Introduction

This document discusses the PIO Example Design and the Downstream Port Model that come with the generation of the Virtex®-5 Integrated PCI Express Block Plus core in the CORE Generator software. The main goal of this document is to provide detailed information on the architecture of the PIO Example Design and the simulation setup consisting of the Downstream Port Model.

The PIO Example Design simulation emulates the packet transaction between a Downstream Port and an endpoint. This document describes how the initialization process takes place, how the configuration transactions are initiated by the Downstream Port Model, and how normal memory read, memory write and I/O read write transactions are initiated by the host. The generation of a Completion packet by the endpoint is also covered.

The latter part of this document goes through the packet analysis of the TLPs generated by the Downstream Port Model and the corresponding Completions generated by the endpoint example design.

PIO Example Design

The PIO Example Design is a simple target-only application that interfaces with the endpoint for PCIe core's Transaction (TRN) interface.

Following are the main features of the PIO Example Design:

- Four transaction-specific 2 kb target region using the internal Xilinx FPGA block RAMs, providing a total target space of 8192 bytes.
- Supports single DWORD payload Read and Write PCI Express transactions to 32-/64-bit address memory spaces and I/O space with support for Completion TLPs.
- Utilizes the core's trn_rbar_hit_n[6:0] signals to differentiate between TLP destination Base Address Registers.
- Provides separate implementations optimized for 32-bit and 64-bit TRN interfaces.

Figure 1 shows different components in the PIO Example Design.
Downstream Port Model

The Downstream Port Model acts as a root complex, but it is not really a "root complex". The model represents only the downstream port interface which allows link training with the connecting endpoint. A complete root complex functionality represents a lot of other things. The Downstream Port Model and the backend testbench only provide just enough tools to configure the endpoint and issue read/write requests to the endpoint user application. The Downstream Port Model is not a full simulation model as a true Bus Functional Model (BFM) available from third-party vendors. However, it enables enough functionality to perform basic testing of the user design. Xilinx recommends obtaining a BFM for advanced design simulation and verification.

The Downstream Port Model initializes the core’s configuration space, creates TLP transactions, generates TLP logs, and provides an interface for creating and verifying tests.

Figure 2 shows a high-level architecture of the Downstream Port Model.
Files Hierarchy

Figure 3 shows the modules hierarchy of the PIO Example Design simulation setup. All the files shown are generated during the generation of the Virtex-5 FPGA Integrated PCI Express Block Plus Core. This hierarchy has been captured by creating an ISE project of the VHDL files provided in the PIO Example Design.

Downstream Port Model Architecture

The Downstream Port Model consists of the following components:
This block essentially acts as a root complex. However, the model should not be strictly treated as a root complex as it does not provide many features that a real root complex would normally provide. The Endpoint PCI Express Block Plus at the user side transmits TLPs across the PCI express link to the Downstream Port (dsport) model. The dsport and the PCI Express Block Plus core are responsible for the data link layer and physical layer processing when communicating across the PCI Express fabric.

dsport_cfg configures the Downstream Port Model.

**RX_APP (pci_exp_usrapp_rx.vhd)**

Following are different procedures defined in RX_APP:

- PROC_READ_DATA
- PROC_DECIPHER_FRAME
- PROC_3DW
- PROC_4DW
- PROC_PARSE_FRAME

**PROC_READ_DATA**

This procedure reads receive transaction data line (trn_rd) and stores it in frame_store_rx as shown below:

```vhdl
for i in 0 to (remain - 1) loop
    data_byte := trn_d( hi_index downto low_index);
    hi_index := hi_index - 8;
    low_index := low_index - 8;
    frame_store_rx(frame_store_rx_idx) := data_byte;
    frame_store_rx_idx := frame_store_rx_idx + 1;
end loop;
end PROC_READ_DATA;
```
PROC_DECIPHER_FRAME

This procedure extracts the information from the data collected by PROC_READ_DATA as shown below:

```plaintext
fmt := frame_store_rx(0)(6 downto 5);
tlp_type := frame_store_rx(0)(4 downto 0);
traffic_class := frame_store_rx(1)(6 downto 4);
td := frame_store_rx(2)(7);
ep := frame_store_rx(2)(6);
attr := frame_store_rx(2)(5 downto 4);
length(9 downto 8) := frame_store_rx(2)(1 downto 0);
length(7 downto 0) := frame_store_rx(3);
end PROC_DECIPHER_FRAME;
```

PROC_3DW/PROC_4DW

These procedures print the frame information to the output log file as shown below:

![Diagram](image)

**Figure 5 - Downstream Port Model Output Logs**
fmt_type := fmt & tlp_type;
case (fmt_type) is

when PCI_EXP_CFG_READ0 | PCI_EXP_CFG_WRITE0 =>

    requester_id := frame_store_rx(4) & frame_store_rx(5);
    tag := frame_store_rx(6);
    byte Enables := frame_store_rx(7);
    completer id := frame_store_rx(8) & frame_store_rx(9);
    register_address(9 downto 8) := frame_store_rx(10)(1 downto 0);
    register_address(7 downto 0) := frame_store_rx(11);

    writeHexToRx ("Requester Id: 0x", requester_id);
    writeHexToRx ("Tag: 0x", tag);
    writeHexToRx ("Last and First Byte Enables: 0x", byte Enables);
    writeHexToRx ("Completer Id: 0x", completer id);
    writeHexToRx ("Register Address: 0x", "00" & register_address);

PROC_PARSE_FRAME

PROC_PARSE_FRAME calls PROC_DECIPHER_FRAME, PROC_4DW and PROC_3DW (last two writes to the tx.dat and rx.dat file).

The following code gives the RX_APP state machine:

process (trn_clk, trn_reset_n)
begin

    if (trn_reset_n = '0') then
        trn_rx_state <= TRN_RX_RESET;
        frame_store_rx_idx := 0;
        rx_tx_read_data <= X"FFFFFFF";
        read_data_valid_int <= '0';
    else
        if (trn_clk'event and trn_clk = '1') then
            case (trn_rx_state) is
                when TRN_RX_RESET =>
                    if (trn_reset_n = '0') then
                        trn_rx_state <= TRN_RX_RESET;
                    else
                        trn_rx_state <= TRN_RX_DOWN;
                    end if;
                when TRN_RX_DOWN =>
                    if (trn_lnk_up_n = '1') then
                        trn_rx_state <= TRN_RX_DOWN;
                    else
                        trn_rx_state <= TRN_RX_IDLE;
                    end if;
    end if;
end process;
when TRN_RX_IDLE =>
  read_data_valid_int <= '0';
  if (trn_reset_n = '0') then
    trn_rx_state <= TRN_RX_RESET;
  elsif (trn_lnk_up_n = '1') then
    trn_rx_state <= TRN_RX_DOWN;
  elsif ((trn_rsof_n = '0') and (trn_rsrdc_rdy_n = '0')
    and (trn_rdst_rdy_n_c = '0')) then
    PROC_READ_DATA (0, trn_rd, trn_rrem_n);
    trn_rx_state <= TRN_RX_ACTIVE;
  else
    trn_rx_state <= TRN_RX_IDLE;
  end if;
when TRN_RX_ACTIVE =>
  if (trn_reset_n = '0') then
    trn_rx_state <= TRN_RX_RESET;
  elsif (trn_lnk_up_n = '1') then
    trn_rx_state <= TRN_RX_DOWN;
  elsif ((trn_rsrdc_rdy_n = '0') and (trn_reof_n = '0')
    and (trn_rdst_rdy_n_c = '0')) then
    PROC_READ_DATA (1, trn_rd, trn_rrem_n);
    PROC_PARSE FRAME (rx_tx_read_data, read_data_valid_int);
    trn_rx_state <= TRN_RX_IDLE;
  elsif ((trn_rsrdc_rdy_n = '0') and (trn_rdst_rdy_n_c = '0')) then
    PROC_READ_DATA (0, trn_rd, trn_rrem_n);
    trn_rx_state <= TRN_RX_ACTIVE;
  elsif ((trn_rsrdc_rdy_n = '0') and (trn_reof_n = '0')
    and (trn_rsrdc_dsc_n = '0')) then
    PROC_READ_DATA (1, trn_rd, trn_rrem_n);
    PROC_PARSE FRAME (rx_tx_read_data, read_data_valid_int);
The `usrapp_tx` block sends TLPs to the Downstream Port Model block for transmission across the PCI Express Link to the Endpoint Design Under Test (DUT). Transaction sequences or test programs are initiated by the `usrapp_tx` block to stimulate the endpoint device's fabric interface.

All test programs are defined inside the `test_interface.vhd`. All transaction sequences are defined in the `tests.vhd` file.

There are different tests that you can perform based on the VHDL or Verilog version of the core.

```vhdl
trn_rx_state <= TRN_RX_SRC_DSC;
else
    trn_rx_state <= TRN_RX_ACTIVE;
end if;
when TRN_RX_SRC_DSC =>
    if (trn_reset_n = '0') then
        trn_rx_state <= TRN_RX_RESET;
    elsif (trn_lnk_up_n = '1') then
        trn_rx_state <= TRN_RX_DOWN;
    else
        trn_rx_state <= TRN_RX_IDLE;
    end if;
when others =>
    trn_rx_state <= TRN_RX_RESET;
end case;
end if;
end if;
end process;
```

**TX_APP (pci_exp_usrapp_tx.vhd)**

The `usrapp_tx` block sends TLPs to the Downstream Port Model block for transmission across the PCI Express Link to the Endpoint Design Under Test (DUT). Transaction sequences or test programs are initiated by the `usrapp_tx` block to stimulate the endpoint device's fabric interface.

All test programs are defined inside the `test_interface.vhd`. All transaction sequences are defined in the `tests.vhd` file.

