XAPP1322 (v1.0) November 7, 2017

Transceiver Link Tuning

Author: Giovanni Guasti, Antonello Di Fresco, and Erik Schidlack

Summary

Manual link fine tuning is known to be a tedious and time-consuming operation. The search for the highest link margin can delay an engineer in the laboratory for days. This application note provides an automated fine tuning method for transceiver links. The automatic link tuning application drives through a step-by-step experience on how to use the GT Debugger methodology for the best equalization setup, power reduction, crosstalk analysis, and measurement documentation.

Download the reference design files for this application note from the Xilinx website. For detailed information about the design files, see Reference Design.

Introduction

In a typical transceiver link tuning analysis the user manually modifies the transmitter setup (amplitude, post-emphasis, and pre-emphasis) according to the signal measured at the receiver end and then chooses the best compromise. The goal is to find the transmitter setup that guarantees the highest receiver margin. In cases where power consumption or crosstalk from multiple aggressors must be controlled or reduced, the problem becomes even more intricate because additional variables are present.

Link tuning is known to be a tedious procedure that should be repeated for every link in the design. Due to the high number of configurations tested, the challenge with manual tuning is to keep a well-ordered report of all experiences.

This application note presents a method to run automatic tuning of the channel. This method is applicable when both sides of the link (transmitter and receiver) are Xilinx parts and are controlled by hardware manager in the Vivado® design suite.

The application note makes extensive use of the GT Debugger tool presented in *Automatic Insertion of Debug Logic for Transceivers in Synthesis DCP* (XAPP1295) [Ref 1]. The GT Debugger offers a set of pre-built Tcl functions enabling transceiver ports probing and driving, and DRP access.

This application note is organized as a walk-through debug session, starting from a guided implementation of the design in *Automatic Insertion of Debug Logic for Transceivers in Synthesis DCP* (XAPP1295) and finally focused on signal integrity, power and noise optimization, and test documentation. Simple procedures are combined together to form a more complex analysis path.

Features

The GT Debugger is a powerful set of Tcl procedures allowing a low-level transceiver debug without adding any additional HDL code to the design under test. For this reason, the GT Debugger is best suited to all cases where the HDL code is protected or encrypted, or where the HDL code is simply not available.

The GT Debugger inserts its logic beside the original design, providing complete control of any transceiver input or output port. The GT Debugger also allows you to modify any of the transceiver registers during runtime and provides high-level procedures for safe read-modify-write operations.

Some higher level Tcl routines combining general-purpose I/O (GPIO) and dynamic reconfiguration port (DRP) access are presented here as examples. These allow flexible control of the transceiver and more sophisticated analysis like the eye scan, or even a full automatic TX and RX link tuning.

GT Debugger Design Flow

For a detailed overview of how to instantiate a GT Debugger, refer to *Automatic Insertion of Debug Logic for Transceivers in Synthesis DCP* (XAPP1295). The GT Debugger insertion works starting from the synthesis design checkpoint.

An automatic procedure analyzes the DCP file and creates a summary file with the extension `.do`. You can manually edit the `.do` file or leave the GUI to edit it according to your preferences (refer to Helping GUI for GT Debugger). It is also possible to parse the `.do` file with a Tcl script, and this is the quickest method in case of repetitive debug activities (see Figure 1).

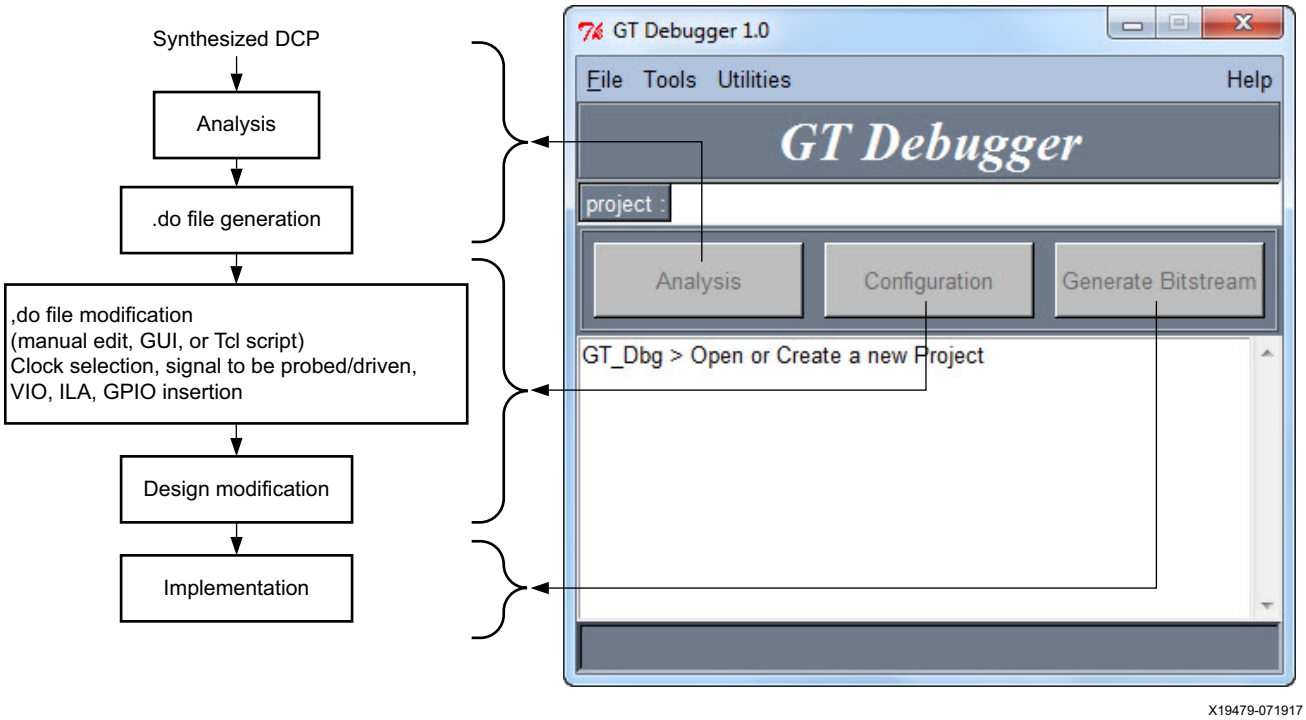

Figure 1: **GT Debugger Implementation Flow and Relative GT Debugger GUI Sections**

Helping GUI for GT Debugger

This application note provides a GUI to drive the user through the implementation of the GT Debugger.

Installation

Unzip the provided reference design file `xapp1322-transceiver-link-tuning.zip` to any location. A directory called `gt_debugger_<version_number>` is created. Here, `<version_number>` is the latest available version and can be 1.0 or higher. The path to the gt_debugger GUI should be stored in a new Windows environment variable named GTDBG (Figure 2).

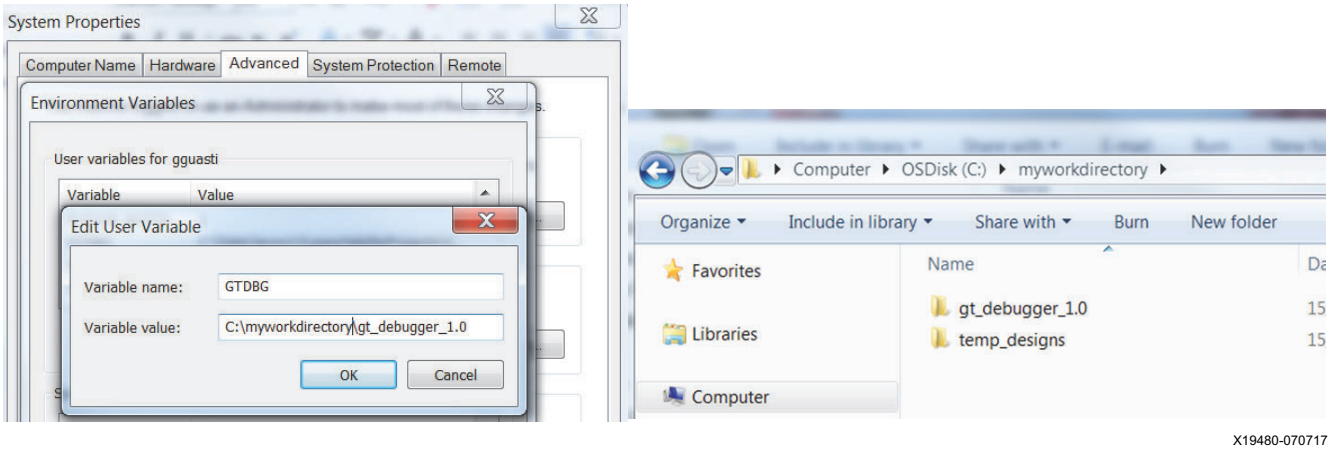

Figure 2: **Environment Variable GTDBG for Windows**

You can create a directory in which to store the modified designs and name it `temp_designs`, for example. In the Linux environment, you can set the variable to `setenv GTDBG /myworkdirectory/gt_debugger_1.0`.

The file `config.tcl` is available in the `gt_debugger_1.0` directory after the tool is installed. In this release, the `config` file is used to define the text editor you want to enable to open the log files. It is possible to define the text editor for both platforms, Linux and Windows. For Linux, it is enough to write the name of the editor. For Windows, the complete path is needed. If not set, the default editor is the native Windows notepad.

