Introduction

Some Xilinx FPGAs contain hard processor cores. This document describes how to debug and trace these cores.

The Xilinx Zynq-7000 and Xilinx UltraScale+ series contain embedded processor systems that include multiple ARM cores.

This document covers several topics for working with TRACE32 and Xilinx-MPSoc-type SoCs such as Zynq-7000 or Zynq UltraScale+.

In This Document:

- Physical connection requirements
- How to export the off-chip trace on Zynq-7000
- How to perform a debugger-based boot sequence on the Zynq-7000
- How to export the off-chip trace on Zynq UltraScale+
- How to perform a debugger-based boot sequence on the Zynq UltraScale+

Overview of TRACE32 Commands used in this Application Note:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzer.PortSize</td>
<td>Set port size of physical trace interface, if it differs from TPIU.PortSize</td>
</tr>
<tr>
<td>TPIU.PortClock</td>
<td>Set the lane rate for HSSTP serial trace</td>
</tr>
<tr>
<td>TPIU.PortMode</td>
<td>Set the HSSTP protocol variant</td>
</tr>
<tr>
<td>TPIU.PortSize</td>
<td>Set the number of data pins driven by the ARM CoreSight hardware;</td>
</tr>
<tr>
<td></td>
<td>Set the number of HSSTP lanes, if applicable</td>
</tr>
<tr>
<td>TPIU.RefClock</td>
<td>Configure a reference clock provided by the debugger to the target</td>
</tr>
</tbody>
</table>

Related Documents:

- “Integration for Xilinx Vivado” (int_vivado.pdf) describes how to use Lauterbach PowerDebug hardware tools with Xilinx Vivado.
- “ARM Debugger” (debugger_arm.pdf) describes the processor-specific settings and features for the Cortex-A/R (ARMv7, 32-bit) debugger.
- “ARMv8-A/-R Debugger” (debugger_armv8a.pdf) describes the processor-specific settings and features for the Cortex-A/R (ARMv8, 32/64-bit) debugger.
Physical Connection Requirements

MPSoC devices use a parallel TPIU trace interface to export trace data. This interface can either be exported though PS pins (MIO) or PL pins via the EMIO interface.

For Zynq UltraScale+ FPGAs, this document also provides instructions on how to use the PL portion of the device to convert the parallel interface into a serial HSSTP interface.

Requirements for Parallel Trace

There are two standard connectors for parallel TPIU trace. The first connector is called Mictor-38, while the second connector is called MIPI-60.

These connectors also include the standard JTAG debug signals. It is possible to either use the JTAG signals on the trace connector or a separate debug connector. We do not recommend routing the JTAG signals to both connectors for reasons of signal integrity.

The required pins for the off-chip trace connection are shown in the table below.

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Description</th>
<th>MANDATORY/OPTIONAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>VREF-TRACE</td>
<td>Reference voltage for the TRACEDATA[...], TRACECLK and TRACECTL lines</td>
<td>MANDATORY</td>
</tr>
<tr>
<td>TRACECLK</td>
<td>Clock line for the trace interface (DDR)</td>
<td>MANDATORY</td>
</tr>
<tr>
<td>TRACECTL</td>
<td>Control line for the trace interface</td>
<td>OPTIONAL</td>
</tr>
<tr>
<td>TRACEDATA[x:0]</td>
<td>Data lines for the trace interface</td>
<td>MANDATORY</td>
</tr>
</tbody>
</table>

For the MPSoC devices, we recommend that you use 16 data lines (TRACEDATA[15:0]). The optional pin TRACECTL is only required for use with Wrapped mode (also called Normal mode in ARM terminology). As all Lauterbach tools work equally well with the Continuous mode, we exclusively use this mode in this application note and don't require TRACECTL. For more information about the connectors and available adaptors, visit:

