Debugging with Reveal Usage Guidelines and Tips

Application Note

FPGA-AN-02060-1.1

April 2024

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUNDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice..

Contents

Figures

Tables

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
DRC	Design Rule Check
DUT	Design Under Test
EBR	Embedded Block RAM
FPGA	Field Programmable Gate Array
ILA	Integrated Logic Analysis
JTAG	Joint Test Action Group
LMMI	Lattice Memory Mapped Interface
LUT	Look-up Table
POR	Power-On Reset
RAM	Random Access Memory

1. Introduction

Reveal is an integrated tool in the Lattice Radiant™ and Lattice Diamond® software used for debugging Lattice FPGAs. On the software side, Reveal consists of two main tools: Reveal Inserter and Reveal Analyzer/Controller. At a high-level, Reveal Inserter is used to insert debug cores into a design, while Reveal Analyzer/Controller is used to detect a debug device and perform debugging.

There are two main types of debug cores, which can be added to Lattice FPGA projects using Reveal Inserter. These are known as Analyzer and Controller cores. Controller cores are typically used to control and manage certain parts of a design, while Analyzer cores are used to tap into signals and perform logic analysis. Some common usages of Analyzer cores include power-on-reset debugging, condition monitoring, and capturing signals. For Controller cores, the main possible usages include virtual switches and LEDs, user memory access, user register control and monitoring, and interfacing with Hard IP.

1.1. Reveal Usage Flow

Although, there are some differences between the Lattice Radiant and the Diamond versions of Reveal in terms of feature support, their general usage is fairly similar.

Once a design is ready to be debugged, the first step is to configure a logic analyzer or controller core using Reveal Inserter.

Next, the debug cores are inserted to the project using Reveal inserter. This requires a new bitstream to be generated. The user needs to rerun the project implementation flow through synthesis, MAP, PAR, and bitgen to generate a new bitstream. Once the new bitstream is generated, the next step is to program the target device using the Lattice Programmer tool.

Lastly, once the project's target device is programmed with the updated bitstream, the Reveal Analyzer/Controller is used to detect the device with Reveal debug cores inserted. With the debug device detected by Reveal Analyzer/Controller, Reveal can then be used to perform various debugging actions depending on the type of debug core(s) that are inserted.

1.1.1. General Reveal Flow Summary

The following is the summary of the Reveal flow:

- Add and configure debug cores using Reveal Inserter
- Perform DRC (design rule check) and insert Reveal to project
- Synthesis, MAP, PAR, and bitstream generation
- Program FPGA using Programmer
- Detect device using Reveal Analyzer/Controller
- Begin debugging

1.2. General Information

1.2.1. Design Rule Checking

A useful feature of Reveal Inserter is its DRC function, which can be used to check certain parts of a Reveal project to ensure that the setup is correct. Before inserting a debug core to a design, always perform a Reveal DRC to avoid mistakes related to the setup of Reveal's debug cores.

Important:

Performing a DRC does not ensure that a design is functional, or that it detects errors outside of Reveal. Reveal DRC only checks a few Reveal-related areas, such as whether a signal has multiple drivers or if a debug clock is selected or not.

Other areas covered by Reveal Inserter's DRC include: device resource availability, number of signals, amount of debug cores, trigger units and expressions syntax, and trigger expression syntax.

1.2.1.1. Performing a Design Rule Check

To perform a DRC using Reveal Inserter, click the  icon as shown in Figure 1.1.

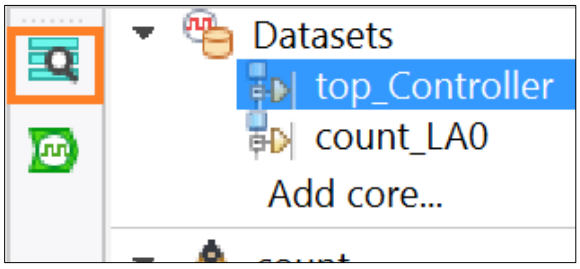


Figure 1.1. Reveal Inserter DRC Icon

If the DRC is successful, the *Design Rule Check PASSED* message is displayed in the Lattice Radiant's console as shown in Figure 1.2. If the design rule check fails, an ERROR message is displayed instead, followed by more error messages indicating what specifically failed the DRC.

```
Checking design rules ...
INFO - The number of EBRs needed is 1.
INFO - The number of DistRAM (logic/ROM/RAM) slices needed is 3.
Design Rule Check PASSED.
```

Figure 1.2. Reveal Successful DRC Console Output

It is important to note that Reveal Inserter's DRC function also reports a high-level estimate of the additional memory resources (EBRs and distributed RAM blocks) required with Reveal inserted to a design.

1.2.2. JTAG Interface Usage

Reveal uses JTAG to connect and debug Lattice FPGA devices, typically utilizing a micro-USB, mini-USB, HW-USBN-2A, or HW-USBN-2B cable connection. By default, Reveal's debug cores are configured to use a hard JTAG connection for debugging. A soft JTAG, however, can also be selected.

For every debug core in a project, either a soft or hard JTAG connection can be selected when setting up debug cores with Reveal Inserter. Reveal debug cores are not limited to using only hard JTAG or only soft JTAG, meaning that a different JTAG connection type can be selected for each debug core. A JTAG hub primitive can also be instantiated for use in the user design. Refer to the Using the JTAG Hub Primitive section for more information.

One last thing to consider when selecting a JTAG connection type for Reveal's debug cores is that the overall debug core limit is different for each JTAG connection. A single JTAG connection (soft or hard) can support up to 23 analyzer cores and one controller cores for Lattice Avant™ devices. For all other devices, up to 15 analyzer cores and a single controller core can be added.

1.2.2.1. Selecting a JTAG Type

To select a different JTAG type, right-click the debug core to modify. From the drop-down menu that appears, select JTAG Interface, followed by Hard JTAG or Soft JTAG depending on the preferred JTAG connection type for that core.

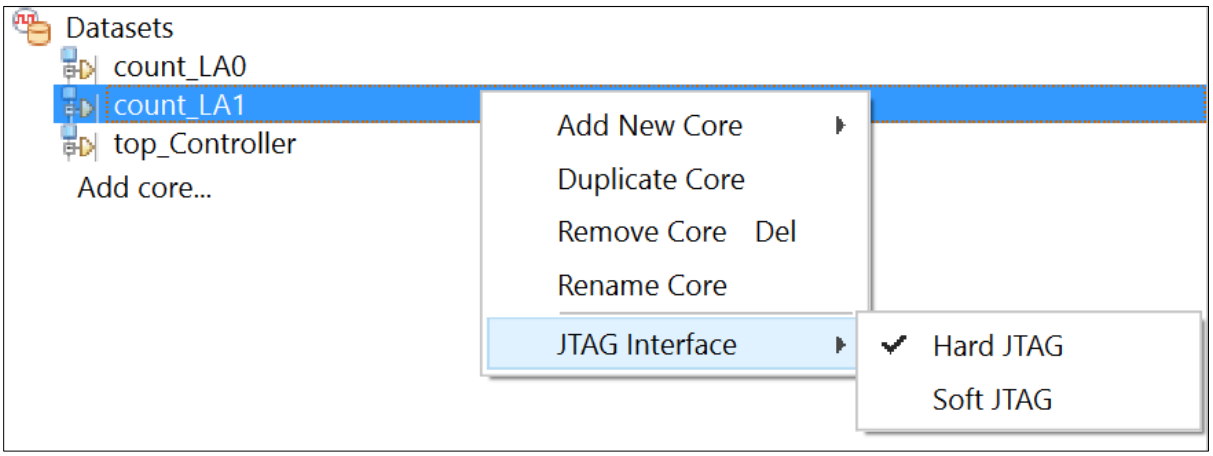


Figure 1.3. JTAG Interface Type Selection in Reveal Inserter

1.2.3. Managing Debug Cores

By default, new Reveal Inserter projects do not contain any cores. To create a new debug core for a Reveal project, click the *Add core…* text below the *Datasets* section of Reveal Inserter. As shown in Figure 1.4, this opens the drop-down list of options allowing the user to select the type of core to create.

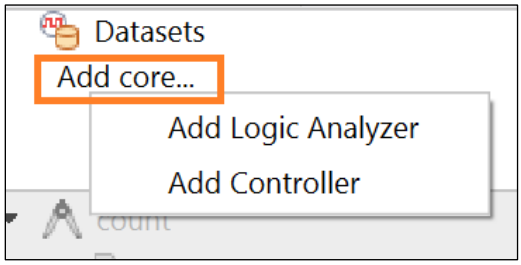


Figure 1.4. Adding Debug Cores in Reveal Inserter

As shown in Figure 1.5, the Reveal debug core appears under the *Datasets* section once added. When adding debug cores to a Reveal project, it is important to consider that up to fifteen analyzer cores and one controller core can be added per JTAG connection type.

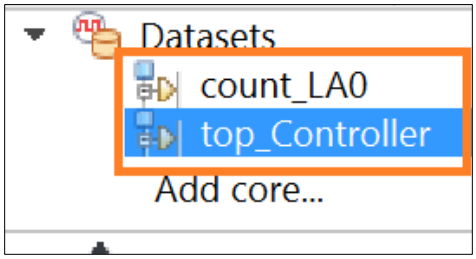


Figure 1.5. Reveal Debug Cores Added to a Project

1.2.4. Detecting a Debug Device

Once the target device for the project is programmed with the updated Reveal bitstream, the next step in the usage flow for Reveal Controller and Reveal Analyzer is to generate the *.rva* project file. This is the Reveal Analyzer/Controller project file that is associated with the active Reveal Inserter debug core. It's important to ensure that these two files are up to date and in sync with each other to prevent issues detecting a debug device.

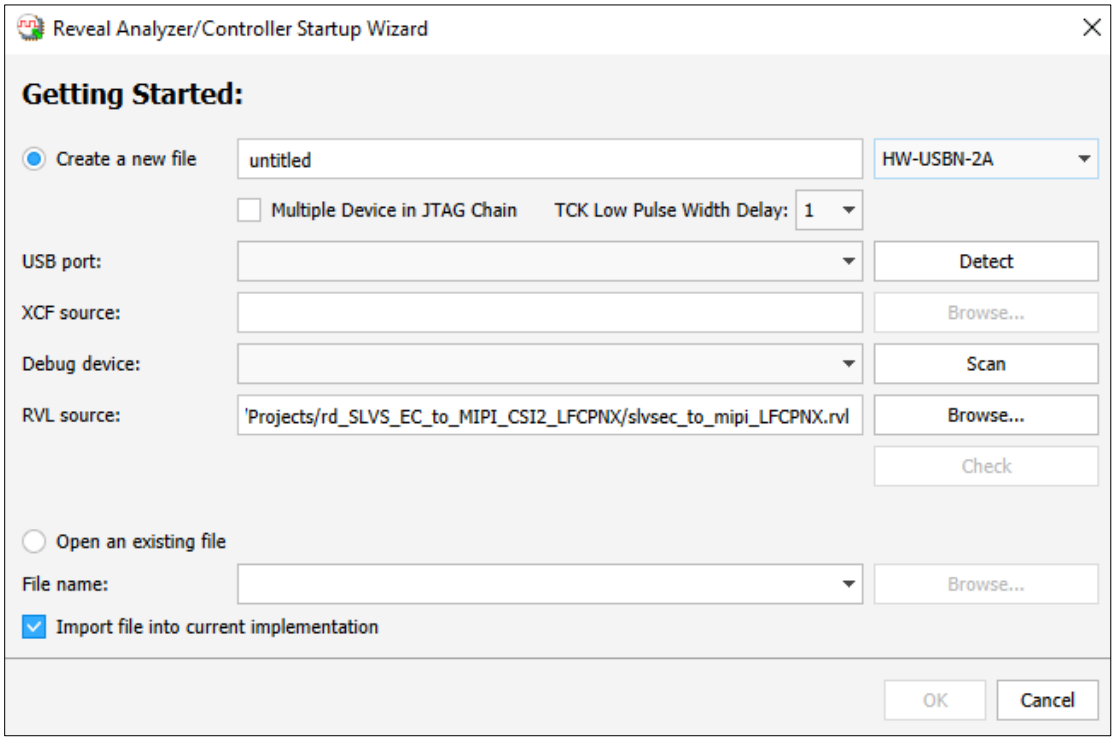


Figure 1.6. Reveal Analyzer/Controller Project Setup Window

To detect a debug device:

