Summary

This application note provides a method to insert debug logic into an existing design without the need to change the HDL code. The debug logic can be accessed in hardware manager with generated Tcl procedures from the Tcl console or virtual input/output (VIO) in the Vivado® Design Suite. The main focus in this document is on debugging transceiver designs, but the provided method can also be used on the flip-flops of a design without transceivers.

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

Introduction

To debug transceiver designs it is often necessary to observe or control ports of the transceiver or its supporting logic that are not easily accessible in the existing application design. Either the ports are not used at all or the logic is not easily changed, for example, an encrypted netlist of an IP core. Also, the transceiver instance is usually at a relatively low level of the hierarchy, and a change could require signals to cross this hierarchy. Some IP cores have the option to bring out a selection of transceiver ports for debug, but they might not be the right ones. All together, it would require a change of the existing design setup and HDL code, and this can be error prone and time consuming.

The Tcl script provided with this application note aims to make this process easier. Its starting point is a design checkpoint (DCP) written out after synthesis of the full application design for the device. No further design files are necessary, so the design setup and HDL code remain untouched. Every transceiver fabric port, except clock ports, can be chosen to be accessible in hardware manager through the Vivado tools Tcl console or VIO. The same control can be added for every flip-flop data port in the design.

To easily access the inserted debug logic through the Tcl console in hardware manager, a Tcl file is generated that provides Tcl procedures for every selected port. This makes it easy to set up more complex debug sequences. As an example, an eye scan script is already included.
In summary, the main features of the script are:

- Access to any transceiver fabric port and flip-flop data port for observation and control
- No change of HDL code necessary
- Access through Tcl console or VIO in HW manager
- Tcl procedures for easy access in Tcl console

---

**Debug Logic Features**

*Figure 1* shows the general debug structure that is added to the design for selected signals.

![Debug Logic Insertion](image)

**Output Port Debug Logic**

When selecting a transceiver or flip-flop output port for debug, the port value can be read back. A multiplexer is also inserted that can be controlled to either feed the port value or an externally set value to the application logic. This structure can be seen in the top left of *Figure 1* and in more detail in *Figure 2*. The registers of the debug logic are clocked with the debug clock. The application clock is not used for synchronization.
Input Port Debug Logic

When selecting a transceiver or flip-flop input port for debug, a multiplexer is inserted that can be controlled to either feed the application logic value or an externally set value to the port. The port value can be read back. This structure can be seen in the bottom left of Figure 1 and in more detail in Figure 3. The registers of the debug logic are clocked with the debug clock. The application clock is not used for synchronization.
Debug Logic Access

There are two possible ways to access the debug logic—through AXI general-purpose I/O (GPIO) registers or VIO. When using AXI GPIO registers, the debug logic can be controlled through the Tcl console in the Vivado tools. A JTAG2AXI and AXI interconnect is used to access the GPIO registers. The width and number of the GPIO registers that are used depends on the number of selected ports. If a port is a bus, its bits might be spread over several GPIO registers. To avoid reading or writing time split values from or to the port, some global control bits are introduced. Setting $ouupdate = 0$ disables changes to the registers that are passed to the multiplexer. When the registers for the complete bus are written, setting $ouupdate = 1$ passes the complete change to the multiplexer in one go. Similarly, on the read side $iupdate = 0$ allows the read of a complete bus value.

When using VIO, the debug logic can be controlled later directly though the standard GUI for the VIO. $ouupdate$ and $iupdate$ have no effect here.

DRP Port Debug Logic

The DRP port of a transceiver instance can only be selected as a full interface. If selected, an AXI2DRP bridge is implemented for this port. If an existing application connection to the port is detected, a dynamic reconfiguration port (DRP) arbiter (DRPMUX) is also added. This arbiter synchronizes the access from the AXI2DRP bridge to the DRP clock used in the application for this port. If the DRP access from the application interferes with the DRP access of the debug logic, a control bit ($drpappdis$) can be used to stop the application access. Each DRPMUX has its own $drpappdis$ bit that can be controlled through the provided Tcl procedures.
ILA Cores within Debug Logic

Additionally, it is possible to add integrated logic analyzer (ILA) cores. Each selected DRP interface can have its own ILA core that is clocked with the DRP clock on that port. All other selected signals can be added to one ILA core that is clocked with the debug clock for the inserted logic.

