Debugging Tools Guide

Version 1.5

Revision History

Revision	Description of Change	Date
v1.0	Initial Release	2/2016
v1.1	Section added for gdb and document renamed from `OpenCPI_FPGA_Vendor_Debug_tool_Integration.pdf`	3/2017
v1.2	Updated for OpenCPI Release 1.2	8/2017
v1.3	Updated for OpenCPI Release 1.3	2/2018
v1.4	Updated for OpenCPI Release 1.4	9/2018
v1.5	Updated for OpenCPI Release 1.5 and expanded SignalTap section	4/2019

Table of Contents

1 References

This document assumes a basic understanding of the Linux command line (or "shell") environment. It requires a working knowledge of OpenCPI, `gdb`, and FPGA Vendors' tools necessary for performing on-chips debug and verification. The reference(s) in Table 1 can be used as an overview of OpenCPI and may prove useful.

Table 1: References

Title	Link
OpenCPI Overview	`Overview.pdf`
Acronyms and Definitions	`Acronyms_and_Definitions.pdf`
Getting Started	`Getting_Started.pdf`
Installation Guide	`RPM_Installation_Guide.pdf`
Xilinx's ChipScope Pro[1]	`http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/chipscope_pro_sw_cores_ug029.pdf`

[1]Full title: "ChipScope Pro Software and Cores (UG029)"

2 Debugging RCC Workers

RCC workers are built as dynamically loadable shared object files, with the .so suffix. When an application uses a worker, it will be loaded on demand, even when the executable is statically linked itself. To debug a worker, it is necessary to first start the debugger on the executable, which is either the ocpirun utility program or an ACI application. In either case the first step is to run the executable under the debugger, establishing a generic breakpoint to enter the debugger at a point after workers are loaded, but before they are actually run. Then breakpoints can be placed in the worker code itself. gdb is now provided as a prerequisite for all embedded platforms. This means that the procedure that is provided below will work on any RCC platform.

2.1 debugging using gdb command line

The initial breakpoint should be placed on the OCPI::RCC::Worker::Worker member function (an internal constructor). This breakpoint will be hit for every worker in the application, after it is loaded, but before it ever is initialized (C) or constructed (C++). Note that although this initial breakpoint is at a constructor, it is not the actual constructor of the C++ worker, and not even in its inheritance hierarchy.

To determine whether the worker about to be constructed is the worker of interest, simply examine the "name" argument at this breakpoint This is the instance name for the worker within the application. If the name indicates a worker of interest you can now establish a breakpoint in the worker, either based on a source line number, or symbols in the worker. in order to do this do the following:

- gdb ocpirun
- (gdb) b OCPI::RCC::Worker::Worker
- (gdb) run -v -d my_application_xml
- Run the following command as many times as RCC workers you have in your application minus one e.g. If your application has 3 RCC workers run 'c' 2 times.

  (gdb) c
- now that all the RCC workers have been loaded into memory we can add a breakpoint in our worker of interest.

  (gdb) b my_worker.cc:135 (e.g. for a C++ worker, by line number)

  or

  (gdb) b my_worker.c:run (e.g. for a C worker, in the run method)

  (gdb) clear OCPI::RCC::Worker::Worker

  (gdb) c

There is now a breakpoint inside the worker of interest and the original breakpoint has been deleted. The worker of interest can now be debugged from here.

2.2 gdb debugging using DDD

If the user prefers they can use a graphical debugging interface such as DDD. This tool can be installed via yum:

- yum install ddd

To run this tool with an OpenCPI application simply type ddd in a console window and use the gdb console at the bottom the window to input commands. The user will use the same commands as in the previous section (debugging using gdb command line) to debug a RCC worker.