- Mictor-38: www.lauterbach.com/adetmmictor.html
- MIPI-60: www.lauterbach.com/adetmmipi60.html
<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/C</td>
<td>1</td>
<td>2</td>
<td>N/C</td>
</tr>
<tr>
<td>N/C</td>
<td>3</td>
<td>4</td>
<td>N/C</td>
</tr>
<tr>
<td>GND</td>
<td>5</td>
<td>6</td>
<td>TRACECLK</td>
</tr>
<tr>
<td>DBGREQ</td>
<td>7</td>
<td>8</td>
<td>DBGACK</td>
</tr>
<tr>
<td>RESET</td>
<td>9</td>
<td>10</td>
<td>EXTRIG</td>
</tr>
<tr>
<td>TDO</td>
<td>11</td>
<td>12</td>
<td>VREF-TRACE</td>
</tr>
<tr>
<td>RTCK</td>
<td>13</td>
<td>14</td>
<td>VREF-DEBUG</td>
</tr>
<tr>
<td>TCK/TCKC/SWCLK</td>
<td>15</td>
<td>16</td>
<td>TRACEDATA7</td>
</tr>
<tr>
<td>TMS/MTSC/SWDIO</td>
<td>17</td>
<td>18</td>
<td>TRACEDATA6</td>
</tr>
<tr>
<td>TDI</td>
<td>19</td>
<td>20</td>
<td>TRACEDATA5</td>
</tr>
<tr>
<td>TRST</td>
<td>21</td>
<td>22</td>
<td>TRACEDATA4</td>
</tr>
<tr>
<td>TRACEDATA15</td>
<td>23</td>
<td>24</td>
<td>TRACEDATA3</td>
</tr>
<tr>
<td>TRACEDATA14</td>
<td>25</td>
<td>26</td>
<td>TRACEDATA2</td>
</tr>
<tr>
<td>TRACEDATA13</td>
<td>27</td>
<td>28</td>
<td>TRACEDATA1</td>
</tr>
<tr>
<td>TRACEDATA12</td>
<td>29</td>
<td>30</td>
<td>GND</td>
</tr>
<tr>
<td>TRACEDATA11</td>
<td>31</td>
<td>32</td>
<td>GND</td>
</tr>
<tr>
<td>TRACEDATA10</td>
<td>33</td>
<td>34</td>
<td>VCC</td>
</tr>
<tr>
<td>TRACEDATA9</td>
<td>35</td>
<td>36</td>
<td>TRACECTL</td>
</tr>
<tr>
<td>TRACEDATA8</td>
<td>37</td>
<td>38</td>
<td>TRACEDATA0</td>
</tr>
<tr>
<td>Signal</td>
<td>Pin</td>
<td>Pin</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>VREF-DEBUG</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>TCK</td>
<td>TCKC</td>
<td>SWCLK</td>
<td>3</td>
</tr>
<tr>
<td>TDI</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>RTCK</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>TRST-</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>DBGACK TRIGOUT</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>TRACECLK</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>15</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>TRACECTL</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[0]</td>
<td>19</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[1]</td>
<td>21</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[2]</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[3]</td>
<td>25</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[5]</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[6]</td>
<td>31</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[7]</td>
<td>33</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[8]</td>
<td>35</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[9]</td>
<td>37</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[10]</td>
<td>39</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[12]</td>
<td>43</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[13]</td>
<td>45</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[14]</td>
<td>47</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[15]</td>
<td>49</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[16]</td>
<td>51</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[17]</td>
<td>53</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>TRACEDATA[18]</td>
<td>55</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>57</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>59</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>
Requirements for Serial HSSTP Trace

When exporting a HSSTP trace interface, a 40-pin SAMTEC connector is commonly used. The connector also includes the standard JTAG debug signals. It is possible to either use the JTAG signals on the trace connector or a separate debug connector. We do not recommend routing the JTAG signals to both connectors for reasons of signal integrity.

The connector includes six TXP/TXN pairs. With the UltraScale+ target, there is no benefit in using more than two lanes.

The TX lanes are terminated by the PowerTrace serial or the serial preprocessor. On the target PCB, they should be routed directly between the connector and the FPGA as a 50 Ω differential pair.

For more information about the target connector, visit www.lauterbach.com/adetmhsstp.html

The FPGA requires a reference clock for its gigabit transceivers. A clock whose frequency is 1/20 of the bit rate can be provided by the debugger on the CLKP/CLKN pins. It is also possible to provide a clock generator on the target. Check the Xilinux documentation for the allowable frequencies.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>TXP[4]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>TXN[4]</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>GND</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>TXP[2]</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>TXN[2]</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>GND</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>TXP[0]</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>TXN[0]</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>GND</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>CLKP</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>CLKN</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>GND</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>TXP[1]</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>TXN[1]</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>GND</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>TXP[3]</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>TXN[3]</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>GND</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>TXP[5]</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>TXN[5]</td>
<td>39</td>
<td>40</td>
</tr>
</tbody>
</table>

©1989-2019 Lauterbach GmbH
Lauterbach supports three methods for exporting the off-chip trace interface of Zynq-7000 devices:

1. Directly using the FixedIO/MIO pins of the Processing System (PS)
2. Routing trace data through the Programmable Logic (PL) portion of the SoC, dividing the clock by two
3. Routing trace data through the PL portion of the SoC, dividing the data width by two and using DDR output registers to increase the bandwidth

Method 1 is the only method that works while the PL is not yet programmed. However, it is limited to a trace port width of 16 bits. In addition, the signal quality of PS pins is worse than on the I/Os belonging to the PL.

