used for this *AXI* interface, which is the *ui_clk* output of the *MIG* itself. You should see a connectivity similar to Figure 13. Once you have an *ILA* connected in the *Block Diagram* editor

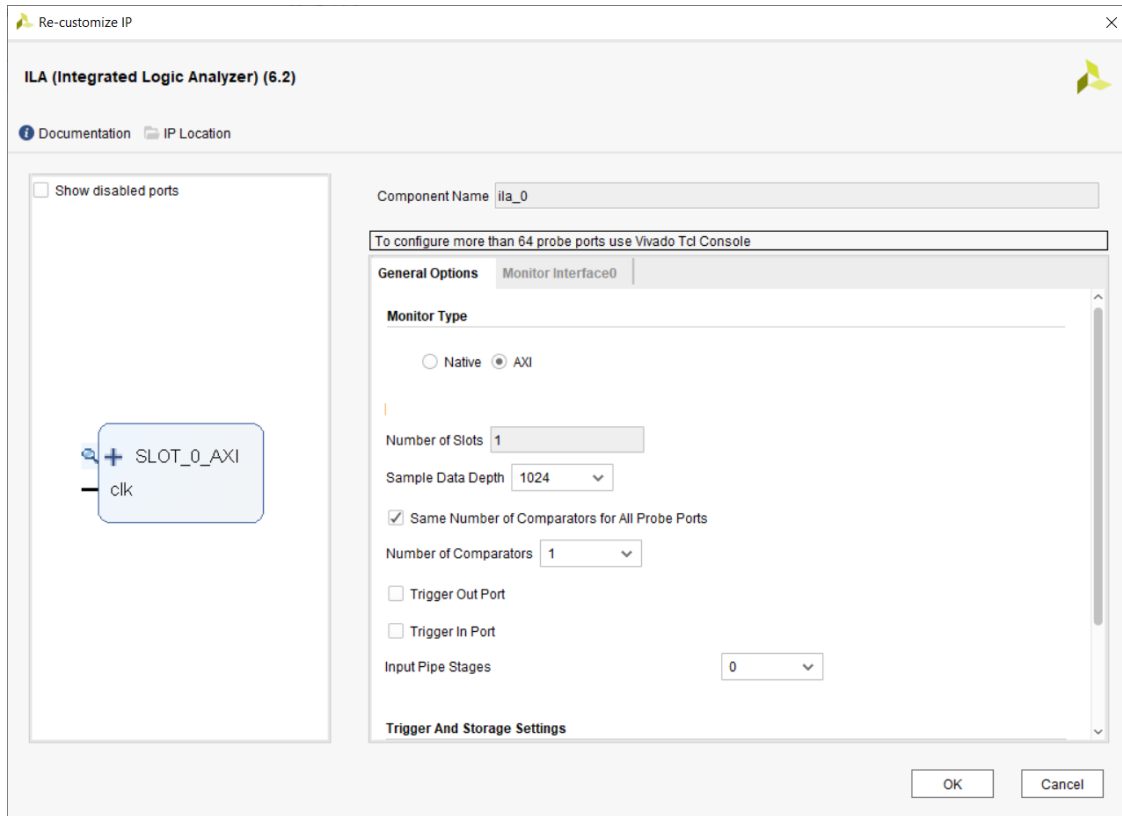


Figure 12: Configuration window for the ILA core

as we've just done, there are no further steps needed to enable the debug of this signal. We can now **Save** the block diagram, run **Validate Design**, and then run **Synthesis**, **Implementation**, and **Generate Bitstream** to create the final bitstream.