3 FPGA Integrated Logic Analyzers

This section describes how to incorporate Xilinx's Vivado Integrated Logic Analyzer, Xilinx's ISE ChipScope PRO and Altera's SignalTap into an OpenCPI design. Below is a summary of the cases that are covered:

- Xilinx Vivado
  - Instance an ILA in any HDL asset using cores from Vivado's IP Catalog
  - Insert an ILA into any HDL asset using the "Set Up Debug" wizard
- Xilinx ISE
  - Integrate an ILA into HDL Worker using CORE Generated cores, used by ChipScope
  - Insert an ILA into HDL Assembly using the Inserter tool, used by ChipScope
- Altera
  - Integrate an Embedded Logic Analyzer into HDL Worker using MegaWizard cores, used by SignalTap II

The developer must have a working knowledge of:

- OpenCPI and how to build HDL Workers and HDL Assemblies for various HDL Targets and HDL Platforms.
- The Xilinx *Debug and Verification* tools: CORE Generator, ChipScope Pro CORE Inserter and Analyzer.
- The Altera Altera SignalTap II Logic Analyzer

3.1 Xilinx Vivado

3.1.1 Case 1: Instance a Debug ILA in an HDL Worker using cores from Vivado's IP Catalog

This case requires that the developer create a debug core with Vivado and write it to an EDIF or DCP file. This can be done in the Vivado GUI:

- Navigate to:
  `Window`→`IP Catalog`→`Debug and Verification`→`Debug`→ ⟨`Core-of-Choice`⟩
- Customize the IP
- Generate IP output products in Global mode
- Run Synthesis and Open Synthesized Design
- Once synthesis completes, enter the Tcl Console, and write the checkpoint file to be included by the worker:

  ```
  > write_checkpoint vivado_ila.dcp
  ```

  Note: you can alternatively use an EDIF netlist (`write_edif`) and stub file

  * See the *Vivado_Usage* document for more information on using Vivado IP with OpenCPI

Note: ISE debug cores (NGCs) can be used in conjunction with Chipscope for debugging *even if Vivado is the tool that OpenCPI is using* to synthesize and implement designs. Reference 3.2.1 for information on including NGC debug cores.

Integrate the debug core into the worker, generate the required files and proceed with compilation as follows:

1. Integrate the *Debug and Verification* cores into the worker's VHDL:
   - Declare and instantiate the component for the core (ILA, VIO, etc)
   - As needed, add signal declarations and assignments (TRIG(Y downto 0), DATA(Z downto 0), etc)
2. (*Only required if using an EDIF instead of DCP*): In the worker's `Makefile`, set "`SourceFiles=`" to include the stub file for the core. [1]. Absolute or relative paths are acceptable. An example is provided:

   ```
   SourceFiles=../vivado_ila/vivado_ila.vhd
   ```

3. In the worker's `Makefile`, set "Cores=" to include the EDIF or DCP file for the core. Absolute or relative paths are acceptable. An example is provided:

   ```
   Cores=../vivado_ila/vivado_ila.dcp
   ```

4. Build HDL worker for target

   Critical: some probe names may not be helpful unless the `flatten_hierarchy` option is set to "`none`" during synthesis of the asset being debugged (in this case the worker). This can be done either in the Vivado GUI or in the OpenCPI worker's `Makefile` (`export VivadoExtraOptions_synth=-flatten_hierarchy none`) as explained in *Vivado_Usage.pdf*.

5. Generate the debug probes file for use in the Logic Analyzer
   - Open the generated `XPR` file located in the worker's `target-*` directory
   - Rerun synthesis now that we are in "project mode"

     Critical: In the project's synthesis settings, make sure `flatten_hierarchy` is set to "`none`"
   - Open the synthesized design
   - In the Tcl Console, generate the *.ltx file containing the debug probe information:

     ```
     > write_debug_probes vivado_ila.ltx
     ```

     * Save this file in a *persistent* location for later use
6. Build HDL assembly for platform

The generated bitstream contains the *Debug and Verification* cores which will be recognized by the Xilinx Vivado Logic Analyzer tool. Once the bitstream has been loaded onto the target FPGA, the Analyzer tool can connect and detect the presence of the *Debug and Verification* core(s). At that point, the LTX debug probes file can be loaded.

[1] Note that this step is *not* necessary if using a DCP file instead of an EDIF netlist because a DCP file includes the EDIF netlist *and* the VHDL stub file.

3.1.2 Case 2: Insert Vivado Debug ILA into an HDL Worker

After building your core, worker, platform, config, assembly, or container, you can add a debug core using the Vivado GUI. The result will be a new netlist containing the debug core. This will replace the netlist generated by OpenCPI. Note that rebuilding or cleaning the worker (or other AV asset) at any time (with "make") will remove any debug functionality added to the Vivado project. The Vivado project files are an artifact of the "make" process, and will be overwritten each time "make" is run for that asset.