To modify the setup of the text editor, edit lines 2 and/or 6 (bold text only) in `config.tcl`.

```
1 # set text editor LINUX. The path is not needed.
2 set text_editor_l "gedit"
3
4 # set text editor path WINDOWS.
5 #If not set the default is the native Windows notepad.
6 set text_editor_w "C:/Program Files (x86)/Notepad++/notepad++.exe"
```

Working with the GUI

The tool has been developed using Tcl/Tk. A separate installation of the Tcl/Tk distribution is not needed in the Windows environment because the GT Debugger uses the distribution that is embedded in the Vivado tools installation.

The Vivado design suite does not have an embedded distribution in Linux, so an installation of Tcl/Tk is needed. Tcl/Tk should come with the Linux distribution by default, so there should be no need to install it separately.

For Windows, a shortcut named gt_dbg is provided. Before starting it, open the properties window of the shortcut and set the correct path to the Vivado tools installation. By default, the tool starts from `c:\Temp`. If this folder doesn't exist, you can create it or modify the path in the **Start in** box of the gt_dbg shortcut properties. By clicking the shortcut, a new Vivado tools Tcl shell opens and the GUI appears after a few seconds (Figure 3).

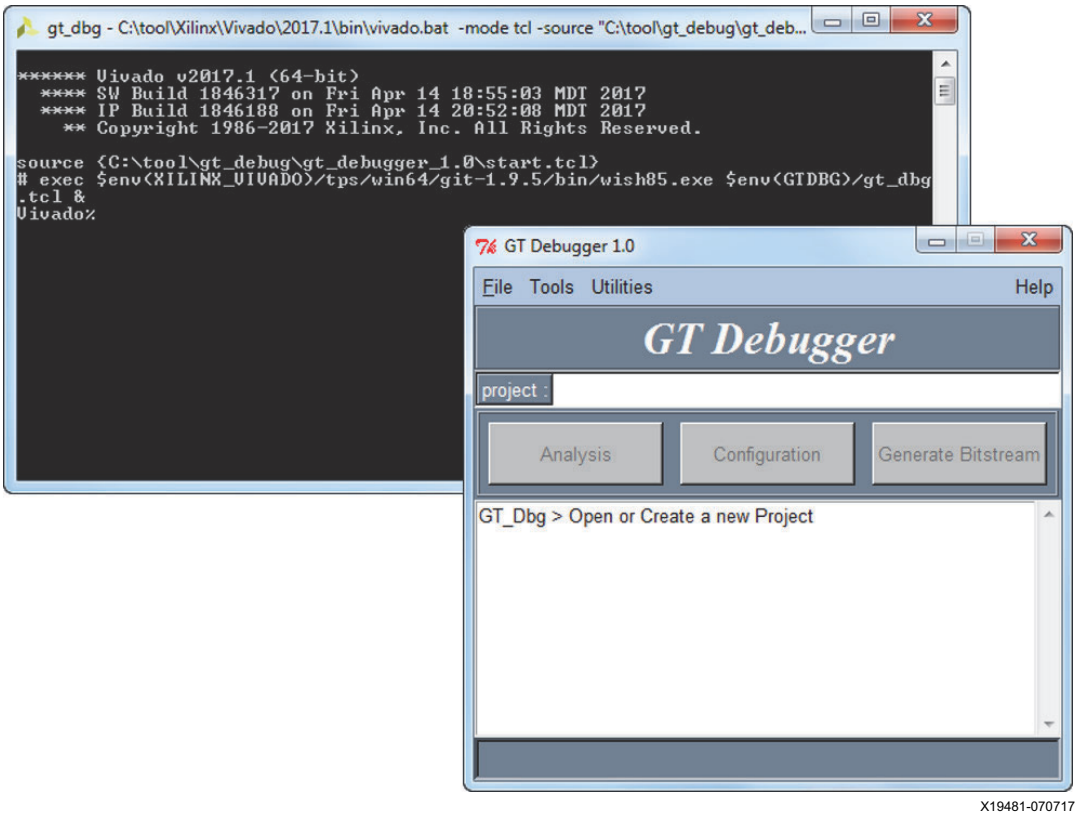

Figure 3: **Initial Appearance of GT Debugger GUI**

To open the GUI in Linux, execute `gt_dbg` in a terminal. From the File menu it is possible to create a new project or open existing projects. If a new project is created, you should also select the post synthesis DCP file (Figure 4).

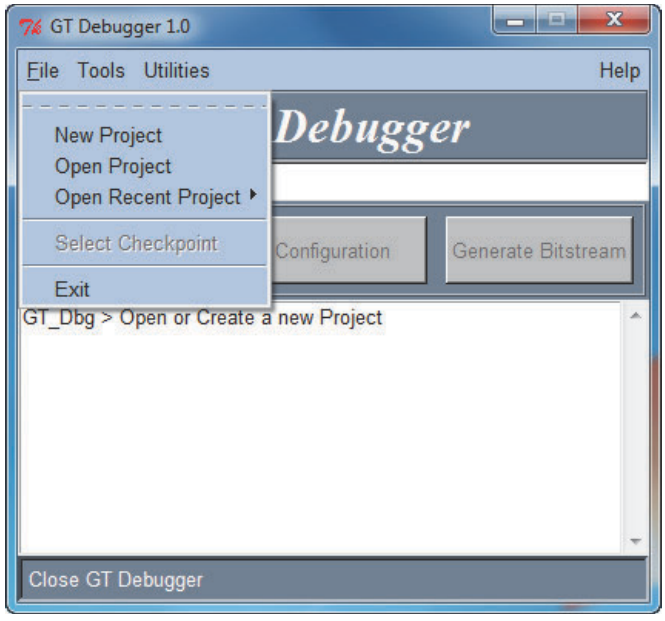

Figure 4: **File Menu**

In a new project, only the **Analysis** button is enabled. The other buttons are enabled as soon as the mandatory steps of the flow are completed. Opening an existing project enables the buttons depending on the steps that have already been completed. The analysis of the DCP file generates the `.do` file, which is a list of available transceivers with all available ports and interfaces (Figure 5).

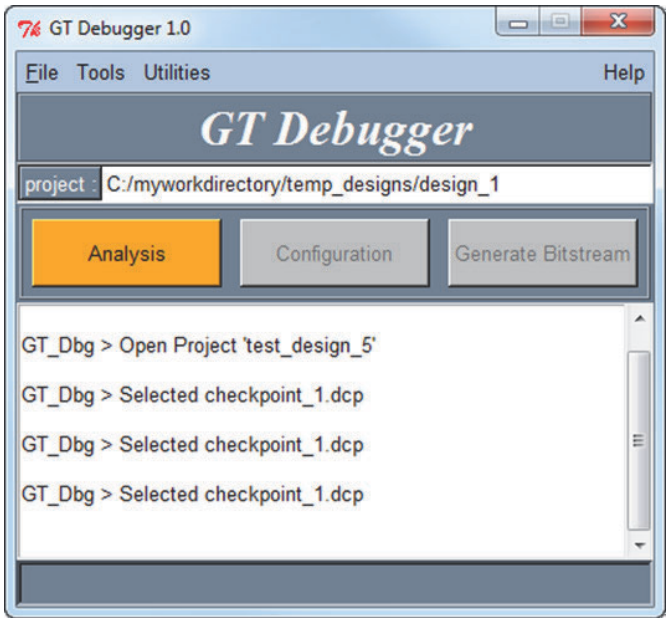

Figure 5: **Running the Initial Analysis on the Post Synthesis DCP**

Probed Port Selection

After the analysis of the DCP files has completed, you can configure the GT Debugger. Here it is possible to select the probes to be assigned and the port or interfaces to be probed. The common and channel entities are shown in the left column (Figure 6).

There are four possible signal connection choices:

- GPIO: Adding a port to this set allows you to drive or read the port from the Tcl console.
- VIO: Adding a port to this set allows you to probe the activity directly with a virtual input/output (VIO).
- GPIO + ILA: This is the same as GPIO, but it also adds the integrated logic analyzer (ILA) feature for a complete signal activity measurement.
- VIO + ILA: This is the same as VIO, but it also adds the ILA feature for a complete signal activity measurement.