1. Load Reveal Analyzer/Controller by selecting **Tools > Reveal Analyzer/Controller**.
2. Enter a name for the Reveal Analyzer/Controller project. There are no naming requirements, except that it cannot begin with an integer.
3. Select the USB Port to which the device with Reveal inserted is connected.
4. Click **Detect** to allow Reveal to scan for devices connected through JTAG. If a device is not detectable, try lowering the TCK Low Pulse Width Delay. This corresponds to a divider value that slows down the JTAG clock.
   - Ensure that the correct cable type is selected in the top right drop down
   - If there are multiple devices in a JTAG chain, only one can be debugged with Reveal Analyzer/Controller at a time. To do this, ensure that the target device is programmed with Reveal debug cores inserted with other devices not containing any Reveal cores.
5. Select a Debug device. Click **Scan** to allow Reveal to search for connected devices with Reveal inserted. Ensure the correct device is selected if there are multiple devices in the JTAG chain.
6. Click **OK**.

If a Reveal Analyzer project already exists but the user is unable to detect the device or encountering other debugging issues, select *Design > Cable Connection Manager* from the toolbar with the Analyzer/Controller project open. This opens the cable connection manager for the Reveal Analyzer project, which can be used to redetect a device to ensure that Reveal Analyzer/Controller can at least detect the debug device.

1.2.5. Other Important Considerations

What is important to note about Reveal is that it adds some additional RTL to a design and increases the overall resource utilization. This is because Reveal generates a new top-level module for the existing design to interface with Reveal's debug logic. With this, it is important to ensure that the critical paths in a design meet timing before using Reveal, and that the project's target device also has enough resources to support Reveal's additional debug logic.

Aside from increased resource utilization, another important thing to note about Reveal are the debug core and signal limits that are supported. As was mentioned in the previous section, up to fifteen analyzer cores and a single controller core can be added per JTAG interface connection. For each analyzer core, up to 512 signals can be traced.

In many cases only a single analyzer core is required to debug a project, however, additional analyzer cores may be required depending on the project. For projects containing multiple clock regions, use a separate analyzer core for each clock region in the design. Another potential reason to add more analyzer cores to a debug project is to trace additional signals in a design if the core signal limit is exceeded for a single analyzer core.

When implementing Reveal to debug a design, consider the GSR. Because Reveal Analyzer disables GSR using a synthesis attribute, it is not recommended that GSR is instantiated or used in user logic.

2. Reveal Analyzer

This section covers the steps for setting up and inserting logic analyzer cores in a Lattice FPGA project using Reveal. Although the screenshots in this section are taken from Lattice Radiant, most of the settings and supported features are the same for Diamond.

2.1. Creating and Inserting Logic Analyzer Cores

2.1.1. Adding Logic Analyzer Cores

To add logic analyzer cores:

1. Open Reveal Inserter. Click **Tools > Reveal Inserter** or the  icon.
2. Select **Add Core > Add Logic Analyzer**.

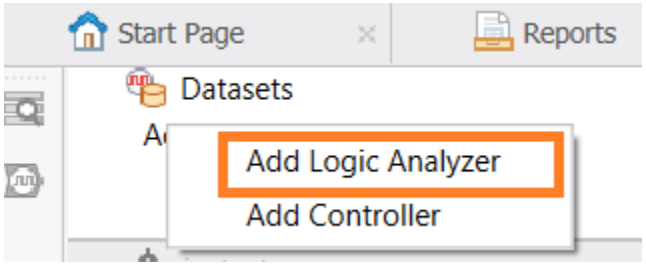


Figure 2.1. Adding Logic Analyzer Core

3. Continue to add cores as desired. Up to 15 logic analyzer cores for each type of JTAG interface may be added.

2.1.2. Setting Up Trace Signals

To set up trace signals:

1. Select the signals to trace. Drag and drop signals from the left side of the Reveal Inserter window to the Trace Signals area as shown in Figure 2.2. Trace signals are annotated using *@Tc*.

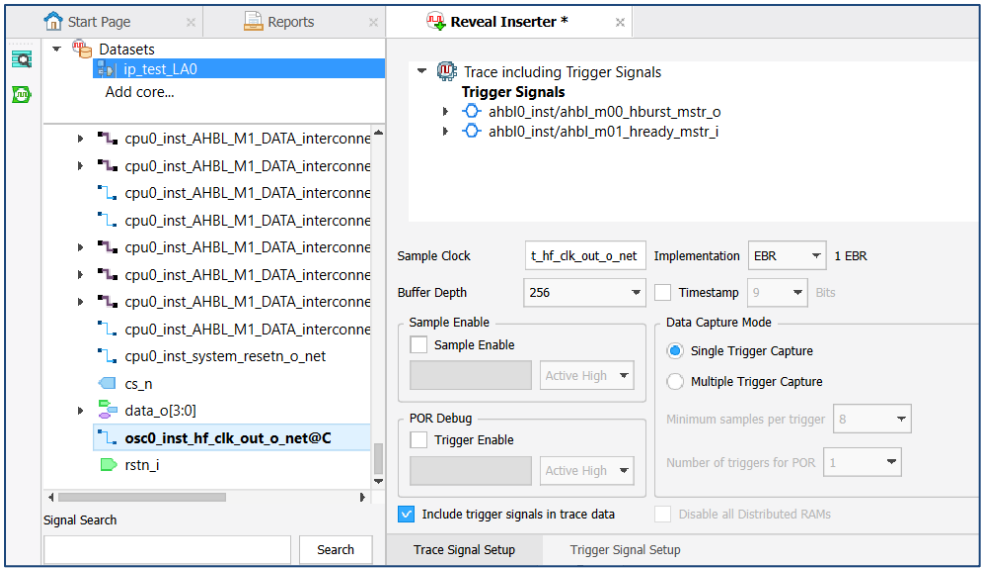


Figure 2.2. Configuring Trace Signals

2. Select the **Sample Clock** for the design. Drag and drop the clock signal from the left side of the Reveal Inserter window to the Sample Clock field. This should be the primary clock used in the design. If a design contains multiple clock regions, it is recommended that each clock region have its own logic analyzer core for each respective clock. Clock signals are denoted using *@C*.
3. Select the **Buffer Depth** corresponding to the amount of data to store each time Reveal Analyzer captures signals. The buffer depth directly corresponds to the number of samples Reveal Analyzer takes. For example, a buffer depth of 256 samples each trace signal 256 times.
4. Select an **Implementation** mode. This controls what memory Reveal Inserter instantiates. Choose between EBR or Distributed RAM. The resource utilization preview for each mode is displayed on the side.
5. Select the **Data Capture Mode** corresponding to the number of times Reveal Analyzer should look for trigger conditions. Single trigger capture mode only samples data the first time a trigger condition is met. Multiple trigger capture mode samples signals multiple times, depending on the *Buffer Depth* and *Minimum Samples per trigger* settings. For example, selecting a buffer depth of 512 and a minimum samples per trigger of 16 captures the signals 32 times (512/16 = 32).

   If POR debugging is enabled, the *Number of Triggers for POR* can be used to capture signals multiple times. This setting is also limited by *Buffer Depth* and *Minimum Samples per trigger*. For example, with a buffer depth of 512 and minimum samples per trigger of 32, 1 to 16 POR triggers may be selected. The POR signal capturing uses the same buffer used to capture other trace signals
6. Configure additional trace signal settings as desired. Timestamp is used to break the captured samples into subsections. If no timestamp is selected, the simulation timescale corresponds to the sample number (for example, 0:142 is the 142nd sample taken by Reveal Analyzer).

   The *POR Debug* is used to monitor a startup signal. Typically, this is used to capture signals soon after a device is reset.
7. Enable **Trigger Enable** to select a signal to monitor as the startup signal.
8. Select the sensitivity of the signal as either **Active High** or **Active Low** depending on your design's logic.
9. Drag and drop the POR debug signal from the left side of the Reveal Inserter window to the box at the bottom of the POR Debug field. This signal is also denoted using *@C*. Enabling *Include trigger signals* in trace data includes the signals in the *Trigger Signal* tab in the list of signals captured by the Reveal Analyzer. The default behavior of Reveal is to capture only trace signals.

2.1.3. Setting up Trigger Conditions

To set up trigger conditions:

1. Switch to the **Trigger Signal Setup** tab.

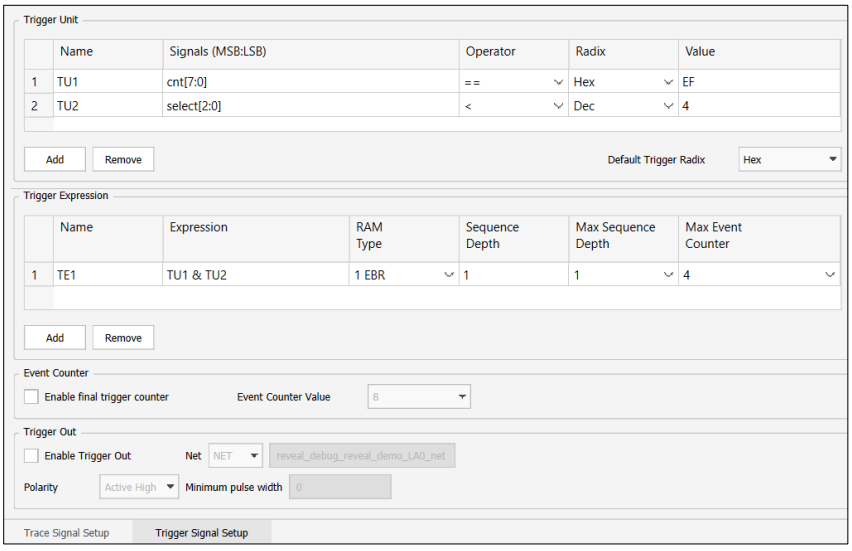


Figure 2.3. Reveal Inserter Setup for Trigger Conditions of Analyzer Cores

2. Add *Trigger Units* as desired. Drag signals from the signal pane to the *Signals* column.
3. Select conditions for trigger unit using the **Operator**, **Radix**, and **Value** fields. These can also be changed later on during debugging with Reveal Analyzer
4. Click the **Add** button to add additional trigger units. Up to 16 different trigger units can be added.
5. Define the **Trigger Expressions** for when to capture signals. Trigger expressions are logical equations featuring one or more trigger units. Up to 16 different trigger expressions can be added per analyzer core.