For our example, we use the complex_mixer.hdl worker built for "zynq":

1. Build the worker:

   ```
   cd ocpiassets/components/dsp_comps/complex_mixer.hdl
   make HdlTarget=zynq VivadoExtraOptions_synth="-flatten_hierarchy none"
   ```

   Note that some probe names may be unhelpful unless the `flatten_hierarchy` option is set to "none" during synthesis of the asset being debugged (in this case the worker). Reference: *Vivado_Usage.pdf*

2. Open up the worker's Vivado project:

   ```
   cd target-zynq
   source /opt/Xilinx/Vivado/2017.1/settings64.sh ; vivado complex_mixer_rv.xpr &
   ```

   Note: because OpenCPI operates in Vivado's Non-Project Mode, you will need to rerun synthesis in Project Mode using the GUI. Refer to the *Vivado_Usage* doc for more information. You may want to set the `flatten_hierarchy` option to `none` via the GUI as well.

3. In the Flow Navigator's Synthesis section, select "Set Up Debug". Choose the debug settings and nets of your choice. You can drag nets in from the Netlist hierarchy, or the Schematic view:

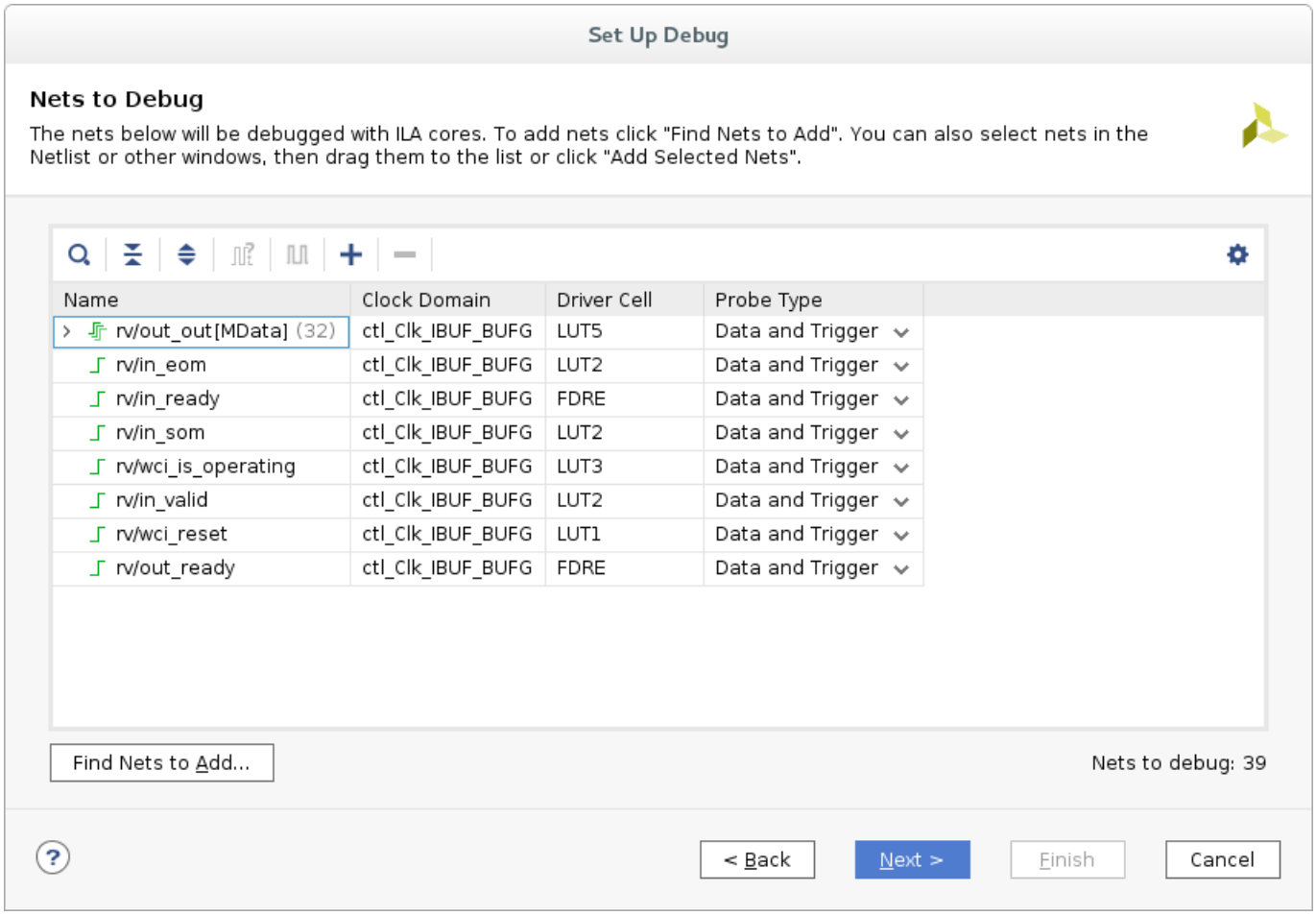

Figure 1: Xilinx Vivado 2017.1 Set Up Debug

4. Confirm that the debug cores are listed:

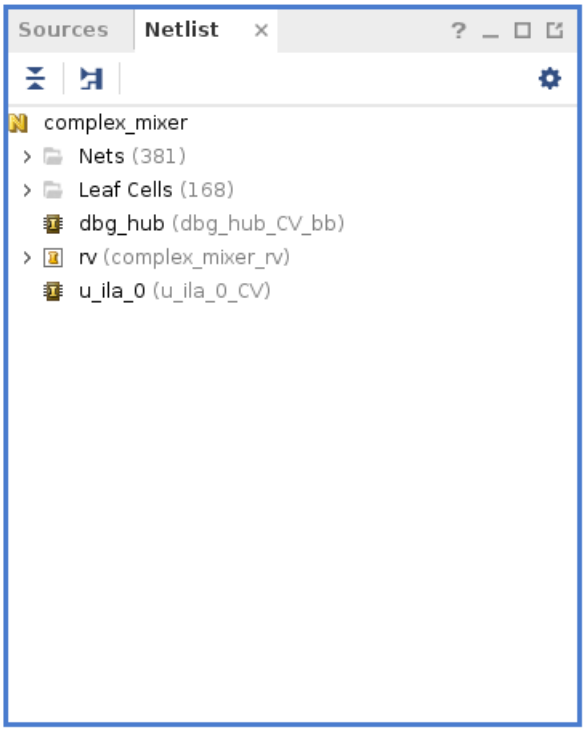

Figure 2: Xilinx Vivado 2017.1 Debug Cores Listed

5. Rerun synthesis. Note that you may once again want to set the `flatten_hierarchy` option set to "none" via the GUI. Observe the debug cores in the worker's netlist:

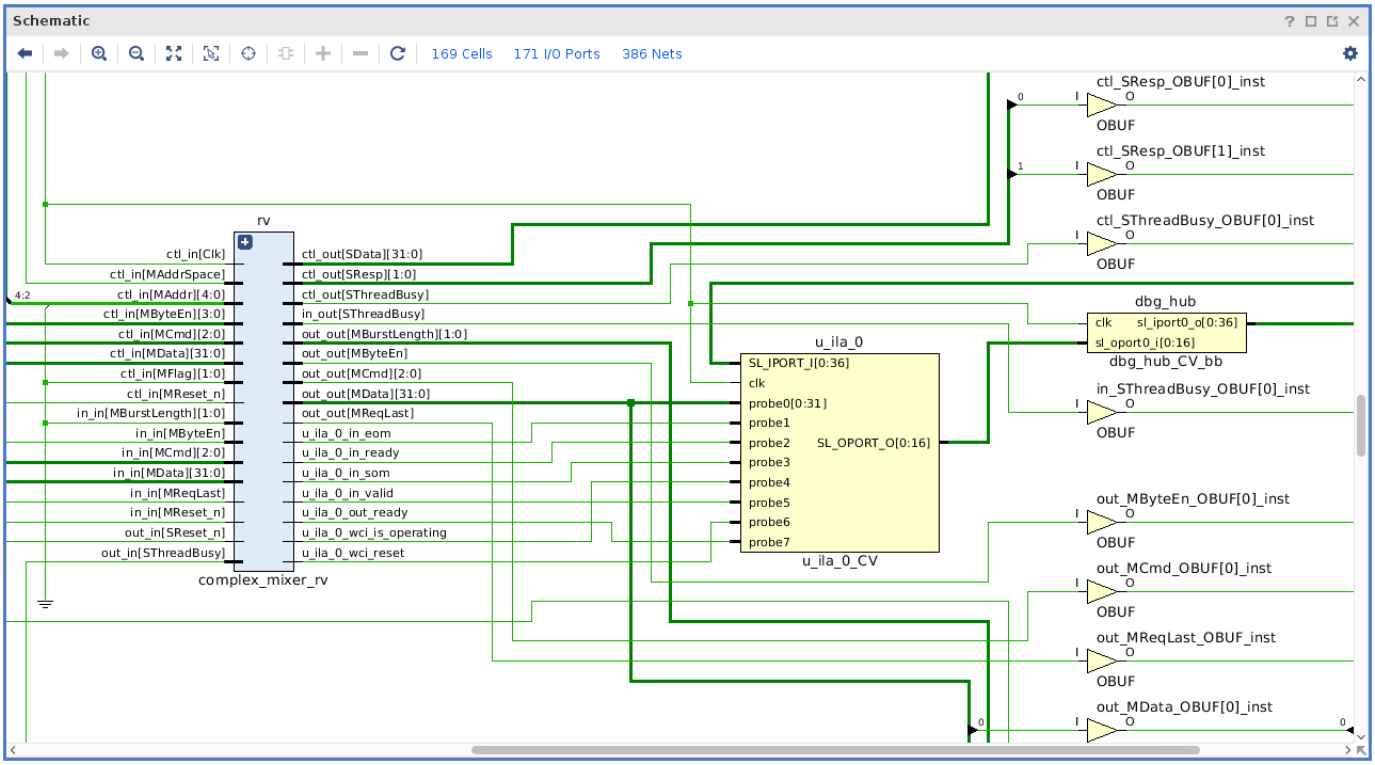

Figure 3: Xilinx Vivado 2017.1 Debug Cores Schematic

6. Enter the Tcl Console, and overwrite the netlist created by the 'make' system in the "target-zynq" directory:

```
> write_edif -security_mode all -force complex_mixer.edf
```

Note: "-force" tells the write_edif command to overwrite the file if it already exists.

Note: "-security_mode all" ensures that partially encrypted designs will still result in a single EDIF file.

7. In the Tcl Console, generate the *.ltx file containing the debug probe information:

```
> write_debug_probes complex_mixer.ltx
```

8. Build an HDL assembly containing this worker

The generated bitstream contains the debug and ILA cores which will be recognized by Xilinx Vivado Integrated Logic Analyzer tool. Once the bitstream has been loaded onto the target FPGA, the Analyzer tool can connect and detect the presence of the debug core(s).

Reiterating an Important Note for Case 2: Rebuilding or cleaning the worker (or other AV asset) at any time (with "make") will remove any debug functionality added to the Vivado project. The Vivado project files are an artifact of the "make" process, and will be overwritten each time "make" is run for that asset.

3.2 Xilinx ISE

3.2.1 Case 1: Integrate ChipScope into HDL Worker using cores from ISE's CORE Generator

This case assumes that the developer has created a Xilinx CORE Generator project and configured the *Debug and Verification* cores as desired. Specifically, these instructions have been verified for the ICON, ILA and VIO cores. Of the many output files generated by CORE Generator for each core, only two (*.vhd, *.ngc) are necessary to be retained for building the HDL worker and subsequently, the HDL assembly.