There are different tests that you can perform based on the VHDL or Verilog version of the core.
Table 1 provides the details of the entire test suite that can be performed with the Downstream Port Model.
<table>
<thead>
<tr>
<th>Test Name</th>
<th>Test in VHDL/Verilog</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_smoke_test0</td>
<td>Verilog and VHDL</td>
<td>Issues a PCI Type 0 Configuration Read TLP and waits for the completion TLP; then compares the value returned with the expected Device/Vendor ID value.</td>
</tr>
<tr>
<td>sample_smoke_test1</td>
<td>Verilog</td>
<td>Performs the same operation as sample_smoke_test0 but makes use of expectation tasks. This test uses two separate test program threads: one thread issues the PCI Type 0 Configuration Read TLP and the second thread issues the Completion with Data TLP expectation task. This test illustrates the form for a parallel test that uses expectation tasks. This test format allows for confirming reception of any TLPs from the customer’s design. Additionally, this method can be used to confirm reception of TLPs when ordering is unimportant.</td>
</tr>
<tr>
<td>pio_writeReadBack_test0</td>
<td>Verilog and VHDL</td>
<td>Transmits a 1 DWORD Write TLP followed by a 1 DWORD Read TLP to each of the example design’s active BARs, and then waits for the Completion TLP and verifies that the write and read data match. The test will send the appropriate TLP to each BAR based on the BARs address type (for example, 32 bit or 64 bit) and space type (for example, IO or Memory).</td>
</tr>
<tr>
<td>pio_testByteEnables_test0</td>
<td>Verilog</td>
<td>Issues four sequential Write TLPs enabling a unique byte enable for each Write TLP, and then issues a 1 DWORD Read TLP to confirm that the data was correctly written to the example design. The test will send the appropriate TLP to each BAR based on the BARs address-type (for example, 32 bit or 64 bit) and space type (for example, IO or Memory).</td>
</tr>
<tr>
<td>pio_memTestDataBus</td>
<td>Verilog</td>
<td>Determines if the PIO design’s FPGA block RAMs data bus interface is correctly connected by performing a 32-bit walking ones data test to the first available BAR in the example design.</td>
</tr>
<tr>
<td>pio_memTestAddrBus</td>
<td>Verilog</td>
<td>Determines whether the PIO design’s FPGA block RAM’s address bus interface is correctly connected by performing a walking ones address test. This test should only be called after successful completion of pio_memTestDataBus.</td>
</tr>
<tr>
<td>pio_memTestDevice</td>
<td>Verilog</td>
<td>Checks the integrity of each bit of the PIO design’s FPGA block RAM by performing an increment/decrement test. This test should only be called after successful completion of pio_memTestAddrBus.</td>
</tr>
<tr>
<td>pio_timeoutFailureExpected</td>
<td>Verilog</td>
<td>Sends a Memory 32 Write TLP followed by Memory 32 Read TLP to an invalid address and waits for a Completion with data TLP. This test anticipates that waiting for the completion TLP times out and illustrates how the test programmer can gracefully handle this event.</td>
</tr>
<tr>
<td>pio_tlp_test0 (illustrative example only)</td>
<td>Verilog</td>
<td>Issues a sequence of Read and Write TLPs to the example design’s RX interface. Some of the TLPs, for example, burst writes, are not supported by the PIO design.</td>
</tr>
</tbody>
</table>
All Downstream Port Model tests follow the same six steps as listed below:

1. Perform conditional comparison of a unique test name.
2. Set up master timeout in case of simulation hangs.
3. Wait for Reset and link-up.
4. Initialize the configuration space of the endpoint.
5. Transmit and receive TLPs between the Downstream Port Model and the Endpoint DUT.
6. Verify that the test succeeded.

An entire source code for TX_APP where needed is presented here along with a relevant description of the procedure. The main objective behind publishing the source code for all procedures defined in the TX_APP is to allow readers to get familiar with the working mechanism of the PIO Example Design without having to generate the core and browse through the source code and the user guide.

pio_writeReadBack_test0 (tests.vhd)

This section provides a description of pio_writeReadBack_test0 test suite provided in the Downstream Port Model. The code consists of a number of procedure calls. An explanation on what each procedure does is given below.

```
-- Testname: pio_write_readBack_test0

elsif (test_selector = String'("pio_writeReadBack_test0")) then
    writeNowToScreen(String'("Running pio_writeReadBack_test0"));
    PROC_SYSTEM_INITIALIZATION(trn_reset_n, trn_lnk_up_n);

PROC_SYSTEM_INITIALIZATION causes the test program to wait for the system reset to deassert as well as the endpoint's trn_lnk_up_n signal to assert. This is an indication that the endpoint is ready to be configured by the test program via the Downstream Port Model.

PROC_BAR_INIT performs a series of Type 0 Configuration Writes and Reads to the endpoint core's Configuration Space, determines the memory and I/O requirements of the endpoint, and then programs the endpoint's Base Address Registers so that it is ready to receive TLPs from the Downstream Port Model.

PROC_BAR_INIT (tx_rx_read_data_valid, rx_tx_read_data_valid, tx_rx_read_data, trn_cd_c,
    trn_tsof_n, trn_teof_n, trn_trem_n_c, trn_tsrc_rdy_n,
    trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
    PROC_TX_CLK_EAT(300, trn_clk);

PROC_BAR_INIT performs a series of Type 0 Configuration Writes and Reads to the endpoint core's Configuration Space, determines the memory and I/O requirements of the endpoint, and then programs the endpoint's Base Address Registers so that it is ready to receive TLPs from the Downstream Port Model.

In the source code shown below, the sample program work together to cycle through all the endpoint's BARs and determine whether they are enabled, and if so, determines their type (e.g., Mem32, Mem64, or I/O).

The BAR can be checked to see if it is enabled or not by probing the BAR_ENABLED[] global array. A non-zero value indicates that the corresponding BAR is enabled. If the BAR is not enabled, then the test program will move on to check the next BAR.

PROC_BAR_INIT fills in the appropriate values into the BAR_ENABLED[ ] array.

If the array element is enabled (that is, non-zero), the element's value indicates the BAR type. A value of 1, 2, and 3 indicate I/O, Memory 32, and Memory 64 spaces respectively.
for i in 0 to 6 loop
    case BAR_ENABLED(i) is
        when 1 =>
            -- Test PIO IO Space
            write (Lglobal, String("Testing PIO IO Space BAR "));
            hwrite (Lglobal, std_logic_vector(to_unsigned(i, 4)));
            writeln (output, Lglobal);
            PROC_TX_IO_WRITE (X"00", BAR(i)(31 downto 0), X"F", X"DEADBEEF",
                trn_td_c, trn_tsof_n, trn_tsof_n, trn_trem_n_c, trn_tarc_rdy_n,
                trn_link_up_n, trn_tdst_rdy_n, trn_clk);
            P_READ_DATA := X"FFFFFFFF"
            PROC_TX_IO_READ (X"01", BAR(i)(31 downto 0), X"F",
                trn_td_c, trn_tsof_n, trn_tsof_n, trn_trem_n_c,
                trn_tarc_rdy_n, trn_link_up_n,
                trn_tdst_rdy_n, trn_clk);
            PROC_WAIT_FOR_READ_DATA (tx_rx_read_data_valid, rx.tx_read_data_valid,
                rx.tx_read_data, trn_clk);
            if (P_READ_DATA = X"DEADBEEF") then
                writeNowToScreen(String("Test Passed. Completion Data = 0xDEADBEEF "));
            else
                success := false;
                writeNowToScreen(String("Test Failed. Completion Data != 0xDEADBEEF. "));
                write (LGlobal, String("Data = 0x"));
                hwrite(LGlobal, P_READ_DATA);
                writeln (output, LGlobal);
            end if;
            PROC_TX_CLK_BAT(100, trn_clk);
        when 2 =>
            -- Test PIO Mem32 Space
            write (Lglobal, String("Testing PIO Mem32 Space BAR "));
            hwrite (Lglobal, std_logic_vector(to_unsigned(i, 4)));
            writeln (output, LGlobal);
            DATA_STORE(0) := X"01";
            DATA_STORE(1) := X"02";
            DATA_STORE(2) := X"03";
            DATA_STORE(3) := X"04";
            PROC_TX_MEMORY_WRITE_32 (X"02", "000", "00000000001", BAR(i)(31 downto 0), X"0", X"F",'0',
                trn_td_c, trn_tsof_n, trn_tsof_n, trn_trem_n_c,
                trn_tarc_rdy_n, trn_terfwd_n, trn_link_up_n, trn_tdst_rdy_n, trn_clk);
P_READ_DATA := X"FFFFFFFF";

PROC_TX_MEMORY_READ_32 ( 
  X"03", "00", "0000000001", BAR(i)(31 downto 0), X"0", X"F", 
  trn_td_c, trn_tsof_n, trn_teof_n, trn_trem_n_c, 
  trn_tsrc_rdy_n, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
PROC_WAIT_FOR_READ_DATA (tx_rx_read_data_valid, 
  rx_tx_read_data_valid, rx_tx_read_data, trn_clk);
if (P_READ_DATA = X"04030201") then 
  writelnNowToScreen(String("Test Passed. Completion Data = 0x04030201"); 
else 
  success := false;
  writelnNowToScreen(String("Test Failed. Completion Data != 0x04030201"); 
  write (LGlobal, String("Data = 0x");
  hwrite(LGlobal, P_READ_DATA);
  writeln (output, LGlobal);
end if;
PROC_TX_CLK_EAT(100, trn_clk);
when 3 =>
-- Test PIO Mem64 Space
write (LGlobal, String("Testing PIO Mem64 Space BAR "); 
  hwrite(LGlobal, std_logic_vector(to_unsigned(i, 4)));
  writeln (output, LGlobal);
DATA_STORE(0) := X"64";
DATA_STORE(1) := X"63";
DATA_STORE(2) := X"62";
DATA_STORE(3) := X"61";
PROC_TX_MEMORY_WRITE_64 { 
  X"04", "00", "0000000001", BAR(i+1)(31 downto 0) & 
  BAR(i)(31 downto 0), X"0", X"F", X"0", 
  trn_td_c, trn_tsof_n, trn_teof_n, trn_trem_n_c, 
  trn_tsrc_rdy_n, trn_terrfwd_n, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);

P_READ_DATA := X"FFFFFFFF";
PROC_TX_MEMORY_READ_64 ( 
  X"05", "00", "0000000001", BAR(i+1)(31 downto 0) & 
  BAR(i)(31 downto 0), X"0", X"F", 
  trn_td_c, trn_tsof_n, trn_teof_n, trn_trem_n_c, 
  trn_tsrc_rdy_n, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
PROC_WAIT_FOR_READ_DATA (tx_rx_read_data_valid, 
  rx_tx_read_data_valid, rx_tx_read_data, trn_clk);
if (P_READ_DATA = X"61626364") then 
  writelnNowToScreen(String("Test Passed. Completion Data = 0x61626364"); 
else 
  success := false;
  writelnNowToScreen(String("Test Failed. Completion Data != 0x61626364"); 
  write (LGlobal, String("Data = 0x");
  hwrite(LGlobal, P_READ_DATA);
  writeln (output, LGlobal);
test_interface.vhd

All procedures called in tests.vhd are defined in test_interface.vhd. The following procedures are defined in this file:

PROC_SYSTEM_INITIALIZATION
PROC_BAR_INIT
PROC_BAR_SCAN
PROC_BUILD_PCIE_MAP
PROC_DISPLAY_PCIE_MAP
PROC_BAR_PROGRAM
PROC_TX_SYNCHRONIZE
PROC_TX_TYPE0_CONFIGURATION_WRITE
PROC_TX_TYPE1_CONFIGURATION_READ
PROC_READ_DATA
PROC_PARSE_FRAME
PROC_DECIPHER_FRAME
PROC_4DW
PROC_TX_IO_WRITE
PROC_TX_IO_READ
PROC_TX_MEMORY_WRITE_32
PROC_TX_MEMORY_READ_32
PROC_WAIT_FOR_READ_DATA
PROC_TX_MEMORY_WRITE_64
PROC_TX_MEMORY_READ_64

PROC_SYSTEM_INITIALIZATION

PROC_SYSTEM_INITIALIZATION waits for transaction interface reset and linkup between the Downstream Port Model and the Endpoint DUT. This task must be invoked prior to endpoint core initialization.

PROC_BAR_INIT

PROC_BAR_INIT performs a series of Type 0 Configuration Writes and Reads to the endpoint core's configuration space, determines the memory and I/O requirements of the endpoint, and then programs the endpoint's Base Address Registers (BAR) so that it is ready to receive TLPs from the Downstream Port Model.
PROC_BAR_SCAN

This procedure performs a sequence of PCI Type 0 Configuration Writes and Configuration Reads to determine the memory and I/O requirements for the endpoint. It stores this information in the global array BAR_RANGE[].

procedure PROC_BAR_SCAN  ( 
    signal tx_rx_read_data_valid : out std_logic;
    signal rx_tx_read_data_valid : in std_logic;
    signal rx_tx_read_data : in std_logic_vector (31 downto 0);
    signal trn_td_c : out std_logic_vector(63 downto 0);
    signal trn_tsof_n : out std_logic;
    signal trn_teof_n : out std_logic;
    signal trn_trem_n_c : out std_logic_vector(7 downto 0);
    signal trn_tsrc_rdy_n : out std_logic;
    signal trn_lnk_up_n : in std_logic;
    signal trn_tdst_rdy_n : in std_logic;
    signal trn_clk : in std_logic
) is

    variable P_ADDRESS_MASK : std_logic_vector((32 - 1) downto 0);
    variable L : line;
    variable DEFAULT_TAG : std_logic_vector(( 8 - 1) downto 0);

begin

    -- TRN MODEL Initialization

    BAR_RANGE (0) := X"FFFFFF804";
    BAR_RANGE(1) := X"FFFFFFFF";
    BAR_RANGE (2) := X"FFFFF800";
    BAR_RANGE(3) := X"00000000";
    BAR_RANGE (4) := X"00000000";
    BAR_RANGE (5) := X"00000000";
    BAR_RANGE (6) := X"FFF00001";

end PROC_BAR_SCAN;
PROC_BUILD_PCIE_MAP

PROC_BUILD_PCIE_MAP performs memory /IO mapping algorithm and allocates Memory-32, Memory-64, and I/O space based on the endpoint's requirement. This procedure has been customized to work in conjunction with the limitations of the PIO Example Design and should only be called after Completion of PROC_BAR_SCAN.

This procedure also checks whether the BAR_RANGE has been defined or not. If it is defined, then the corresponding BAR is enabled. If it is not defined, the BAR is disabled.

```
procedure PROC_BUILD_PCIE_MAP
is
  variable i     : INTEGER;
  variable L     : line;
  variable RANGE_VALUE : std_logic_vector(31 downto 0);
begin
  writelnToScreen(String("PCI EXPRESS BAR MEMORY/IO MAPPING PROCESS BEGAN."));

  BAR(0) := '0' & X"10000000";
  BAR(1) := '0' & X"20000000";
  BAR(2) := '0' & X"30000000";
  BAR(3) := '0' & X"40000000";
  BAR(4) := '0' & X"50000000";
  BAR(5) := '0' & X"60000000";
  BAR(6) := '0' & X"70000001";  -- bit 0 must be set to enable the EROM

  i := 0;
  while (i <= 6) loop
    RANGE_VALUE := BAR_RANGE(i);
    if (RANGE_VALUE = X"00000000") then
      BAR_ENABLED(i) := 0;  -- Disabled
      BAR(i) := '0' & X"00000000";
    elsif ((RANGE_VALUE(0) = '1') and (i /= 6)) then
      BAR_ENABLED(i) := 1;  -- IO
      NUMBER_OF_IO_BARS := NUMBER_OF_IO_BARS + 1;
      if (pio_check_design and (NUMBER_OF_IO_BARS > 1)) then
        writeln(String("Warning: PIO design only supports 1 IO BAR. Testbench will disable BAR"));
        write(L, std_logic_vector(to_unsigned(i, 4)));
        writeln (output, L);
        BAR_ENABLED(i) := 0;  -- Disabled
      end if;
    else
      if (RANGE_VALUE(2) = '1') then
        BAR_ENABLED(i) := 3;  -- Mem64
        BAR_ENABLED(i+1) := 0;  -- Mem64 uses upper BAR so set as disabled
        NUMBER_OF_MEM64_BARS := NUMBER_OF_MEM54_BARS + 1;
        if (pio_check_design and (NUMBER_OF_MEM64_BARS > 1)) then
          writeln(String("Warning: PIO design only supports 1 MEM64 BAR. Testbench will disable BAR"));
        end if;
```
hwrite(L, std_logic_vector(to_unsigned(i, 4)));
write_line (output, L);
BAR_ENABLED(i) := 0; -- Disabled
end if;
i := i + 1;
else
if (i /= 6) then
  NUMBER_OF_MEM32_BARS := NUMBER_OF_MEM32_BARS + 1;
end if;
BAR_ENABLED(i) := 2; -- Mem32
if (pio_check_design and (NUMBER_OF_MEM32_BARS > 1)) then
  write (L, String'("Warning: PIO design only supports 1
    MEM32 BAR. Testbench will disable BAR"));
  hwrite(L, std_logic_vector(to_unsigned(i, 4)));
  write_line (output, L);
  BAR_ENABLED(i) := 0; -- Disabled
end if;
end if;
i := i + 1;
end loop;
end PROC_BUILD_PCIE_MAP;
PROC_BAR_PROGRAM

procedure PROC_BAR_PROGRAM (  

  signal trn_td_c  : out std_logic_vector(63 downto 0);  
  signal trn_tsof_n : out std_logic;  
  signal trn_teof_n : out std_logic;  
  signal trn_trem_n_c : out std_logic_vector(7 downto 0);  
  signal trn_tsrc_rdy_n : out std_logic;  
  signal trn_lnk_up_n : in std_logic;  
  signal trn_tdst_rdy_n : in std_logic;  
  signal trn_clk : in std_logic  
) is

  variable L : line;  
  variable DEFAULT_TAG : std_logic_vector({ 8 - 1} downto 0);  

begin

  DEFAULT_TAG := X"0f";

  write (L, String"("[ "") ; write (L, now);  
  write (L, String"(" [ Setting Core Configuration Space. .."));  
  writeln (output, L);

  -- Program BAR0

  PROC_TX_TYPE0_CONFIGURATION_WRITE (  
    DEFAULT_TAG, --tag : in std_logic_vector (7 downto 0);  
    X"010", --reg_addr 12'h10  
    BAR(0) (31 downto 0), --reg_data : in std_logic_vector (31 downto 0);  
    X"F", --first_dw_be : in std_logic_vector (3 downto 0);  
    trn_td_c, trn_tsof_n, trn_teof_n, trn_trem_n_c, trn_tsrc_rdy_n,  
    trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);  
  DEFAULT_TAG := X"10";
  PROC_TX_CLK_EAT(100, trn_clk);
PROC_TX_SYNCHRONIZE

The main function of this procedure is to synchronize the trn_clk and trn_tdst_rdy_n signal. Before a TLP is transferred, it waits for trn_clk positive edge and trn_tdst_rdy_n to be asserted.

```cpp
wait until (trn_clk'event and trn_clk = '1');
if ((trn_tdst_rdy_n = '1') and (first = 1)) then
  while (trn_tdst_rdy_n = '1') loop
    wait until (trn_clk'event and trn_clk = '1');
  end loop;
end if;

PROC_TX_SYNCHRONIZE calls PROC_READ_DATA and PROC_PARSE_FRAME. PROC_PARSE_FRAME calls PROC_DECIPHER_FRAME, PROC_4DW and PROC_3DW. These procedures log the outgoing TLPs into the output log file (i.e., tx.dat) which is shown in the following source code snippet:

```cpp
if (first = 1) then
  if (trn_trem_n_c = X"00") then ---"00000000"
    last := 0;
  else
    last := 1;
  end if;
  PROC_READ_DATA(last, trn_td_c, trn_trem_n_c);
end if;
if (last_call = 1) then
  PROC_PARSE_FRAME;
end if;
end PROC TX SYNCHRONIZE:
```

PROC_TX_TYPE0_CONFIGURATION_WRITE

This procedure sends a Type 0 PCI Express Configuration Write TLP from the Downstream Port Model to reg_addr_ of the endpoint with tag_ and first_dw_be_ inputs. The Completion for the Configuration Write TLP returned from endpoint uses contents of global COMPLETE_ID_CFG as the Completion ID.

The inputs to this procedure are as follows:

```cpp
tag : in std_logic_vector (7 downto 0);
reg_addr : in std_logic_vector (11 downto 0);
reg_data : in std_logic_vector (31 downto 0);
first_dw_be : in std_logic_vector (3 downto 0);
```

**Figure 6 - Request Header Format for Configuration Transactions**
First of all, PROC_TX_SYNCHRONIZE is called to synchronize the trn_clk and trn_tdst_rdyn_n signals. After that, the TLP information is put on trn_td_c. The second PROC_TX_SYNCHRONIZE call synchronizes the signals, as well as logs the TLP information into local buffer to be parsed and sent to the output log. The entire outgoing TLP information is sent to the output log after the transmission of the last TLP data.

```
PROC_TX_SYNCHRONIZE(0, 0, trn_lnk_up_n, trn_tdst_rdyn_n, trn_clk);
trn_td_c <=
  '0' &
  "10" &
  "00100" &
  '0' &
  "000" &
  "0000" &
  '0' &
  '0' &
  "00" &
  "00" &
  "000000001" &
  COMPLETE_ID_CFG &
tag(7 downto 0) &
  "0000" &
  first_dw_be(3 downto 0);
trn_tsof_n <= '0';
trn_tsac_rdy_n <= '0';
PROC_TX_SYNCHRONIZE(1,0, trn_lnk_up_n, trn_tdst_rdyn_n, trn_clk);

trn_td_c <= COMPLETE_ID_CFG &
  "0000" &
  reg_addr(11 downto 2) &
  "00" &
  reg_data(7 downto 0) &
  reg_data(15 downto 8) &
  reg_data(23 downto 16) &
  reg_data(31 downto 24);
trn_tsof_n <= '1';
trn_teof_n <= '0';
trn_trem_n_c <= X"00";
PROC_TX_SYNCHRONIZE(1, 1, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
trn_teof_n <= '1';
trn_tsac_rdy_n <= '1';
end PROC_TX_TYPE0_CONFIGURATION_WRITE;
```

**PROC_TX_TYPE1_CONFIGURATION_READ**

This procedure sends a Type 1 PCI Express Configuration Read TLP from the Downstream Port Model to reg_addr_ of the endpoint with tag_ and first_dw_be_ as the inputs. CplID (Completion with Data) returned from the endpoint uses COMPLETE_ID_CFG as the Completion ID.

