Xilinx Vivado ILA Lab
Part of DAT096
January 20, 2016

Dmitry Knyaginin

Room 4103, D&IT, dmitryk@chalmers.se

CSE, Chalmers University of Technology

Updated: February 16, 2018 by Vikram Jain (vikramj@student.chalmers.se)

---

1 Introduction

Debugging FPGA designs can be complex and time consuming. Synthesis can take hours, and so a debug process with multiple iterations can cost you a lot of time. Thus it is important to use an efficient debug methodology. There are a few choices:

- HDL simulation
- HDL co-simulation
- Hardware-in-the-loop
- In-circuit debugging

HDL simulation requires an HDL testbench that generates inputs for the Design Under Test (DUT) and compares its outputs. The simulation process might be very slow. Inputs must be such that they can be imported or generated from code, which is a limitation since some designs might require real-life inputs that are difficult to reproduce.

HDL co-simulation is similar to HDL simulation, but the testbench is implemented as a high-level model (e.g., in MathWorks Simulink). The testbench includes a generator of inputs, a reference model of the DUT[1], and means to compare the outputs of the reference model and those of the DUT. During co-simulation, inputs are communicated down to the HDL simulation of the DUT, and outputs are communicated back to the testbench. Co-simulation might be slower than HDL

[1] The DUT is typically created from its reference model, either generated or by hand.

simulation, since the reference model has to be executed as well. The advantages of this methodology are that 1) the DUT is ran against its reference model, and 2) tools like Simulink offer rich means for generating inputs and analyzing outputs.

Hardware-in-the-loop runs the DUT on the target FPGA, and thus is much faster than HDL simulation or HDL co-simulation. Similarly to co-simulation, the testbench is implemented as a high-level model. A limitation is that there is no way to inspect the internals of the DUT, and so this methodology is useless when bugs are not trackable from outputs.

In-circuit debugging is irreplaceable when a design must be debugged on the target FPGA, i.e., when Simulink models or HDL testbenches cannot reproduce real-life inputs. It is also irreplaceable when the behavior of the DUT on the target FPGA is not as expected, e.g., due to timing violations. An Integrated Logic Analyzer (ILA) is instantiated in the DUT and thus can probe its internal signals, capture the signals of interest for a specified number of samples, and then transfer the samples up to the host machine via the JTAG interface. A drawback is that an ILA core occupies FPGA resources, can affect timings, and changes to its configuration require re-implementing the DUT (from synthesis to bit-file generation). However, this methodology is key to efficient debugging, and thus it is the subject of this lab.

Xilinx Vivado has integrated debug capabilities [1], and in this lab we will study the ILA. There are multiple debug flows [1] and for brevity we will discuss the following two:

1. Configure an ILA core from the IP Catalog → Instantiate the ILA core in the DUT → Synthesize the DUT → Implement → Program → Set triggers and acquire waveforms;
2. (optional) Apply the mark debug attribute to the signals of interest → Synthesize the DUT → (if not marked) Mark as **Debug** signals of interest via the GUI → Open the synthesized design and **Set Up Debug** via the GUI → Implement → Program → Set triggers and acquire waveforms.