1. Integrate the *Debug and Verification* cores into the worker's VHDL:
   - Declare and instantiate the component for each core (ICON, ILA, VIO, etc)
   - As needed, add signal declarations and assignments (CONTROL(35 downto 0), TRIG(Y downto 0), DATA(Z downto 0), etc)
2. Edit the worker's `Makefile` to include the path and file name of the instantiated cores (ICON, ILA, VIO, etc) *.vhd files. Use the framework's Makefile variable "SourceFiles=" to include the path and name of the VHDL file of each core. Absolute or relative paths are acceptable. An example is provided:

```
SourceFiles=../../chipscope/icon1.vhd ../../chipscope/ila_trig32_data128_16384.vhd
```

3. Edit the HDL Assembly's `Makefile` to include the path and file name of the instantiated cores (ICON, ILA, etc) *.ngc files. Use the framework's Makefile variable "Cores=" to include the path and name of the NGC file of each core. Absolute or relative paths are acceptable. An example is provided:

```
Cores=../../../../components/dsp_comps/cic_dec.hdl/chipscope/icon1.ngc
../../../../components/dsp_comps/cic_dec.hdl/chipscope/ila_trig32_data128_16384.ngc
```

4. Build HDL worker for target.
5. Build HDL assembly for platform.

The generated bitstream contains the *Debug and Verification* cores which will be recognized by the Xilinx ChipScope Pro Analyzer tool. Once the bitstream has been loaded onto the target FPGA, the Analyzer tool can connect and detect the presence of the *Debug and Verification* core(s).

3.2.2 Case 2: Integrate ChipScope into HDL Assembly using the Inserter tool

1. If the HDL assembly has already been built, proceed to step 2. Otherwise start the the HDL assembly build process. Once the build process has completed the *ngdbuild* step, the build process can be canceled.

2. Launch the Xilinx ChipScope Pro *Inserter* tool and create a new project.

   Note: The versions of the *Inserter* and *Analyzer* tools must match.

3. Select the *Input Design Netlist* by browsing to the HDL assembly's container's target directory and selecting the -b.ngc file: A example is provide:

   ```
   /data/ocpi_baseassets/ocpiassets/applications/FSK/assemblies/fsk_filerw/
   container-fsk_filerw_matchstiq_base/target-zynq/fsk_filerw_matchstiq_base-b.ngc
   ```

4. The default name and location of the *Output Design Netlist* is acceptable.

5. The default name and location of the *Output Directory* is acceptable.

6. Save the project file. When selecting a location to save the project file, it is recommended to not save project in an OpenCPI artifact directory, as they are deleted upon execution of a *make clean* process.

7. Continue with the Inserter tool process to:

   - Add signals that are to be monitored

   - Generate the cores and NGO file:

     - The output folders and files will be generate in *Output Directory* directory.

   - cs_icon_pro/
   - cs_ila_pro_0/
   - dump.xst/
   - fsk_filerw_matchstiq_base-b.ngo
   - icon_pro.ngc
   - ila_pro_0.ngc

8. Used the NGO file to regenerate the NGD

   i) - Replace the NGC with the NGO by copying *-b.ngo over *-b.ngc.

   ii) - Rebuild the NGD file based upon *ngdbuild.out*.

   Within the container's target directory, open the *ngdbuild.out* file and locate the *ngdbuild* command including all of its options necessary for execution. Note that the command in *ngdbuild.out* provides a relative path for ngdbuild. Copy the *ngdbuild* command, modify the command to include the full path to ngdbuild, and execute it from the container's target directory. An example is provided below. Note that this should not be executed from a shell where a Xilinx settings32.sh or settings64.sh script has been sourced.

   ```
   /opt/Xilinx/14.7/ISE_DS/ISE/bin/lin64/ngdbuild -verbose -uc
   /data/ocpi_baseassets/ocpiassets/applications/FSK/assemblies/../../../hdl/platforms/matchstiq/lib/matchstiq.ucf -p xc7z020-1-clg484 -sd
    ../../../../../../hdl/platforms/matchstiq/lib/hdl/zynq -sd
    ../../../../../../hdl/platforms/matchstiq/lib/hdl/zynq -sd
    ../../../../../../../../ocpi_baseproject/exports/lib/devices/hdl/zynq -sd ../../lib/hdl/zynq -sd
    ../../../../../../components/dsp_comps/complex_mixer.hdl/chipscope -sd
    ../../../../../../components/dsp_comps/complex_mixer.hdl/chipscope -sd
    ../../../../../../../../ocpi_baseproject/exports/lib/adapters/hdl/zynq -sd
    ../../../../../../components/util_comps/lib/hdl/zynq -sd
    ../../../../../../components/dsp_comps/lib/hdl/zynq -sd
    ../../../../../../components/dsp_comps/lib/hdl/zynq -sd
    ../../../../../../components/dsp_comps/lib/hdl/zynq -sd
    ../../../../../../components/dsp_comps/lib/hdl/zynq -sd
    ../../../../../../components/dsp_comps/lib/hdl/zynq -sd
    ../../../../../../components/dsp_comps/lib/hdl/zynq -sd
    ../../../../../../../../ocpi_baseproject/exports/lib/adapters/hdl/zynq -sd
    ../../../../../../../../ocpi_baseproject/exports/lib/devices/hdl/zynq -sd
    ../../../../../../../../ocpi_baseproject/exports/lib/components/hdl/zynq
    fsk_filerw_matchstiq_base-b.ngc fsk_filerw_matchstiq_base.ngd
   ```