   Valid operators: & (AND), | (OR), ^(XOR), !(NOT), THEN, NEXT, #(count), ##(consecutive count)
   - TU1 #3 means TU1 must occur three times before this expression is true.
   - TU1 ##3 means TU1 must occur three clock cycles consecutively for this expression to be true.
6. Select the **Max Sequence Depth**. To minimize the amount of resources used when Reveal is inserted, minimize this value. This value should be greater than or equal to value in the **Sequence Depth** column.
7. Select the **Max Event Counter**. This value must be greater than the largest count value if # or ## are used in any trigger expressions.
8. Select the additional optional settings. The *Event counter* is used to change the number of times an event (including all trigger expressions) must occur before capturing signals. *Trigger out* is used to drive a signal once an event occurs.
9. Select the type of signal to drive using the **Net** field.
10. Select the **Polarity** and **Minimum pulse width**, which corresponds to the number of clock cycles of the sample clock.

2.2. Debugging with Reveal Analyzer

With the project's target device programmed with the updated bitstream containing Reveal analyzer debug cores, the next step is to begin capturing signals. Depending on how the analyzer core is setup in Reveal Inserter, how signals are captured at this point may vary slightly.

If Reveal Analyzer is configured for POR debugging, then Analyzer starts capturing signals as soon as the POR trigger signal is active. Typically, this is a signal that becomes active shortly after a design is reset.

If trigger units and expressions are used to set up an Analyzer core, then there are two main ways to begin capturing signals. The first way is to click the … icon in the top right of the Analyzer user interface to begin debugging. Doing this causes the Reveal Analyzer to monitor the trigger signals in a Reveal project until a trigger expression is enabled.

Aside from that Reveal analyzer can also be forced to capture signals. To do this, click the … icon towards the top of the Reveal Analyzer user interface. This Analyzer functionality is especially useful in cases where trigger expressions are not being enabled, as it allows users to force signal capture.

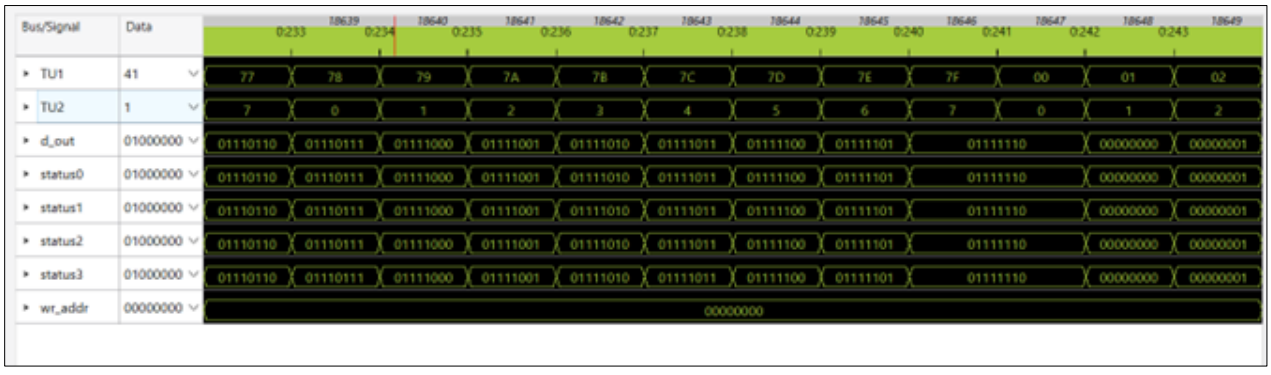


Figure 2.4. Reveal Analyzer Waveform Display

Once signals are captured, the Reveal Analyzer switches to the *Waveform Display* tab to show the captured signals. Depending on how the Analyzer debug core is configured, different number of samples are captured. However, one takeaway is that each unit in the display corresponds to a sample. For example, Figure 2.4 shows the sample at 0:234 was the 234th sample captured by the Reveal Analyzer. This is useful to know when setting a trigger position for Reveal Analyzer. The user can specify the specific sample for the waveform display cursor to appear under.

In addition, the drop-down next to each signal name in the *Data* column can be used to change the radices displayed for a signal in the waveform.

2.3. Modifying Trigger Conditions

Reveal Analyzer supports some level of customization for trigger expressions and units while debugging. As shown in Figure 2.5, some settings that can be changed for trigger units include the comparison operator and radix for the comparison value. Certain trigger expressions can also be enabled or disabled using the checkbox next to each expression in the *Trigger Expression* field.

The trigger conditions for Reveal Analyzer can also be customized using the settings in the trigger options field in the bottom left of the tab. The first setting, *Enable TE*, is used to either OR ALL or AND ALL the trigger expressions to capture signals. The default behavior for Reveal Analyzer is to capture signals with OR ALL, so any trigger expression being enabled triggers this. Next, the samples per trigger and number of trigger settings can also be modified depending on the buffer depth. Both these setting are codependent, which means that lowering one allows a higher value for the other.

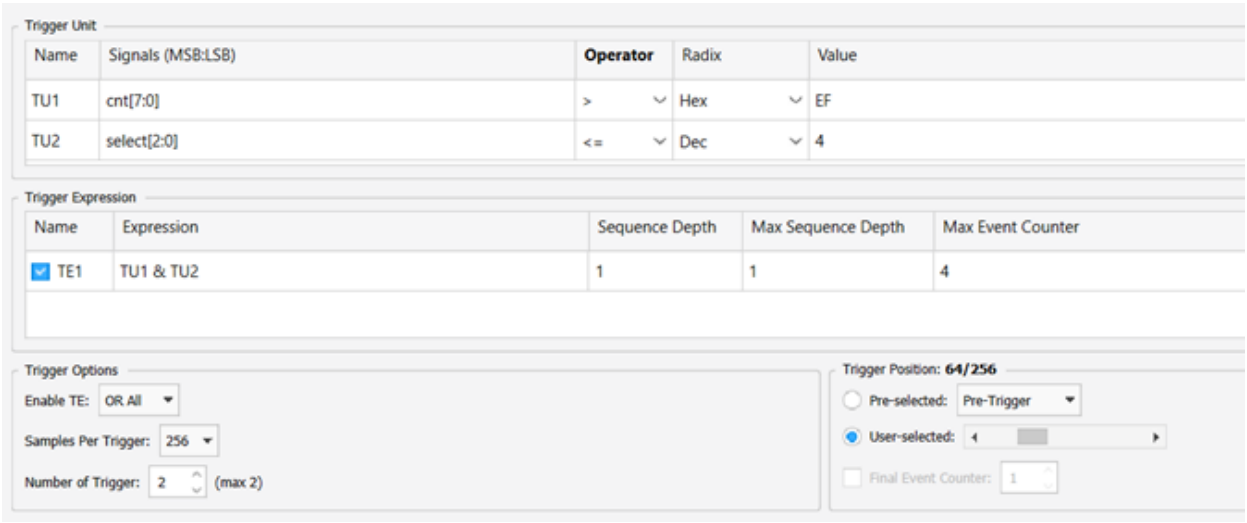


Figure 2.5. Reveal Analyzer Trigger Expression and Unit Settings

Lastly, the location of the waveform display cursor can be selected using the settings in the *Trigger Position* section. For pre-selected, either pre-trigger (cursor centered at 1/16th point of samples), center-trigger (cursor centered at ½ point of sample), or post-trigger (cursor centered at 15/16th point of samples. The *User-selected* option can also be used to select a specific sample number for the cursor to be centered on when the waveform display window opens.

3. Reveal Controller

This section covers the usage guidelines for setting up and debugging Lattice Radiant projects using the Reveal Controller. This section assumes that a Reveal inserter project has already been created, and that a controller debug core is added. For more information about these two processes, refer to Managing Debug Cores and Detecting a Debug Device sections.

3.1. Virtual Switches and LEDs

The Reveal Controller's virtual switches and LEDs are other useful features for debugging Lattice FPGA projects. Virtual switches can be used to toggle user logic, while virtual LEDs can be used to monitor signals in a design. Both virtual switches and LEDs support up to 32-bits of signals. The main design consideration to make when using this controller function, is that the signals used must be wire types.

One last thing to consider when using virtual switches to drive user logic is that they should be treated as asynchronous switches since they are synchronous with the JTAG clock, which typically runs between 1 to 5 MHz. Because of this, it is recommended that a handshake is implemented in the user logic to account for this lower frequency clock, especially if the rest of the design is running at a significantly higher frequency.

3.1.1. Adding Virtual Switches and LEDs

The process for adding virtual switches and LEDs is essentially the same. As mentioned in the Virtual Switches and LEDs section, the main thing to consider when implementing these is to only use wire type signals. Additionally, ensure that the signals being used as virtual switches and LEDs are not being driven elsewhere in the design to prevent errors in the implementation flow due to multiple drivers.

To add virtual switches and LEDs:

1. Enable **Virtual Switch Setting** or **Virtual LED Setting** depending on what is to be used.
2. Select the **Width** corresponding to the number of signals to be assigned to the virtual switches and LEDs.
3. Drag and drop the signals from the **Signal Pane**. Once added, the signals should appear in their assigned cell as shown in Figure 3.1. A group of signals can also be added by pressing **SHIFT** or **CTRL**, and then clicking multiple signals and dragging them to the **Switch List** or **LED List**.

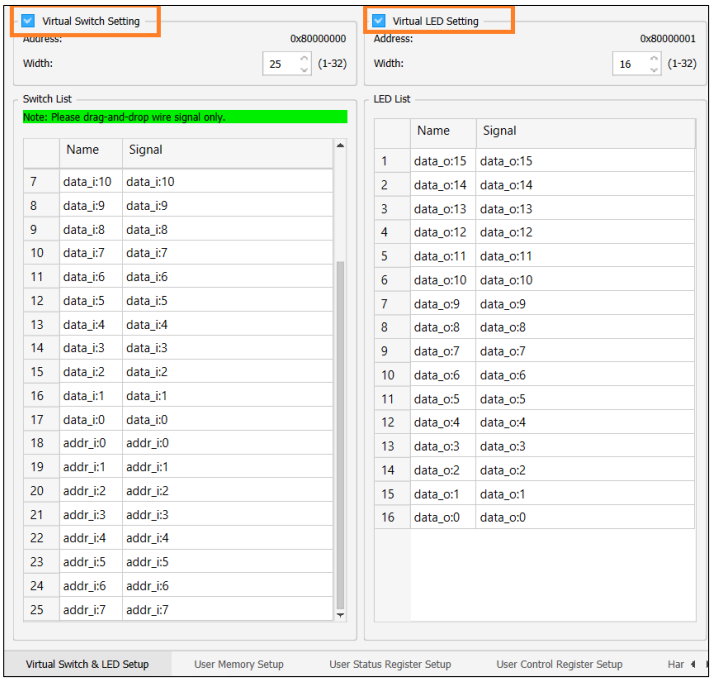


Figure 3.1. Virtual Switches and LED Setup in Reveal Inserter

4. Rename signals as desired to improve readability when debugging with Reveal Controller. To rename a signal, click the cell next to it in the **Name** column. The signal names provided here are the same signal names to be displayed when using Reveal Controller.
5. Perform a Design Rule Check by clicking the icon. Insert the controller debug core using the icon, underneath DRC. Ensure **top_controller** is active and that the **Activate Reveal file in project** is enabled.
6. Generate a new bitstream and program device.
7. Once a bitstream is generated with the Reveal Controller debug core inserted and the target device is programmed, the next step is to create a new Reveal Controller project. For more information on how to do this, refer to Detecting a Debug Device section.