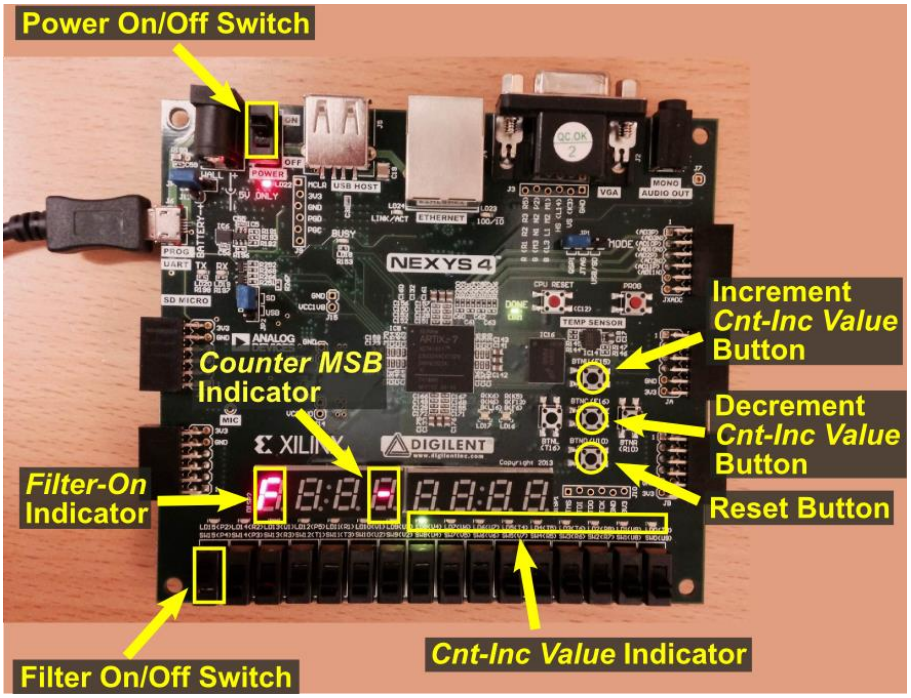

Figure 1: Nexys4 board

2 The Board and the Project

We will use the Digilent Nexys4 board [2]. Unpack the projects from dat096-ila-labs.zip into a directory of your choice, further referred to as $PROJECT_HOME. There are three projects: baseline, lab1 and lab2. All of them implement the same DUT but with different debug capabilities, that we will discuss later. The DUT implements a free-running counter with an increment (*Cnt-Inc*) controlled by the user. Figure 1 shows the board and its buttons, switches, and indicators that we use as follows (please locate the component names on the board itself):

- BTNU – increments the Cnt-Inc value by a power of two (shift left by one bit).
- BTNC – decrements the Cnt-Inc value by a power of two (shift right by one big).
- BTND – resets the DUT.
- SW15 – turns on/off the filter (debouncer).
- LD8-LD0 (the array of LEDs) – displays the binary Cnt-Inc value, where LD8 shows the Most-Significant Bit (MSB).
- The left-most 7-segment indicator of DISP2 – shows F when the filter (debouncer) is on.

- The middle segment of the right-most 7-segment indicator of DISP2 – displays the MSB of the free-running counter (the greater the Cnt-Inc value, the faster the segment blinks).

You can remap the functions in the Nexys4-DDR-Master.xdc file located in $PROJECT HOME/lab1/lab1.srcs/constrs 1/new

Controlling the design:

- The increments and decrements of the Cnt-Inc value are wrapped around. I.e., an increment of 2^8 results in 2^0, and a decrement of 2^0 results in 2^8.
- To enable/disable the filter (debouncer), set SW15 in the top/bottom position, respectively.

3 Problem Statement

Let's identify the problem:

- Set SW15 into the bottom position (turn off the filter).
- Open the lab1.xpr project in Xilinx Vivado, generate the bit file, and program the FPGA.
- Observe that the left-most 7-segment indicator of DISP2 is blank, the Cnt-Inc value is 1 (LD0 glows), and the middle segment of the right-most 7-segment indicator blinks approximately once per second.
- Increment and decrement the Cnt-Inc value by pushing BTNU and BTNC.
- Does the Cnt-Inc value change as expected? _____
- If no, how can you explain the observed behavior? _____

You might have observed that sometimes a single push of BTNU shifts the Cnt-Inc value left by more than one bit. Likewise, a single push of BTNC shifts the Cnt-Inc value right by more than one bit. This is the bug that we address in this lab. If we used an HDL simulator, we would not observe this bug (try yourself if curious and have spare time). Apparently, there is a problem with the buttons and the shape of the signal they produce. This is where in-circuit debugging comes irreplaceable.

The probability of the bug depends on the wear of the buttons. The boards are relatively new, and so the bug might not show up easily (but of course we must take care of all bugs, even the rare ones). In the worst case you can try to remap the increment and decrement functions to other buttons (or switches), hoping that they would expose the bug easier.

Let's look inside the FPGA and inspect the signals coming from BTNU and BTNC. Do:

- Make sure that SW15 is still in the bottom position.
- In top.vhd, uncomment lines 219-242. These lines instantiate ila, the ILA core that samples at 100MHz (the system clock), and ila slow, the ILA core that samples at about 1.56MHz (the system clock divided by 2^6).

Warning: You should never use a divided clock signal for clocking circuits. The right way is to use ether 1) the system clock with a clock-enable signal or 2) a synthesized clock (see the Clocking Wizard [3]) when possible. However, the ILA core does not have a clock-enable input and the divided clock is too slow to be synthesized. Since the ILA core is a non-critical component, it is acceptable to clock it like we do in this project.