9. Continue the HDL Assembly build process.

   - Change from the container target directory back to the assembly directory and re-run make

The generated bitstream contains the *Debug and Verification* cores which will be recognized by the Xilinx ChipScope Pro Analyzer tool. Once the bitstream has been loaded onto the target FPGA, use the Analyzer tool can connect and detect the presence of the *Debug and Verification* core(s). The saved project file can be imported to automatically populates the names of the signals being monitored.

3.3 Altera SignalTap II

3.3.1 Limitations

In versions of Quartus after 14.1, the OpenCPI build flow of exporting QXPs and including SignalTap at the Worker level causes a build failure. Per Altera's website, you can force Quartus to use the legacy SignalTap flow. More details can be found here:

`https://www.altera.com/support/support-resources/knowledge-base/solutions/rd07012015_904.html`

In order to use this build flow, the file /opt/opencpi/cdk/include/hdl/quartus.mk must be modified. An example diff of the change needed can be seen below:

```
<  ) > $(Core).qsf; echo fit_stratixii_disallow_slm=On > quartus.ini;
---
>  ) > $(Core).qsf; echo fit_stratixii_disallow_slm=On > quartus.ini; echo sci_use_legacy_sld_flow=On >> quartus.ini;
```

3.3.2 Create SignalTap II instance

This signal tap implementation can be reused with different workers but not in the same bitstream.

1. Run Quartus
    - The default location for the Quartus executable can be found at `/opt/Altera/17.1/quartus/bin/quartus`
2. Open SignalTap in Quartus
    - Go to tools - IP Catalog
    - The ipcatalog will now be visible on the right
    - Select the device family ex. Stratix IV
    - In the search bar type in SignalTap and select it from the filtered list.
    - Signal tap will say New IP Variation.
3. Create IP Variation
    - Entity Name: signal_tap
    - Save in folder: Navigate to the workers's directory and create a folder called signal_tap
    - Family: Stratix IV
    - Device: Select yours or Unknown
4. Configure SignalTap core
    (a) Data:
        i. Data Input Port Width: Select how many signals to watch. You will connect the signals to this port from your worker.
        ii. Sample Depth: The count of how many total captures will occur. Basically how many clock cycles of capture.
    (b) Trigger:
        i. Trigger Input Port Width: Select the number of different signals that should be observed to trigger the start of capturing.
        ii. Trigger Conditions: How many combinations, aka if watching multiple signals then you may want a combination of when enable is high and another signal is a rising edge, in that case use 2 trigger conditions.
    (c) Storage Qualifier:
        i. Input Port: Toggles capturing signals using an enable signal.
        ii. Continuous: Once the trigger condition is met it will take samples from then on. In the Logic Analyzer you can configure to take the samples before and after the trigger condition or near the beginning or near the end.

(d) Segmented Acquisition:

i. Allows specifying groups of continuous segments. Meaning if Number of Segments = 2 then half the sample depth will be used for the first time the trigger condition is met and then the next half will be used for the next time the trigger condition is met.

5. Click Generate HDL

- A Generation Popup will show. Select VHDL instead of Verilog. (Note: You can use Verilog as well but for our purposes workers are more commonly implemented in VHDL)
- The remaining defaults are sufficient
- Click Generate.

3.3.3 Integrate SignalTap II instance into HDL Worker

These instructions have been verified for the sld_signaltap core. Many files are generated but only two of them are required for our purposes. The (*_inst.vhd) file contains the component declaration and the (*.vhdl) file is used to build the SignalTap core into the HDL worker and subsequently, the HDL assembly.