Clock Changes

For certain transceiver clock inputs it is possible to connect them to the debug clock if they are unconnected in the application. The following is a list of clock inputs for which this is currently supported:

- DMONITORCLK
- CLKRSVD[n]
- CPLLLOCKDETCLK
- PLL0LOCKDETCLK
- PLL1LOCKDETCLK
- QPLLLOCKDETCLK
- QPLL0LOCKDETCLK
- QPLL1LOCKDETCLK
- TXLATCLK
- RXLATCLK
- SIGVALIDCLK
- TXPHDLYTSTCLK

Other clock connections cannot be changed.

Requirements and Limitations

There are some requirements for the debug clock described earlier. This clock must already be available in the application design and it must be free-running. Otherwise, the debug logic is not detected correctly in the hardware manager. If the frequency of this clock is outside the DRP limits of the used transceiver, the clock is divided by 2, 4, or 8, depending on which setting brings the frequency first into this limit. If there are already debug cores in the application, the clock for the existing debug hub has to be used, and it will be preselected.

There is one restriction for port selection. If the net connected to a selected port is connected to a debug core in the application (implying a DONT_TOUCH), this connection cannot be altered and the insertion might fail.

The number of ports that can be selected for GPIO or DRP control is restricted by the implemented AXI interconnect, which provides a maximum of 64 master interfaces that can be
Debug Logic Insertion

This section describes the necessary steps for the debug logic insertion:

- **Preparation**: Get the files and software ready.
- **Design Analysis**: Run analysis on the design and prepare changes in the generated `insert_gt_dbg.do` file.
- **Design Modification**: Run the modification.
- **Design Implementation**: Implement the changed design.

**Preparation**

The first step for the debug insertion is to write out a DCP of the full application design after synthesis. Then, do the following:

1. Create a new directory (`<dbgdir>`) that is not within an existing Vivado tools project structure and copy the DCP file into that directory.
2. Unpack the provided `insert_gt_dbg_<version>.7z` file in this directory.

You should now have the following two entries in `<dbgdir>`:

- The DCP file
- A directory named `insert_gt_dbg`

3. Open the Vivado tools and make sure you are in the `<dbgdir>` directory in the Tcl console (`pwd`).
4. Execute the following command in the Tcl console:

```
source ./insert_gt_dbg/insert_gt_dbg.tcl
```

You have now added the `insert_gt_dbg` command (Figure 4).
Design Analysis

To analyze the application DCP file, execute the following command in the Tcl console:

```tcl
insert_gt_dbg analyze <DCP file>
```

This opens the DCP file and extracts the necessary information to build the debug design. No change is done to the DCP (Figure 5). (If you need to come back to this step at a later stage, close all checkpoints/projects in the Vivado tools session beforehand.)
When finished, you can see a list of all transceiver instances in the log and the following line (Figure 6):

To run the "modify" step, please edit "insert_gt_dbg.do" now.
To run the modify step, open the file `<dbgdir>/insert_gt_dbg/insert_gt_dbg.do` in the text editor of your choice and make the necessary edits. You only need to change entries in the first column of a line. Leave everything else untouched.

The `.do` file is separated into several sections.

[section clock]

Here all clocks present in the application design are listed with their buffer instance, frequency, and clock net (Figure 7). If there is a debug hub already in the design, its clock is preselected with a `Y` in the first column. Otherwise there will be an `N` in the first column.

Select only one clock from the list. It must be free running.
This section allows you to select the data depth of the ILA core when choosing to add a signal to the signal ILA core (Figure 8). Change the first column from \( n \) to \( y \) for the selection. If two \( y \) are found, the last selection is taken. If nothing is selected, 1024 is used. The available resources in the design limit the ILA depth that you can select.

This section allows you to select the data depth of the ILA core when choosing to add an ILA core for a DRP port (Figure 9). Change the first column from \( n \) to \( y \) for the selection. The last selection is taken. If nothing is selected, 65536 is used. The available resources in the application design might determine what you can select.
This section is divided into subsections for every transceiver instance where you can do the port selections for signals that you want instrumented for debug (Figure 10).
### GT Instances

**Figure 10:**

**[section GT: <GT index> <instance name>]**

- **<GT index>:** This is the index used by the tool consisting of gt<integer>.
- **<instance name>:** This gives the instance name used in the application.

The first line after the `[section ...]` shows spaces as the first characters on the line:

```
<GT index> <GT type> <GT block> <GT location> Instance: <instance name>
```

### selections for GPIO access (with and without ILA)

```
<table>
<thead>
<tr>
<th>dbg</th>
<th>Index Type</th>
<th>Block</th>
<th>Port</th>
<th>DesignConnection</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
</tbody>
</table>
```

### selections for VIO access (with and without ILA)

```
<table>
<thead>
<tr>
<th>dbg</th>
<th>Index Type</th>
<th>Block</th>
<th>Port</th>
<th>DesignConnection</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
<tr>
<td>gtl</td>
<td>OTPGP2</td>
<td>CHE203</td>
<td>u_iert_oeve/inst/INNO[0]</td>
<td>u.ch/iov[0]</td>
</tr>
</tbody>
</table>
```
<GT type>: For example GTPE2 or GTYE4.

<GT block>: Can be CHANNEL or COMMON.

<GT location>: Gives the device location for the instance, for example X0Y3. If not available in the application design, it is set to undef.

Nothing should be done to this line. It is there for information purposes.

The next line allows you to select the DRP port.

n       <GT index> <GT type> <GT block> DRP

To do so change n to y. To add an additional ILA port for this DRP port, set it to yi.

After this point, all transceiver ports for the current transceiver instance are listed. The list is sorted alphabetically after the port name with the inputs first and then the outputs.

n       <GT index> <GT type> <GT block> <GT port name> <application net name>

<Application net name>: Shows the name of the net connected to the port in the application design. If empty, the port is not connected in the application design.

To select the port to be controlled through the GPIO, change n to g. To add the port to the signal ILA, set it to gi.

To select the port to be controlled through the VIO, change n to v. To add the port to the signal ILA, set it to vi.

To select a clock port to be connected with the debug clock, change n to c. For the DMONITOR clock input only, if it is necessary to connect the DMONITOR clock input with the clock provided from the transceiver, change n to ci (DMONITOROUT should also be selected from bit 0 upwards. A separate ILA is added for it, running with the internal clock).

[Section ILA hierarchies]

Separate to the above debug logic, this section allows insertion of ILA cores depending on the application design hierarchy (Figure 11). If a hierarchy is selected all flip-flop data ports in that hierarchy are added to an ILA, except for flip-flops that have the ASYNC_REG property set. Inputs and outputs can be selected separately.

The top of the section lists the clocks used in the design:

n       <pin driving the clock> <clock net>

If you want to have all resulting signals in one ILA, change the n to y in the line of the clock you want to use for the ILA.

If no clock is selected here, the default behavior is to check for every flip-flop what clock it is related to. For every clock that is used for the selected hierarchies, a separate ILA is created to allow synchronous data capture in the ILA.
The hierarchy is listed after the clocks. Each hierarchy has two lines, one for flip-flop inputs and one for flip-flop outputs. The varying indentation between the lines shows the level of hierarchy. An example is shown here:

```
  nffi /gt_exdes_top
  nffo /gt_exdes_top
  nffi /gt_exdes_top/example_wrapper_inst
  nffo /gt_exdes_top/example_wrapper_inst
  nffi /gt_exdes_top/example_wrapper_inst/common_wrapper_inst
  nffo /gt_exdes_top/example_wrapper_inst/common_wrapper_inst
```

To select a hierarchy, change `nffi` to `yffi` or `nffo` to `yffo`. Do not remove the leading space characters in the line.

```
[section FF instances]
```
n  <FF index> <FF type> D <FF instance name> <FF input net> <FF clock net> <driving clock port>

n  <FF index> <FF type> Q <FF instance name> <FF output net> <FF clock net> <driving clock port>

The same selections as for GT ports can be done here (g, gi, v, vi).

Design Modification

There are two possibilities for the modify command. If still using the same Vivado tools session in which the analysis was run, execute the following command in the Tcl console (Figure 13):

```
insert_gt_dbg modify
```

If a new Vivado tools session was started as mentioned in Preparation, page 6, and the analysis was already run before in <dbg_dir>, execute the following command (Figure 14):

```
insert_gt_dbg modify <DCP file>
```
First some IP cores can be generated for the provided Verilog modules necessary to implement functionality. After this, a block design is built containing the debug logic. This block design is then inserted into the application DCP and connected. The resulting design is saved in a DCP file in the `<dbgdir>/insert_gt_dbg` directory.

Several windows might open and close in the process. Leave them untouched until the following message appears in the log (Figure 15):

```
Synthesized design contains debug logic now. Implement and generate bitfile as usual.
```

In the Hardware Manager source the script `<project_directory>/..insert_gt_dbg/insert_gt_dbg_hwproc.tcl`. This provides procedures to access selected DRPs and signals.

**Figure 14:** Log of Modify Step Start when Analysis Was Not the Previous Step in Tcl Console

**Figure 15:** Log of Modify Step End

### Design Implementation

To implement the modified DCP, the command provides the option to start a default implementation with the following:
This includes the steps opt_design, place_design, route_design, report_timing_summary -verbose, write_bitstream, and write_debug_probes without any specific further options. After each step a checkpoint is saved. Output files can be found in the directory <dbgdir>/insert_gt_dbg (igd_impl.timrptsum, igd_dbg.bit, igd_dbg.ltx).

If more options are necessary for the implementation of the design, these steps would have to be done manually.

---

**Usage in Hardware Manager**

**Preparation**

Open the Vivado tools, start the hardware manager, and program the device with the generated bitstream and probe file. If only VIO was selected to control the signals, you don’t need anything else. For DRP and GPIO control, make sure you are in the <dbgdir> directory in the Tcl console.

Source the generated file insert_gt_dbg_hwproc.tcl as follows:

```
source ./insert_gt_dbg/insert_gt_dbg_hwproc.tcl
```

All provided Tcl procedures have the prefix igd_. All global variables used have the prefix _igd_.

**List of Tcl Procedures**

- `igd_axi_res`
- `igd_axi_rd <base address> <address>`
- `igd_axi_wr <base address> <address> <data>`
- `igd_axi_wrm <base address> <address> <data> <mask>`
- `igd_axi_rmw <base address> <address> <data> <mask>`
- `igd_data_modify <original data> <new data> <mask> <width>`
- `igd_data_slice <data> <width> <from> <to>`
- `igd_data_concat {<data> <width>} [{<data> <width>} ...]`
- `igd_gpio_ctr_wr <GPIO channel> <data> [<mask>]`
- `igd_gpio ctr_drpmux reset rst`
- `igd_gpio_ctr_drpmux reset set yes`
- `igd_gpio_ctr_oupdate rst`
- `igd_gpio_ctr_oupdate set`
• igd_gpio_ctr_iupdate_rst
• igd_gpio_ctr_iupdate_set
• igd_drp_<GT identifier>_rd <DRP address>
• igd_drp_<GT identifier>_wr <DRP address> <data>
• igd_drp_<GT identifier>_rmw <DRP address> <data> <mask>
• igd_gpio_ctr_<GT identifier>_drpappdis_rst
• igd_gpio_ctr_<GT identifier>_drpappdis_set
• igd_drp_rd <GT identifier> <attribute name>
• igd_drp_wr <GT identifier> <attribute name> <value>
• igd_gpio_<GT/FF identifier>_<signal name>_rd_gt_d
• igd_gpio_<GT/FF identifier>_<signal name>_rd_reg_d
• igd_gpio_<GT/FF identifier>_<signal name>_wr_d <data> <mask>
• igd_gpio_<GT/FF identifier>_<signal name>_rd_m
• igd_gpio_<GT/FF identifier>_<signal name>_wr_m <data> <mask>
• igd_gpio_<GT/FF identifier>_<signal name>_rst_m
• igd_gpio_<GT/FF identifier>_<signal name>_set_m

General Tcl Procedures

igd_axi_res

This procedure allows you to reset hw_axi, should that become necessary. It does not have any further options.

igd_axi_rd <base address> <address>

This procedure reads an AXI register and returns the read data.

igd_axi_wr <base address> <address> <data>

This procedure writes an AXI register and returns the written data.

igd_axi_wrm <base address> <address> <data> <mask>

This procedure writes bits of an AXI register depending on the mask and returns the written data. It is used for GPIO access. The current output value of a GPIO register cannot be read back from hardware. The output value is saved in the Tcl dictionary _igd_gpio_wrv, which is used to generate the masked value.
**Usage in Hardware Manager**

\[
\text{igd\_axi\_rmw} \ <\text{base address}> \ <\text{address}> \ <\text{data}> \ <\text{mask}>
\]

This is the read-modify-write operation of an AXI register depending on the mask. This procedure returns the written data and is used for DRP access.

\[
\text{igd\_data\_modify} \ <\text{original data}> \ <\text{new data}> \ <\text{mask}> \ <\text{width}>
\]

This procedure modifies bits of \(<\text{original data}>\) to bits of \(<\text{new data}>\) depending on \(<\text{mask}>\). The \(<\text{width}>\) is used to get the correct length for the return value as a hexadecimal value.

\[
\text{igd\_data\_slice} \ <\text{data}> \ <\text{width}> \ <\text{from}> \ <\text{to}>
\]

This procedure cuts a bit range out of \(<\text{data}>\). The \(<\text{width}>\) is the bit width of the range, \(<\text{from}>\) is the LSB of the range, and \(<\text{to}>\) is the MSB. This procedure returns a list with the format \{<\text{slice value}> <\text{width}>\}.

\[
\text{igd\_data\_concat} \ \{<\text{data}> <\text{width}>\} \ \{<\text{data}> <\text{width}>\} \ ...
\]

This procedure combines several data values to a single value. The arguments to the procedure are lists containing the data value and the bit width of the current part of the data to connect. The first argument is placed at the MSB of the resulting value and the last at the LSB. This procedure returns a list for the combined data in the format \{<\text{value}> <\text{width}>\}.

\[
\text{igd\_gpio\_ctr\_wr} \ <\text{GPIO channel}> \ <\text{data}> \ [<\text{mask}>]
\]

This procedure writes the selected channel of the GPIO control register for the debug logic. It returns the value of that channel after the write.

\(<\text{mask}>\): If omitted, it defaults to 0xffffffff.

GPIO channel 1 bit[0]: DRPMUX reset for all implemented arbiters. This bit should not be necessary to set. Use it only when you are sure that a DRPMUX does not recover normally. This can corrupt the DRP protocol on the application side. Use with \text{igd\_gpio\_ctr\_drpmux\_reset\_rst} and \text{igd\_gpio\_ctr\_drpmux\_reset\_set}.

GPIO channel 1 bit[1]: Oupdate. When 0, writes to GPIO registers have no effect on debug logic. When 1, previous and further writes have immediate effect. Use with \text{igd\_gpio\_ctr\_oupdate\_rst} and \text{igd\_gpio\_ctr\_oupdate\_set}.

GPIO channel 1 bit[2]: Iupdate. Used as an enable for the read flip-flop. Consistent bus values can be read when set to 0. Use with \text{igd\_gpio\_ctr\_iupdate\_rst} and \text{igd\_gpio\_ctr\_iupdate\_set}. These are used in the procedures for signal access (see \text{igd\_gpio\_ctr\_iupdate\_rst} and \text{igd\_gpio\_ctr\_iupdate\_set}). It should not be necessary to use them explicitly.

Further bit usage depends on DRP arbiter usage. Every DRP arbiter gets its own \text{drpappdis} bit which, when set, disables the DRP access from the application side. Use with the specific reset and set procedure for this bit generated for the specific DRP port.
Because there are 64 bits available in the GPIO control register over the two channel register, there are a maximum of 61 DRP arbiters that the script can currently implement. This should suit most purposes.

**igd_gpio_ctr_drpmux_reset_rst**

This procedure resets the reset for the DRP arbiters and returns nothing. For more information, see *igd_gpio_ctr_wr GPIO channel* <data> [mask].

**igd_gpio_ctr_drpmux_reset_set yes**

This procedure sets the reset for the DRP arbiters and returns nothing. For more information, see *igd_gpio_ctr_wr GPIO channel* <data> [mask]. The *yes* is introduced to prevent an accidental reset when unintentionally calling the procedure.

**igd_gpio_ctr_oupdate_rst**

This procedure resets ounupdate and returns nothing. For more information, see *igd_gpio_ctr_wr GPIO channel* <data> [mask].

**igd_gpio_ctr_oupdate_set**

This procedure sets ounupdate and returns nothing. For more information, see *igd_gpio_ctr_wr GPIO channel* <data> [mask].

**igd_gpio_ctr_iupdate_rst**

This procedure resets ounupdate and returns nothing. For more information, see *igd_gpio_ctr_wr GPIO channel* <data> [mask].

**igd_gpio_ctr_iupdate_set**

This procedure sets ounupdate and returns nothing. For more information, see *igd_gpio_ctr_wr GPIO channel* <data> [mask].

**Tcl Procedures for DRP Access**

There are three procedures provided for every DRP port instrumented for read, write, and read-modify-write:

**igd_drp_<GT identifier>_rd <DRP address>**

**igd_drp_<GT identifier>_wr <DRP address> <data>**

**igd_drp_<GT identifier>_rmw <DRP address> <data> [mask]**

<GT identifier>: If the location of the transceiver instance is known during analysis it is used as an identifier here, e.g., for CHANNEL X0Y0 it is cX0Y0 (*igd_drp_cX0Y0_rd*) and for COMMON X0Y1 it is qX0Y1 (*igd_drp_qX0Y1_rd*). If the location is not known, the index used in the .do file is used (*igd_drp_gt2_rd*).
The return value for the read is the value read from hardware. The return value for writes is the actual written data.

When a DRP arbiter is used for the instance, two additional procedures are provided to enable (rst) or disable (set) application access:

\textbf{igd_gpio_ctr\_<GT identifier\>_drpappdis\_rst}

This procedure enables application access to the DRP port and returns nothing.

\textbf{igd_gpio_ctr\_<GT identifier\>_drpappdis\_set}

This procedure disables application access to the DRP port and returns nothing.

Additionally there are two general procedures to read/write transceiver attributes:

\textbf{igd_drp\_rd \<GT identifier\> \<attribute name\>}

This procedure reads the attribute value from the transceiver \<GT identifier\>. It performs the necessary igd_drp\_<GT identifier>_rd calls to read the full attribute. The address mapping is provided through Tcl files in the \<dbgdir>/insert_gt_dbg directory which are sourced through insert_gt_dbg_hwproc.tcl. This procedure returns the attribute value as a hexadecimal value. For example:

\texttt{igd_drp\_rd cX0Y0 ES\_SDATA\_MASK}

\textbf{igd_drp\_wr \<GT identifier\> \<attribute name\> \<value\>}

This procedure writes the attribute value of the transceiver \<GT identifier\>. It performs the necessary igd_drp\_<GT identifier>_rmw calls to write the full attribute. The address mapping is provided through Tcl files in the \<dbgdir>/insert_gt_dbg directory which are sourced through insert_gt_dbg_hwproc.tcl. This procedure returns the written attribute value as a hexadecimal value. For example:

\texttt{igd_drp\_wr cX0Y0 ES\_SDATA\_MASK 0xffffffffffffffff}

\section*{Tcl Procedures for Signal Access}

This section provides a set of procedures for each instrumented signal, beginning with a list of procedures that are meant to be used by the user. Other procedures are supportive of these initial procedures.

\textbf{<signal name>:} For a single-bit transceiver port, this is the port name (e.g., GTRXRESET). For a transceiver port bus or part of the bus, it is the port name plus the LSB of the selected part (e.g., for TXDATA\[31:24\] it is TXDATA\_24). For a flip-flop port it is the port name plus the register instance name (e.g., D\_GTRXRESET\_0_reg).

\textbf{igd_gpio\_<GT/FF identifier\>\_<signal name\>_rd\_gt\_d}

This procedure reads and returns the signal value from hardware. It handles \texttt{iupdate} and the tristate port of the GPIO IP automatically.
igd_gpio_<GT/FF identifier>_<signal name>_rd_reg_d

This procedure reads and returns the written GPIO register value for the signal from the Tcl dictionary _igd_gpio_wrv.

igd_gpio_<GT/FF identifier>_<signal name>_wr_d <data> <mask>

This procedure writes the GPIO register value for the signal in hardware and updates the dictionary _igd_gpio_wrv. It resets ouptdate before writing and recovers its value in the end. The procedure returns nothing.

igd_gpio_<GT/FF identifier>_<signal name>_rd_m

This procedure reads and returns the written GPIO register value for the signal MUX control from the Tcl dictionary _igd_gpio_wrv.

igd_gpio_<GT/FF identifier>_<signal name>_wr_m <data> <mask>

This procedure writes the GPIO register value for the signal MUX control and updates the dictionary _igd_gpio_wrv. It resets ouptdate before writing and recovers its value in the end. The procedure returns nothing.

igd_gpio_<GT/FF identifier>_<signal name>_rst_m

This procedure resets all bits of the GPIO register value for the signal MUX control. It returns nothing.

igd_gpio_<GT/FF identifier>_<signal name>_set_m

This procedure sets all bits of the GPIO register value for the signal MUX control. It returns nothing.