The definition of this procedure is the same as for PROC_TX_TYPE0_CONFIGURATION_WRITE.
PROC_READ_DATA / PROC_PARSE_FRAME / PROC_DECIPHER_FRAME / PROC_4DW

These are general procedures that are used to log the TLP information in the output log (i.e., tx.dat and rx.dat).

PROC_TX_IO_WRITE

This procedure sends a PCI Express I/O Write TLP from the Downstream Port Model to IO address addr[31:2] of the endpoint. The CplD returned from the endpoint uses the contents of global COMPLETE_ID_CFG as the Completion ID. The code snippet for PROC_TX_IO_WRITE is as follows:

```vhdl
PROC_TX_SYNCHRONIZE(0, 0, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
trn_td_c <= '0' &
    "10" &
    "0010" &
    '0' &
    "0000" &
    "0000" &
    '0' &
    '0' &
    "00" &
    "000000001" &
    COMPLETER_ID_CFG &
    tag(7 downto 0) &
    "0000" &
    first_dw_be(3 downto 0);

trn_tsof_n <= '0';
trn_tsof_n <= '1';
trn_trem_n_c <= X"00";
trn_tarc_rdy_n <= '0';
```

Figure 7 - Request Header Format for I/O Transactions
PROC_TX_MEMORY_WRITE_32

The inputs for this procedure are as follows:

| tag      | : in std_logic_vector (7 downto 0); |
| tc       | : in std_logic_vector (2 downto 0); |
| len      | : in std_logic_vector (9 downto 0); |
| addr     | : in std_logic_vector (31 downto 0); |
| last_dw_be | : in std_logic_vector (3 downto 0); |
| first_dw_be | : in std_logic_vector (3 downto 0); |
| ep       | : in std_logic; |

Figure 8 shows the header format for 32-bit address memory write TLP.

The code snippet for the PROC_TX_MEMORY_WRITE_32 is shown below:
if (len = "0000000000") then
    length := "1000000000";  --1024
else
    length := len;
end if;

PROC_TX_SYNCHRONIZE(0, 0, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
trn_td_c <= '0' &
    "10" &
    "00000" &
    '0' &
    tc(2 downto 0) &
    "0000" &
    '0' &
    '0' &
    "00" &
    "00" &
    len(9 downto 0) &
    COMPLETER_ID_CFG &
    tag(7 downto 0) &
    last_dw_be(3 downto 0) &
    first_dw_be(3 downto 0);

trn_tsof_n <= '0';
trn_teof_n <= '1';
trn_trem_n_c <= X"00";
trn_tsrc_rdy_n <= '0';

PROC_TX_SYNCHRONIZE(1,0, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
trn_td_c <= addr(31 downto 2) &
    "00" &
    DATA_STORE(0) &
    DATA_STORE(1) &
    DATA_STORE(2) &
    DATA_STORE(3);

trn_tsof_n <= '1';
if (length /= "0000000001") then
    unsigned_length := unsigned(length);
    int_length := to_integer(unsigned_length);
i := 4;
while [i < (int_length * 4)) loop
    PROC_TX_SYNCHRONIZE(1,0, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
PROC_WAIT_FOR_READ_DATA

This procedure waits for the next Completion with data TLP that is sent by the Endpoint DUT. On a successful
Completion, the first DWORD of data in the CplD is stored in the global P_READ_DATA. This task should be called
immediately following any of the read tasks in the TLP that request Completion with Data TLPs to avoid any race
conditions.

By default, this task will locally timeout and terminate the simulation after 1000 transaction interface clocks. The global
cpld_to_finish can be set to zero so that local timeout returns execution to the calling test and does not result in simulation
timeout. For this case, test programs should check the global cpld_to, which when set to one indicates that this task has
timed out and that the contents of P_READ_DATA are invalid.

Xilinx Answer 46888 – Virtex-5 Endpoint Block Plus - Debugging and Packet Analysis Guide
PROC_WAIT_FOR_READ_DATA is called from the main test (pio_writeReadBack_test0) code as follows:

```c
PROC_TX_MEMORY_READ_64 {
   X"05", "000", "0000000001", BAR(i+1)(31 downto 0) &
   BAR(i)(31 downto 0), X"0", X"P",
   trn_td_c, trn_tsof_n, trn_tsof_n, trn_trem_n_c,
   trn_tsrc_rdy_n, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
PROC_WAIT_FOR_READ_DATA (tx_rx_read_data_valid,
   rx_tx_read_data_valid, rx_tx_read_data, trn_clk);
if (P_READ_DATA = X"61626364") then
   writeNowToScreen(String'("Test Passed. Completion Data = 0x61626364"))
```

**PIO Example Design Packet Analysis**

In the previous section, the pio_writeReadBack_test0 test provided in the PIO Example Design was discussed. In this section, a detailed description of TLP packet analysis will be presented by simulating the example design based on pio_writeReadBack_test0 Downstream Port Model test suite. The example design work flow mechanism is shown in Figure 9.
Figure 9 - PIO Example Design Working Mechanism
Fmt and Type

To perform PCI Express packet analysis, it is important to understand the Fmt and Type value in a TLP a. Table 2 and Table 3 show the Fmt and Type encodings.

### Table 2 - Fmt[1:0] Field Values

<table>
<thead>
<tr>
<th>Fmt[1:0]</th>
<th>Corresponding TLP Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>3 DW header, no data</td>
</tr>
<tr>
<td>01b</td>
<td>4 DW header, no data</td>
</tr>
<tr>
<td>10b</td>
<td>3 DW header, with data</td>
</tr>
<tr>
<td>11b</td>
<td>4 DW header, with data</td>
</tr>
</tbody>
</table>

### Table 3 - Fmt [1:0] and Type [4:0] Field Encodings

<table>
<thead>
<tr>
<th>TLP Type</th>
<th>Fmt [1:0]</th>
<th>Type [4:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRd</td>
<td>00</td>
<td>0 0000</td>
<td>Memory Read Request</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRdLk</td>
<td>00</td>
<td>0 0001</td>
<td>Memory Read Request-Locked</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MWr</td>
<td>10</td>
<td>0 0000</td>
<td>Memory Write Request</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IORd</td>
<td>00</td>
<td>0 0010</td>
<td>I/O Read Request</td>
</tr>
<tr>
<td>IOWr</td>
<td>10</td>
<td>0 0010</td>
<td>I/O Write Request</td>
</tr>
<tr>
<td>CfgRd0</td>
<td>00</td>
<td>0 0100</td>
<td>Configuration Read Type 0</td>
</tr>
<tr>
<td>CfgWr0</td>
<td>10</td>
<td>0 0100</td>
<td>Configuration Write Type 0</td>
</tr>
<tr>
<td>CfgRd1</td>
<td>00</td>
<td>0 0101</td>
<td>Configuration Read Type 1</td>
</tr>
<tr>
<td>CfgWr1</td>
<td>10</td>
<td>0 0101</td>
<td>Configuration Write Type 1</td>
</tr>
<tr>
<td>Msg</td>
<td>01</td>
<td>1 0r2, r3</td>
<td>Message Request – The sub-field r[2:0] specifies the Message routing mechanism (see Table 2-11).</td>
</tr>
<tr>
<td>MsgD</td>
<td>11</td>
<td>1 0r2, r3</td>
<td>Message Request with data payload – The sub-field r[2:0] specifies the Message routing mechanism (see Table 2-11).</td>
</tr>
<tr>
<td>Cpl</td>
<td>00</td>
<td>0 1010</td>
<td>Completion without Data – Used for I/O and Configuration Write Completions and Read Completions (I/O, Configuration, or Memory) with Completion Status other than Successful Completion.</td>
</tr>
<tr>
<td>CplD</td>
<td>10</td>
<td>0 1010</td>
<td>Completion with Data – Used for Memory, I/O, and Configuration Read Completions.</td>
</tr>
<tr>
<td>CplLk</td>
<td>00</td>
<td>0 1011</td>
<td>Completion for Locked Memory Read without Data – Used only in error case.</td>
</tr>
<tr>
<td>CplDLk</td>
<td>10</td>
<td>0 1011</td>
<td>Completion for Locked Memory Read – otherwise like CplD.</td>
</tr>
</tbody>
</table>

All encodings not shown above are Reserved.
Configuration Write

Figure 10 is a close-up view of the example design simulation output with the default settings.

The first TLP that is initiated by the Downstream Port Model is a Configuration Write transaction. Opening tx.dat shows the following information of the first Configuration Write transaction:

```
[ 75162 ns ] : Config Write Type 0 Frame
  Traffic Class: 0x0
  ID: 0
  EP: 0
  Attributes: 0x0
  Length: 0x001
  Requester Id: 0x0100
  Tag: 0x0F
  Last and First Byte Enables: 0x0F
  Completer Id: 0x0100
  Register Address: 0x010

  0x0C
  0x0C
  0x0C
  0x10
```

This Configuration Write transaction is called from the PROC_BAR_PROGRAM. The first call to PROC_TX_TYPE0_CONFIGURATION_WRITE is to program BAR0. The code snippet with the parameters for the first call is shown below. The main goal here is to track and map the parameters in the tx.dat with results in the simulation waveform.
PROC_TX_TYPE0_CONFIGURATION_WRITE:

```vhdl
DEFAULT_TAG := X"0f":

write (L, String'("[ ") )); write (L, now):
write (L, String'(" ] : Setting Core Configuration Space...") );
write (output, L);

-- Program E800

PROC_TX_TYPE0_CONFIGURATION_WRITE ()

DEFAULT_TAG, --tag : in std_logic_vector (7 downto 0);
X"010", --reg_addr 12'h10
BAR(0) (31 downto 0), --reg_data : in std_logic_vector (31 downto 0);
X"F", --first_dw_be : in std_logic_vector (3 downto 0);
trn_td_c, trn_tsof_n, trn_teof_n, trn_trem_n_c, trn_tsrc_rdy_n,
\  trn_link_up_n, trn_tdst_rdy_n, trn_clk);

DEFAULT_TAG := X"10":
PROC_TX_CLK_EAT(100, trn_clk):

PROC_TX_TYPE0_CONFIGURATION_WRITE is defined as follows:

PROC_TX_SYNCHRONIZE(0, 0, trn_link_up_n, trn_tdst_rdy_n, trn_clk):

trn_td_c <=
   '0' &
   "10" &
   "00100" &
   '0' &
   "000" &
   "0000" &
   '0' &
   '0' &
   "00" &
   "00" &
   "0000000001" &
   COMPLETER_ID_CFG &
   tag(7 downto 0) &
   "0000" &
   first_dw_be(3 downto 0);

trn_tsof_n <= '0';
trn_tsrc_rdy_n <= '0';

PROC_TX_SYNCHRONIZE(1, 0, trn_link_up_n, trn_tdst_rdy_n, trn_clk):

trn_td_c <= COMPLETER_ID_CFG &
   "0000" &
   reg_addr(11 downto 2) &
   "00" &
   reg_data(7 downto 0) &
   reg_data(15 downto 8) &
   reg_data(23 downto 16) &
   reg_data(31 downto 16);
Figure 11 shows Configuration Write header format:

```
R = 0
Fmt = 10
Type = 00100
R = 0
TC = 000
Reserved = 0000
TD = 0
EP = 0
Attr = 00
R = 00
Length = 000000001
Requester ID = COMPLETER_ID_CFG (this value is the global constant definition)
Tag = 0f (this is passed from the PROC_TX_TYPE0_CONFIGURATION_WRITE call)
Last DW BE = 0000
1st DW BE = f (this is passed from the PROC_TX_TYPE0.Configuration_Write call)
Bus Number/Device Number/ Function Number = COMPLETER_ID_CFG
Reserved = 0000
Ext Reg Number / Register Number = x010 (provided in the procedure call)
The reg_data is the content to be programmed into BAR (0) as shown below:
```
BAR(0) := '0' & X"10000000";
BAR(1) := '0' & X"20000000";
BAR(2) := '0' & X"30000000";
BAR(3) := '0' & X"40000000";
BAR(4) := '0' & X"50000000";
BAR(5) := '0' & X"60000000";
BAR(6) := '0' & X"70000001"; -- bit 0 must be set to enable the EROM

reg_data(7 downto 0) &
reg_data(15 downto 8) &
reg_data(23 downto 16) &
reg_data(31 downto 24);

In the above, BAR(0) = reg_data

Therefore,

Reg_data (7 downto 0) = 0x00
Reg_data (15 downto 8) = 0x00
Reg_data (23 downto 16) = 0x00
Reg_data (31 downto 24) = 0x10

Now, check the waveform in Figure 12 to see if the above parameter values are reflected in the waveform or not.

--- DPORT TRN signals
- trn_ck
- trn_reset_n
- trn_link_up_n
- trn_td
- trn_lsrc_rdy_n
- trnᵗst_rdy_n
- trn_rc
- trn_rsrc
- trn_rseq
- trnᵗst_rseq

![Waveform](image)

**Figure 12 - Downstream Port Configuration Write Transaction**

The first 64 bits of trn_td in binary representation of the waveform is as follows:

```
0100110000000000000000000000000000000000000000000000111
```

The second 64 bits of trn_td in binary representation of the waveform is as follows:

```
00000001111100000000000000000000000000000000000000000000
```

If you break this down and put it in the header format above, you should get the following:
There are a total of 9 Configuration Writes which are also seen in the waveform below.

**Figure 13 - Configuration Write Header Analysis**

The yellow box in Figure 14 shows the Configuration Write Completions. Since there were 9 Configuration Writes, there are also 9 corresponding Configuration Write Completions.

**Figure 14 - Configuration Writes Completions**

Now, check rx.dat for the output log of the first Configuration Write Completion.
Figure 15 shows the Completion header format:

![Figure 15 - Completion Header Format](image)

If you zoom in to the waveform in Figure 15 to trace the first Configuration Write Completion TLP, you should see the waveform as shown in Figure 16:

![Figure 16 - Configuration Write Completion](image)

In the waveform, both completer ID and Requester ID are the same (i.e., ‘01A0’). The important field is the Tag. In the first Configuration Write, the Tag was ‘0F’. The same tag is seen in the Completion as well, indicating that this Completion is for the specific Configuration Write TLP.

Now, take a look at the second Configuration Write log and the corresponding Configuration Write Completion log, and verify if both have the same Tag field or not. The following is from tx.dat:
The following is from rx.dat:

```
[ 78458 ns ] : Config Write Type 0 Frame
  Traffic Class: 0x0
  TD: 0
  EP: 0
  Attributes: 0x0
  Length: 0x001
  Requester Id: 0x01A0
  Tag: 0x10
  Last and First Byte Enables: 0x0F
  Completor Id: 0x01A0
  Register Address: 0x014

0x00
0x00
0x00
0x20
```

The following is from rx.dat:

```
[ 81370 ns ] : Completion Without Data Frame
  Traffic Class: 0x0
  TD: 0
  EP: 0
  Attributes: 0x0
  Length: 0x000
  Completor Id: 0x01A0
  Completion Status: 0x0
  Requester Id: 0x01A0
  Tag: 0x10
```

**Memory Write 64 / Memory Read 64 / Completion**

A yellow box in Figure 17 is a 64-bit memory write TLP.

![Figure 17 - 64-bit Memory Write TLP](image)

Figure 18 shows a close-up view that has been broken into two parts for clarity.
The first two hexadecimal bits (60) indicate that this TLP is a memory write TLP with 4DW header (i.e., it is addressing a memory location with 64 bits address).

Before performing further analysis of the waveform above, check how the BARs were configured. The following is from the ModelSim console:

8522.1 ns ] : Transaction Reset is De-asserted
75066.1 ns ] : Transaction Link is Up
75066.1 ns ] : PCI EXPRESS BAR MEMORY/IO MAPPING PROCESS BEGUN.
   BAR 0 = 0x00000000 RANGE = 0xFFFFFFFF MEM64 MAPPED
   BAR 1 = 0x20000000 RANGE = 0xFFFFFFFF DISABLED
   BAR 2 = 0x30000000 RANGE = 0xFFFFFFFF MEM32 MAPPED
   BAR 3 = 0x40000000 RANGE = 0x00000000 DISABELED
   BAR 4 = 0x50000000 RANGE = 0x00000000 DISABLED
   BAR 5 = 0x60000000 RANGE = 0x00000000 DISABLED
   BAR 6 = 0x70000000 RANGE = 0xFFFFFFFF MEM32 MAPPED

As seen here, in the current PIO Example Design configuration, the first BAR has been enabled for 64-bit address. Hence, the second bar is automatically disabled. The third BAR (i.e., BAR 2) has been enabled to map to 32-bit memory address location. If you refer to the flowchart in Figure 9, the first TLP that is to be sent will be by calling PROC_TX_MEMORY_WRITE_64. This procedure is defined as follows:
PROC_TX_SYNCHRONIZE(0, 0, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
trn_td_c <= '0' & "11" & "00000" & '0' & tc(2 downto 0) & "0000" & '0' & '0' & "00" & "00" & len(9 downto 0) & COMPLETER_ID_CFG & tag(7 downto 0) & last_dw_be(3 downto 0) & first_dw_be(3 downto 0);

trn_tsof_n <= '0';
trn_teof_n <= '1';
trn_trem_n_c <= X"000";
trn_tsrc_rdy_n <= '0';
PROC_TX_SYNCHRONIZE(1,0, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
trn_td_c <= addr(63 downto 2) & "00";
trn_tsof_n <= '1';
unsigned_length := unsigned(length);
int_length := to_integer( unsigned_length);
if (int_length = 1) then
   DATA_STORE(4) := X"00";
   DATA_STORE(5) := X"00";
   DATA_STORE(6) := X"00";
   DATA_STORE(7) := X"00";
end if;
i := 0;
while (i < (int_length * 4)) loop

PROC_TX_SYNCHRONIZE(1,0, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
trn_td_c <= DATA_STORE(i+0) & DATA_STORE(i+1) & DATA_STORE(i+2) & DATA_STORE(i+3) & DATA_STORE(i+4) & DATA_STORE(i+5) & DATA_STORE(i+6) & DATA_STORE(i+7);
if ((i+7) >= ((int_length*4)-1)) then
   trn_tcof_n := '0';
   if (ep = '1') then
      trn_treffwd_n := '0';
   end if;
end if;
PROC_TX_MEMORY_WRITE_64 is called in pio_writeReadBack_test0 as follows:

PROC_TX_MEMORY_WRITE_64 (X"04", "000", "0000000001", B&I | 31 downto 0)
& BAR(i) | 31 downto 0, X"00", X"0", '0',
trn_td_c, trn_tsof_n, trn_teof_n, trn_trem_n_c,
trn tsr_c rd_n, trn terrfw_n, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);

Figure 19 shows the 4DW Memory Request header format.

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>+0</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Fmt</td>
<td>Type</td>
<td>TC</td>
<td>Reserved</td>
</tr>
<tr>
<td>Byte 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 19 - 64-bit Addressing Header Format

Now, check the corresponding parameters for the TLP. Some of the fields are assigned in PROC_TX_MEMORY_WRITE_64, and the others are input parameters passed during this procedure call.

R = 0
Fmt = 11
Type = 00000
R = 0
TC = 000 (passed during the procedure call)
Reserved = 0000
TD = 0
EP = 0
Attr = 00
R = 00
Length = 0000000001 (passed during the procedure call)
Requester ID = COMPLETER_ID_CFG (this value is defined as a global constant)

constant COMPLETER_ID_CFG : std_logic_vector(15 downto 0) := X"0100";

Tag = 04 (passed during the procedure call)
Last DW BE = 0000 (passed during the procedure call)
1st DW BE = 0xf (passed during the procedure call)
Bus Number/Device Number/ Function Number = Completer_ID_CFG

Address [63:32] = BAR (1) = 0x20000000
Address [31:2] = BAR (0) = 0x10000000
R = 0

Put the first 32 bits from the above parameters together and see if this matches with what is in the waveform in Figure 18:
0110_0000_0000_0000_0000_0000_0000_0001

From the waveform in Figure 18, we get [01100000000000000000000000000000] in binary format, which is the second 32-bit of the TLP. The corresponding hexadecimal value is as follows.
01A0_040f = 01A0040f

Figure 20 shows this value in the waveform in Figure 18:

![Figure 20 – First 64 bits of the 64-bit Memory Write TLP in Figure 18](image)

The next 64 bits is the memory address. From the above parameter definition, it should be:
20000000_10000000

Figure 21 shows the same value in the waveform in Figure 18:

![Figure 21 - Second 64 bits of the 64-bit Memory Write TLP in Figure 18](image)

In this TLP, the length of the payload size is specified to be ‘1’. Therefore, the payload data of 1DW is attached with this TLP.

In pio_writeReadBack_test0, DATA_STORE is assigned the following value:

```c
DATA_STORE[0] := "44";
DATA_STORE[1] := "63";
DATA_STORE[3] := "61";
```

Now, go back to the PROC_TX_MEMORY_WRITE_64 definition.
if (int_length = 1) then
  DATA_STORE(4) := X"00";
  DATA_STORE(5) := X"00";
  DATA_STORE(6) := X"00";
  DATA_STORE(7) := X"00";
end if;
i := 0;
while (i < (int_length * 4)) loop
  FPROC_TX_SYNCHRONIZE(1,0, trn_lnk_up_n, trn_tdst_rdy_n, trn_clk);
  trn_lk_c <= DATA_STORE(i+0) &
             DATA_STORE(i+1) &
             DATA_STORE(i+2) &
             DATA_STORE(i+3) &
             DATA_STORE(i+4) &
             DATA_STORE(i+5) &
             DATA_STORE(i+7);
  i := i+8;
end loop;

The first condition in the 'if' statement is true, therefore, DATA_STORE (4,5,6,7) is filled with ‘0’. The while loop runs only once. The above code snippet transfers 64 bits of data with the first 32 bits as “64636261” and the rest as ‘0’.

Figure 22 shows this data in the waveform in Figure 18:

The output log in tx.dat for this TLP is as shown below:

```
[ 14335B ns ] : Memory Write-64 Frame
  Traffic Class: 0xC
  TD: 0
  EP: 0
  Attributes: 0x0
  Length: 0x001
  Requester Id: 0x0100
  Tag: 0x04
  Last and First Byte Enables: 0x0F
  Address High: 0x2C000000
  Address Low: 0x10000000

0x64
0x63
0x62
0x61
```

The yellow box in Figure 23 shows the TLP sent from the Downstream Port Model (as shown in Figure 18) appearing on the trn receive interface of the endpoint.
A close-up view of the yellow box in Figure 23 is shown in Figure 24:

The data in the yellow box in Figure 24 is garbage data. Since the length of the payload is 1 (i.e., 1DW), this value does not count.

In pio_writeReadBack_test0, the Write TLP is followed by a Read TLP just to make sure that the value is written correctly. In Figure 45, the blue box is the Write TLP discussed above, and the yellow box is the Read TLP (this can be verified by checking the first two hexadecimal bits which is ‘20’ here).

The output log for this read TLP in tx.dat is as follows:
Memory Read is a non-posted TLP, therefore, a Completion packet must be sent upstream by the Endpoint Example Design user application. This Completion packet should be visible on the trn transmit interface of the Endpoint Example Design and also on the trn receive interface of the Downstream Port Model in the testbench. Now, trace the packet on these two interfaces in simulation.

In Figure 26, the yellow box shows the Completion packet on the trn transmit interface of the endpoint.

![Figure 26 – Completions Packets on trn Transmit Interface of the Endpoint](image)

Figure 27 shows a close-up view of the yellow box in Figure 26.

![Figure 27 - Completion Packet](image)

Before analyzing this packet, check the header format of the Completion packet shown in Figure 28.
In the first 8 bits in the Completion packet of Figure 27, ‘4A’ indicates it is a ‘Completion with Data’ packet. Below is the received output log for this Completion packet:

```
[ 146522 ns ] : Completion With Data Frame
  Traffic Class: 0x0
  TD: 0
  EP: 0
  Attributes: 0x0
  Length: 0x001
  Completer Id: 0x0110
  Completion Status: 0x0
  Requester Id: 0x0110
  Tag: 0x05
  0x64
  0x63
  0x62
  0x61
```

As you can see in the waveform in Figure 27, the correct value is returned in the Completion packet. The main interest here is the Tag field. In the Memory Read packet (shown in Figure 25), the Tag field was 0x05 (shown below in the log file for reference).

```
[ 143354 ns ] : Memory Read-64 Frame
  Traffic Class: 0x0
  TD: 0
  EP: 0
  Attributes: 0x0
  Length: 0x001
  Requester Id: 0x01A0
  Tag: 0x05
  Last and First Byte Enables: 0x0F
  Address High: 0x20000000
  Address Low: 0x10000000
```

If you check the Tag field in the Completion packet, the value is the same (i.e., 0x05) indicating that this Completion belongs to the Memory Read packet that was sent earlier (shown in Figure 25).

Also, the Completion status field is 0x0, indicating that the Completion was successful.
Table 4 - Completion Status Field Value

<table>
<thead>
<tr>
<th>Completion Status[2:0] Field Value</th>
<th>Completion Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>000b</td>
<td>Successful Completion (SC)</td>
</tr>
<tr>
<td>001b</td>
<td>Unsupported Request (UR)</td>
</tr>
<tr>
<td>010b</td>
<td>Configuration Request Retry Status (CRS)</td>
</tr>
<tr>
<td>100b</td>
<td>Completer Abort (CA)</td>
</tr>
<tr>
<td>all others</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The yellow box in Figure 29 shows the byte count field in the Completion packet in Figure 27.

The Byte Count field indicates the remaining number of bytes required to complete the read request, including the number of bytes returned with the Completion packet (except when the BCM field is set).

Figure 30 shows the Completion packet being received on the receive trn interface of the Downstream Port Model.

If you run into issues related with Completions, the following requirement (as specified in the PCI Express Base Specification) should be checked to make sure it is satisfied by the Completion packet:

“Completion headers must supply the same values for the Requester ID, Tag, Attribute and Traffic Class as were supplied in the header of the corresponding Request”
Memory Write 32 / Memory Read 32 / Completion

The yellow box in Figure 31 shows a memory write TLP that addresses 32-bit memory address. The memory write TLP is immediately followed by a memory read TLP to the same memory location.

Figure 31 - 32-bit Memory Write TLP

Figure 32 is a close-up view of the yellow box in Figure 31. It shows a memory write TLP followed by a memory read TLP.

Figure 32 - 32-bit Memory Write followed by a 32-bit Memory Read

The value of ‘40’ in the red box of Figure 32, indicates 32-bit memory write TLP. The value of ‘00’ in the yellow box of the same figure indicates a 32-bit memory read TLP.

The output log for the memory write packet in tx.dat is as follows:

```
[ 149914 ns ] : Memory Write-32 Frame
    Traffic Class: 0x0
    TD: 0
    EP: 0
    Attributes: 0x0
    Length: 0x001
    Requester Id: 0x01A0
    Tag: 0x02
    Last and First Byte Enables: 0x0F
    Address Low: 0x30000000

    0x01
    0x02
    0x03
    0x04
```

The output log for the memory read packet in tx.data is as follows:
The yellow box in Figure 33 shows the Completion for the above memory read packet.

Figure 33 - Completion Packet on trn Receive Interface of the Downstream Port Model

Figure 34 is a close-up view of the Completion packet as shown in Figure 33.

Figure 34 – Close-up view of Completion Packet in Figure 33

The output log for this packet in rx.dat is:
The yellow box in Figure 35 shows memory write and memory read from the Expansion ROM which is mapped to BAR6.
Limitations and Features of Downstream Port Model

Following are the limitations of the Downstream Port Model. For more details, please see the Virtex-5 FPGA Integrated PCI Express Block Plus Core User Guide.

- The PIO Example Design was created to support one I/O BAR at the most, one Mem64 BAR, and two Mem32 BARs (one of which must be the EROM space); by default, the Downstream Port Model makes a check during device configuration that verifies the core has been configured to meet this requirement. A violation of this check causes a warning message to be displayed as well as for the offending BAR to be gracefully disabled in the test bench. This check can be disabled by setting the pio_check_design variable to zero in the pci_exp_usrapp_tx.v file.
- The Downstream Port Model consists of a parallel test that involves more than one process thread. The test “sample_smoke_test1” is an example of a parallel test with two process threads. Parallel tests are very useful in verifying that a specific set of events has occurred, however the order of these events are not known. Currently, the VHDL version of the Downstream Port Model Test Bench does not support Parallel tests.
- The Downstream Port Model has a 128-byte MPS capability in the receive direction, and a 512-byte MPS capability in the transmit direction.
- The Downstream Port Model testbench provides just enough tools to perform Writes and Reads to the user design. The Downstream Port Model is not a full blown simulation model as a true BFM (available from third-party vendors). However, it enables enough functionality to perform basic testing of the user design.

Limitations of PIO Example Design

Following are the limitations of the PIO Example Design. For more details, please see the Virtex-5 FPGA Integrated PCI Express Block Plus Core User Guide.

- The PIO design is a simple target-only application that interfaces with the endpoint for PCIe core’s Transaction (TRN) interface.
- The PIO design only supports single DWORD payload Read and Write PCI Express transactions to 32-/64-bit address memory spaces, and I/O space with support for Completion TLPs.
- Each space is implemented with a 2 KB memory. If the corresponding BAR is configured for a wider aperture, accesses beyond the 2 KB limit wrap around and overlap the 2 KB memory space.
- The PIO design successfully processes single DWORD payload memory read and memory write TLPs and I/O read and write TLPs. Memory read or memory write TLPs of lengths larger than one DWORD are not processed correctly by the PIO design. However, the core does accept these TLPs and passes them along to the PIO design. If the PIO design receives a TLP with a length of greater than one DWORD, the TLP is received completely from the core and discarded. No corresponding Completion is generated.
- PIO design handles memory Writes and I/O TLP Writes in different ways; the PIO design responds to I/O writes by generating a Completion without Data (cpl), a requirement of the PCI Express specification.
- The PIO_32_TX_ENGINE and PIO_64_TX_ENGINE modules generate Completions for received memory and I/O read TLPs. The PIO design does not generate outbound read or write requests. However, you can add this functionality to further customize the design.
Modifying Downstream Root Port Model Testbench Packet Generation

Length Parameter Modification

As stated in the earlier section, the PIO Example Design will not generate Completion for a TLP whose payload is greater than ‘1’. Check the behavior of the simulation to see if a memory read TLP has the length parameter of ‘2’. Change the parameter value in the pio_writeReadBack_test0 as shown below:

```vhdl
PROC_TX_MEMORY_READ_32 (  
    X"03", "00", "0000000000", B\&R(i)(31 downto 0), X"0", X"F",  
    trn_tdc, trn_tsof_n, trn_tsof_n, trn_trem_n_c,  
    trn_tsrc_rdy_n, trn_ink_up_n, trn_tdst_rdy_n, trn_clk);
```

If you simulate this modified design, the simulation times out. The TLP is sent from the dsport to the endpoint. It appears at the receive trn interface of the endpoint, but the Completion is never generated, hence the simulation times out.

In the yellow box in Figure 36, you can see that the TLP with the modified length parameter leaves the trn transmit interface at the Downstream Port Model.

In the yellow box of Figure 37, the TLP does arrive at the user side, but the corresponding Completion is never generated.

![Figure 36 - TLP with Modified Length Parameter at the trn Interface of the Downstream Port Model](image-url)
Take a closer look at the modified TLP and check whether the new length value shows up in the outgoing packet or not. As expected, it does appear as shown in Figure 38.

Figure 37 - TLP with Modified Length Parameter at the trn Interface of the Endpoint

Figure 38 - New Length Field Value in the Modified TLP
Now, what happens if a 32-bit memory read TLP is poisoned? A TLP is poisoned by assigning the value of ‘EP’ as ‘1’ as shown below. The following code snippet is from PROC_TX_MEMORY_READ_32 procedure in test_interface.vhd.