1. Integrate the *Debug and Verification* cores into the worker's VHDL: Navigate to the workers signal_tap subfolder and copy the _inst.vhd component declaration and component instantiation to the architecture section and begin section respectively.

a) Place the component declaration in the declarative area of the architecture section

```
component capture_v2_signaltap is
port (
    acq_data_in    : in std_logic_vector(71 downto 0) := (others => 'X'); -- acq_data_in
    acq_trigger_in : in std_logic_vector(4 downto 0)  := (others => 'X'); -- acq_trigger_in
    acq_clk        : in std_logic                     := 'X';             -- clk
    storage_enable : in std_logic                     := 'X'              -- storage_enable
);
end component capture_v2_signaltap;
```

b) Place the component instantation under the begin section as a concurrent statement

```
u0 : component capture_v2_signaltap
port map (
        acq_data_in    => CONNECTED_TO_acq_data_in,    -- tap.acq_data_in
        acq_trigger_in => CONNECTED_TO_acq_trigger_in, -- acq_trigger_in
        acq_clk        => CONNECTED_TO_acq_clk,        -- acq_clk.clk
        storage_enable => CONNECTED_TO_storage_enable  -- storage_qualifier.storage_enable
);
```

2. Connect signals, as needed, from the worker to the component instance

```
u0 : component capture_v2_signaltap
port map (
        acq_data_in    => ctl_in.is_operating & in_in.eom &
                          in_in.som & in_in.valid & out_in.ready &
                          std_logic_vector(time_in.seconds) &
                          std_logic_vector(time_in.fraction), -- tap.acq_data_in
        acq_trigger_in => ctl_in.is_operating & in_in.som & in_in.eom &
                          in_in.valid & out_in.ready, -- acq_trigger_in
        acq_clk        => ctl_in.clk,          -- acq_clk.clk
        storage_enable => ctl_in.is_operating  -- storage_qualifier.storage_enable
);
```

3. Edit the Worker's `Makefile` to include the path and file name of the instantiated core *.vhdl files. Use the framework's Makefile variable "SourceFiles=" to include the path and name of the VHDL file of the core. Absolute or relative paths are acceptable. SourceFiles must be before the "include" statement. An example is provided:

```
SourceFiles=.signaltap/synthesis/capture_v2_signaltap.vhd
include $(OCPI_CDK_DIR)/include/worker.mk
```

4. Build the HDL worker for the target.

   ```
   ocpidev build -d <path/worker.hdl> --hdl-target <target>
   ```

5. Generate SignalTap format file using Quartus GUI

   - Start Quartus and open the Quartus Project File (.qpf) which is located in the built Worker's target-stratix4 directory. Click File - Create/Update - Create Signal Tap II file from Design Instance(s) and save the file. This will open signal tap logic analyzer. If you have already created a stp file you can open this in Quartus by File - open and selecting the *.stp file from the workers directory.

6. Build the HDL assembly for the target platform.

   ```
   ocpidev build -d <path/assembly> --hdl-platform <platform>
   ```

3.3.4 Monitor signals with SignalTap II Logic Analyzer

The generated bitstream contains the *Debug and Verification* cores which will be recognized by the Altera Quartus SignalTap II Logic Analyzer tool. The executable for this tool (quartus_stpw) is located in the Quartus installation directory. Once the bitstream has been loaded onto the target FPGA, use the tool can connect and detect the presence of the *Debug and Verification* core(s).

1. Setup the JTAG Connection:

   - Select USB-Blaster
   - Select your device
   - If you cannot connect to your device, check the *JTAG Daemon* section of the Alst4_Getting_Started_ Guide

2. Set the triggers to specify when signal collection begins:

   - Go to the setup tab and scroll down to the triggers. Select the signal you want to set with a trigger then right click the box with a lattice pattern to specify the trigger criteria.

3. Enable Collection

   - In the instance manager select the SignalTap instance. The run Analysis and Autorun Analysis buttons should be enabled.
   - Press the Autorun Analysis button to record continuously.

4. Run your application or test and the signals will appear in the data tab.