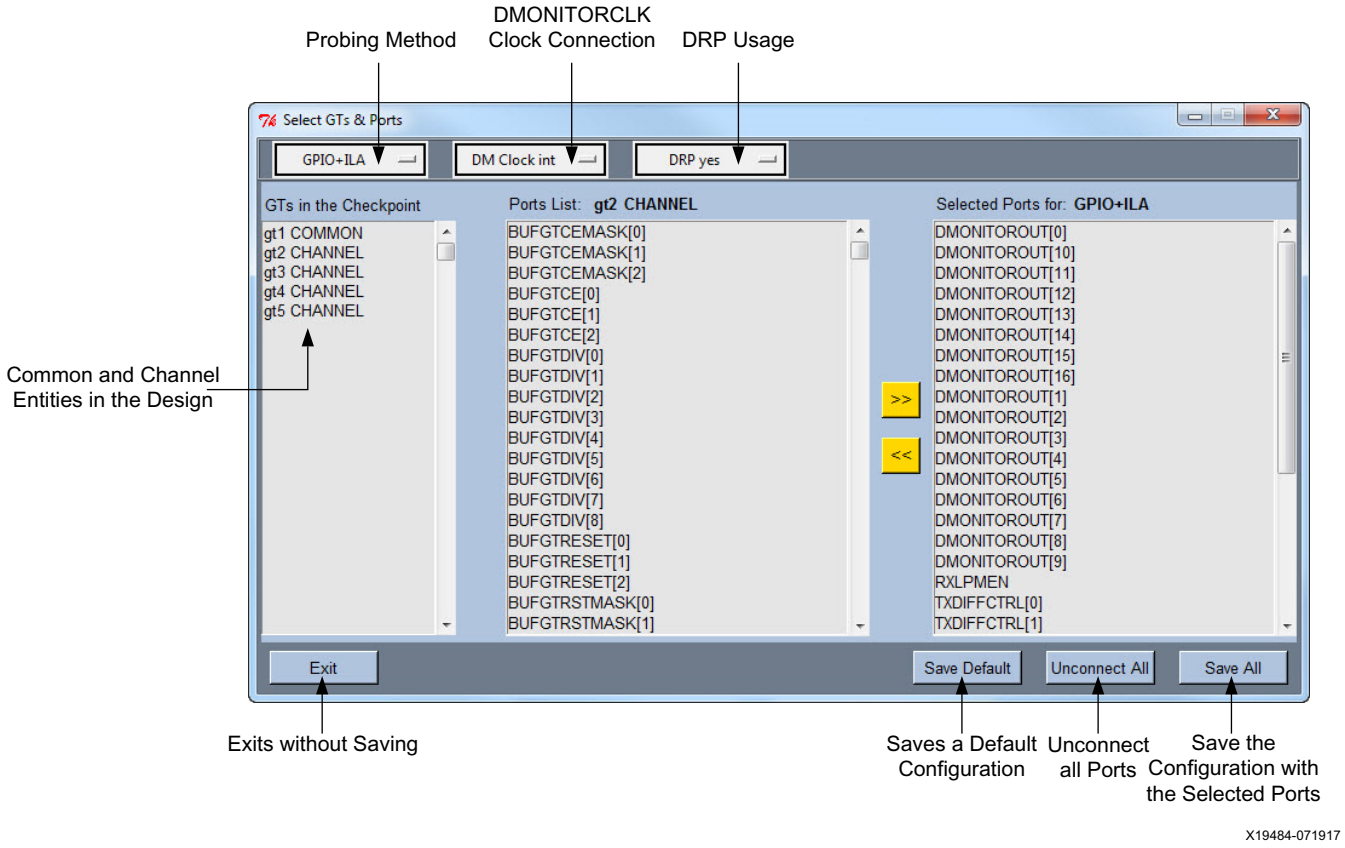

Figure 6: **Selection of Ports to be Probed or Driven in the GT Debugger**

The GT Debugger also allows you to probe the DMONITOR port. The DMONITOR port requires a free running clock to be connected to DMONITORCLK. This clock can be internal, external, or neither. Connect the DMONITORCLK only if the DMONITOR is used.

The DRP port can also be accessed with read and write operations. You can select how to connect the DMONITORCLK and the DRP port in the GUI according to this sequence of steps:

1. Select the channel or common to configure. The port column in the middle is populated.
2. Select the probe (i.e., GPIO). The assignment arrow is activated.
3. Move the ports to be probed to the right column.
4. Save and eventually exit.

There is a quicker method to connect some ports that allows you to skip the steps described above. By clicking the button **Save Default**, the tool loads and saves a default configuration. In the root directory of the tool is a file called `default.tcl`. You can edit this file to define a default configuration. If **Save Default** is used after having connected some ports, the tool disconnects all ports and then connects them accordingly to `default.tcl`.

After the **Save Default** operation, you can exit without clicking **Save All** or connecting other ports according to the steps described above. With the last step, the selected ports are appended in the `.do` file. At this point, you should click **Save All** before exiting.

Clicking the **Unconnect All** button cleans the `.do` file by removing all the connections.

When the configuration is done, it is possible to move to the next step and run Generate Bitstream. This step launches the synthesis, implementation, and bitstream generation. After generating the bitfile (Figure 7), the GUI can be closed and the GT Debug can proceed with the Vivado hardware manager.

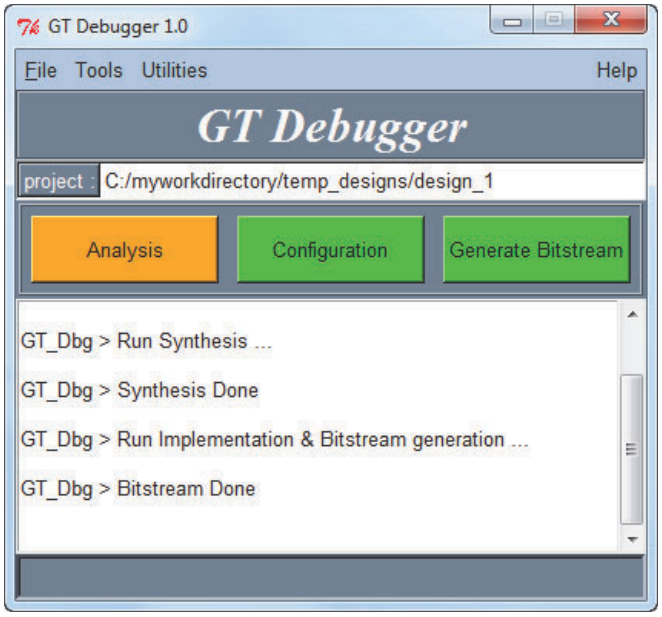

Figure 7: **Completion of Bitfile Synthesis, Implementation, and Bitfile Generation**

GT Debugger GUI - Tools

The following options are available (Figure 8) under the Tools menu:

- Report:
  - Analysis LogFile: Opens the analysis log file.
  - Synthesis LogFile: Opens the synthesis log file.
  - Implementation LogFile: Opens the implementation log file.
  - Configuration DumpFile: Extracts the connected ports from the `.do` file and opens the file.
- Edit Configuration: Opens the `.do` file to be edited manually in the text editor.
- Run Synthesis: Run synthesis only.
- Run Implementation: Runs implementation and bitstream generation.

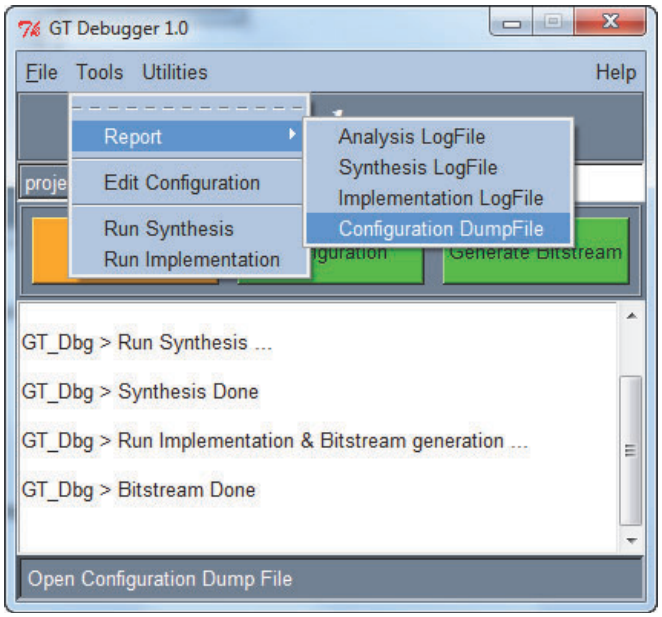

Figure 8: **Menu Tools**

GT Debugger GUI - Utilities

From the Utilities menu, you can directly open the Vivado hardware manager and automatically load the setup needed to debug the transceivers (Figure 9).

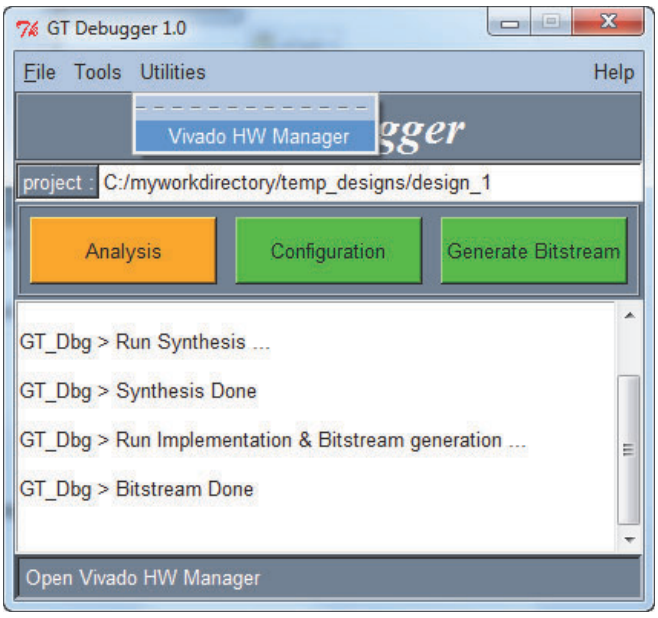

Figure 9: **Built-in Utilities**

Working with GT Debugger

This section describes a set of experiments of growing complexity. The experiments allow you to develop confidence in carrying out the basic procedures while developing new functionalities. Xilinx recommends running the experiments in sequential order from start to finish. All experiments are listed in the `TopDemo.tcl` file.

Copy the commands from the `TopDemo.tcl` file and paste them into the Vivado tools Tcl shell. The command execution order is important, and the first operation required is to source the file `tuningproc.tcl`. This is the repository for the procedures called during the experiments. A third file, `configuration.tcl`, describes the link connections and the tuning strategy.

Tuning and Debugging Procedures

The file `tuningproc.tcl` is the repository for the Tcl procedures called during the experiments. With the aim of making simple procedures for simple tasks, a nested procedure hierarchy results in few cases. Table 1 provides a quick description of these procedures and passed parameters. Calls to other procedures are also shown, when present.