Method 2 allows trace port widths up to 32 bits and trace clock frequencies up to 125 MHz. It consumes a small amount of PL resources for pipeline registers and routing. A clock divider is used to generate the TRACECLK signal from the internal single data rate trace clock. The maximum operating frequency is limited by the maximum clock frequency of the PS-PL TPIU interface.

Method 3 is similar to method 2, but instead of dividing the trace clock by two, the width of the data path is reduced internally. This allows using an external 16-bit trace port by internally utilizing the full 32-bit PS-PL TPIU interface. Alternatively, an external 8-bit port can also be created. The maximum frequency achievable using this method is 250 MHz, which corresponds to 1 GB/s (decimal) when used with a 16-bit trace port.

**NOTE:** We recommend using method 2 or 3. Note that these methods are interchangeable for a given PCB design. Designing a PCB to use method 1 limits the achievable trace bandwidth.

The remainder of this section contains a step-by-step procedure to each of these methods, followed by instructions related to board bring-up using a debugger.
1. Create a new Vivado project with an instance of the Zynq processing system.
2. Enter the configuration of the Zynq processing system.
3. Enable the trace pin export via MIO by selecting the desired port size, see [A] in the figure below.
4. Select the external pins that are connected to the trace port, see [B] in the figure below.
5. Select an internal clock source (ARM PLL, DDR PLL or IO PLL) and the desired frequency for the TPIU (Trace Port Interface Unit).

Please note that the exported TRACECLK is a DDR clock signal whose actual frequency will be half the frequency selected in this configuration window:
6. Finish your Vivado design and export the project to the SDK.
7. Generate or regenerate your FSBL (first-stage boot loader).
8. Do one of the following:
   - Either program the resulting FSBL to the boot device,
   - Or perform a debugger-based boot (see Performing a Debugger-Based Boot on the Zynq-7000).

You are now ready to debug and trace your target with TRACE32.
1. Create a new Vivado project with an instance of the Zynq processing system.
2. Enter the configuration of the Zynq processing system.
3. Enable the trace pin export via MIO by selecting the desired port size and the pins that will be connected to the trace connector.

4. Activate at least one of the `FCLK_CLK<x>` clock signals, which will later be used as the TPIU clock. The exported TRACECLK will be half this frequency. Select **External** as the TPIU clock source.
5. Connect and export the signals as follows:

To use continuous mode, we do not need the `TRACE_CTL` signal. The `TRACE_CLK_OUT` signal will be generated by the HDL wrapper. We rename `FCLK_CLK0` to `TRACE_CLK_SDR` and will use this signal to sample `TRACE_DATA`.

6. Finish your block design and generate the HDL wrapper.

7. Modify the HDL wrapper to include the pipeline registers and DDR clock generation for routing the PS trace interface to PL pins:

```vhdl
entity zynq_wrapper is
  port (
    oTraceClkDdr: out std_logic;
    oTraceData: out std_logic_vector(15 downto 0)
  );
end entity;

architecture SDR of zynq_wrapper is
  signal wTraceClkSdr: std_logic;
  signal wTraceData: std_logic_vector(15 downto 0);
  signal rTraceClkDdr: std_logic;
  signal rTraceData_q: std_logic_vector(15 downto 0);
  signal rTraceData_qq: std_logic_vector(15 downto 0);

begin
  zynq_i: entity work.zynq port map (
    TRACE_CLK_SDR => wTraceClkSdr,
    TRACE_DATA => wTraceData
  );

  trace_pipeline: process(wTraceClkSdr)
  begin
    if rising_edge(wTraceClkSdr) then
      rTraceClkDdr <= not rTraceClkDdr;
      rTraceData_qq <= rTraceData_q;
      rTraceData_q <= wTraceData;
    end if;
  end process;

  oTraceData <= rTraceData_qq;
  oTraceClkDdr <= rTraceClkDdr;
end architecture;
```
8. Assign the `oTraceData[x:0]` and `oTraceClkDdr` to the appropriate FPGA pins matching your board layout. Select the same I/O standard for all pins and the slew rate appropriate for the desired trace port speed.

9. Finish your Vivado design and export the project to the SDK.

10. Generate or regenerate your FSBL (first-stage boot loader).

11. Do one of the following:
   - Either program the resulting FSBL to the boot device,
   - Or perform a debugger-based boot (see “Performing a Debugger-Based Boot on the Zynq-7000”, page 17).

You are now ready to debug and trace your target with TRACE32.
1. Create a new Vivado project with an instance of the Zynq processing system.