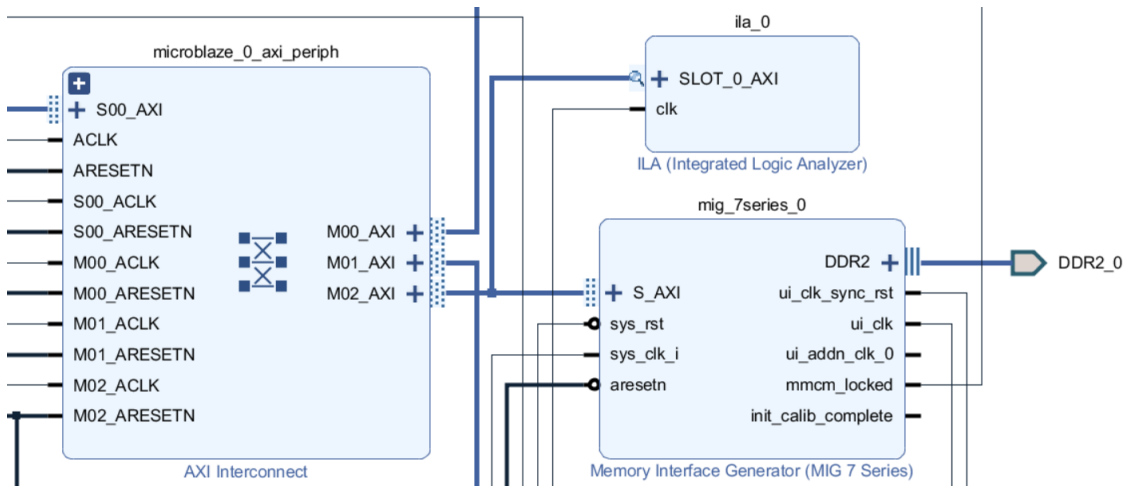


Figure 13: The ILA core connect to the AXI interface of the MIG

5 Using an ILA with an AXI Interface in Hardware Manager

Open **Hardware Manager** and program the FPGA with your newly created bitstream, making sure that the probes file is automatically selected for you as we did in Section 3. In the *Trigger Setup* tab, press **+** button to add a new probe for a trigger event. From the list, select the AXI signal that ends in **WVALID**. Repeat this step and add the signal that ends in **WREADY**. Set the *Value* field for both triggers to *1 (logical one)*. These two signals correspond to the write data channel, where we expect the write data to be transmitted. The *valid* signal indicates that there is valid data being transmitted, and the *ready* signal indicates that the data is accepted. If we set the trigger to **global AND** by pressing the **AND** button, we will trigger on valid data transmitted. **Run** the *ILA* to wait for the trigger event.

We cannot trigger the event from the switches, so we have to run an application on the MicroBlaze to trigger the *ILA*. Without closing the *Hardware Manager*, **Export** the Hardware and start **Vivado SDK**. Open the *helloworld* application created in tutorial 3. Note, sometimes when we change the MicroBlaze system, our SDK applications fail to compile. We need to refresh the include files included in the board support package, so open an include file in the *hello_bsp* directory (*xil_printf.h* for example) and press **F5** to refresh the files; you should now be able to run the application. Open the *Run Configuration* created in tutorial 3, but don't run it quite yet. Uncheck the **Program FPGA** option and then press **Apply** (See Figure 14). We don't want to reprogram the FPGA, as that will reset our ILA. Press **Run** to run the application.

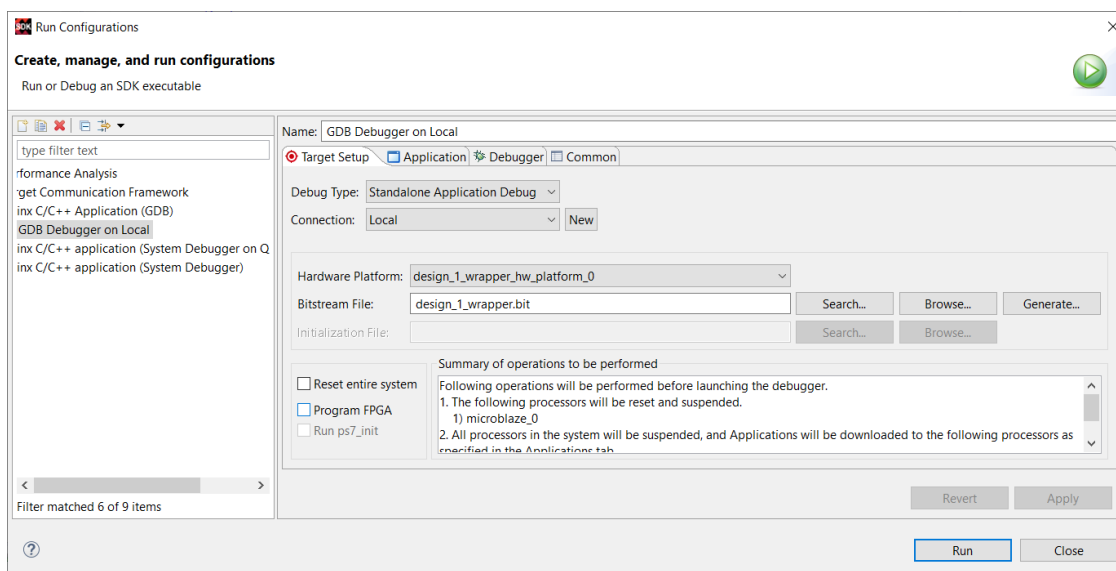


Figure 14: Run Configuration dialog, with Program FPGA unchecked

Return to *Hardware Manager*, the event should have triggered and we should see some output in the *Waveform Viewer*. Find the signals that end in *WVALID*, *WREADY*, and *WDATA*. You should see the values we wrote to the memory on the *WDATA* signal, with subsequent values appearing after both *WVALID* and *WREADY* go high at the same time.

6 Summary

The *ILA* is an effective way to debug our systems as they run in real-time on the FPGAs. *ILAs* can either be automatically inserted based on the Vivado Debug flow, or manual inserted as an actual *ILA* core. With properly setup trigger values, we can capture our probe values when a specific event happens. The ILA can even be used in AXI mode, to capture the AXI transactions and verify the reads and writes done by AXI master devices (like the MicroBlaze).

References

- [1] *Integrated Logic Analyzer Product Guide*, https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf.