XILINX®

Xilinx Answer 53776
Generating Quick Test Cases for Xilinx Integrated PCI Express Block and Serial RapidIO Cores Verilog Simulation

Important Note: This downloadable PDF of an Answer Record is provided to enhance its usability and readability. It is important to note that Answer Records are Web-based content that are frequently updated as new information becomes available. You are reminded to visit the Xilinx Technical Support Website and review (Xilinx Answer 53776) for the latest version of this Answer.

Introduction

This document primarily focuses on techniques to create test cases in simulation by forcing certain data pattern on core interfaces. When designing a system with IPs such as PCI Express and Serial Rapid IO, designers may run into issues where the system halts due to a certain incoming packet or incorrect toggling of signals. To debug such issues in hardware could be difficult and time consuming as this would need debugging using tools such as Chipscope. The best option is to try to reproduce the issue in simulation by writing a specific testbench. The problem with this is you will need to write a comprehensive code in the testbench to capture that particular use case scenario. This takes time and designers mostly wouldn't be in condition to afford much time due to time critical nature of their project. This document describes how a designer can drive custom packets on an interface of designs with PCI Express and Serial RapidIO cores with '*force'* command in Verilog. '*force'* is a powerful Verilog command which you can use to drive signals at any timestamp of your simulation.

The first part of this document describes different Verilog simulation statements such as '*force-release'*, '*initial-begin'*, ' *display'*, '*monitor'* etc. The remainder of the document presents specific test cases to illustrate how a packet can be injected into Xilinx Integrated PCI Express Block and Serial Rapid IO cores for quick simulation.

Verilog Simulation

Verilog provides powerful features that allow users to model designs for particular use case and do required analysis. There are number of Verilog features, tailored for simulation, a designer can use. This section describes some major features that are helpful in reproducing design issues in simulation, seen in hardware:

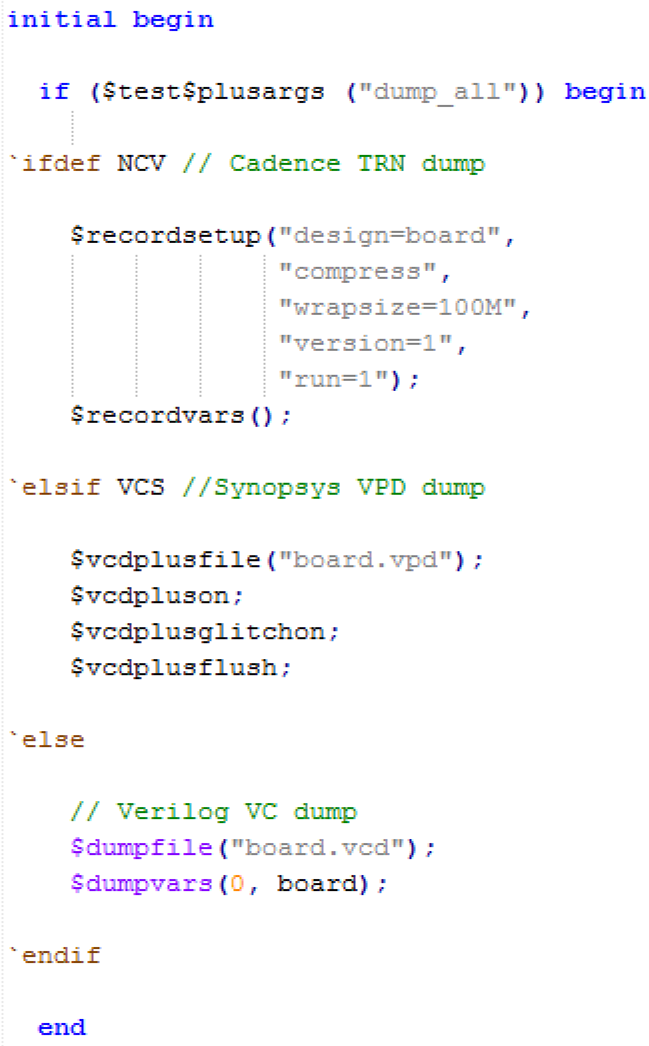
1. Procedural Continuous Assignments
2. Conditional Compilation
3. System Tasks
4. Looping Constructs
5. Structured Procedures
6. fork-join

Procedural Continuous Assignments

Procedural continuous assignments allow values to be driven continuously onto a register or a net for a limited period of time. It is different from procedural assignments where the value will be there in the register after the assignment until a new procedural assignment assigns a new value. *force-release* is a powerful procedural continuous assignment feature in Verilog which will be used in all the test cases described in this document. The *force* assignment is used to assign a new value to a register or a net. The value assigned will be retained till it is released with *release* command or with another *force* assignment. *force-release* is used only for design verification and is not synthesizable.