```vhdl
trn_td_c <=
  '0' &
  "00" &
  "00000" &
  '0' &
  tc(2 downto 0) &
  "0000" &
  '1' &
  "00" &
  "00" &
  len(9 downto 0) &
  COMPLETER_ID_CFG &
  tag(7 downto 0) &
  last_dw_be(3 downto 0) &
  first_dw_be(3 downto 0);
```

The packet is generated and passed to the endpoint backend user application. The user side detects that a TLP with poisoned data has been received. The message is printed on the console window as shown below:

```
# XILINX_PCIE_311 : RECEIVED POISONED TLP
# XILINX_PCIE_312 : TRANSMITTED POISONED TLP
# [153114 ns] : PROC_PARSE_FRAME on Receive
# [153210 ns] : Test Passed. Completion Data = 0x04030201
# Testing PIO Mem32 Space BAR 6
# [158506 ns] : PROC_PARSE_FRAME on Transmit
# [158602 ns] : PROC_PARSE_FRAME on Transmit
# XILINX_PCIE_311 : RECEIVED POISONED TLP
# XILINX_PCIE_312 : TRANSMITTED POISONED TLP
# [159706 ns] : PROC_PARSE_FRAME on Receive
# [159802 ns] : Test Passed. Completion Data = 0x04030201
# ** Failure: Simulation Stopped.
```

As mentioned in the user guide, all received TLPs with the Data Poisoning bit in the header set (EP=1) are presented to the user. The core asserts the trn_rerrfwrd_n signal for the duration of each poisoned TLP (illustrated in Figure 39).
Figure 40 shows the same behavior in the simulation as illustrated in Figure 39.
Poisoning Configuration Write Request

What happens when a Configuration Write request is poisoned? This is done by modifying the ‘EP’ bit in PROC_TX_TYPE0_CONFIGURATION_WRITE procedure in test_interface.vhd as shown below:

```vhdl
trn_tdc <= '0' &
"10" &
"00100" &
'0' &
"000" &
"0000" &
'1' &
"00" &
"00" &
"0000000001" &
COMPLETER_ID_CFG &
tag(7 downto 0) &
"0000" &
first_dw_be(3 downto 0);
trn_tsaf_n <= '0';
trn_tsrc_rdy_n <= '0';
```

According to the PCI Express Base Specification, a poisoned Configuration Write request must be discarded by the Completer, and a Completion with a Completion Status of UR is returned.

This behavior can be seen in the simulation as well. The following is a message printed on the ModelSim console when a poisoned Configuration Write is transmitted from the Downstream Port Model:

```
# [51734 ns]: PROC_PARSE_FRAME on Transmit
# XILINX_PCIE_311: RECEIVED POISONED TLP
# XILINX_PCIE_312: TRANSMITTED UNSUPPORTED REQUEST
# [84330 ns]: PROC_PARSE_FRAME on Receive
# [65350 ns]: PROC_PARSE_FRAME on Transmit
# XILINX_PCIE_311: RECEIVED POISONED TLP
# XILINX_PCIE_312: TRANSMITTED UNSUPPORTED REQUEST
# [87382 ns]: PROC_PARSE_FRAME on Receive
# [68348 ns]: PROC_PARSE_FRAME on Transmit
# XILINX_PCIE_311: RECEIVED POISONED TLP
# XILINX_PCIE_312: TRANSMITTED UNSUPPORTED REQUEST
# [91250 ns]: PROC_PARSE_FRAME on Receive
# [91842 ns]: PROC_PARSE_FRAME on Transmit
# XILINX_PCIE_311: RECEIVED POISONED TLP
# XILINX_PCIE_312: TRANSMITTED UNSUPPORTED REQUEST
# [94554 ns]: PROC_PARSE_FRAME on Receive
```

In Figure 41, the red box is the poisoned Configuration Write request. The yellow box is the corresponding Completion for this request. First, see whether the Configuration Write request is poisoned or not. Then, look at the incoming Completion to see whether the ‘Completion Status’ field is set to ‘001’ or not, which indicates ‘Unsupported Request’.
Figure 41 - Poisoned Configuration Write and the Corresponding Completion

Figure 42 shows a close-up view of the first outgoing Configuration Write request with the poisoned bit ‘EP’ set to ‘1’.

The header format for the configuration request TLP is shown in Figure 43 for a reference.

Figure 43 - Configuration Request Header Format

Figure 44 shows the corresponding Completion with the Completion Status set to ‘Unsupported Request’.

Figure 44 - Completion with Completion Status Field set to ‘Unsupported Request’
The header format for the Completion TLP is shown in Figure 45 for a reference.

![Completion TLP Header Format](image)

Figure 45 - Completion TLP Header Format

**Testing with a new TLP**

Insert a new TLP that writes to BAR mapped to 32-bit memory address and then read it back. This is done by modifying the tests.vhd file as follows:

```vhdl
DATA_STORE1(0) := X"DE";
DATA_STORE1(1) := X"AD";
DATA_STORE1(2) := X"BE";
DATA_STORE1(3) := X"EF"
```

-------------------Test TLP -------------------------------
PROC_TX_MEMORY_WRITE_321 (
  X"02", "00", "0000000001", BAR(i)(31 downto 0), X"0", X"F",'0',
  trn_td_c, trn_tsof_n, trn_tsof_n, trn_trem_n_c,
  trn_tsrc_rdy_n, trn_terrfwd_n, trn_ink_up_n, trn_tdst_rdy_n, trn_clk);
```

P_READ_DATA := X"FFFFFFFFF";

PROC_TX_MEMORY_READ_32 ( 
  X"03", "00", "0000000001", BAR(i)(31 downto 0), X"0", X"F",
  trn_td_c, trn_tsof_n, trn_tsof_n, trn_trem_n_c,
  trn_tsrc_rdy_n, trn_ink_up_n, trn_tdst_rdy_n, trn_clk);
PROC_WAIT_FOR_READ_DATA (tx(cx_read_data_valid,
  rx_read_data_valid, rx_tx_read_data, trn_clk);
```

You are not creating an entirely new TLP here, you are just replicating the PROC_TX_MEMORY_WRITE_32 TLP to read and write different data sets to the same memory location as would be done by PROC_TX_MEMORY_WRITE_32. For this, a new procedure is defined in test_interface.vhd called PROC_TX_MEMORY_WRITE_321. The content of this procedure basically remains the same, except for DATA_STORE. A new array has been defined called DATA_STORE1. A different set of values is stored in this array as shown in the code snippet above.

Take a look at the resulting waveform. In the previous simulation, there were three TLPs going down from the Dsport Model to the endpoint. In this case, there are five TLPs.
Take a closer look at the new outgoing TLP. You should see the following data payload with this TLP:

\[
\begin{align*}
\text{DATA\_STORE1(0)} & \ := \ X"\text{DE}"; \\
\text{DATA\_STORE1(1)} & \ := \ X"\text{AD}"; \\
\text{DATA\_STORE1(2)} & \ := \ X"\text{BE}"; \\
\text{DATA\_STORE1(3)} & \ := \ X"\text{EF}";
\end{align*}
\]

The output log in tx.dat for this TLP is as shown below:

```
[ 149914 ns ] : Memory Write-32 Frame
  Traffic Class: 0x0
  TD: 0
  EP: 0
  Attributes: 0x0
  Length: 0x001
  Requester Id: 0x01A0
  Tag: 0x02
  Last and First Byte Enables: 0x0F
  Address Low: 0x30000000

  0xDE
  0xAD
  0xBE
  0xEF
```

A memory read TLP is issued. Check to see if the same data pattern in the Completion is sent back from the endpoint for this Memory Read TLP.
The output log in rx.dat for this Completion packet is shown below:

```
[ 153114 ns ] : Completion With Data Frame
  Traffic Class: 0x0
  TD: 0
  EP: 0
  Attributes: 0x0
  Length: 0x001
  Completer Id: 0x0110
  Completion Status: 0x0
  Requester Id: 0x0110
  Tag: 0x03