Table 1: **Tuning and Debugging Procedures**

Procedure Name	Description	Called Procedures
Equalizer and Eye Scan Procedures		
monitoreq {rx}	Monitor for the status of the channel decision feedback equalizer (DFE) or low-power mode (LPM) equalizer. The *rx* is the channel to be probed (i.e., cX0Y8).	monitordfe monitorlpm
monitordfe {rx}	Monitor for the DFE equalizer. The *rx* is the channel to be probed (i.e., cX0Y8).	igd_gpio_ctr_oupdate_set igd_gpio_ctr_iupdate_set igd_drp_${rx}_rmw igd_data_slice
monitorlpm {rx}	Monitor for the LPM equalizer. The *rx* is the channel to be probed (i.e., cX0Y8).	igd_gpio_ctr_oupdate_set igd_gpio_ctr_iupdate_set igd_drp_${rx}_rmw igd_data_slice
igd_data_slice {data width from to}	Modified version of the GT_Debugger igd_data_slice. The procedure extracts an offset slice of data: • *data*: Input data where the slice should be taken. • *width*: Slice width (not used). • *from*: Starting point. • *to*: End point.	
explore {}	Explores transceiver resources in local and remote boards. Also initializes the eye scan function at local and remote receivers.	sendcom igd_eyescan

Table 1: **Tuning and Debugging Procedures *(Cont'd)***

Procedure Name	Description	Called Procedures
eyescan {link prescale vect}	Runs the vector eye scan on the link with a defined prescale. • *link*: Identifies the receiver link according to `configuration.tcl`. • *prescale*: Refer to the "RX Margin Analysis" section in the UltraScale transceiver user guides [Ref 2]. • *vect*: Determines the number of directions for the vector eye scan (4 generates an eye plot with four vertices).	igd_eyescan sendcom
man_scan {link txswing txpost txpre prescale}	Given the link, manually sets the transmitter, runs an eye scan, and saves the result: • *link*: Identifies the receiver link according to `configuration.tcl`. • *txswing:* Main cursor of the transmitter. • *txpost*: Post-cursor of the transmitter. • *txpre*: Pre-cursor of the transmitter. • *prescale:* Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2].	man_tx eyescan
aperture {ber_d rx prescale}	Width and height aperture measurement after eye diagram: • *ber_d*: A dictionary containing the result of the eye scan. • *rx*: Channel to be probed (i.e., cX0Y8). • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2].	
findmaxap {prescale linkname}	Returns the configuration with maximum aperture: • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2]. • *linkname*: Identifies the receiver link according to `configuration.tcl`.	
Link Tuning Procedures		
drpdump {}	Returns all transceiver channel attributes via DRP read operations. The ordered attributes are written in tabular format in the file `drp_dump.txt`.	igd_drp_rd

Table 1: **Tuning and Debugging Procedures *(Cont'd)***

Procedure Name	Description	Called Procedures
tune {link prescale new}	Main tuning loop with convergence strategy. The strategy is decided in the configuration file dictionary `strategy_d`: • *link:* Identifies the receiver link, according to `configuration.tcl`. • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2]. • *new*: If new = 1, new measurements replace old measurements. If new = 0, old measurements are kept. Only new cases are added.	sendcom BF_scan Sweetspot
sweetspot {link prescale}	Brings the TX to the best setup and runs an eye scan. Updates the `best_d` dictionary by appending the current configuration: • *link:* Identifies the receiver link, according to `configuration.tcl`. • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2].	man_scan
BF_scan {link minswing maxswing swd minpost maxpost pod minpre maxpre prd prescale new}	Brute force scan: • *link:* Identifies the receiver link according to `configuration.tcl`. • *minswing:* Signal main-cursor initial value. • *maxswing*: Signal main-cursor final value. • *swd*: Main-cursor variation step. • *minpost:* Signal post-cursor initial value. • *maxpost*: Signal post-cursor final value. • *pod*: Post-cursor variation step. • *minpre*: Signal pre-cursor initial value. • *maxpre*: Signal pre-cursor final value. • *prd*: Pre-cursor variation step. • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2]. • *new*: If new = 1, new measurements replace old measurements. If new = 0, old measurements are kept. Only new cases are added.	man_mux man_scan
man_tx {link txswing txpost txpre}	Sets the transmitter TXDIFFCTRL, TXPOSTCURSOR, and TXPRECURSOR: • *link:* Identifies the receiver link according to `configuration.tcl`. • *txswing:* Signal main-cursor value. • txpost: Signal post-cursor value. • *txpre:* Signal pre-cursor value.	igd_gpio_${tx_gt}_TXDIFFCTRL_0_wr_d igd_gpio_${tx_gt}_TXPOSTCURSOR_0_wr_d igd_gpio_${tx_gt}_TXPRECURSOR_0_wr_d sendcom

Table 1: **Tuning and Debugging Procedures *(Cont'd)***

Procedure Name	Description	Called Procedures
man_mux {link}	Turns on the GT Debugger multiplexers for manual GPIO change. • *link*: Identifies the receiver link according to `configuration.tcl`.	igd_gpio_ctr_oupdate_set igd_gpio_${tx_gt}_TXDIFFCTRL_0_wr_m igd_gpio_${tx_gt}_TXPOSTCURSOR_0_wr_m igd_gpio_${tx_gt}_TXPRECURSOR_0_wr_m sendcom
defaulttx {}	Removes TXELECIDLE if present. Sets all TX to a default value.	man_tx
generatebestd {prescale}	Generates the dictionary `best_d` according to measurements. This is the best configuration for each link: • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2].	findmaxap
delmeasurement {}	Deletes the measurement stored in dictionary `margin_d`.	
savetofile {filename link prescale}	Saves the measurement to file for report. Saves `margin_d` (apertures dictionary) to a file in the format *swing post pre width height*: • *filename*: Name of the file to be saved. • *link*: Identifies the receiver link according to `configuration.tcl`. • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2].	
savemeas {filename}	Saves measurement in `margin_d` to file.	
loadmeas {filename}	Recalls measurement from file.	
Crosstalk Analysis Procedures		
resetRX {link}	Toggles the VIO driving hb_gtwiz_reset_all_vio_int in the Wizard example design. Replace the script with the command generated by the hardware manager when the hb_gtwiz_reset_all_vio_int VIO button toggles Low-High-Low. The reset is needed during crosstalk analysis.	
txidle {link status}	Controls the transmitter electrical idle status. • *link*: Identifies the receiver link, according to `configuration.tcl`. • status idle is 1; active is 0.	igd_gpio_ctr_oupdate_set igd_gpio_${txgt}_TXELECIDLE_wr_m igd_gpio_${txgt}_TXELECIDLE_wr_d sendcom
linksoff {}	Switches all transmitters to electrical idle.	txidle
xtalk01 {prescale lista}	Tunes one single link with all other aggressors off. After xtalk01 completes, the best setup is stored in ::best_d: • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2]. • *lista*: Link set where the crosstalk is analyzed.	linksoff resetRX tune txidle

Table 1: **Tuning and Debugging Procedures *(Cont'd)***

Procedure Name	Description	Called Procedures
xtalk02 {prescale}	xtalk02 switches on one aggressor at a time, and repeats this for each victim and for each aggressor. It finally fills the table of influence. • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2].	linksoff man_mux man_tx txidle resetRX
tree2olist {tree prescale linkname width}	Sorting procedure used from Xtalk procedures. Converts from a dictionary to an ordered list. • *tree*: Dictionary of margin measurements. • *prescale*: Refer to the Eye Scan chapter in the UltraScale transceiver user guides [Ref 2]. • *linkname*: Identifies the receiver link, according to `configuration.tcl`. • *width*: Eye width.	
xtalk03 {}	Finds the worst aggressor link.	tree2olist
Socket Procedures		
startclient {}	Starts the socket client	sendcom
startserver {}		vwait
accept {s a port}	Runs another procedure when a fileevent occurs: • *s*: Socket identifier. • *port*.	echo
echo {s}	Echoes and executes a message. • *s*: Socket identifier.	
sendcomm {s comm}	Send a command through the socket channel • *s*: Socket identifier. • *comm*: Command string.	
GetData {chan}	Gets data from a channel: • *chan*: Socket identifier.	

Hardware Description

All figures and examples in this application note refer to a test case designed for the GTH transmitter and receiver in UltraScale devices. The method is general and can be replicated with any combination of transceivers from the 7 series, UltraScale, and UltraScale+ devices.

For a fully functional test, two asynchronous sets of transceivers should be present. In the reference design as shown in Figure 10, two asynchronous sets of transceivers on two different boards (near-end (NE) and far-end (FE) side) are configured and controlled with two independent Vivado hardware managers using independent JTAG links.