Conditional Compilation

The compiler directive for conditional compilation are: *`ifdef, `ifndef, `else, `elsif* and *`endif*. Conditional compilation is useful to conditionally output the debug message on the terminal or an output file. Figure 1 illustrates how conditional compilation directives could be used to conditionally output the waveform dump depending on which simulator was used.



```
initial begin

  if ($test$plusargs ("dump_all")) begin

`ifdef NCV // Cadence TRN dump

    $recordsetup("design=board",
                 "compress",
                 "wrapsize=100M",
                 "version=1",
                 "run=1");
    $recordvars();

`elsif VCS //Synopsys VPD dump

    $vcdplusfile("board.vpd");
    $vcdpluson;
    $vcdplusglitchon;
    $vcdplusflush;

`else

    // Verilog VC dump
    $dumpfile("board.vcd");
    $dumpvars(0, board);

`endif

  end
```

Figure 1 - Conditional Compilation

System Tasks

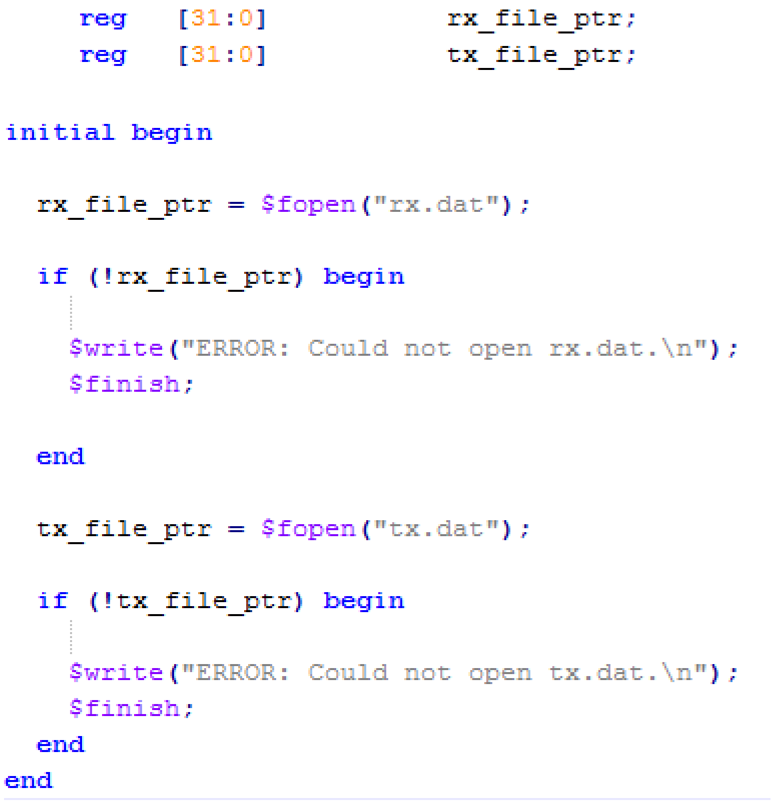
System tasks for outputting simulation result into a file can be useful to analyze the simulation output. Verilog provides tasks to record the output in a file. The ***$fopen*** function is used to open a file. The call to this function returns a 32 bit descriptor. This descriptor is unique for each file and is the main communication bridge between the simulator and the file. If the file doesn't already exist, ***$fopen*** will create a new file with the provided name in the default folder or a folder given in the full path description.

$fclose can be called to close the all opened files. However, explicit call to this function is not necessary as all open files are closed before the simulator terminates. This happens by default.