  0xDE
  0xAD
  0xBE
  0xEF
```

Points to Note

- If the Header Length field indicates a transfer is more than 1DW, the first DW Byte Enabled must have at least one bit enabled.
- A Write Request with a transfer length of 1DW and no byte enables set is legal, but has no effect on the Completer.
- If a Read Request of 1 DW is issued with no byte enable bits set, the Completer returns a 1DW data payload with undefined data. This can be used as a Flush mechanism. Because of ordering rules, a flush can be used to force all previously posted writes to the memory before the Completion is returned.
- The first byte of the data in the payload (immediately after the header) is always associated with the lowest (start) address.
- Requests must not mix combinations of start address and transfer length which causes a memory space access to crash a 4KB boundary. While checking is optional in this case, receivers checking for violations of this rule will report it as a Malformed TLPs.

Conclusion

This document presented a detailed analysis of the PIO Example Design and the Downstream Port Model that come with the generation of Virtex-5 FPGA Integrated PCI Express Block Plus Core. The main purpose of this document is to give you a clear understanding of what is provided in the PIO Example Design and how you can modify the example design (if
required) to conduct custom tests. After reading this document, you should be able to perform in-depth debugging of issues related to the Virtex-5 FPGA Integrated PCI Express Block Plus Core. Although the document specifically mentions the Virtex-5 FPGA Integrated PCI Express Block Plus Core, the same principal applies to all other Xilinx PCI Express Cores.

If this document does not help resolve your problem, please open a WebCase with Xilinx Technical Support that contains the details of your investigation and analysis.

Appendix

Some reference content is given in this section to make it easier for you to perform packet analysis to debug your design.

Generic TLP Header Fields

I/O Request Header Format

Memory Request Header Format
Configuration Request Header Format

+0  +1  +2  +3  
R   x 0  |  Type  |  TC  |  Reserved  |  Attr  |  Length  
  7  6  5  4  3  2  1  0  |  7  6  5  4  3  2  1  0  |  7  6  5  4  3  2  1  0  |  0  0  0  0  0  0  0  1  

Byte 0: 
- R: Requester ID
- x: Tag
- 0: Last DW BE
- TC: Device Number
- Reserved
- Attr: Function Number

Message Request Header Format

+0  +1  +2  +3  
R   x 1  |  Type  |  TC  |  Reserved  |  Attr  |  Length  
  7  6  5  4  3  2  1  0  |  7  6  5  4  3  2  1  0  |  7  6  5  4  3  2  1  0  |  0  0  0  0  0  0  0  1  

Byte 0: 
- R: Requester ID
- x: Tag
- 1: Message Code
- TC: Device Number
- Reserved
- Attr: Function Number

Completion Header Format

+0  +1  +2  +3  
R   x 0  |  Type  |  TC  |  Reserved  |  Attr  |  Length  
  7  6  5  4  3  2  1  0  |  7  6  5  4  3  2  1  0  |  7  6  5  4  3  2  1  0  |  0  0  0  0  0  0  0  1  

Byte 0: 
- R: Requester ID
- x: Tag
- 0: Last DW BE
- TC: Device Number
- Reserved
- Attr: Function Number
- Length: Byte Count

Byte 4: 
- Completion ID
- Compl. Status
- Byte Count

Byte 8: 
- Requester ID
- Tag
- R
- Lower Address
## Packet Header Field Description

<table>
<thead>
<tr>
<th>Header Field</th>
<th>Size (bits)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fmt</td>
<td>3</td>
<td><strong>Format</strong>: Indicates header size and if packet contains payload and TLP prefix</td>
</tr>
<tr>
<td>Type</td>
<td>5</td>
<td><strong>Transaction variant</strong></td>
</tr>
<tr>
<td>TC</td>
<td>3</td>
<td><strong>Traffic Class</strong>: Binary-encoded traffic class; Default=000b</td>
</tr>
<tr>
<td>Attr[2]</td>
<td>1</td>
<td>ID Based Ordering</td>
</tr>
<tr>
<td>TH</td>
<td>1</td>
<td><strong>TLP Processing Hints (TPH)</strong></td>
</tr>
<tr>
<td>TD</td>
<td>1</td>
<td><strong>Digest Present</strong>: Optional 1DW TLP Digest Field (ECRC) present</td>
</tr>
<tr>
<td>EP</td>
<td>1</td>
<td><strong>Poisoned Data</strong>: Payload should be considered invalid</td>
</tr>
<tr>
<td>Attr[1:0]</td>
<td>2</td>
<td><strong>Attributes</strong>: {Relaxed ordering, No Snoop}</td>
</tr>
<tr>
<td>AT</td>
<td>2</td>
<td><strong>Address Translation</strong>: 00b=Default, 01b=Translation Request, 10b=Translated, 11b=reserved</td>
</tr>
<tr>
<td>Length</td>
<td>10</td>
<td>TLP payload size, in DW (1=1DW, 0=1024DW)</td>
</tr>
<tr>
<td>Requester ID</td>
<td>16</td>
<td>Identifies the requester for matching completions with requests - consists of {Bus[7:0], Device[4:0], Function[2:0]}</td>
</tr>
<tr>
<td>Completer ID</td>
<td>16</td>
<td>The completer for the transaction (captured by endpoint for Cfg)</td>
</tr>
<tr>
<td>Tag</td>
<td>8</td>
<td>Identifier for each outstanding request - sequential for NP</td>
</tr>
<tr>
<td>1st DW BE</td>
<td>4</td>
<td>Byte-enables for 1st DW - bits map 1:1 with bytes</td>
</tr>
<tr>
<td>Last DW BE</td>
<td>4</td>
<td>Byte-enables for last DW - bits map 1:1 with bytes For single-DW, aligned transfers, Last DW BE=0000b</td>
</tr>
<tr>
<td>Address</td>
<td>30/52</td>
<td>Upper 32/62 bits of start address - bottom two bits are reserved, forcing the start address to be DW-aligned</td>
</tr>
<tr>
<td>Register Number</td>
<td>4+6</td>
<td>Configuration space offset for config requests</td>
</tr>
<tr>
<td>Byte Count</td>
<td>12</td>
<td>Remaining byte count until a read request is satisfied</td>
</tr>
<tr>
<td>BCM</td>
<td>1</td>
<td><strong>Byte Count Modified</strong>: only used by PCI-X</td>
</tr>
<tr>
<td>CS</td>
<td>3</td>
<td><strong>Completion Status</strong>: 000b=SC, 001b=UR, 010b=CRS, 100b=CA</td>
</tr>
<tr>
<td>Lower Address</td>
<td>7</td>
<td>Lower 7 bits of address for 1st enabled byte of read</td>
</tr>
<tr>
<td>Message Code</td>
<td>8</td>
<td>Type of message</td>
</tr>
</tbody>
</table>
## Packet Types

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>TLP Type</th>
<th>Hdr Size</th>
<th>P/NP/Cpl</th>
<th>Fmt</th>
<th>Type</th>
<th>Fmt/Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRd</td>
<td>Memory Read Request</td>
<td>3DW</td>
<td>NP</td>
<td>000</td>
<td>0</td>
<td>0x00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4DW</td>
<td>NP</td>
<td>001</td>
<td>0</td>
<td>0x20</td>
</tr>
<tr>
<td>MRdLk</td>
<td>Memory Read Lock Request</td>
<td>3DW</td>
<td>NP</td>
<td>000</td>
<td>0</td>
<td>0x01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4DW</td>
<td>NP</td>
<td>001</td>
<td>0</td>
<td>0x21</td>
</tr>
<tr>
<td>MWr</td>
<td>Memory Write Request</td>
<td>3DW</td>
<td>P</td>
<td>010</td>
<td>0</td>
<td>0x40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4DW</td>
<td>P</td>
<td>011</td>
<td>0</td>
<td>0x60</td>
</tr>
<tr>
<td>IORd</td>
<td>IO Read Request</td>
<td>3DW</td>
<td>NP</td>
<td>000</td>
<td>0</td>
<td>0x02</td>
</tr>
<tr>
<td>IOWr</td>
<td>IO Write Request</td>
<td>3DW</td>
<td>NP</td>
<td>010</td>
<td>0</td>
<td>0x42</td>
</tr>
<tr>
<td>CfgRd0</td>
<td>Config Type 0 Read Request</td>
<td>3DW</td>
<td>NP</td>
<td>000</td>
<td>0</td>
<td>0x04</td>
</tr>
<tr>
<td>CfgWr0</td>
<td>Config Type 0 Write Request</td>
<td>3DW</td>
<td>NP</td>
<td>010</td>
<td>0</td>
<td>0x44</td>
</tr>
<tr>
<td>CfgRd1</td>
<td>Config Type 1 Read Request</td>
<td>3DW</td>
<td>NP</td>
<td>000</td>
<td>0</td>
<td>0x05</td>
</tr>
<tr>
<td>CfgWr1</td>
<td>Config Type 1 Write Request</td>
<td>3DW</td>
<td>NP</td>
<td>010</td>
<td>0</td>
<td>0x45</td>
</tr>
<tr>
<td>Msg</td>
<td>Message Request</td>
<td>4DW</td>
<td>P</td>
<td>001</td>
<td>1</td>
<td>0x3x</td>
</tr>
<tr>
<td>MsgD</td>
<td>Message Request W/Data</td>
<td>4DW</td>
<td>P</td>
<td>011</td>
<td>1</td>
<td>0x7x</td>
</tr>
<tr>
<td>Cpl</td>
<td>Completion</td>
<td>3DW</td>
<td>Cpl</td>
<td>000</td>
<td>0</td>
<td>0xA</td>
</tr>
<tr>
<td>CplD</td>
<td>Completion W/Data</td>
<td>3DW</td>
<td>Cpl</td>
<td>010</td>
<td>0</td>
<td>0x4A</td>
</tr>
<tr>
<td>CplLk</td>
<td>Completion-Locked</td>
<td>3DW</td>
<td>Cpl</td>
<td>000</td>
<td>0</td>
<td>0x0B</td>
</tr>
<tr>
<td>CplDLk</td>
<td>Completion-Locked W/Data</td>
<td>3DW</td>
<td>Cpl</td>
<td>010</td>
<td>0</td>
<td>0x4B</td>
</tr>
<tr>
<td>FetchAdd</td>
<td>Fetch and Add AtomicOp Req</td>
<td>3DW</td>
<td>NP</td>
<td>010</td>
<td>0</td>
<td>0x4C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4DW</td>
<td>NP</td>
<td>011</td>
<td>0</td>
<td>0x6C</td>
</tr>
<tr>
<td>Swap</td>
<td>Unconditional Swap AtomicOp Req</td>
<td>3DW</td>
<td>NP</td>
<td>010</td>
<td>0</td>
<td>0x4D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4DW</td>
<td>NP</td>
<td>011</td>
<td>0</td>
<td>0x6D</td>
</tr>
<tr>
<td>CAS</td>
<td>Compare and Swap AtomicOp Req</td>
<td>3DW</td>
<td>NP</td>
<td>010</td>
<td>0</td>
<td>0x4E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4DW</td>
<td>NP</td>
<td>011</td>
<td>0</td>
<td>0x6E</td>
</tr>
<tr>
<td>LPrfx</td>
<td>Local TLP Prefix</td>
<td>1DW</td>
<td>Cpl</td>
<td>100</td>
<td>0</td>
<td>0xE</td>
</tr>
<tr>
<td>Eprfx</td>
<td>End-End TLP Prefix</td>
<td>1DW</td>
<td>Cpl</td>
<td>100</td>
<td>1</td>
<td>0x9</td>
</tr>
</tbody>
</table>
## Control Symbols and Ordered Sets

### Gen1/2 Ordered Sets

<table>
<thead>
<tr>
<th>8b10b</th>
<th>Hex</th>
<th>Sym</th>
<th>TS1</th>
<th>TS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K28.5</td>
<td>BC</td>
<td>COM</td>
<td>1</td>
<td>COM</td>
</tr>
<tr>
<td>K27.7</td>
<td>FB</td>
<td>STP</td>
<td>2</td>
<td>ink</td>
</tr>
<tr>
<td>K26.2</td>
<td>5C</td>
<td>SDP</td>
<td>3</td>
<td>ln</td>
</tr>
<tr>
<td>K25.7</td>
<td>FD</td>
<td>END</td>
<td>4</td>
<td>nFTS</td>
</tr>
<tr>
<td>K30.7</td>
<td>FE</td>
<td>EDB</td>
<td>5</td>
<td>rate</td>
</tr>
<tr>
<td>K23.7</td>
<td>F7</td>
<td>PAD</td>
<td>6</td>
<td>tc</td>
</tr>
<tr>
<td>K26.0</td>
<td>1C</td>
<td>SKP</td>
<td>7</td>
<td>4Ah</td>
</tr>
<tr>
<td>K26.1</td>
<td>3C</td>
<td>FTS</td>
<td>8</td>
<td>4Ah</td>
</tr>
<tr>
<td>K28.3</td>
<td>7C</td>
<td>IDL</td>
<td>9</td>
<td>4Ah</td>
</tr>
<tr>
<td>K28.7</td>
<td>FC</td>
<td>EIE</td>
<td>10</td>
<td>4Ah</td>
</tr>
<tr>
<td>K26.4</td>
<td>9C</td>
<td>N/A</td>
<td>11</td>
<td>4Ah</td>
</tr>
<tr>
<td>K.28.6</td>
<td>DC</td>
<td>N/A</td>
<td>12</td>
<td>4Ah</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>4Ah</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>4Ah</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>4Ah</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>4Ah</td>
</tr>
</tbody>
</table>

### References

1. PCI Express® Base Specification Revision 1.1, March 28, 2005
2. LogiCORE™ IP Endpoint Block Plus v1.x for PCI Express® (UG341)

### Revision History

03/20/2012 - Initial release