3.1.2. Debugging with Virtual Switches

As mentioned previously, Reveal Controller's virtual switches are synchronous with respect to the FPGA's JTAG clock, and should be treated as asynchronous switches in user design logic. With that in mind, the two main modes for virtual switches are direct and indirect mode, which can be toggled using the Direct Mode checkbox as shown in Figure 3.2.

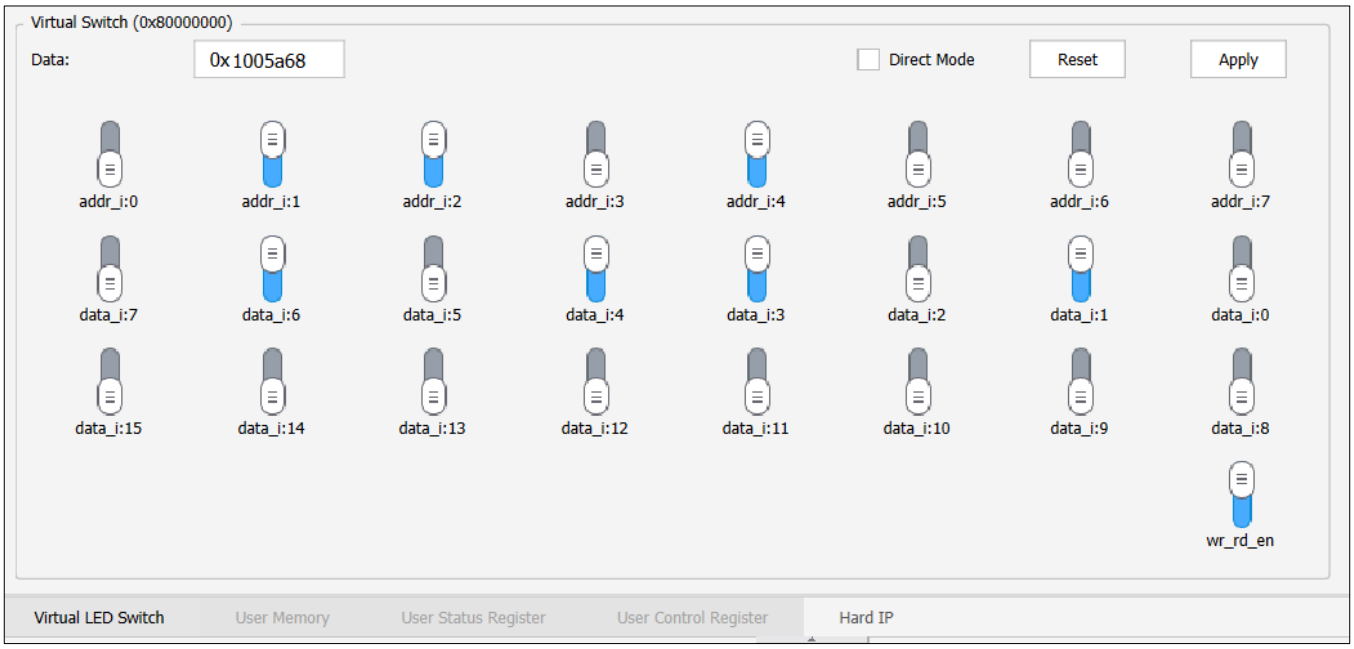


Figure 3.2. Virtual Switches Controller User Interface

In direct mode, the Reveal Controller writes data to the virtual switches as soon as a switch is toggled in the controller user interface. For indirect mode, switches are toggled once the Apply button in the top right of the user interface is clicked. To toggle a virtual switch, click the switch to toggle, which updates the Data field in the top left of the user interface to reflect the data being written out. Similarly, the Data field also accepts user entry, and can be used to directly input the data to write out in hex format.

To reset all user inputs in the controller user interface back to the default state (all 0's), click the *Reset* button located towards the top right.

3.1.3. Debugging with Virtual LEDs

Similar to virtual switches, the Reveal Controller's virtual LEDs also have two main modes: continuous and single polling.

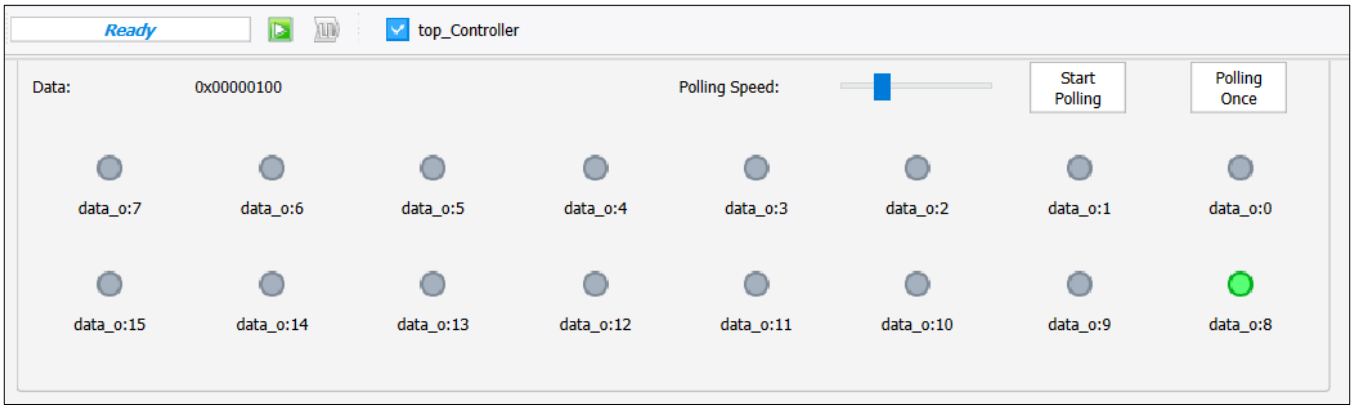


Figure 3.3. Virtual LEDs Controller User Interface

When in continuous polling mode, the virtual LEDs continually update depending on the selected polling speed. To use virtual switches in continuous polling mode, select a Polling Speed by dragging the slider bar (left is slower, right is faster). Once a polling speed is selected, click the *Start Polling* button to begin polling continuously. It is important that about continuous polling also occupies the JTAG to read signals.

In single polling mode, the virtual LEDs update only when the *Polling Once* button on the top right of the controller user interface is clicked. Since the button operates with respect to the JTAG clock, it may need to be pressed multiple times to capture the desired data.

3.2. User Memory Access

The Reveal Controller's user memory access function is another useful feature for debugging Lattice FPGA projects. The primary use for user memory access is to provide an interface to a memory component, which can be used to read or write data. Before user memory access can be used, a project must already contain a memory block (EBR, distributed memory, or PMI).

Something to consider when using user memory access is that if a single port memory component is used, the signals assigned in Reveal Controller cannot be used to drive user logic elsewhere in the design. Because of this, it is recommended that dual port memory components are used when debugging with user memory access. This way, one interface can be used to read or write to the memory with the Reveal Controller, while the other can be used for memory access by user logic.

To configure the memory block for user memory access:

1. Select the **User Memory Setup** tab.
2. Enable **User Memory Setting**.
3. Assign the mandatory signals from the signal pane to the setting list.
   - **Clock** – input clock
   - **Clock_enable** – clock enable signal
   - **Wr_Rdn** – active high write enable signal (low is read enable)
   - **Address** – memory address (16-bit max)
   - **WData** – data to write to memory (32-bit max)
   - **RData** – data to read from memory (32-bit max)
4. To avoid incorrect signal selection, directly assign the signals from the user memory component in the design as shown in Figure 3.4.

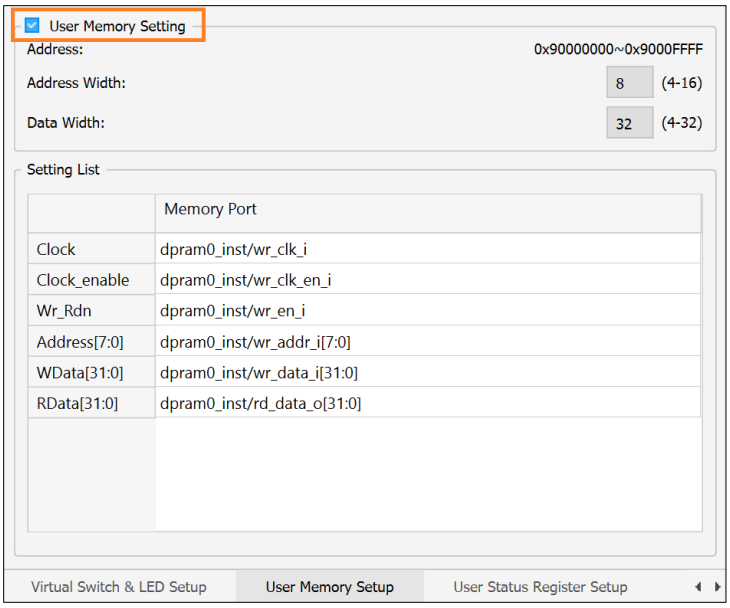


Figure 3.4. Reveal Inserter User Interface Setup for User Memory Access

5. Insert the controller debug core using the icon, underneath DRC.
6. Ensure **top_controller** is active and that **Activate Reveal file in project** is enabled.
7. Generate a new bitstream and program device. Once a bitstream is generated with the Reveal Controller debug core inserted and the target device is programmed, the next step is to create a new Reveal Controller project. For detailed information on how to do this, refer to Detecting a Debug Device section.

3.2.1. Debugging with User Memory Access

Once a new bitstream with the Reveal debug cores is generated and programmed to the target device, the Reveal Controller can be used to debug a project. The user interface of the Reveal Controller's user memory access function is shown in Figure 3.5.

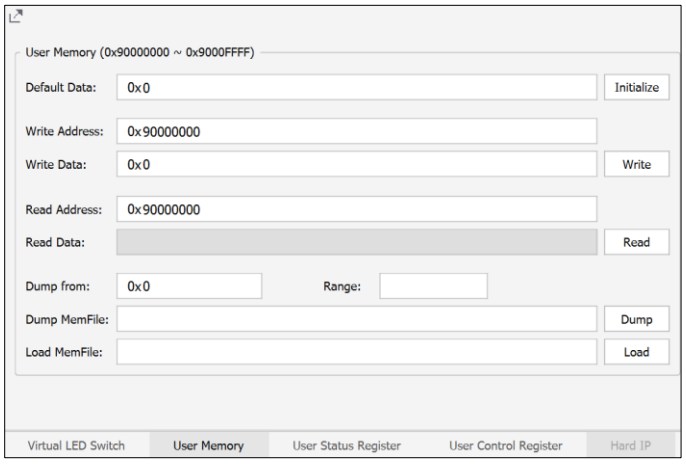


Figure 3.5. Reveal Controller User Memory Access User Interface

The Default Data field is used to initialize an entire memory to the same value. Once the initial value is selected, click Initialize to initialize the memory component.

To read from a specific address in a memory component, select the target address using the Read Address field. Once the address is selected, click the Read button to read from that address space. The Read Data field updates to reflect the value that was read.

Aside from initializing an entire memory to the same value, there are two other ways to write to the memory spaces in a Hard IP. The first way is to select the address to write to using the Write Address field. The base address that appears here is offset, so it is important to check the memory range for each component at the top of the user interface. This ensures writing to the correct memory space. Once a target address is selected, input the data in hex format using the Write Data field and click *Write* to write that data to the memory component.

The second way to write to a memory component using user memory access is initialize it using a memory initialization file. To do this, select the Load button to select an initialization file. The only requirements for this file are that it uses a .mem extension and lists each register assignment as a separate line using the <memory address>:<memory data> format. Once a file is selected, click Load again to load the memory initialization file and finish initializing the Hard IP's registers.

Lastly, user memory access features dump memory, which is used to generate a memory initialization file. Select the address range to dump values from using the *Dump from* and *Range* fields. If no range is selected, then all data from a memory component are exported. Next, click the Dump button. This opens the file explorer window, which requires the user to select a name for the memory initialization file to generate. Select a name and click Save to generate a memory initialization file.

3.3. User Status and Control Registers

Similar to Virtual Switches and LEDs, the Reveal Controller's user status (read-only) and control registers (read-write) can be used to read and write to some signals in a design. These two controller functions work similarly and require all signals interfaced with control and status registers to be of the wire type.

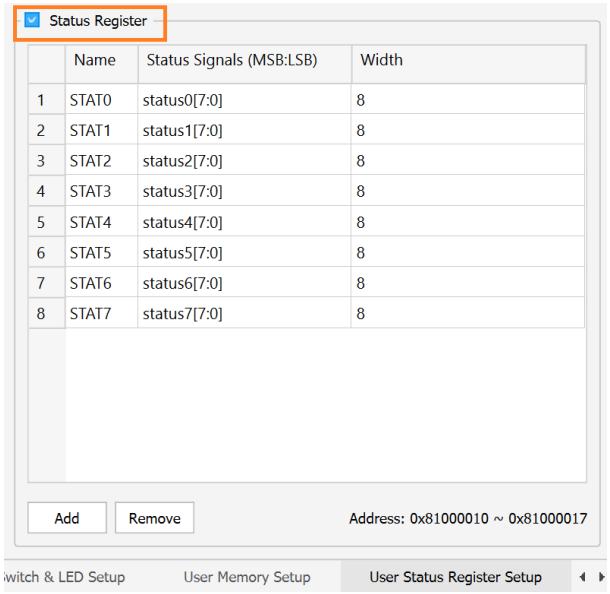


Figure 3.6. Reveal Inserter User Status Register Setup

The main function of the user status and control registers is that they support up to eight 8-bit signals as shown in Figure 3.6 and Figure 3.7, which is enhanced to support 32 8-bit signals in a future release of Lattice Radiant.

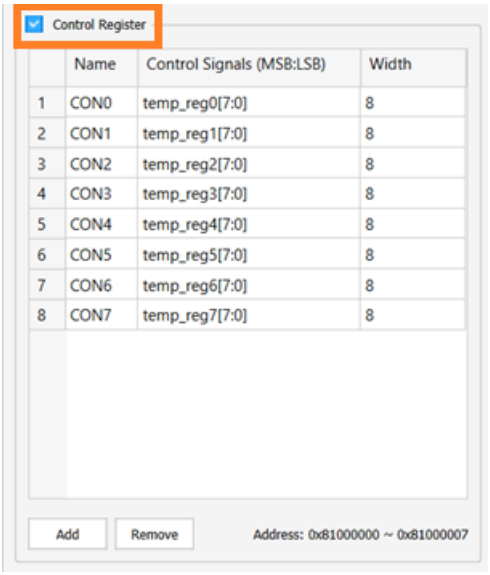


Figure 3.7. Reveal Inserter User Control Register Setup

To use user status and control registers, enable either the *Status Register* and/or *Control Register* settings in each of the respective tabs in Reveal inserter. With either controller function enabled, drag and drop up to eight 8-bit signals from the signal list on the left side of the Reveal Inserter user interface into the *Status Signals* and *Control Signal* fields. To rename a status or control register, double-click the *Name* column next to the register the user wants to configure.

3.3.1. Debugging with User Status Registers

Once a device is programmed with the updated Reveal bitstream containing the controller debug cores, the user status register user interface is displayed as shown in Figure 3.8.

The main usage of the Reveal Controller function is to read from the status registers. To do this, click the *Rd* button next to the status register to read from.

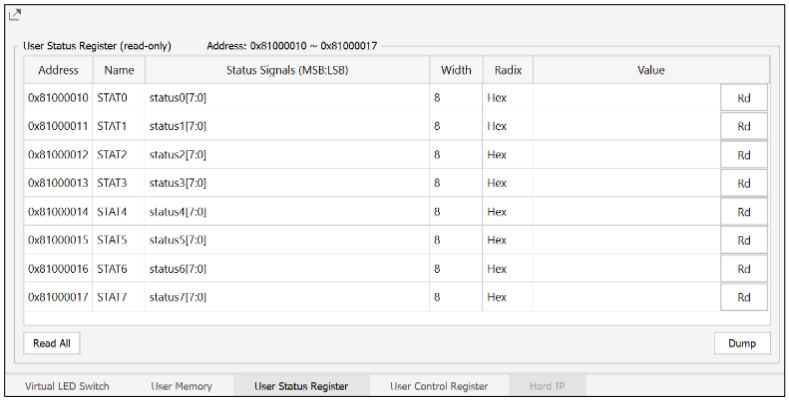


Figure 3.8. Reveal Controller User Status Register User Interface

Similarly, the *Read All* button in the bottom left of the user interface can be used to concurrently read from all the status registers in the Reveal Controller. Once a register is read from, the *Value* field updates to reflect the contents of that register.

When at least one register is read from, the *Dump* button can be used to generate a memory initialization file, which can be used to initialize a memory component or control registers. To do this, click the *Dump* button. This opens the file manager on Linux window requiring the user to select a name for the memory initialization file. Once the name is selected, click *Save* to generate a memory initialization file.

3.3.2. Debugging with User Control Registers

Once the device is programmed with the updated Reveal bitstream containing the controller debug cores, the user control register user interface is displayed as shown in Figure 3.9.

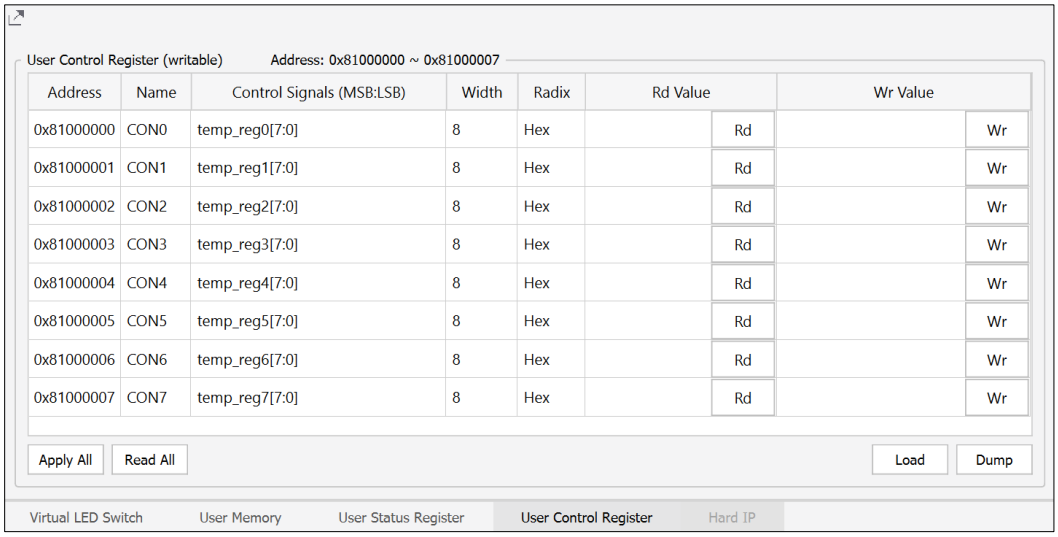


Figure 3.9. Reveal Controller User Controller Register User Interface

As mentioned previously, one of the main differences of the user control registers is that it can be used to read and write to signals in a design. To read from a signal, click the *Rd* button next to the signal's name, or click the *Read All* button to read from all the signals in this tab. Once a signal has been read from, its value is updated in the *Rd Value* field.

Similarly, the *Wr Value* field is used to specify an 8-bit hex value to write to the selected control register. Once an 8-bit value is entered, click the *Wr* icon in order to apply that value to the selected signal. Once the value is applied, read from the same control register to ensure that the value is correctly applied. Aside from that, the *Apply All* button can be used to apply all values in the *Wr* value field to every control register.

Lastly, the Dump button can be used to generate a memory initialization file, which can be used to initialize a memory component or control registers. To do this, click the Dump button. This opens the file explorer window which requires the user to select a name for the memory initialization file to generate. Select a name and click *Save* to generate a memory initialization file. This file can be used to initialize control registers later on using the Load button.

3.4. Configuring Hard IP

The Reveal Controller's Hard IP function is used to interface with the Hard IP on a device by providing an interface to its LMMI ports to dynamically change its functionality. Note that in order to prevent multiple drivers, any existing user logic interfaced with a Hard IP's LMMI ports is disconnected when Reveal is inserted to a design with this enabled. An exception is when a PCIELL Hard IP is used in the design. In this scenario, Reveal inserts a MUX which handles the arbitration between LMMI signals in the user design and the generated LMMI bus from the Reveal controller as shown in Figure 3.10. This arbitration method allows user logic to function as normal. Reveal only accesses the LMMI interface when it needs to perform read or write operations by asserting then de-asserting the usr_lmmi_ready_o signal.

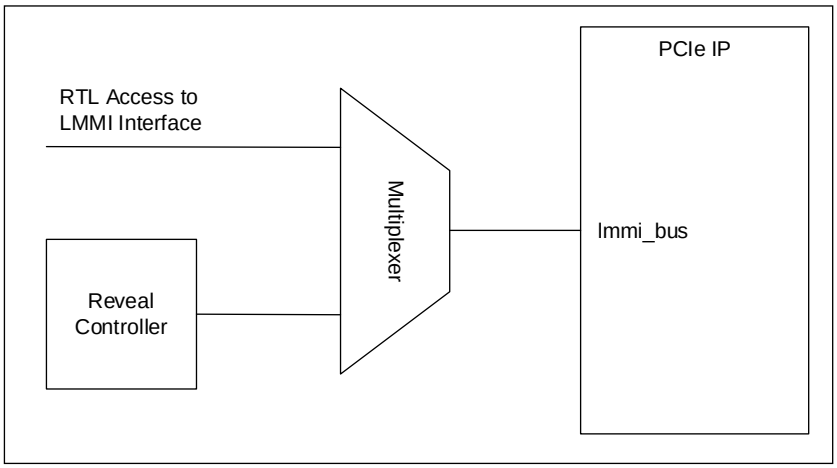


Figure 3.10. Reveal Controller LMMI Arbitration Scheme for PCIELL

Another important thing to know about the Hard IP function is that the IPs that are controllable depend on the specific target device. For Crosslink™-NX, Certus™-NX, and CertusPro™-NX devices, the Hard IPs are: PLL, DPHY, I2CFIFO, SGMIICDR, PCIELL, PCSX1, PCSX2, and PCSX4. For Lattice Avant™, the configurable Hard IP is PLL.

Before using the Reveal Controller's Hard IP function, the Hard IP to configure must already be in the active design. As mentioned before, ensure that no important user design logic is hooked up with the target device's LMMI interface.

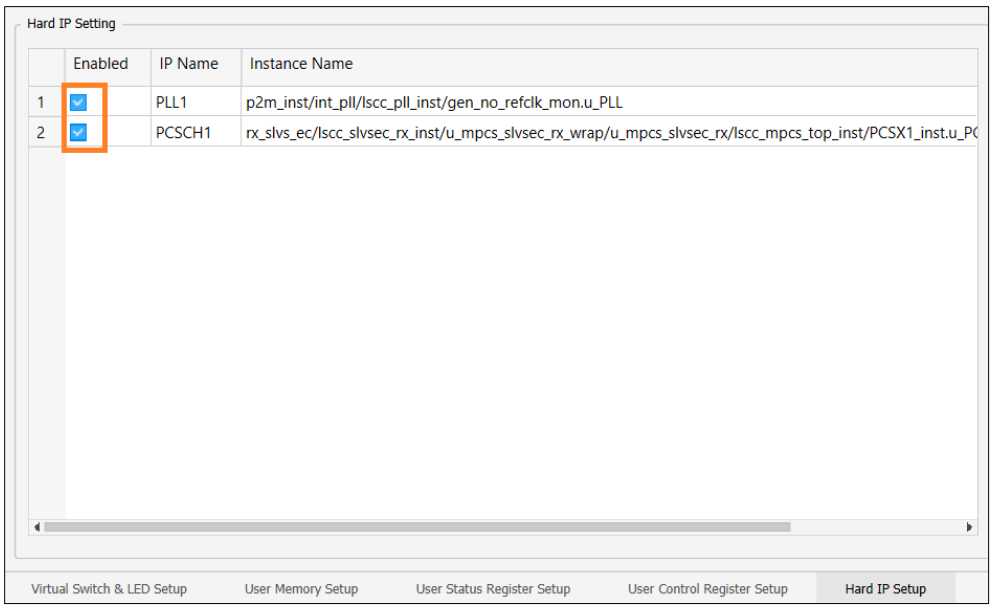


Figure 3.11. Hard IP Setup Tab in Reveal Inserter

Once the project is ready for debugging with Reveal Controller, add a new Controller debug core. For more information on how to add a Controller debug core, refer to Managing Debug Cores section. Once the controller core is added, select the Hard IP Setup tab shown in Figure 3.11.

In this section of the Reveal Controller user interface, the list of configurable Hard IPs that are detected in the current project appears. For each Hard IP block in a design, a separate entry appears in this section. For example, in a two-channel PCIe design, two separate PCS channels are shown.

To enable an IP for configuration using Reveal Controller, select the box next to the Hard IP's name.

The address range for each enabled Hard IP is displayed under the Address column, as shown in Figure 3.12. It is important to note this address range as the Hard IP function also provides an interface for writing to any of the address spaces in this range when debugging later on.

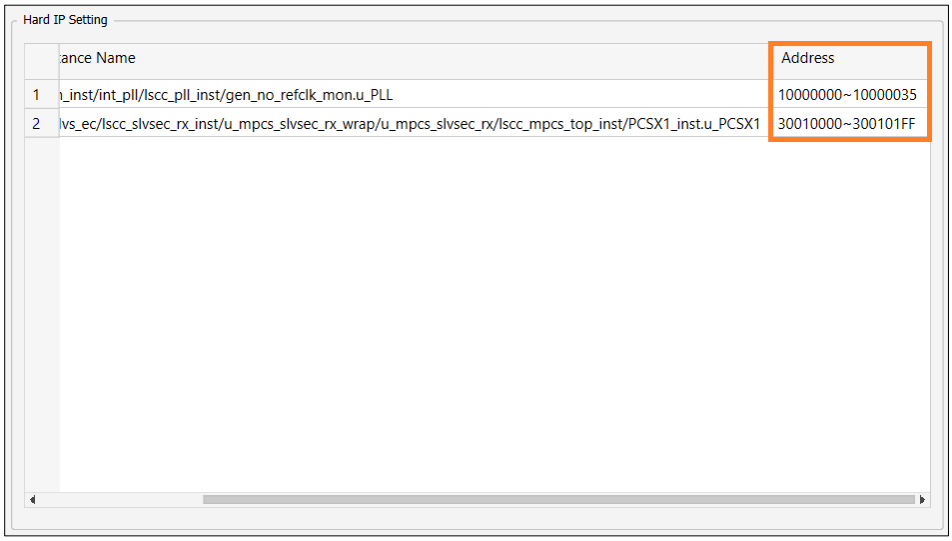


Figure 3.12. Address Ranges for the Enabled Hard IP in Reveal Inserter

Once the Hard IP to configure is selected, perform the DRC using the *Design Rule Check* icon on the left side of the user interface. If the DRC is successful, insert the controller debug core and generate a new bitstream.

3.4.1. Dynamically Updating Hard IP

Depending on the IP that is enabled for the Hard IP configuration with the Reveal Controller, the exact appearance of the Hard IP tab varies. Figure 3.13 shows the user interface of the MPCS IP for CertusPro-NX and Figure 3.14 shows the tab for the PLL IP.

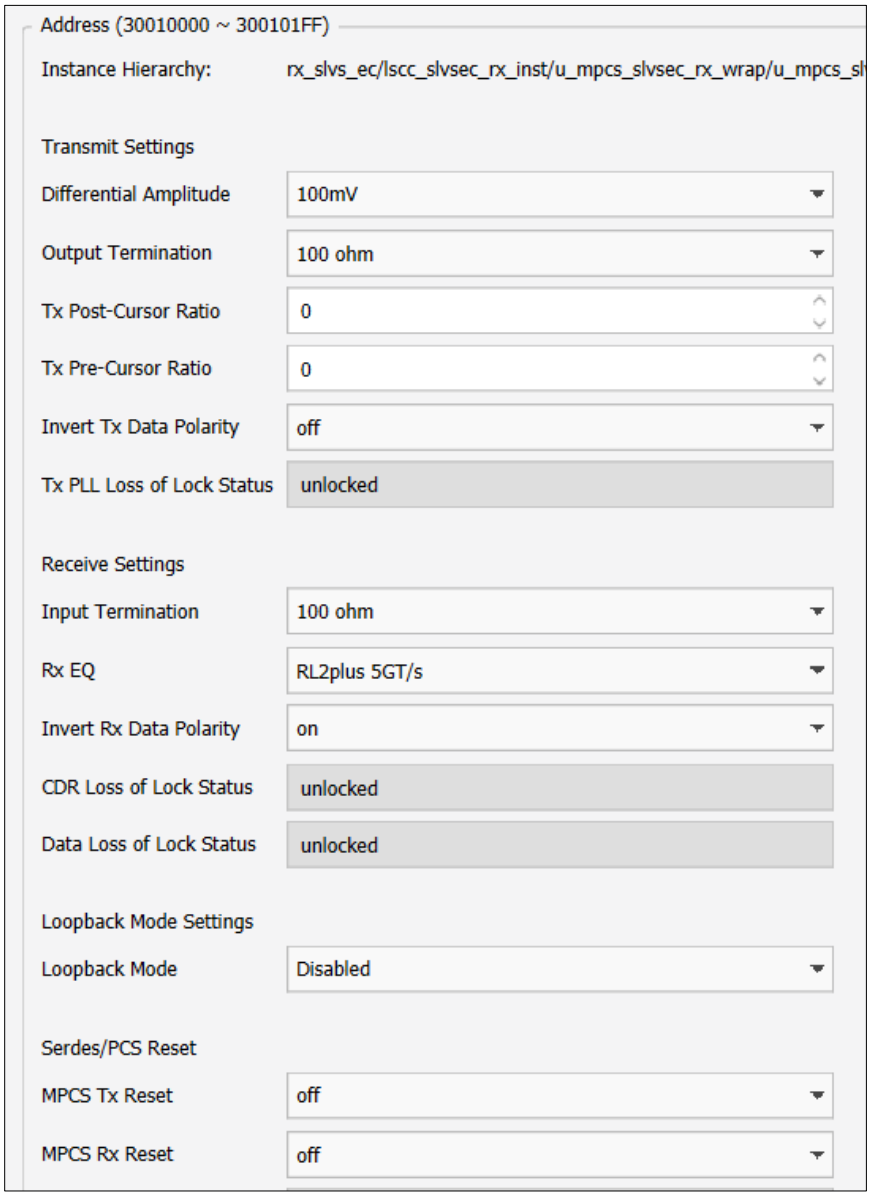


Figure 3.13. Reveal Controller Hard IP Settings for MPCS

Every Hard IP configurable by Reveal Controller has a user memory access section, which allows users to directly write and read from the registers in a Hard IP and functions similar to the user memory access function discussed in the User Memory Access section. For specific information about the properties or addresses of the registers for a Hard IP, refer to the component's IP user guide.

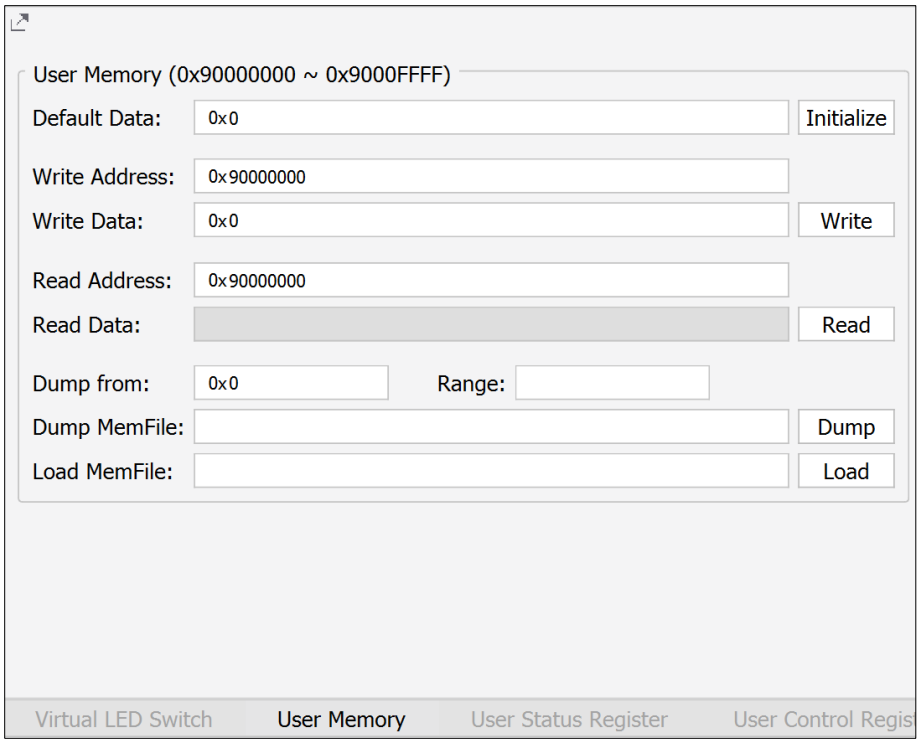


Figure 3.14. Reveal Controller Hard IP Memory Access User Interface

The Default Data setting is used to initialize all registers to the same initial value. Once an initial value is specified by inputting a hex value, click the *Initialize* button to initialize all registers to that value.

To read from a register, select the target register's address using the Read Address field. Once a register address is selected, click the *Read* button to read from that register. The Read Data field updates to reflect the value of the register that was read from.

There are two ways to write to the registers in a Hard IP. The first way is to select the register address the user wants to write to using the Write Address field. The base address for the registers in a Hard IP are offset in the Reveal Controller, so it is important to check the memory range for each component and cross reference it with the IP user guide to ensure the user writes to the correct register. Once the target address is selected, input the data in hex format that the user wants to write using the Write Data field, and click *Write* to write to that register.

Another way to write to the registers in a Hard IP is to use a memory initialization file. To do this, select the Load button to select the memory initialization file. The only requirements for this file are that it uses a .mem extension and lists each register assignment as a separate line using the <register address>:<register data> format. Once a file is selected, click Load again to load the memory initialization file and finish initializing the Hard IP's registers.

Aside from that, certain Hard IP contains user interface settings that can be used to easily change the settings written to various registers. As shown in Figure 3.13, the Hard IP user interface for MPCS contains a few additional settings before the user memory access section, allowing users to modify things like differential output, RX equalizer, and loopback mode.

A useful feature of the Reveal Controller's Hard IP configuration function is that a memory initialization file can easily be generated after some registers were modified. To do this, select the address range first to dump values from using the Dump from and Range fields. If no range is selected, then the values from all the registers in a Hard IP are included in the generated memory initialization file. Click the *Dump* button next, which opens the file explorer window requiring the user to select the name for the memory initialization file to generate. Select a name and click *Save* to generate a memory initialization file. This file can be used later on to reinitialize the registers in a Hard IP using the Load function that was previously discussed.

3.4.2. Eye-Opening Monitor

For designs containing one, two, or four PCS channels in PCIe mode Gen1, Gen2, or Gen3, the performance for each channel can be analyzed using Reveal Controller's Eye-Opening Monitor function. The purpose of this function is to measure the relative performance of a SerDes through comparison, analyzing data regarding the height and width of the eye diagram, and where the passing eye begins and ends. The main thing to know about the functionality of Eye-Opening monitor is that data is collected on the RX side of the PCS block.

To use the Reveal Eye-Open monitor, user must enable the PCS channel to analyze in the Reveal Inserter for any IP containing PCS channels. As discussed in the Configuring Hard IP section. Once the device is programmed with the updated Reveal bitstream, the next step is to load the Reveal Controller and start capturing data for the eye diagram. To do this, click the *Eye-Opening Monitor* button as shown in Figure 3.15.

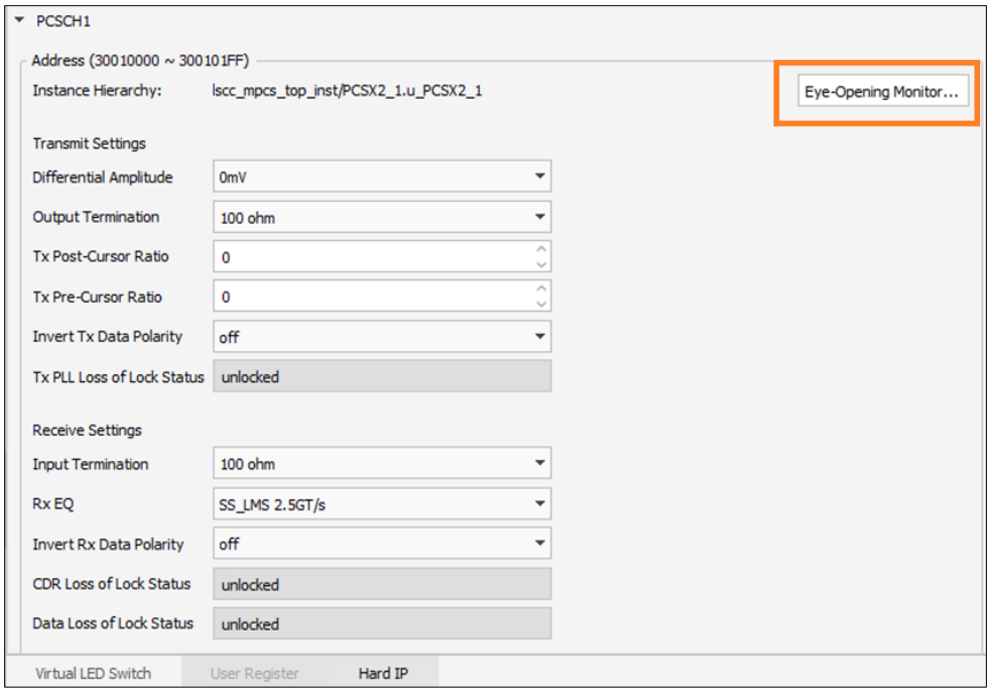


Figure 3.15. PCS Channel Hard IP Eye-Opening Monitor Setting Location

Doing this opens the *Eye-Opening Monitor Configuration* dialog box as shown in Figure 3.16, where the quality and detail of the desired eye diagram can be selected. To generate the eye diagram, it is recommended that either low or normal quality are initially selected, as a high-quality eye diagram can significantly increase the required runtime. Once the correct eye diagram quality is selected, click *Run* to start capturing data for the eye diagram.

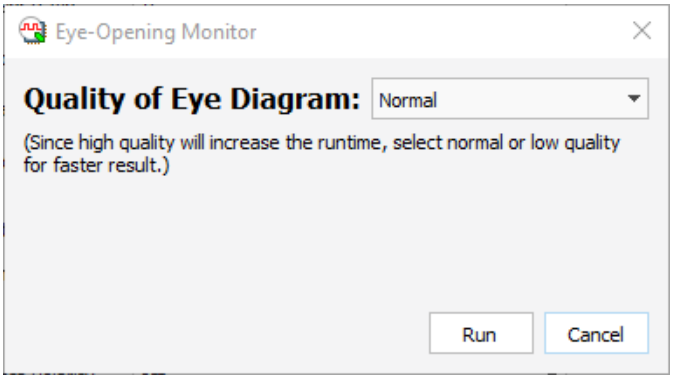


Figure 3.16. Eye Diagram Quality Selection Window

Once Eye-Opening Monitor finishes generating the eye diagram, the Eye Diagram appears as shown in Figure 3.17. The window displays the actual eye diagram and can be used to visually determine the quality of the captured eye.

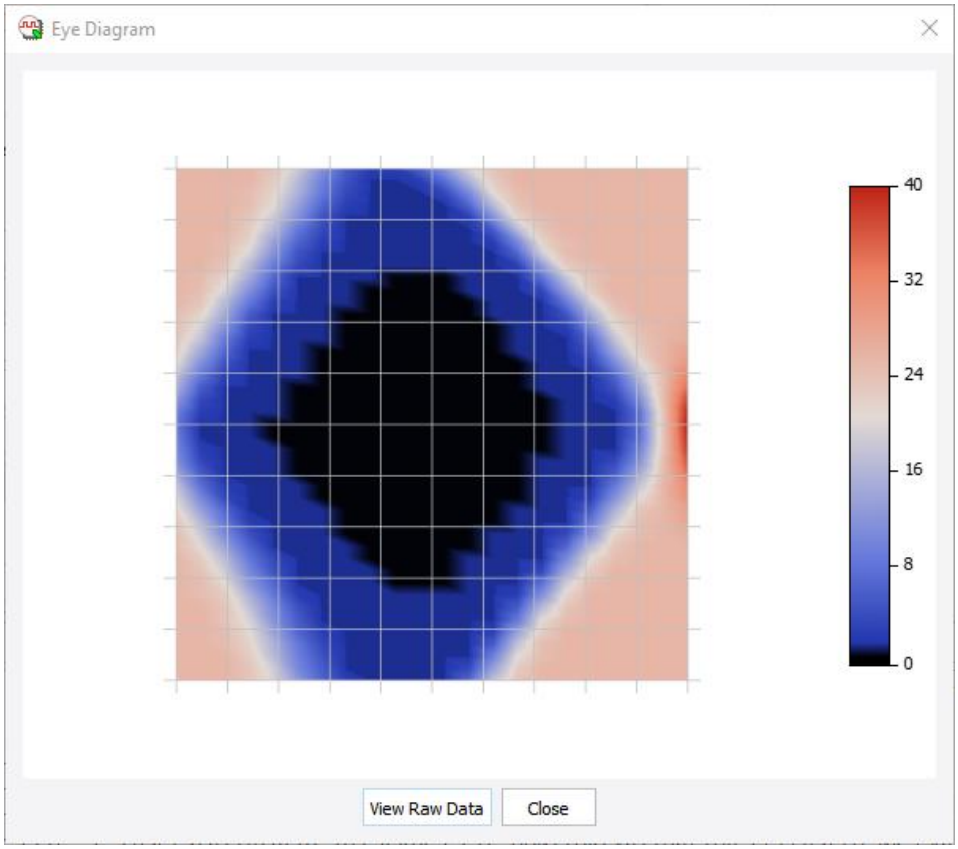
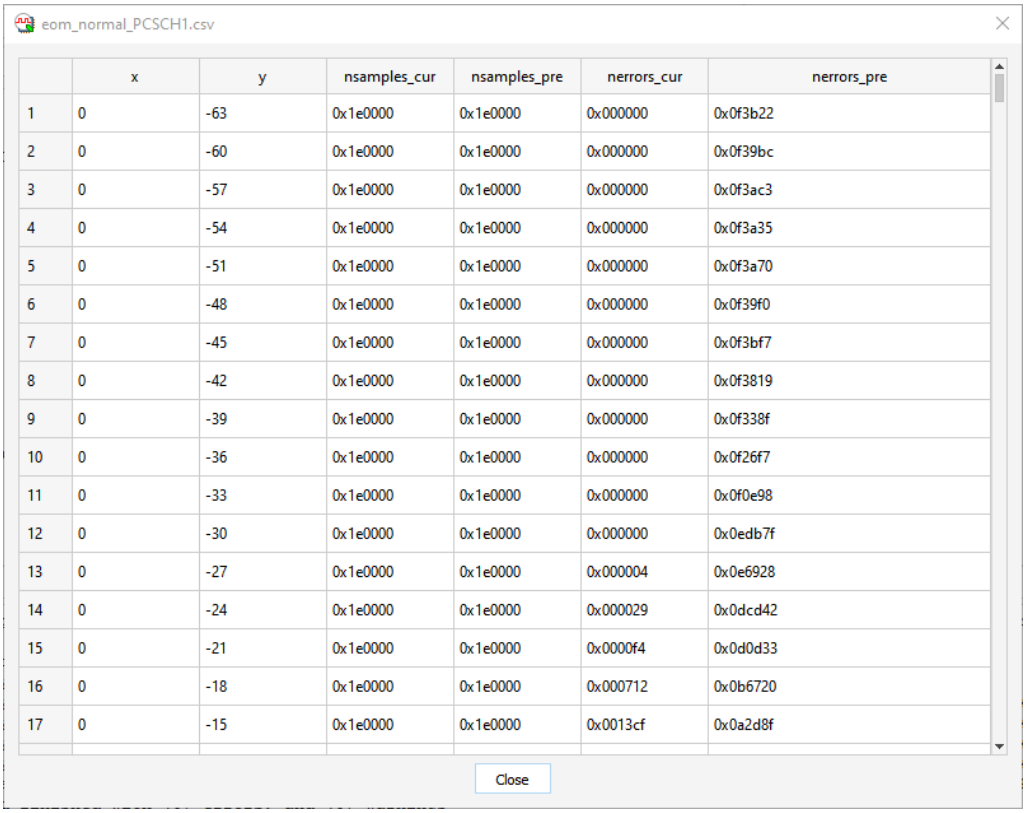


Figure 3.17. Generated Eye Diagram by Reveal Eye Open Monitor

At the bottom of the eye-diagram window is the *View Raw Data* button. Clicking this opens the new window containing the data used to generate the eye diagram as shown in Figure 3.18.