- Re-implement the design.
- In order to run the ILA at a lower clock frequency, we need to reduce the JTAG clock frequency to its minimum. In the **Program and Debug** section, click on **Open Target** and select **Open New Target**, click next. Check that the name of the Host is set to Local server and click next to continue. In the resulting window, select the JTAG Clock Frequency at 125 KHz and click next. Click Finish to continue and select **Program Device** to dump the code to the board.

  Note: Do not use Auto Connect to connect the hardware.
- After successful programming, the Hardware Manager window should automatically open two ILA Dashboards named hw ila 1 and hw ila 2, that connect to ila and ila slow, respectively. Click on the **Float** icon  of both Dashboards, place them on different screens, and maximize them for ease of viewing.
- The Waveform window of hw ila 1 shows the following signals:
  - clk div2 – system clock divided by two (just a reference).

- inc – signal from BTNU.
- dec – signal from BTNC.
- inc int le – output of the leading edge detector for inc.
- dec int le – output of the leading edge detector for dec.
- cnt inc[8:0] – the Cnt-Inc value (a 9-bit bus).
- cnt[26:0] – free-running counter value (a 27-bit bus).

- The Waveform window of hw ila 2 shows the following signals:
  - clk slow div2 – system clock divided by 2^{15} (just a reference).
  - fil – signal from SW15.
  - inc – signal from BTNU.
  - dec – signal from BTNC.
  - inc int – input of the leading edge detector for inc.
  - dec int – input of the leading edge detector for dec.
  - cnt inc[8:0] – the Cnt-Inc value.
  - cnt[26:0] – the free-running counter value.

- Why are inc int le and dec int le of no interest in hw ila 2? _____

- Click the **Run trigger for this ILA core** icon andthenthe **Toggle auto re-trigger Mode for this ILA core** icon inboth hw ila 1 and hw_ila 2.Ignorethefollowing error message (since we know that the core clock is slow):

> ERROR: [Labtools 27-1395] Unable to arm ILA 'hw ila 2'. The core clock is slow or no core clock connected for this ILA or the ILA core may not meet timing.

- Make sure that the core status switches to **Waiting for Trigger** in both hw ila 1 and hw_ila 2. Push BTNU or BTNC once and analyze the waveforms that get uploaded.

If you want to stop the trigger for an ILA core, remember to first disable the auto re-trigger mode by clicking the **Toggle auto re-trigger mode for this ILA core** icon  again.

- What is the shape of the inc and dec signals? _____

  How many times did cnt inc change? _____

If cnt is not properly displayed, copy and paste it (Ctrl+C, Ctrl+V) in the **Waveform** window. This seems to be a bug of the GUI.

- Repeat the last two steps until you observe that cnt inc changes more than once. How can you explain this behavior? _____

  _____

You have just observed signal bouncing that is characteristic for buttons, switches, and other mechanical inputs. The ILA lets us expose such problems. Figures 2 and 3 show example waveforms. Each waveform contains 4096 acquired samples for each of the signal of interest.

The sample that triggered acquisition is highlighted with the red marker. Figure 2 shows that dec is bouncing, causing multiple changes of cnt inc: first from x001 to x100 (recall that the decrement is wrapped around), then to x080, and so on until x020. Figure 3 does not show that dec is bouncing (ila slow is too slow to capture that) but it shows the entire duration of BTNC being pressed and the final cnt inc value of x020.

Figure 2: Bouncing dec captured by ila

Figure 3: Bouncing dec captured by ila slow

4 Solution

The filter mentioned earlier solves the problem of bouncing signals. It is a simple synchronous circuit that does not propagate its input until it has stayed stable for a user-defined number of cycles (see filter.vhd for implementation details). Do:

- Set SW15 to the top position (turn on the filter), observe that the left-most 7-segment indicator of DISP2 shows F.
- Push BTND once to reset the design.
- Make sure that both hw ila 1 and hw ila 2 are **Waiting for Trigger**. Push BTNU or BTNC once and analyze new waveforms.
- What is the shape inc and dec? _____

  What is the shape of inc int and dec int? _____

  What is the shape of inc int le and dec int le? _____

  How many times did cnt inc change?

You should have observed that now inc_int and dec int transition only after a significant delay that eliminates bouncing. Now the design is robust and behaves as intended: each press on BTNU or BTNC changes the Cnt-Inc value exactly once. Figures 4 and 5 show example waveforms. Figure 4 still shows that dec is bouncing, but now dec int le is low and cnt inc does not change (the delay

of the filter is much longer than 4096 cycles at 100MHz). Figure 5 shows that now dec int is delayed and cnt inc changes only once.