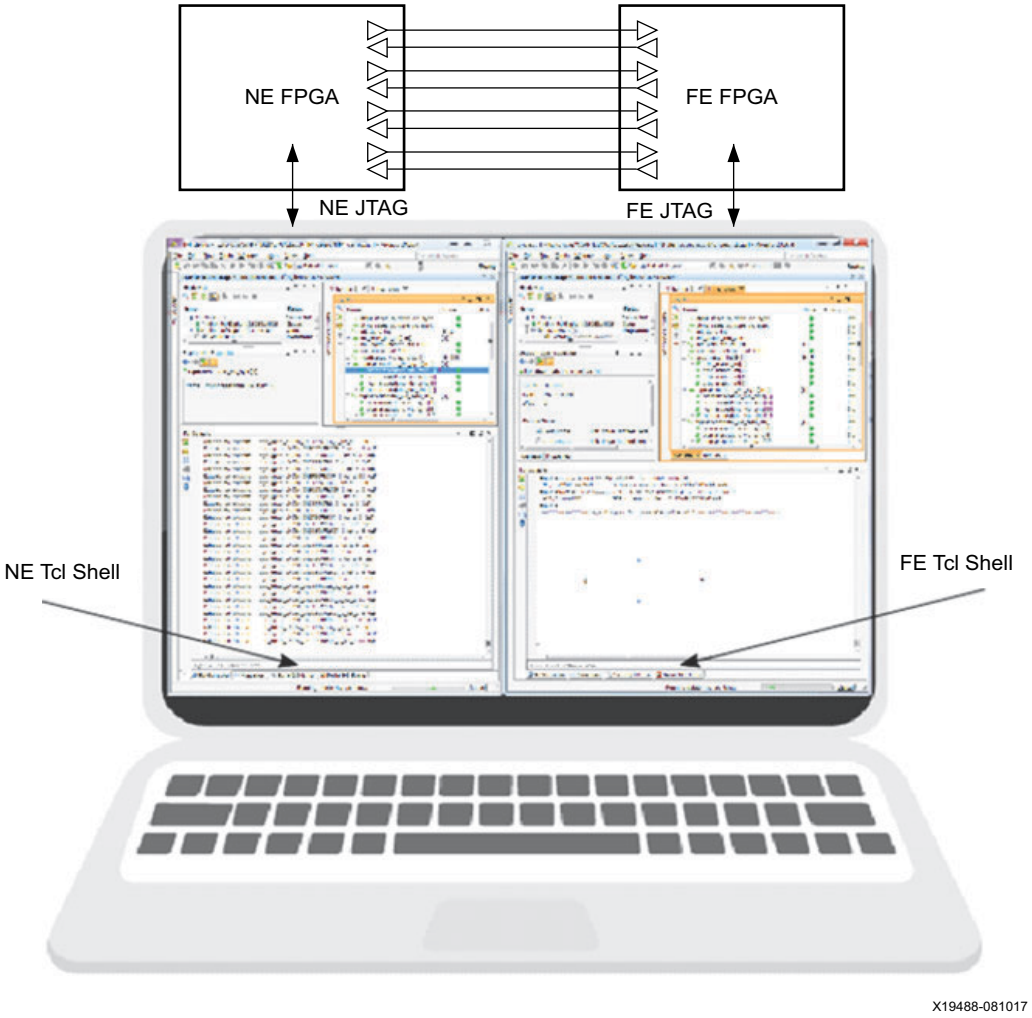

Figure 10: **Hardware Configuration**

Basic Design Generation

The GT Wizard example design is an easy starting point because it generates a design with all the necessary resources for clock distribution, reset, pattern generators, and checkers. The GT Wizard example design generated here consists of four independent bidirectional 10 Gb/s data rate channels with PRBS31 pattern generators and checkers.

The two FPGAs in the test referred to as the NE and FE FPGAs can be programmed with the same design. In this application note, two transceiver characterization boards, UC1250 and UC1287, were used (Figure 20).

Although the Vivado tools allow you to target multiple JTAG servers and multiple FPGAs with a single hardware manager, due to some temporary restrictions the GT Debugger still requires that one hardware manager be dedicated to only one FPGA.

Automatic resets on errors should be prevented. Figure 11 shows the VIO gating the resets due to PRBS errors (resetonerror). Table 2 summarizes the main characteristics of the GTH Wizard example design. Raw data is transmitted and received (with no encoder and decoder), and both TX and RX buffers are used.

Table 2: **Example Design Characteristics**

Field	Value
Data rate	10 Gb/s
Pattern	PRBS31
Quads in the design	1
Transceiver	GTH transceiver in UltraScale device
REFCLK	250 MHz
DRPCLK	100 MHz

Customization of GT Wizard Example Design for Link Tuning

The following HDL modifications to the plain Wizard example design were necessary:

- Differential free-running clock: The original GT Wizard example design has a single-ended free-running clock. The input buffer has been replaced with a differential input buffer.
- Gated reset on error: This is necessary to prevent unwanted automatic resets following data errors. The signal prbs_error_any_sync is gated by the signal resetonerror. A VIO drives the resetonerror signal.

The modified file `gtwizard_ultrascale_0_example_top.v` is included in the application note for reference, and the modified lines are highlighted with comments. For the experiments described in this application note, GPIOs, VIOs, and ILAs were added to some transceiver ports. These ports are used in this application note:

- DRP
- RXLPMEN
- TXDIFFCTRL
- TXPOSTCURSOR
- TXPRECURSOR
- DMONITOROUT
- DMONITORCLK
- LOOPBACK

The 100 MHz free-running clock for the reset sequence also sources the DRP interface and the DMONITORCLK. The GUI provided with this application note makes the port selection immediate. From the GUI, it is possible to load a default set of ports or interfaces to be probed and driven, and it is possible to manually customize the connections. If a signal is marked in the HDL with KEEP or DONT_TOUCH, this signal should not be added to the probed port list, otherwise it generates an error later during implementation.

Experiments with GT Debugger

Xilinx recommends that the links under test be in a stable condition before running the experiments in the application note for the first time. Possibly, no data errors should occur at least while you are developing confidence with this debugging method (Figure 11).

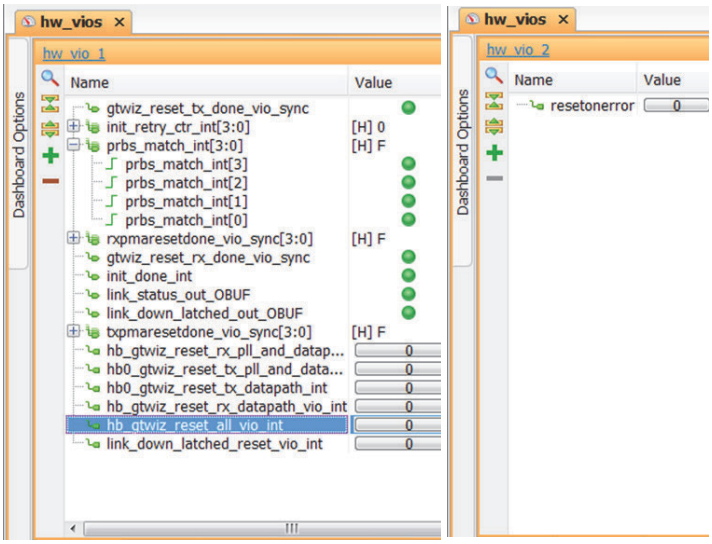

Figure 11: **Stable Condition without Data Errors and Reset on Error Function Gated**

In the file `configuration.tcl`, a dictionary called "links" describes the connections between the transceivers and should match the cable connections. The same transceivers can at the same time send data to themselves (through NE PMA loopback) and to any other FPGA receiver (through the cable connection). For example, in Figure 12 link 1 represents a cable connection existing from FE transmitter cX0Y8 to NE receiver cX0Y8, and link 5 is a near-end loopback on FE FPGA cX0Y8. When the NE PMA LOOPBACK is active, the signal is still present on the output pin so one transmitter can indeed transmit the signal to two receivers.

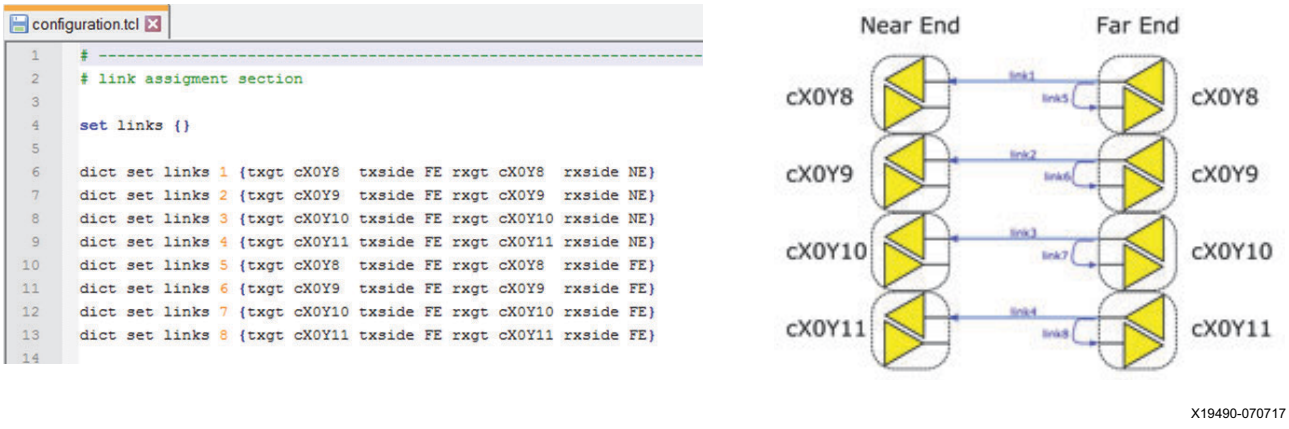

Figure 12: **Example in `configuration.tcl` and Relative Cable Connection between NE FPGA and FE FPGA**

Note: The FE FPGA transceivers in Figure 12 are all set to PMA Loopback mode.