	x	y	nsamples_cur	nsamples_pre	nerrors_cur	nerrors_pre
1	0	-63	0x1e0000	0x1e0000	0x000000	0x0f3b22
2	0	-60	0x1e0000	0x1e0000	0x000000	0x0f39bc
3	0	-57	0x1e0000	0x1e0000	0x000000	0x0f3ac3
4	0	-54	0x1e0000	0x1e0000	0x000000	0x0f3a35
5	0	-51	0x1e0000	0x1e0000	0x000000	0x0f3a70
6	0	-48	0x1e0000	0x1e0000	0x000000	0x0f39f0
7	0	-45	0x1e0000	0x1e0000	0x000000	0x0f3bf7
8	0	-42	0x1e0000	0x1e0000	0x000000	0x0f3819
9	0	-39	0x1e0000	0x1e0000	0x000000	0x0f338f
10	0	-36	0x1e0000	0x1e0000	0x000000	0x0f26f7
11	0	-33	0x1e0000	0x1e0000	0x000000	0x0f0e98
12	0	-30	0x1e0000	0x1e0000	0x000000	0x0edb7f
13	0	-27	0x1e0000	0x1e0000	0x000004	0x0e6928
14	0	-24	0x1e0000	0x1e0000	0x000029	0x0dcd42
15	0	-21	0x1e0000	0x1e0000	0x0000f4	0x0d0d33
16	0	-18	0x1e0000	0x1e0000	0x000712	0x0b6720
17	0	-15	0x1e0000	0x1e0000	0x0013cf	0x0a2d8f

Figure 3.18. Raw Data Used to Calculate and Generate the Eye Diagram

Appendix A. Using the JTAG Hub Primitive

Similar to other Lattice primitives, the JTAGhub primitive used by Reveal is instantiated in custom user logic. To use a soft JTAG hub, choose the JTAGH25SOFT primitive. To use a hard JTAG, choose the JTAGH25 primitive. The usage for both primitives is essentially the same. The main difference is the way each JTAGhub is implemented. The hard JTAG uses the hardened JTAG block on an FPGA, while the soft JTAG is implemented in RTL.

When using JTAGhub, JTAG primitive cannot be used if JTAGH25 primitive is already included in the design. JTAGH25 already contains a JTAG primitive and the synthesis tool produces an error if more than one JTAG primitive is detected.

Figure A.1 depicts the block based diagram of input and output ports for the JTAGhub primitive. This diagram is the same for both JTAGH25 and JTAGH25SOFT.

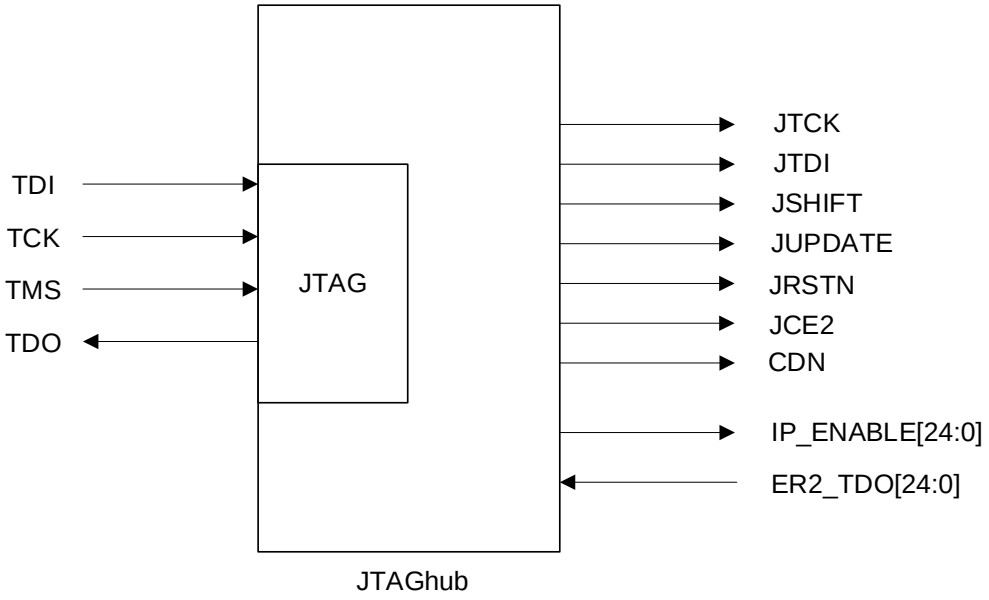


Figure A.1. JTAGhub Primitive Block Based Diagram

When implementing the JTAGhub primitive in user logic, the TCK, TMS, TDI and TDO ports should be connected to external ports on the FPGA, while all other ports should be connected internally within the user logic. Refer to Table A.1 for more information on the input and output ports of the JTAGhub primitive.

Table A.1. JTAGH19 and JTAGH25SOFT Lattice Primitives Signal Descriptions

Signal	Description
JTCK	The same signal as TCK.
JTDI	Registered version of the TDI signal.
JSHIFT	Indicates if valid data is available in JTDI. Is asserted and de-asserted on the positive edge of JTCK.
JUPDATE	Used to capture shifted data by JTDI. A positive pulse is output one-cycle after JSHIFT is de-asserted. Signal is asserted and de-asserted on the positive edge of JTCK.
JRSTN	Active low reset signal. Set low initially and then remains high after initialization.
JCE2	Signal indicating valid data is available in JTDI for the selected core. Remains high as long as data is valid. Signal is asserted one clock before JSHIFT and is de-asserted at the same time as JSHIFT.
CDN	Unused signal.
IP_ENABLE[18:0]	Signal enabling one bit from each debug core. For Avant, this signal is 25-bits instead of 19-bits.
ER2_TDO[18:0]	Input data signal to be shifted out of TDO. Each respective bit corresponds to the same bit of IP_ENABLE. For Avant, this signal is 25-bits instead of 19-bits.

Figure A.2 and Figure A.3 depict valid timing diagrams for these signals in the JTAGhub primitive.

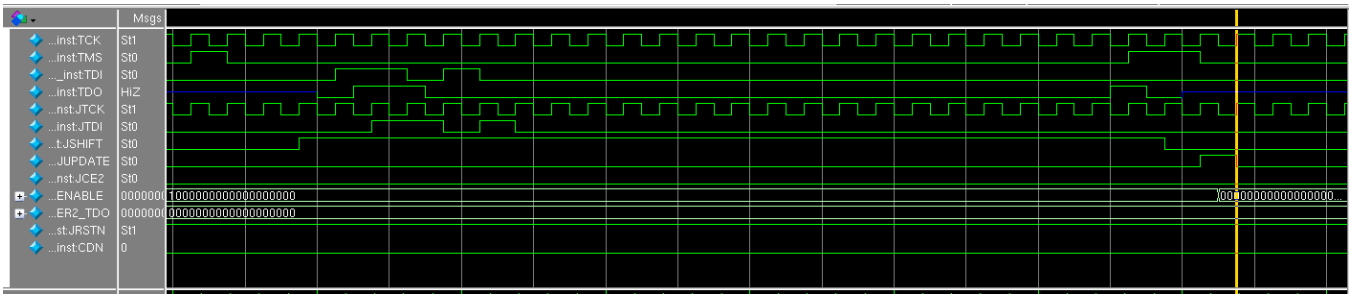


Figure A.2. JTAGhub Primitive Timing Diagram

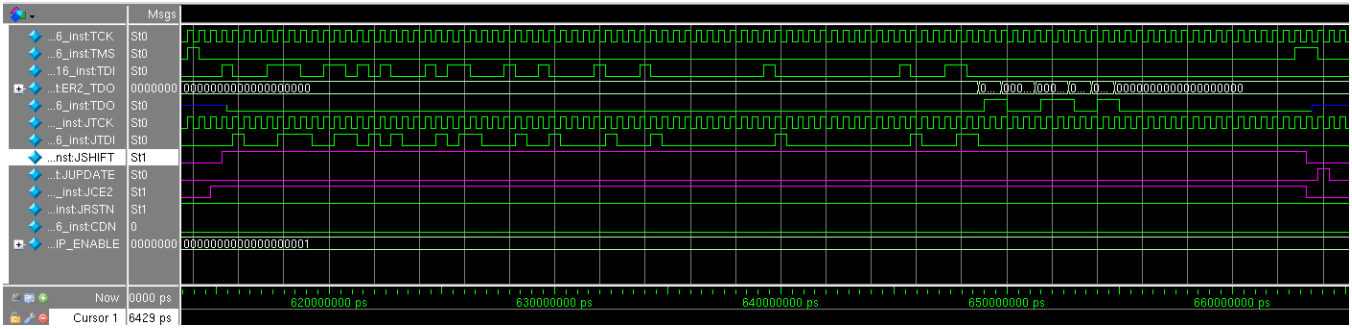


Figure A.3. Additional JTAGhub Primitive Timing Diagram

Whenever a bit of the IP_ENABLE signal is active high at the same time as JCE2 and JSHIFT, valid data from JTDI is shifted-out with its corresponding bit from ER2_TDO.

Some additional considerations to keep in mind when using the JTAGhub primitive is that TCK does not work like a typical clock. Instead, the TCK signal is pulsed whenever data is transmitted and received.

Aside from the signals mentioned above, the JTAGhub component also has several parameters corresponding to each of the debug cores within the JTAG hub. For Avant devices, up to 25 debug cores are supported, while for all other devices, up to 19 debug cores are supported. However, it is recommended that cores 17 and 18 are used in custom user RTL. The reason for this is because the first few debug cores are typically reserved for other types of debugging, such as Reveal Analyzer and GDB. For portability from other device families, it is also recommended to use core 17 and core 18 for Avant devices so that the only thing that needs to be updated is the primitive itself.

To enable a debug core for use with a primitive, change the corresponding parameter for the debug core from 0. All cores for use with Reveal Analyzer are initialized with the parameter value of 0x43. All other cores intended for custom use should be parameterized with 0x1.

For example, to use core 17 of the JTAGhub primitive, initialize the parameter “HUB_17” to 0x1, and connect the IP_ENABLE[17] and ER2_TDO[17] signals accordingly.

The following are examples of JTAG commands:

- Reading the ID of debug cores

```
! enable ER1
SIR 8 TDI (32); //8-bit data with hex value of 32
! set reveal core
SDR 24 TDI (800006); //24-bit data with hex value of 800006
```

- Enabling CORE0

```
! enable ER1
SIR 8 TDI (32);
SDR 24 TDI (000016);
```

- Enabling CORE2

```
! enable ER1
SIR 8 TDI (32);
SDR 24 TDI (000046);
```

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, refer to the Lattice Answer Database at www.latticesemi.com/Support/AnswerDatabase.

Revision History

Revision 1.1, April 2024

Section	Change Summary
Disclaimers	Updated disclaimers.
Reveal Analyzer	Minor editorial fixes.

Introduction	• In the JTAG Interface Usage section:
	  • Added discussion on instantiating a JTAG hub primitive and cross reference to appendix.
	  • Removed discussion on using only a hard JTAG for debugging with Reveal.
	• Moved the Using the JTAG Hub Primitive section into the appendix.

Reveal Controller	• In the Configuring Hard IP section:
	  • Added discussion on PCIELL Hard IP exception in relation to LMMI ports when using the Reveal Controller's Hard IP function.
	  • Added Figure 3.10. Reveal Controller LMMI Arbitration Scheme for PCIELL.
	• Updated discussion on the purpose of the Reveal Controller's Eye-Opening Monitor function in the Eye-Opening Monitor section.

Appendix A	Created appendix section.

Revision 1.0, January 2023

Section	Change Summary
All	Initial release.

www.latticesemi.com