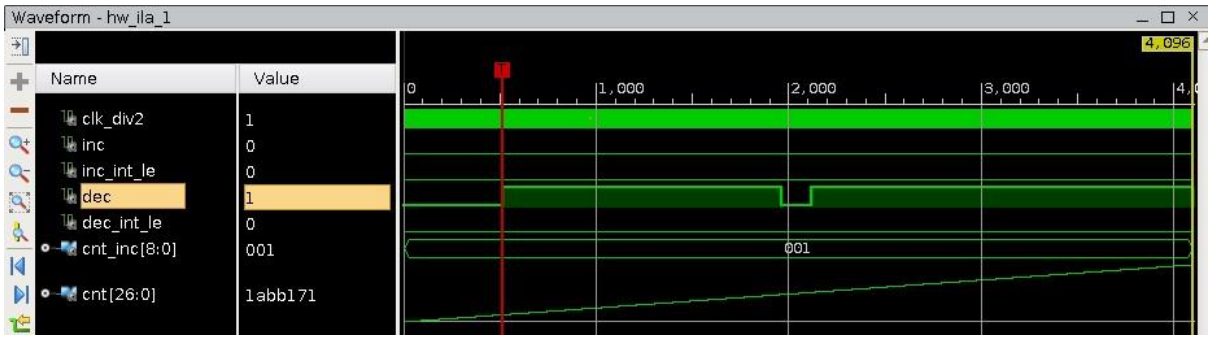

Figure 4: Filtered dec captured by ila

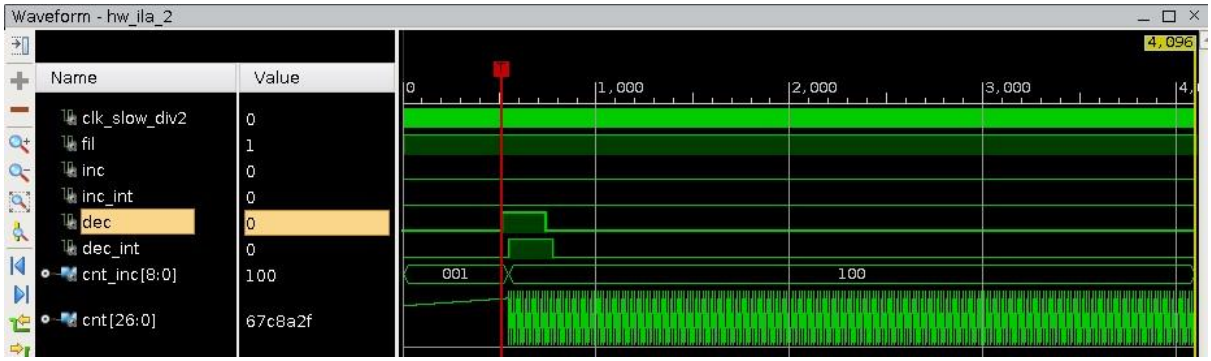

Figure 5: Filtered dec captured by ila slow

5 In-Depth Look

Let's go over the steps required to create the ILA cores. In the very end of Section 1 you read about two debug flows. We will start with the first one, i.e., when we explicitly instantiate the ILA cores in top.vhd. As you might have realized, this is the debug flow that we have been following so far.

Disclaimer: This lab has been created using Vivado 2015.2 for Linux. There might be insignificant differences between this version and the version that you are running.

5.1 Flow 1

In this debug flow we manually instantiate ILA cores. The respective project is lab1.xpr.

You can use baseline.xpr if you want to start from the baseline without ILA cores.

Tip: Execute Tcl command reset_project to clean up a project.

The debug flow has the following steps:

- Plan which signals to probe. Consider adding reference signals like the sampling clock divided by two (e.g., clk_div2). It is convenient to have such signals in the waveform, since we cannot use the sampling clock itself (it would appear as a constant in the waveform).
- Plan how many ILA cores to use. We obviously need a core like ila that samples at the system clock rate to capture detailed waveforms. When we know that we also want to sample slow signals (e.g., inc and dec), we can improvise and add a core like ila_slow. Let's start with ila.
- Find **Project Manager** in the **Flow Navigator** window (top-left corner of the default Vivado window layout). Click on **IP Catalog** there. This opens an **IP Catalog** tab in the **Project Manager** window. Type debug in its search field. Find **ILA (Integrated Logic Analyzer)** under **Vivado Repository** / **Debug & Verification** / **Debug** and double-click on it to start the ILA core generator. The **Customize IP** window appears where you can choose **Component Name**, **General Options**, and **Probe Ports**.
- On the **General Options** tab you only need to set the number of probes (i.e., signals to be sampled) and the sample data depth (the number of samples per signal). Mind that the core temporarily stores acquired samples and thus utilizes FPGA memory resources.