The following is a list of procedures that are used within the above procedures:

- igd_gpio_<GT/FF identifier>_<signal name>_rd_d_<GPIO block number>_<GPIO channel number>
- igd_gpio_<GT/FF identifier>_<signal name>_wr_d_<GPIO block number>_<GPIO channel number> <[list data width]> <[list mask width]>
- igd_gpio_<GT/FF identifier>_<signal name>_wr_t_<GPIO block number>_<GPIO channel number> <[list data width]> <[list mask width]>
- igd_gpio_<GT/FF identifier>_<signal name>_wr_m_<GPIO block number>_<GPIO channel number> <[list data width]> <[list mask width]>

The numbering for the GPIO block and channel depends on how the signal is mapped to them. This can change for every build. There can be several procedures for buses, depending on the mapping. It should not be necessary to call these procedures manually.
For example, to set the near-end PMA loopback for CHANNEL X0Y1, use these procedures:

- `igd_gpio_cX0Y1_LOOPBACK_0_wr_d 0x2 0x7`
- `igd_gpio_cX0Y1_LOOPBACK_0_set_m`
- `igd_gpio_ctr_oupdate_set`

**Eye Scan**

Because doing an eye scan in an existing application is one of the most desirable debug methods for transceiver channels, a separate script is provided to do an eye scan based on the debug logic and procedures discussed above. There is no additional logic in hardware used to speed up a full eye scan. Thus, two methods are used to improve scan time: vector scan and matrix scan.

The vector scan is done along vectors starting from the center of the eye. The number of vectors can be chosen as a power of 2. For example, 2 would show the horizontal eye opening, and 4 would show the horizontal and vertical eye openings. Increasing numbers would divide the eye more and show more of its details, but the opening can already be recognized with a low number of vectors.

The full length of each vector is not looked at, but the edge of the eye is searched for with a dividing algorithm. The starting point is the center, then the midpoint of the vector is looked at. If there are no errors, the midpoint of the outer half is looked at. Otherwise, the midpoint of the inner half is tested. This goes on until two adjacent points are found with one passing and one failing. If a point is already measured from another vector due to rounding of the integer coordinates, the previous result is used. Both methods together reduce the number of points that need to be tested for a reasonable eye diagram.

Two approaches can be used for the vector scan. First, a range of prescale settings can be set and for each a separate scan is done. Valid prescale settings are –1 to 31. The values 0 to 31 are the prescale settings described in the relevant transceiver user guide [Ref 1] [Ref 2] and are used with the COUNT branch of the eye scan finite state machine (FSM). This provides BER results. A setting of –1 selects the ARMED branch of the FSM and provides the quickest possible scan with just pass/fail results.

Second, the BER exponent can be set for the error rate to test for. Here, the vector is first scanned through the ARMED branch to find the eye border. Then the relevant prescale value is set for the wanted BER, and from the previously found border the actual border for the BER is searched for in small steps along the vector.

The matrix scan allows you to scan the full eye at equidistant points. The number of points used can be defined separately for the horizontal and vertical directions, starting from the center point out to the edge. Additionally, an offset can also be given in the horizontal and vertical
directions. For example, when setting the number of points to 1 in one direction, a horizontal or vertical line can be scanned, and with the offset this line can be moved.

Source the provided Tcl script `igd_eyescan.tcl` to use the eye scan.

```
source ./insert_gt_dbg/igd_eyescan.tcl
```

The main procedure to use the scan is `igd_eyescan`. When called by itself, this procedure gives the help message shown in Figure 16.

```
source ./insert_gt_dbg/igd_eyescan.tcl
```

![Figure 16: Help Message for igd_eyescan](X18510-041217)

This gives an overview of all options for the procedure. The `init` function should be called in the beginning followed by an RX reset. Then the `run` function can be called several times.