2. Enter the configuration of the Zynq processing system.

3. Enable the trace pin export via MIO by selecting twice the desired port size and the pins that will be connected to the trace connector. To use this method, you must select d[0:1] ... d[16:31] for an external 16-bit trace port or d[0:1] ... d[8:15] for an external 8-bit trace port.

4. Activate at least one of the FCLK_CLK<XX> clock signals, which will later be used as the TPIU clock. The exported TRACECLK will be using this frequency. Select External as the TPIU clock source.
5. Connect and export the signals as follows:

```
processing_system7_0
```

To use continuous mode, we do not need the `TRACE_CTL` signal. The `TRACE_CLK_OUT` signal will be generated by the HDL wrapper. We rename `FCLK_CLK0` to `TRACE_CLK_SDR` and will use this signal to sample `TRACE_DATA`.

6. Finish your block design and generate the HDL wrapper.

7. Locate the file `~/demo/arm/hardware/zynq-7000/parallel_trace_adapter.vhd` from your TRACE32 installation and add it to the project.

8. Modify the HDL wrapper to include the parallel_trace_adapter for routing the PS trace interface to
9. Assign the `oTraceData[x:0]` and `oTraceClkDdr` to the appropriate FPGA pins matching your board layout. Select the same I/O standard for all pins and the slew rate appropriate for the desired trace port speed.

10. Finish your Vivado design and export the project to the SDK.

11. Generate or regenerate your FSBL (first-stage boot loader).

12. Do one of the following:
   - Either program the resulting FSBL to the boot device,
   - Or perform a debugger-based boot (see “Performing a Debugger-Based Boot on the Zynq-7000”, page 17).
NOTE: To use the off-chip trace port from TRACE32, you need to apply the following special settings:

```
TPIU.PortSize     32 ; internal port size (PS -> PL),
                 ; same as gBitsIn
Analyzer.PortSize 16 ; external port size (PL -> TRACE32),
                 ; same as gBitsOut
```

You are now ready to debug and trace your target with TRACE32.
This section focuses on the **JTAG-BOOT** mode of the Zynq-7000. In contrast to all other boot modes, this mode is only intended for development. The basic idea is that the CPUs will wait in an endless loop after executing the boot ROM, allowing the JTAG probe to perform all further initialization.

**To perform a debugger-based boot:**

1. Set the boot mode to **JTAG-BOOT** using the MIO lines.
2. Reset the SoC, for example by asserting the RESET line. Not all boards have a RESET line connected to the SoC, thus a power cycle or similar might be required.
3. Execute the boot ROM.
4. Load the FSBL boot code using the debugger.
5. Execute the FSBL boot code.
6. Optionally load the FPGA fabric using the debugger.

You are now ready to load the next stage boot loader, OS, etc., and to use the optional off-chip trace.

Example files following the above sequence are included in the TRACE32 installation directory under `~/demo/arm/hardware/zynq-7000`
On the Zynq UltraScale+ series, Lauterbach currently supports four ways to export trace data:

1. Parallel trace exported using the FixedIO/MIO pins of the Processing System (PS)
2. Parallel trace routed through the EMIO pins and the Programmable Logic (PL) portion of the SoC
3. HSSTP (serial) trace via PL gigabit transceivers (GTH)
4. PCIe (serial) trace via PS gigabit transceivers (GTR)

Each method uses a different kind of SoC resource that is not available to the application while the application is being traced. The following table provides an overview of different configurations, their resource usage, achievable data rate and required Lauterbach hardware.

<table>
<thead>
<tr>
<th>Method</th>
<th>SoC Resources</th>
<th>PL Used</th>
<th>Data Rate</th>
<th>Lauterbach Trace Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (16 bit @ 125 MHz DDR)</td>
<td>17 PS I/Os</td>
<td>no</td>
<td>500 MB/s</td>
<td>PowerTrace or PowerTrace II/PX with AutoFocus preprocessor</td>
</tr>
<tr>
<td>2 (16 bit @ 250 MHz DDR)</td>
<td>17 PL I/Os</td>
<td>yes</td>
<td>1000 MB/s</td>
<td></td>
</tr>
<tr>
<td>2 (8 bit @ 250 MHz DDR)</td>
<td>9 PL I/Os</td>
<td>yes</td>
<td>500 MB/s</td>
<td>PowerTrace II/PX with serial preprocessor or PowerTrace Serial</td>
</tr>
<tr>
<td>3 (2 Lanes @ 6.25 Gbps)</td>
<td>2 GTH</td>
<td>yes</td>
<td>1000 MB/s</td>
<td>PowerTrace II/PX with serial preprocessor or PowerTrace Serial</td>
</tr>
<tr>
<td>3 (1 Lane @ 6.25 Gbps)</td>
<td>1 GTH</td>
<td>yes</td>
<td>625 MB/s</td>
<td></td>
</tr>
<tr>
<td>3 (1 Lane @ 10Gbps)</td>
<td>1 GTH</td>
<td>yes</td>
<td>1000 MB/s</td>
<td>PowerTrace Serial</td>
</tr>
<tr>
<td>4 (PCIe v2 x2)</td>
<td>2 GTR</td>
<td>no</td>
<td>1000 MB/s</td>
<td></td>
</tr>
<tr>
<td>4 (PCIe v2 x1)</td>
<td>1 GTR</td>
<td>no</td>
<td>500 MB/s</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:**
- Lauterbach recommends using one of the methods with a data rate of 1000 MB/s, especially if a multi-core program is to be traced.
- Due to the complexity involved in setting up PCIe trace, we only recommend this method if a PCIe expansion slot is already present on the target board.