  In large projects there might be not enough resources for both the DUT and the ILA core. In addition, transferring acquired samples from the FPGA to the host machine takes time. Thus, it is recommended to choose the least number of probes and the least sample data depth that are sufficient for obtaining meaningful waveforms[2]. E.g., the sample data depth of 4096 is informative enough for the purposes of this lab. You do not need to change the other parameters in this tab, but you should read about them when you have time [1].

[2] If you have spare time, you can try to choose the maximum sample data depth and try to generate a bit file. [3]You could keep the default **Synthesis Options** but this would start a rather long synthesis process.

- On the **Probe Ports** tab you need to define the width of each probe port according to the widths of the signals that you intend to sample. Keep the number of comparators set to 1 for all probe ports.

  Why did we set the width of **PROBE 5** to 9 in ila but to 1 in ila slow? _____

  _____

- Click **OK**. This opens the **Generate Output Products** window. Set **Synthesis Options** to **Global**[3] and click **Generate**. The generated ILA core will appear under **Design Sources** in the **Sources** / **Hierarchy** window of the **Project Manager** window.

> You can always reconfigure the core by double-clicking on it.

- Among other files, an instantiation template is generated. Locate it in the **Sources** / **IP Sources** window of the **Project Manager** window, under **IP** / core name / **Instantiation Template**. Use the template to declare the core as a component and then to instantiate it in top.vhd. Connect the probe ports appropriately, where the clk port has to be connected to the sampling clock signal.

> Some versions of Vivado might generate std logic _vector(0 downto 0) instead of std logic for probe ports that are just one bit wide. The simplest workaround is to slice the probe port when mapping it, e.g., probe3(0) => probe3 int, where probe3 is of type std logic vector(0 downto 0) and probe3 int is of type std _logic.

- Repeat the above steps for ila slow.
- Synthesize and implement the design and program the FPGA.
- Upon successful programming, one dashboard per ILA core opens automatically in the **Hardware Manager** window. Note that Vivado might rename signals by adding suffixes. This does not change the functionality, and you should be able to identify all of the signals of interest. You can always rename the signals back to their original names.
- Set up triggers in the dashboard's **Trigger Setup**. You can add and remove probes used as triggers by clicking on the **Add probe(s)** icon  and the **Remove selected probe(s)** icon . We are interested in signal transitions from low to high, and so the **Compare Value** should be "== [B] R" (equal to rising edge, i.e., **0-to-1 transition**).

When done, click on the **Set trigger condition to Global AND, OR, NAND or NOR function** icon  and select **Set Trigger Condition to 'Global OR'**.

- The dashboard's **Settings** allow us to split the available sample data depth (the total number of samples) into multiple sampling windows, such that one window is captured per trigger and all of the windows are shown in one compound waveform. We can also choose the trigger position in the window, such that the ILA core captures a number of samples right before the sample that triggers acquisition. This is a convenient way to observe signal values before and after the trigger.
- Change the waveform format as you like:
  - Create buses from multiple signals.
  - Reorder signals and buses by dragging them to new positions.
  - Change their names, color, radix, and the bus bit order.
  - Duplicate signals and buses (Copy / Paste), add or remove them.
- Now you are ready to trigger the cores and capture the waveforms like we did in Section 3.

5.2 Flow 2

In this debug flow the ILA cores are instantiated semi-automatically. The respective project is lab2.xpr. Again, you can use baseline.xpr to start from the baseline without ILA cores. Do:

- If you choose to continue with lab1.xpr:
  - Comment out lines 99-124 and 219-242 in top.vhd since now we do not want to explicitly instantiate the ILA cores. Delete ila and ila_slow from the **Design Sources** in the **Sources** window of **Project Manager**.
  - Uncomment lines 136-149 in top.vhd to apply the mark_debug attribute for the signals of interest that protects them from getting optimized away during synthesis.
- Synthesize the DUT (but do not implement it yet).
- Open the synthesized design from the **Flow Navigator** window.