Initial HW Manager Tcl Shell Setup

Unzip the reference design in a dedicated directory (i.e., `C:/XAPP1322`). Open the file `TopDemo.tcl` in a text editor (Figure 13). The path to the GT Debugger directory should be assigned to the variable **NEpath** or **FEpath** depending on the target PCB. The path to the XAPP1322 procedures should be assigned to the variable **demopath**.

- Copy and paste from line 15 to line 21 in the NE Tcl shell.
- Copy and paste from line 29 to line 35 in the FE Tcl shell.

```
10  # ------------------------------------------------------------------------------
11  # NE side (UC1287) procedures sourcing
12  #
13  # please copy and paste the following commands in the NE side TCL shell
14  # from here:
15  set NEpath "C:/XAP1322/UC1287"
16  set demopath "C:/XAP1322"
17  cd $NEpath
18  source "$demopath/configuration.tcl"
19  source "$demopath/tuningproc23.tcl"
20  source ./insert_gt_dbg/insert_gt_dbg_hwproc.tcl
21  source ./insert_gt_dbg/igd_eyescan.tcl
22  # to here
23
24  # ------------------------------------------------------------------------------
25  # FE side (UC1250) procedures sourcing
26  #
27  # please copy and paste the following commands in the FE side TCL shell
28  # from here:
29  set FEpath "C:/XAPP1322/UC1250"
30  set demopath "C:/XAP1322"
31  cd $FEpath
32  source "$demopath/configuration.tcl"
33  source "$demopath/tuningproc23.tcl"
34  source ./insert_gt_dbg/insert_gt_dbg_hwproc.tcl
35  source ./insert_gt_dbg/igd_eyescan.tcl
36  # to here
37
38  # ------------------------------------------------------------------------------
```

X19491-070717

Figure 13: **Screen Capture of `TopDemo.tcl` File**

Experiment 1 - Explore the GT Configuration

For each experiment, copy and paste the experiment command lines in the target Tcl shell. The first experiment allows you to download and save the actual DRP configuration. This simple operation is fundamental to knowing the real-time setup of each transceiver. The Tcl script follows this sequence of operations:

1. Complete DRP space download (DRP read operations)
2. Extract the attribute values from the DRP space
3. Reorder the attributes by name
4. Save the attributes in a tabular format

The actual transceiver attributes can differ from what is stated in the HDL. The DRP space can be overwritten (for example, during the startup sequence or due to real-time protocol replacement) and modified from the original setup. The attribute file is stored in the GT Debug design directory (Figure 14).

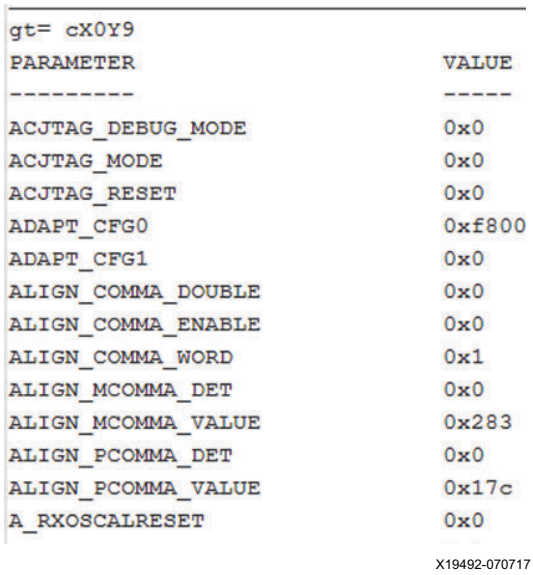

Figure 14: **Output of DRP Dump Operation in Experiment 1**

This script works fine with GTH and GTY transceivers in UltraScale devices but might require an adjustment for other devices, depending on the DRP map.

Experiment 2 - Probe the Equalizer Adaptation

Xilinx FPGA families allow a great design simplification—the RX adaptive algorithms tune the equalizers automatically and in real time. This circuit activity is transparent to the user and in most cases, the user does not even know they exist. However, an accurate link tuning always requires the user to probe the equalizer status. The procedure in this application note recognizes whether an LPM or DFE equalizer is employed and also outputs their status while comparing with the range (Figure 15).

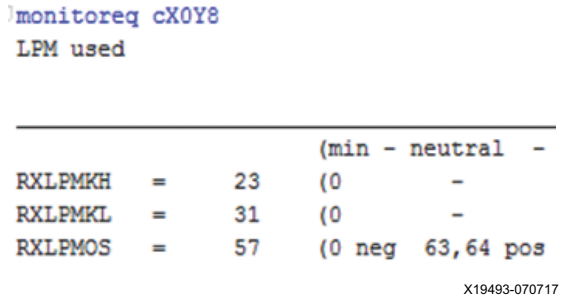

Figure 15: **Output from Equalizer Monitor Procedure**

This script works fine with GTH and GTY transceivers in UltraScale devices but might require an adjustment for other devices, depending on how the DFE and LPM are mapped in the DMONITOR.

Experiment 3 - Initiate a Socket Connection

This is a preparatory experiment in which a socket is created and closed. Currently, no useful operation is executed in this experiment.

The socket opens a TCP network connection and allows a client and server to exchange information. After the socket link is initiated, the Vivado tools become a powerful environment allowing an efficient information exchange between Tcl shells. In this application note, the socket is the method used to send and receive information between two Vivado tools Tcl shells. There are multiple examples of possible socket configurations available on the Internet. A complete socket description is beyond the scope of this application note.

The example design contains two Vivado hardware managers driving one FPGA each via JTAG, and a serial data link is present between the two boards. The link is established by configuring a server and a client. The server is created by sourcing the `FE_server.tcl` script at the far-end Tcl shell. The `FE_server.tcl` scripts contains the following instruction:

```
set s [socket -server accept 2828]
vwait x
```

This command opens a network socket and returns a channel identifier. In this case a server side socket is opened on port 2828, and the `accept` command is run. The procedure `accept` contains a fileevent statement that allows it to process the incoming data whenever something is present in the channel. The `vwait` command enters the Tcl event loop to process events until some event handler sets the value of variable `x`, blocking the application if no events are ready.

Thus, after the FE side socket server is started, the relative Tcl shell remains blocked until someone closes the channel. The client is activated by sourcing `NE_client.tcl` at the near-end Tcl shell. After the channel is active, because the FE hardware manager shell is blocked, all commands must be routed to the NE hardware manager Tcl shell only.

In the next set of experiments, the procedure `sendcomm` executed at the NE side can send a command line through the socket, have it executed at the FE side, and receive feedback from the FE. The experiment completes by closing the channel. Always follow the steps in the order presented here:

- On the NE side (client) type the following commands:

  ```
  close $s
  ```

  At this point, the FE Tcl shell is released, but you should still close the channel at the FE side.

- Type the following at the FE (server) side:

  ```
  close $s
  ```

The system is now ready to initiate a new socket channel.

In summary, the socket allows you to send any instruction from the NE (client) to the FE (server) and have this instruction executed at the FE side. This feature is particularly important, for example, when a variable on the FE side should be known by the NE side.

The socket is extensively used in this application note to join two hardware managers and drive multiple FPGAs at the same time. This method allows you to share information between independent hardware managers managed by different laptops if the laptops are connected to the same network.

Experiment 4 - Vector Eye Scan with FE TX Configuration

In this experiment, a receiver at the NE side and a transmitter at the FE side are used. The NE side:

- Sends the desired configuration (amplitude, post-emphasis, and pre-emphasis) to a transmitter on the FE side via the active socket
- Starts a vector eye scan

The vector eye scan differs from the brute force scan in integrated bit error ratio test (IBERT), where the offset sampler is made moving over the whole eye space. In the vector eye scan, the measurement points are laid on a vector starting from the center of the eye. The eye is explored by $N = 2^M$ vectors with $M \in N$ (Figure 16).

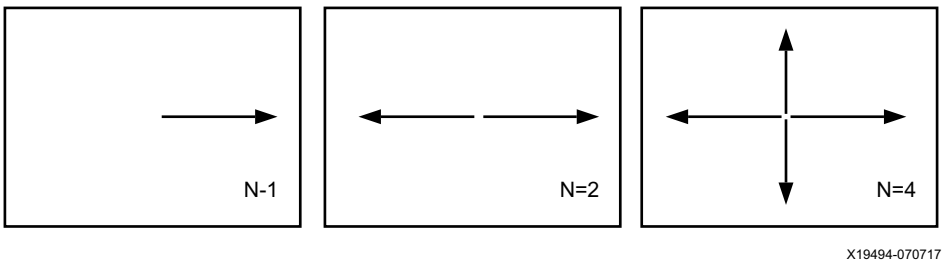

Figure 16: **Positions of Exploring Vectors in the Vector Eye Scan Procedure**

The algorithm used is binary search, also known as half-interval or logarithmic search. It is assumed that the bit error rate (BER) grows if the distance between the offset sampler and data sampler increases.

In Figure 17, the circle represents a measurement with no bit errors, and a cross is a measurement with bit errors. The measurement point is initially chosen in the middle of the vector. If there are no bit errors (measurement 1), the measurement point is moved farther from the center in the middle of the remaining vector segment (measurement 2).

Because no errors are found in measurement 2, the next measurement point is chosen in the middle of the right-most remaining segment (measurement 3). If there are bit errors, the next measurement point is moved closer to the center, again in the middle of the remaining segment (measurement 4).

In a few steps (O(log n) comparisons, on average) the center-most failing point is found (measurement 4). This is a great advantage for a slow Tcl-based eye scan.