This section includes a step-by-step introduction for each method as well as a step-by-step introduction for performing a debugger-based boot sequence.
1. Create a new Vivado project with an instance of the Zynq processing system.
2. Enter the configuration of the Zynq processing system.
3. Enable the trace pin export via MIO by selecting the desired port size and the pins that will be connected to the trace connector.
4. Select an internal clock source (IOPLL, DPLL, or APLL) and the desired frequency for DBG Trace. Please note that the exported TRACECLK is a DDR clock signal whose actual frequency will be half the frequency selected in this configuration window.
5. Finish your Vivado design and export the project to the SDK.
6. Generate or regenerate your FSBL (first-stage boot loader).
7. Do one of the following:
   - Either program the resulting FSBL to the boot device,
- Or perform a debugger-based boot (see “Performing a Debugger-Based Boot on the Zynq UltraScale+”, page 37).

You are now ready to debug and trace your target with TRACE32.
1. Create a new Vivado project with an instance of the Zynq processing system.

2. Enter the configuration of the Zynq processing system.

3. Enable the trace pin export via EMIO. Select 32Bit as the trace width. This setting will later be overridden by the debugger.

4. Activate at least one of the PL Fabric Clocks (for example PL0), which will later be used as the TPIU clock. The exported TRACECLK will be using this frequency. Select the same frequency for the DBG_TRACE clock.

5. Connect and export the signals as follows
To use continuous mode, we do not need the TRACE_CTL signal. The trace_clk_out signal will be generated by the HDL wrapper. We rename pl_clk0 to TRACE_CLK_SDR and will use this signal to sample TRACE_DATA.

6. Finish your block design and generate the HDL wrapper.

7. Locate the file ~/demo/arm64/hardware/zynq_ultrascale/hdl/parallel_trace_adapter.vhd from your TRACE32 installation and add it to the project.

8. Modify the HDL wrapper to include the parallel trace adapter that exports the trace port without a clock divider:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity zynq_wrapper is
  port (  
    oTraceClk: out std_logic;
    oTraceData: out std_logic_vector(15 downto 0)
  );
end entity;

architecture behavioral of zynq_wrapper is
  signal wTraceClkSdr: std_logic;
  signal wTraceDataSdr: std_logic_vector(31 downto 0);
begin
  yZynq: entity work.zynq port map (  
    TRACE_CLK_SDR => wTraceClkSdr,
    TRACE_DATA => wTraceDataSdr  
  );