- Click on **Set Up Debug** to open a wizard and then click **Next**. The wizard will try to trace clock domains. Since clk slow does not create a clock domain, we will not be able to select it as the sampling clock. Thus, this wizard does not let us create an equivalent of ila _slow. Though, there is a way to do it and we will discuss it soon.
- The next window shows **Nets to Debug**, where some nets might have an undefined clock domain (e.g., the system clock clk and the unused bits of cnt _inc). Here we need to choose signals to sample. We have marked as debug signals sampled by both ila and ila _slow, and so now we need to delete signals that we do not intend to sample by this ILA core. Since we are configuring an equivalent of ila, we should delete the signals intended for ila _slow (clk slow _div2, fil, inc int, dec int). In addition, we need to delete the clock signal (clk) and any unused signals (e.g., the unused bits of cnt inc).
- Click **Next** to move on to **ILA Core Options**. Choose a sample data depth and keep the other parameters unchanged. Click **Next** and then click **Finish**.
- The wizard generates the ILA core but does not modify top.vhd. Instead, it modifies Nexys4 _Master.xdc (the constraints file). In order to see the changes, click on the **Save Constraints (Ctrl+S)** icon  (top-left corner of the Vivado window), open Nexys4 _Master.xdc, and scroll to the very end.
- Now find the **Debug** window in the **Synthesized Design** window. To be able to reuse the existing hw ila 1 dashboard, order the port assignments of u ila _0 in the same way as they are assigned to ports of ila in top.vhd, lines 222-228. You can drag-and-drop signals from the **Netlist** window to the desired probe debug channel. To remove a signal from a probe debug port, drag-and-drop it to the **Unassigned Debug Nets** in the **Debug** window. Do not worry about unconnected debug ports and channels while re-assigning signals. Once done, just right-click on u ila _0 in the **Debug** window and select **Remove Unconnected Debug Ports and Channels** to clean up.

> For the purposes of this lab it is important that we create ILA cores that are identical to ila and ila _slow: as has been mentioned, if the respective cores are identical, we can reuse the existing dashboards.

- Now let us create an equivalent of ila slow. In the **Debug** window, click the **Create Debug Core** icon  .
- In the **Create Debug Core** window that appears, set C DATA _DEPTH (the sample data depth) to 4096 and leave the other parameters unchanged. Click **OK**. This creates u ila 1 that you can find in the **Debug** window.
- u-ila 1-has only one probe and both clk and probe0 are unassigned. Drag-and-drop clk slow from the **Netlist** window to **Ch 0** of clk. This way, you assign clk slow as the sampling clock for u ila 1.
- Next, drag-and-drop clk slow div2 to Ch 0 of probe0. Right now the goal is to reproduce the probes of ila slow in the same order as in top.vhd, lines 234-241. To add a probe port, click on the **Create Debug Port** icon in the **Debug**  window. In the **Create Debug Port** window that appears, select u ila 1 from the list of debug cores (click on the **Select debug core** icon ) and click **OK**. Note that the port's width will change automatically when you drag-and-drop signals into it. Repeat this process for all of the ports (this is the tedious part) and finally clean up by right-clicking on u ila 1 and selecting **Remove Unconnected Debug Ports and Channels**.
- Click on the **Save Constraints (Ctrl+S)** icon  to append the new ILA core configuration to Nexys4-DDR-Master.xdc.
- Now you can implement the design and program the FPGA. After that, you will see the familiar ILA dashboards in the **Hardware Manager** window.

Flow 2 is more interactive than Flow 1 but as you have probably observed it can be more tedious and it imposes limitations (e.g., the automatic clock domain selection). Typically it is more efficient to use Flow 1.

6 Conclusion

We have gone through a case study for in-circuit debugging using Xilinx Vivado ILA. We did not use all of its debug features but instead focused on the most essential ones. Thus it is important that you continue with reading the documentation [1] to get prepared for advanced in-circuit debugging.

References

[1] Xilinx, Inc., “Vivado Design Suite User Guide. Programming and Debugging.” http://www.xilinx.com, Apr. 2015.

[2] Digilent, Inc., “Nexys4 FPGA Board Reference Manual.” http://www.digilentinc.com, Nov. 2013.

[3] Xilinx, Inc., “Clocking Wizard v5.1. LogiCORE IP Product Guide. Vivado Design Suite.” http://www.xilinx.com, Apr. 2015.