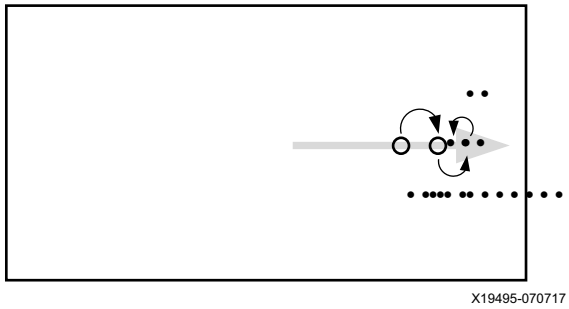

Figure 17: **First Error Search Algorithm in the Vector Eye Diagram**

The tool calculates the actual BER and marks the point with a number corresponding to the negative BER exponent. For example if a BER = 10^{-8} is measured, the spot is marked with 8 (Figure 18). The achievable BER depends on the prescale setting, and this in turn affects the measurement time.

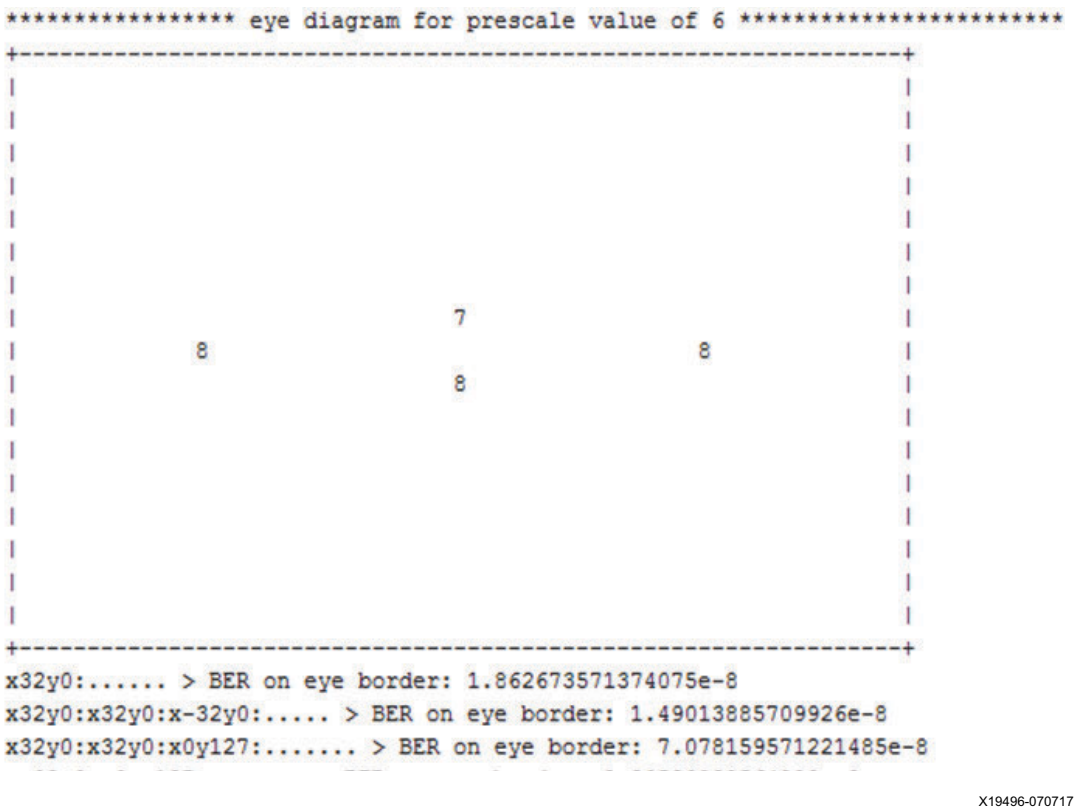

Figure 18: **Vector Eye Scan with N = 4**

Experiment 5 - BF_Scan

The BF_scan procedure repeats the single vector eye scan over a set of transmitter configurations. When calling the procedure, you should declare the ranges of TXDIFFCTRL, TXPOSTCURSOR, and TXPRECURSOR, the incremental step, and the eye scan prescale.

This procedure needs the socket channel to connect the NE and FE boards. A receiver is used on the NE board, and a transmitter is used on the FE board. The NE board is where the BF_Scan should be launched. The NE Tcl shell:

- Sends the new configuration to the FE transmitter
- Runs a vector eye scan on the NE receiver
- Records the eye height and amplitude in the variable margin_d

The dictionary margin_d keeps a memory of all measurements ordered by prescale, link name, and transmitter setup. It is possible to skip the repetition of the measurement if the same setup was already examined in the past.

Experiment 6 - Progressive Tuning

In the case of a 10 Gb/s data rate link, each vector eye scan with N = 4 requires about 5 seconds. Exploring all transmitter possibilities with a GTH transceiver in an UltraScale+ device requires 16 x 32 x 20 = 10,240 scans. That is equivalent to 14 hours per each channel if we employ a brute force scan. A smarter strategy is needed to avoid useless measurements. The goal of this experiment is not to give the smartest strategy, but rather to show how Tcl scripting can make a tedious tuning process quicker.

Progressive tuning explores the link behavior with a "zooming" strategy. It runs an initial, rough analysis on the full variables extension, then it progressively zooms on the best position found and increases the sampling density at the same time.

The search strategy is set in the `configuration.tcl` file, in the strategy_d dictionary. The search for the best configuration goes by steps. In each step, the TXDIFFCTRL, TXPOSTCURSOR, and TXPRECURSOR variation range and number of increments is defined.

In the example shown in Figure 19, the search is divided into three steps. The TXDIFFCTRL is initially explored on 100% of the range, in the second run on 40% of the range, and finally on 20% of the range in the last run. The step size is reduced progressively from 5 to 3, and finally to 1. At every new step, the focus is moved on the best position from the previous scan. The number of steps, number of increments, and range are configurable. The script recognizes when a setup has already been measured, and if it has, the measurement is skipped.

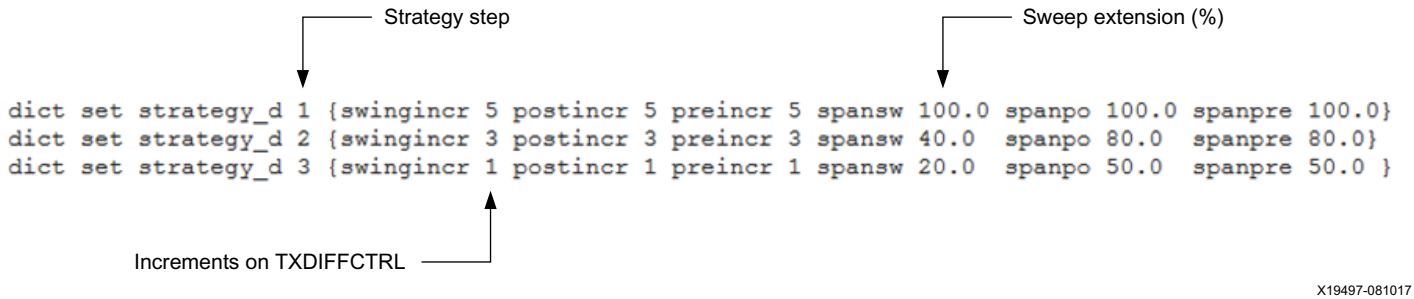

Figure 19: **Strategy Dictionary in `configuration.tcl`**

Experiment 7 - Crosstalk Analysis and Reduction

This experiment requires the use of a channel with existing crosstalk between two links (link 1 and link 2) as shown in Figure 20.

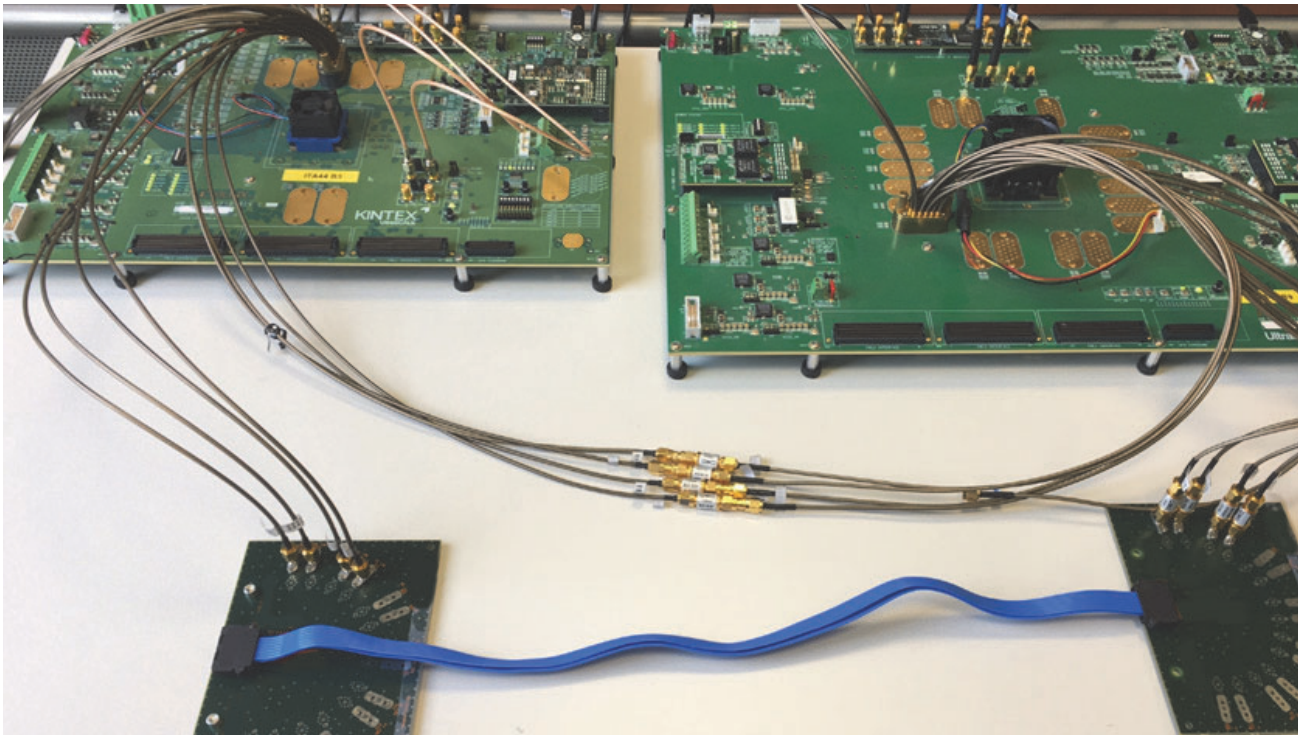

Figure 20: **Crosstalk between Links**

Note: In Figure 20, the blue cable has been used to allow some coupling between links 1 and 2. Most of the coupling is actually due to the connectors.