After each run, a downscaled eye plot is printed in the log as shown in Figure 17 for the vector scan and in Figure 18 for the matrix scan. The scan data is stored in two dictionaries. The `ber_d` dictionary contains the full scan data, and the `ber_plot_d` dictionary contains the scaled version.
Additional Scripts for More Complex Use Cases

```
igi_d_eyescan run cXYB pre scale_min=-1 pre scale_max=-1 max_circle_div=4 dfe=1 init_window=1
x32y0:...... > BER on eye border: 2.0
x-32y0:...... > BER on eye border: 2.0
x0y127:...... > BER on eye border: 1.0
x0y-127:...... > BER on eye border: 1.0

********************************** scaled eye diagram for pre scale value of -1 **********************************

X

Z

Z

X

igi_d_eyescan run cXYB ber=10 max_circle_div=4 dfe=1 init_window=1
x32y0:...... > eye border found in armed state.... > BER on eye border: 4.656683928435187e-10
x-32y0:...... > eye border found in armed state.... > BER on eye border: 9.167846484106776e-10
x0y127:...... > eye border found in armed state.... > BER on eye border: 1.3096923548723964e-10
x0y-127:...... > eye border found in armed state.... > BER on eye border: 1.45521372639990e-11

********************************** eye diagram for target BER of 1e-10 **********************************

.

.

9.

X.9

a
```

**Figure 17:** Vector Scan Eye Plot for `igi_d_eyescan`
With the ploteye procedure, either the full eye or the scaled eye can be printed in the Tcl console.

`ploteye <GT identifier> <selection> <prescale or BER coefficient> [scaled]`

The `<GT identifier>` is the same as for DRP procedures. The `<selection>` can be either ps or ber. When ps is used, the next parameter is `<prescale>`. When ber is used, the next parameter is the BER coefficient (e.g., 8 for $10^{-8}$). The `<prescale>` can be any value as the ones defined above for `prescale_min` and `prescale_max`. The `<scaled>` can be 1 or 0. It defaults to 1, which prints the scaled eye. The scaled plot only shows the edge for the eye of each vector (Figure 19). The full eye shows all measured points (Figure 20). The characters in the plot represent the following:

- X – fail, measured with ARMED scan (1.0)
Additional Scripts for More Complex Use Cases

- . – pass, no errors (0.0)
- 0-9 – represent measured BER with exponent $10^{-0}$ to $10^{-9}$
- a-v – represent measured BER with exponent $10^{-10}$ to $10^{-32}$
- z – anything else

**Figure 19: Scaled Eye Plot**

**Figure 20: Start and Horizontal Vectors of Full Eye Plot**
Reference Design

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

Table 1 shows the reference design matrix.

<table>
<thead>
<tr>
<th>Table 1: Reference Design Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameter</strong></td>
</tr>
<tr>
<td><strong>General</strong></td>
</tr>
<tr>
<td>Developer name</td>
</tr>
<tr>
<td>Target devices</td>
</tr>
<tr>
<td>Source code provided</td>
</tr>
<tr>
<td>Source code format</td>
</tr>
<tr>
<td>Design uses code and IP from existing Xilinx application note and reference designs or third party</td>
</tr>
<tr>
<td><strong>Simulation</strong></td>
</tr>
<tr>
<td>Functional simulation performed</td>
</tr>
<tr>
<td>Timing simulation performed</td>
</tr>
<tr>
<td>Test bench used for functional and timing simulations</td>
</tr>
<tr>
<td>Test bench format</td>
</tr>
<tr>
<td>Simulator software/version used</td>
</tr>
<tr>
<td>SPICE/IBIS simulations</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
</tr>
<tr>
<td>Synthesis software tools/versions used</td>
</tr>
<tr>
<td>Implementation software tools/versions used</td>
</tr>
<tr>
<td>Static timing analysis performed</td>
</tr>
<tr>
<td><strong>Hardware Verification</strong></td>
</tr>
<tr>
<td>Hardware verified</td>
</tr>
<tr>
<td>Hardware platform used for verification</td>
</tr>
</tbody>
</table>

Conclusion

This application note provides an easy way to insert debug logic into an existing design and to allow automated debug through Tcl scripts.
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. 7 series transceivers user guides:
   - 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476)
   - 7 Series FPGAs GTP Transceivers User Guide (UG482)

2. UltraScale architecture transceivers user guides:
   - UltraScale Architecture GTH Transceivers User Guide (UG576)
   - UltraScale Architecture GTY Transceivers User Guide (UG578)


Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>09/19/2017</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
</tbody>
</table>
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 http://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 http://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. AMBA, AMBA Designer, ARM, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, and MPCore are trademarks of ARM in the EU and other countries. All other trademarks are the property of their respective owners.