  yAdapter: entity work.parallel_trace_adapter generic map (  
    gPlatform => "ULTRASCALE",
    gBitsIn => 32,
    gBitsOut => 16
  ) port map (  
    iClk => wTraceClkSdr,
    iData => wTraceDataSdr,
    oClk => oTraceClk,
    oData => oTraceData
  );
end architecture;
```

9. Assign the oTraceData[x:0] and oTraceClk to the appropriate FPGA pins matching your board layout. Select the same I/O standard for all pins and the slew rate appropriate for the desired
trace port speed.

10. Finish your Vivado design and export the project to the SDK.

11. Generate or regenerate your FSBL (first-stage boot loader).

12. Do one of the following:
   - Either program the resulting FSBL to the boot device,
   - Or perform a debugger-based boot (see "Performing a Debugger-Based Boot on the Zynq UltraScale+", page 37).

13. Use the following TPIU and Analyzer commands in your PRACTICE start-up script (*.cmm) to configure the trace:

   ```
   TPIU.PortSize 32 ; internal port size (PS -> PL),
   ; same as gBitsIn
   Analyzer.PortSize 16 ; external port size (PL -> TRACE32),
   ; same as gBitsOut
   ```

   You are now ready to debug and trace your target with TRACE32.
Use the following step-by-step procedure to produce an FPGA design that supports tracing the ARM cores via HSSTP at a bit rate of 6.25 Gbps or lower. This is the maximum supported data rate of the serial preprocessor LA-7988. If you use the PowerTrace Serial, LA-3520, you may want to consider using a higher bit rate, as described in the next section, “Exporting the UltraScale+ Trace Interface via HSSTP (10 Gbps)”, page 30.

Note that the bandwidth of the trace infrastructure of the PS is limited to 1000 MB/s. Therefore, using more than two lanes is only beneficial if the lane rate is limited by other factors.

1. Perform steps 1 through 6 of the previous section, “Exporting the UltraScale+ Trace Interface via FPGA Fabric/PL”, page 21. Depending on the desired number of lanes, select an appropriate trace port frequency to avoid FIFO overflows:
   - For one lane at 6.25 Gbps, set the frequency to 150 MHz.
   - For two lanes at 6.25 Gbps, set the frequency to 250 MHz.
   - For any other setup, calculate the maximum frequency as:
     \[ f_{\text{max}} = (\text{number of lanes}) \cdot (\text{lane rate}) \cdot \frac{1 \text{ B}}{10 \text{ bit}} \cdot \frac{1 \text{ Hz}}{4 \text{ G/s}} \]
     The selected frequency must be lower than this frequency and at most 250 MHz.

2. Create a second block design to hold the Xilinx Aurora encoder and a FIFO. In this example, it is called aurora_and_fifo.
3. In the block design, take the following steps to instantiate an Aurora 8B10B core:
   - Select the desired line rate and set the Lane width to 4.
   - Choose the number of lanes and the location of the GT lane(s) that are connected to the trace.
- Select appropriate clock sources for the reference and initialization clocks.
- It is also important to select **Little Endian Support**.

4. On the second page of the configuration window, select **include Shared Logic in core**.

5. In the same block design, instantiate an AXI4-Stream Data FIFO. Set **TDATA Width** to four times the number of lanes. The FIFO is used for synchronisation between the Aurora user clock and the trace clock and to buffer data while the Aurora core is busy sending clock compensation sequences and frame delimiters. It is sufficient to make it 32 entries deep.
6. Connect the two components and export the pins as follows:

7. Locate the directory `~/demo/arm64/hardware/zynq_ultrascale/hdl/` from your TRACE32 installation and add the following files to the project:

   - `serial_trace_adapter_axis.vhd`
   - `stream_width_expander.vhd`
8. Modify the generated wrapper for the Zynq UltaScale+ MPSoC to include both block designs and an adapter from the HDL files imported in step 7:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity zynq_wrapper is
  generic (
    -- adapt this to match the FIFO configuration
    gBytes: positive := 8;
  );
  port (
    -- input signals
    iInitClk: in std_logic;
    iRst: in std_logic;
    iAuroraRefN: in std_logic;
    iAuroraRefP: in std_logic;
    -- output signals
    oAuroraTxN: out std_logic_vector(0 to gBytes - 1);
    oAuroraTxP: out std_logic_vector(0 to gBytes - 1)
  );
end entity;

architecture behavioral of zynq_wrapper is
  signal wTraceClkSdr: std_logic;
  signal wTraceDataSdr: std_logic_vector(31 downto 0);
  signal wFifoData: std_logic_vector(8*gBytes - 1 downto 0);
  signal wFifoValid: std_logic;
  signal wFifoLast: std_logic;
  signal wFifoKeep: std_logic_vector(gBytes - 1 downto 0);
  signal wRstN: std_logic;

begin
  yDesign: entity work.mpsoc port map (  
    TRACE_CLK_SDR => wTraceClkSdr,
    TRACE_DATA => wTraceDataSdr
  );

  yAdapter: entity work.serial_trace_adapter_axis generic map (  
    gBytes = gBytes
  ) port map (  
    iClk => wTraceClkSdr,
    iRst => iRst,
    iData => wTraceDataSdr,
    oData => wFifoData,
    oValid => wFifoValid,
    oLast => wFifoLast,
    oKeep => wFifoKeep
  );
  -- continued on next page
```
9. Finish your Vivado design and export the project to the SDK.

10. Generate or regenerate your FSBL (first-stage boot loader).

11. Do one of the following:
    - Either program the resulting FSBL to the boot device,
    - Or perform a debugger-based boot (see “Performing a Debugger-Based Boot on the Zynq UltraScale+”, page 37).