With GT Debugger it is possible to estimate crosstalk by comparing the signal eye with or without the aggressor presence. This crosstalk reduction script identifies the main aggressors and helps to modify the aggressor signal energy. However, crosstalk is not only a matter of signal amplitude but also and mainly a matter of signal spectrum. Xilinx recommends completing the crosstalk analysis with a signal and channel frequency domain measurement as a preferred way to mitigate the crosstalk noise.

The first procedure xtalk01 runs a tuning of all links in the list when all aggressors are switched off. The second procedure xtalk02 compares the ideal eyes (with no crosstalk) and real eyes (with crosstalk) and represents the result in two tables.

The coupling between channel 1 and channel 2 reduces the eye width and eye height. This is reported by the jitter influence and amplitude influence tables (Figure 21).

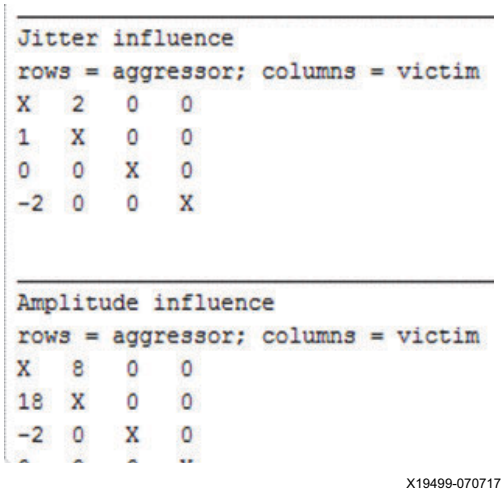

Figure 21: **Jitter Influence and Amplitude Influence Tables**

Note: The jitter influence and amplitude influence tables in Figure 21 show how much the presence of an active aggressor reduces the jitter margin and the eye amplitude. A small negative number should be disregarded and is due to the fact that ideal (no aggressor present) and real (aggressor active) measurements happened at different times.

From this crosstalk measurement, the highest crosstalk is measured between channels 1 and 2. In particular, when the aggressor is channel 1, the victim channel 2 eye width is reduced by two steps, and the height is reduced by 8 steps. When the aggressor is channel 2, the victim channel 1 eye width is reduced by about one step, and the height is reduced by 18 steps. Each link transmitter and eye aperture at the receiver side can be recalled by entering the commands shown in Figure 22.

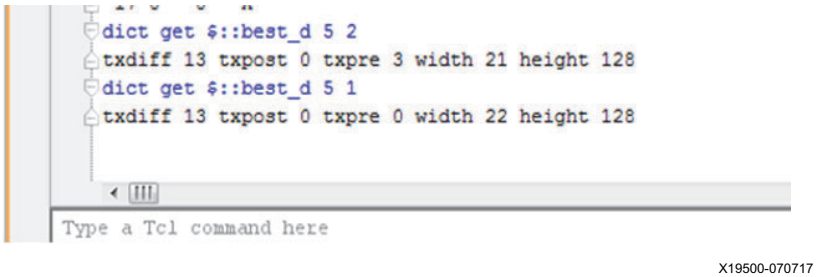

Figure 22: **Recalling the Status for Links 2 and 1**

You can check if an alternative setup with the same or similar eye width but less signal energy is possible. For example, you can check all configurations that have the same eye width equal to 21 or 22 steps, and select one with less energy (Figure 23). The procedure `tree2olist` creates an ordered by height list of all measurements having the same prescale, link, and width:

```
showlist [tree2olist $margin_d 5 1 22]
```

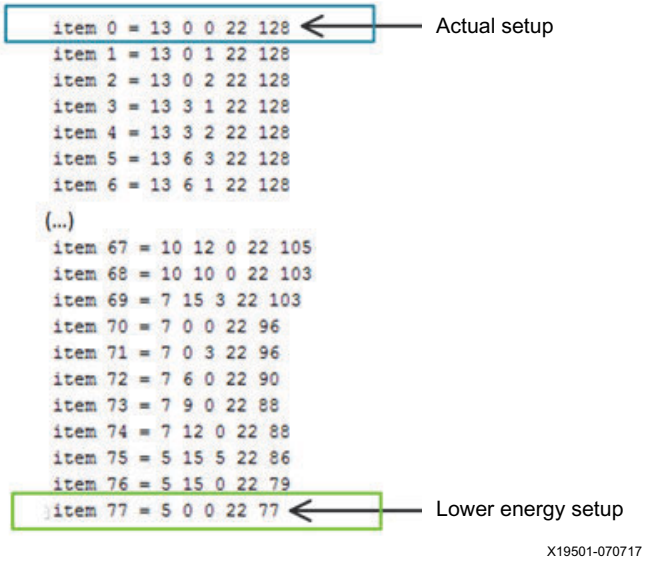

Figure 23: **Comparison of Two Configurations with Same Eye Width but Different Signal Energy**

From Figure 23, the link 1 original configuration appears at position 0: TXDIFFCTRL = 13; TXPOSTCURSOR = 0; TXPRECURSOR = 0. The eye width is 22 and eye height is 128. List item 77 shows that configuration TXDIFFCTRL = 5; TXPOSTCURSOR = 0; TXPRECURSOR = 0 still has the same eye width and acceptable eye height of 77. The same approach can be repeated to reduce the energy on link 2. After having replaced the link 1 and link 2 setup in the best_d dictionary, you can run the crosstalk analysis again (Figure 24).

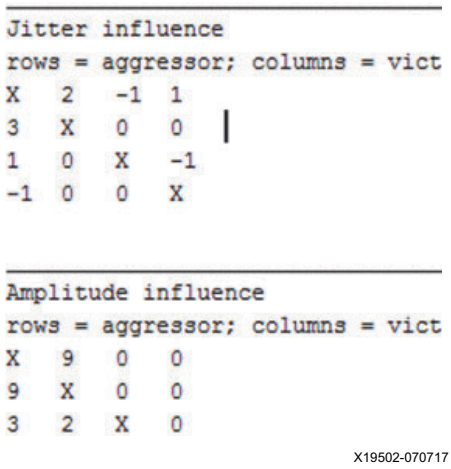

Figure 24: **New Aggressor Tables after Link 1 and Link 2 Energy Optimization**

With the new setup for link 1 and 2, the crosstalk was greatly reduced on victim 1 but remained almost the same on victim 2. You might wonder if this is really a success, because you started from a lower height margin. However, this simple example with just one aggressor and one victim should be extended to a general case with many aggressors. When many aggressors are present, reducing their strength provides a significant benefit on the victim signal. In general, avoiding wastage of signal power helps to keep the overall crosstalk contribution low and mitigates reflections caused by transmission line impedance mismatches.

Reference Design

Download the reference design files for this application note from the Xilinx website.

Table 3 shows the reference design matrix.

Table 3: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Xilinx
Target devices	7 series, UltraScale, and UltraScale+ devices
Source code provided	Yes
Source code format	Tcl
Design uses code and IP from existing Xilinx application note and reference designs or third party	No
Simulation	
Functional simulation performed	No
Timing simulation performed	No
Test bench used for functional and timing simulations	No
Test bench format	N/A
Simulator software/version used	N/A
SPICE/IBIS simulations	N/A
Implementation	
Synthesis software tools/versions used	Vivado tools 2017.2
Implementation software tools/versions used	Vivado tools 2017.2
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	Yes
Hardware platform used for verification	UC1287 and UC1250 characterization boards

Conclusion

This application note provides an easy way to use the powerful GT Debugger and gives some hints about what the GT Debugger can do for channel analysis, tuning, power, and noise optimization. The tool also shows how to share information between multiple Vivado hardware managers, which is a precious resource if the optimized link transmitter and receiver do not belong to the same FPGA.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado® IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

References

1. *Automatic Insertion of Debug Logic for Transceivers in Synthesis DCP* (XAPP1295).
2. UltraScale Architecture Transceivers User Guides:
    - *UltraScale Architecture GTH Transceivers User Guide* (UG576)
    - *UltraScale Architecture GTH Transceivers User Guide* (UG578)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/07/2017	1.0	Initial Xilinx release.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx’s Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (“SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (“SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

© Copyright 2017 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.