12. Use the following commands in your PRACTICE start-up script (*.cmm) to configure the trace:

```
TPIU.PortSize 32
TPIU.PortSize 1Lane ; or 2Lane
TPIU.PortMode Continuous
TPIU.PortClock 6250Mbps ; Vivado uses the term ‘lane rate’ here
TPIU.RefClock 1/20 ; to enable the PortClock/20
                   ; reference clock
```

If your design requires the reference clock from the debugger, you must make sure that the FPGA is reset or programmed after the debugger has been connected and configured.

You are now ready to debug and trace your target with TRACE32.
Use the following step-by-step procedure to produce an FPGA design that supports tracing the ARM cores via HSSTP at a lane rate of 10 Gbps. Because the Xilinx Aurora 8b10b core does not support this lane rate, this step-by-step procedure uses a custom Aurora encoder coupled with the UltraScale FPGAs transceivers wizard.

Note that the bandwidth of the trace infrastructure of the PS is limited to 1000 MB/s. Therefore, using a higher lane rate than 10 Gbps or multiple lanes does not provide any benefit.

We recommend that users first verify their hardware using the previous section, “Exporting the UltraScale+ Trace Interface via HSSTP (up to 6.25 Gbps)”, page 24, before following these steps.

1. Perform steps 1 through 6 as described in “Exporting the UltraScale+ Trace Interface via FPGA Fabric/PL”, page 21. Use a TPIU clock frequency of 250 MHz.
2. From the IP catalog, choose the UltraScale FPGAs Transceivers Wizard.
   - Select the preset GTH-Aurora_8B10B.
   - Set the line rate to 10 GB/s.
   - Choose a suitable reference clock. The debugger can provide a 500 MHz reference clock.
Choose **8B/10B** as the encoding.

Since the wizard does not have a simplex option, we also need to configure the receiver, even though it is not used. It is simplest to use the same settings as for the transmitter.

3. **On the Physical Resources tab**, select the channel and reference clock to be used. You also need to provide a free-running clock for initialization and specify its frequency.

4. **On the Structural Options tab**, set all auxiliary components to be included in the core.
5. Locate the directory `~/demo/arm64/hardware/zynq_ultrascale/hdl/` from your TRACE32 installation and add the following files to the project:

- `serial_trace_adapter_gt.vhd`
- `tpiu_to_stream.vhd`
- `fifo_inferred.vhd`
- `synchronizer.vhd`
- `util_pkg.vhd`
- `aurora_encoder.vhd`
- `aurora_idle_generator.vhd`
6. Modify the generated wrapper for the Zynq UltraScale+ MPSoC to include the Processing System block design, the transceiver IP generated in steps 2 to 4 and an adapter contained in the HDL files imported in step 5:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

library UNISIM;
use UNISIM.Vcomponents.all;

entity zynq_wrapper is
  port (  
    iInitClk: in  std_logic;
    iRst:    in  std_logic;
    iAuroraRefN: in  std_logic;
    iAuroraRefP: in  std_logic;
    oAuroraTxN: out std_logic_vector(0 to 0);
    oAuroraTxP: out std_logic_vector(0 to 0)
  );
end entity;

architecture behavioral of zynq_wrapper is
  signal wTraceClkSdr:  std_logic;
  signal wTraceDataSdr: std_logic_vector(31 downto 0);
  signal wAuroraRef:    std_logic;
  signal wUserClk:      std_logic;
  signal wTxData:       std_logic_vector(31 downto 0);
  signal wTxDataK:      std_logic_vector(3 downto 0);

  begin
    yDesign: entity work.mpsoc port map (  
      TRACE_CLK_SDR   => wTraceClkSdr,
      TRACE_DATA      => wTraceDataSdr
    );

    yAdapter: entity work.serial_trace_adapter_gt port map (  
      iTraceClk    => wTraceClkSdr,
      iTraceRst    => iRst,
      iUserClk     => wUserClk,
      iUserRst     => iRst,
      iData        => wTraceDataSdr,
      oData        => wTxData,
      oDataK       => wTxDataK
    );
    -- continued on next page
```
7. Finish your Vivado design and export the project to the SDK.

8. Generate or regenerate your FSBL (first-stage boot loader).

9. Do one of the following:
   - Either program the resulting FSBL to the boot device,
   - Or perform a debugger-based boot (see “Performing a Debugger-Based Boot on the Zynq

```vhdl
-- continued from previous page

entity work.gt_1lane is
  port map
   (gtwiz_reset_clk_freerun_in(0) => iInitClk,
    gtwiz_reset_all_in(0) => iRst,
    gtrenclk00_in(0) => wAuroraRef,
    gtwiz_userclk_tx_reset_in => (others => '0'),
    gtwiz_userclk_tx_usrclk2_out(0) => wUserClk,
    gtwiz_reset_tx_pll_and_datapath_in => (others => '0'),
    gtwiz_reset_tx_datapath_in => (others => '0'),
    gtwiz_userdata_tx_in => wTxData,
    tx8b10ben_in => (others => '1'),
    txctrl10_in => (others => '0'),
    txctrl11_in => (others => '0'),
    txctrl2_in(7 downto 4) => (others => '0'),
    txctrl2_in(3 downto 0) => wTxDataK,
    gthtxn_out => oAuroraHpc1TxN,
    gthtxp_out => oAuroraHpc1TxP,
    gtwiz_userclk_rx_reset_in => (others => '0'),
    gtwiz_reset_rx_pll_and_datapath_in => (others => '0'),
    gthrxn_in => (others => '0'),
    gthrxp_in => (others => '0'),
    rx8b10ben_in => (others => '0'),
    rxbufreset_in => (others => '0'),
    rxcommadeten_in => (others => '0'),
    rxmcommaalignen_in => (others => '0'),
    rpxcommaalignen_in => (others => '0')));

yBufAurora: IBUFDS_GTE4 port map
                (O => wAuroraRef,
                 I => iAuroraRefP,
                 IB => iAuroraRefN,
                 CEB => '0');

end architecture;
```
10. Use the following TPIU commands in your PRACTICE start-up script (*.cmm) to configure the trace:

```
TPIU.PortSize 32
TPIU.PortSize 1Lane
TPIU.PortMode Continuous
TPIU.PortClock 10000Mbps
TPIU.RefClock 1/20 ; if you require the 500 MHz reference clock
```

If your design requires the reference clock from the debugger, you must make sure that the FPGA is reset or programmed after the debugger has been connected and configured.

You are now ready to debug and trace your target with TRACE32.
Off-chip trace via PCIe differs from the other trace methods because it requires assistance from the target code. The debugger acts as a slave device to the PCIe root complex in the PS. The operating system running on the target must enumerate the debugger and assign an address to it.

This means that the debugger must already be ready and configured while the target is booting. Trace capture is only possible after the device has booted and the debugger must not be disconnected from the target while it is running.

1. Create a new Vivado project with an instance of the Zynq processing system.
2. Enter the configuration of the Zynq processing system.
3. Enable the PCIe root port and select the correct number of lanes. Using four lanes is possible, but will not increase the bandwidth available for trace.

4. Finish your Vivado design and export the project to the SDK.
5. Generate or regenerate your FSBL (first-stage boot loader).
6. When setting up the target's operating system, make sure that the appropriate drivers are enabled for PCIe support.
7. Verify that PCIe is operational using an expansion card such as a PCIe-to-USB converter.
8. Do one of the following:
   - Either program the resulting FSBL to the boot device,
   - Or perform a debugger-based boot (see “Performing a Debugger-Based Boot on the Zynq UltraScale+”, page 37).

You are now ready to debug and trace your target with TRACE32.
Using the Example Design for the ZCU102

For the ZCU102 evaluation board, Lauterbach provides an example Vivado project for download at www.lauterbach.com/support/static/armdemos/20170313ZCU102_trace_example.xpr.zip.

The design contains five top-level entities for different trace setups:

- top_parallel_8bit
- top_parallel_16bit
- top_hsstp_1lane_6250mbps
- top_hsstp_2lane_6250mbps
- top_hsstp_1lane_10000mbps

Use “Set as Top” on one of these entities in the project manager before generating a bitstream.

Parallel trace is exported on the P6 connector on the ZCU102, while serial trace requires an adapter to be plugged into the J4 port (HPC1). Contact Lauterbach to obtain this adapter.

Performing a Debugger-Based Boot on the Zynq UltraScale+

This section focuses on the **JTAG-BOOT** mode of the ZynqUltraScale+. In contrast to all other boot modes, this mode is only intended for development. The basic idea is that the CPUs will wait in an endless loop after executing the boot ROM, allowing the JTAG probe to perform all further initialization.

To perform a debugger-based boot:

1. Set the boot mode to **JTAG-BOOT** using the MIO lines.
2. Reset the SoC, for example by asserting the RESET line. Not all boards have a RESET line connected to the SoC, thus a power cycle or similar might be required.
3. Execute the boot ROM.
4. Load the FSBL boot code using the debugger.
5. Execute the FSBL boot code.
6. Optionally load the FPGA fabric using the debugger.

You are now ready to load the next stage boot loader, OS, etc., and to use the optional off-chip trace.

Example files following the above sequence are included in the TRACE32 installation directory under ~/demo/arm64/hardware/zynq_ultrascale.