$fdisplay is used to write to files. A file descriptor returned when opening the file with ***$fopen*** is passed as an argument to ***$fdisplay***. Like ***$fdisplay***, there is another system task called ***$fwrite***. Both of them write data to a file whenever they are executed. The difference between ***$display*** and ***$fwrite*** is that the former inserts a new line after every execution and the latter doesn't.

The system tasks ***$fdisplay*** and ***$fwrite*** are for writing to a file. To display the output only on the console, the system task used is ***$display***. Similar to ***$display***, there are two other system tasks called ***$monitor*** and ***$strobe***. ***$display*** and ***$strobe*** write to the output only when they are executed. The difference between the two is that ***$display*** displays the result right after it is executed whereas ***$strobe*** displays the result at the end of the current simulation time instant i.e. after all the events that have been scheduled for the current simulation time have been completed. The other system task similar to ***$display*** and ***$strobe*** is ***$monitor***. The main feature of ***$monitor*** is that it displays in the standard output or the log file everything when any of its parameter changes.

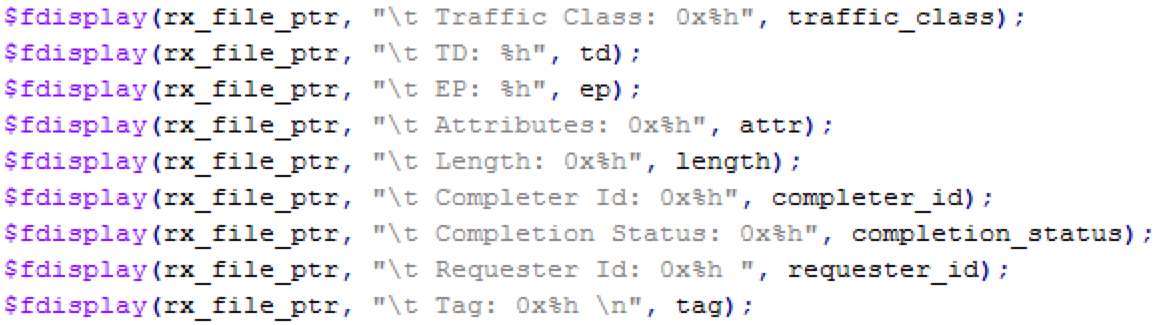
In Figure 2, rx_file_ptr and tx_file_ptr are the file handles for receive data log file and transmit data log file. Both of them are defined as 32 bit reg since ***$fopen*** returns 32 bit file descriptor.



```
     reg    [31:0]            rx_file_ptr;
     reg    [31:0]            tx_file_ptr;

initial begin

  rx_file_ptr = $fopen("rx.dat");

  if (!rx_file_ptr) begin

    $write("ERROR: Could not open rx.dat.\n");
    $finish;

  end

  tx_file_ptr = $fopen("tx.dat");

  if (!tx_file_ptr) begin

    $write("ERROR: Could not open tx.dat.\n");
    $finish;
  end
end
```

Figure 2 - File System Tasks in Verilog

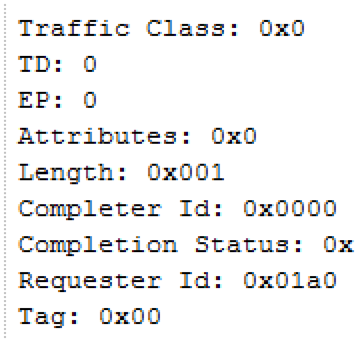
After a file has been successfully opened, the next step is to write to that file. It is done by calling ***$fdisplay*** system task as shown in Figure 3.



```
$fdisplay(rx_file_ptr, "\t Traffic Class: 0x%h", traffic_class);
$fdisplay(rx_file_ptr, "\t TD: %h", td);
$fdisplay(rx_file_ptr, "\t EP: %h", ep);
$fdisplay(rx_file_ptr, "\t Attributes: 0x%h", attr);
$fdisplay(rx_file_ptr, "\t Length: 0x%h", length);
$fdisplay(rx_file_ptr, "\t Completer Id: 0x%h", completer_id);
$fdisplay(rx_file_ptr, "\t Completion Status: 0x%h", completion_status);
$fdisplay(rx_file_ptr, "\t Requester Id: 0x%h ", requester_id);
$fdisplay(rx_file_ptr, "\t Tag: 0x%h \n", tag);
```

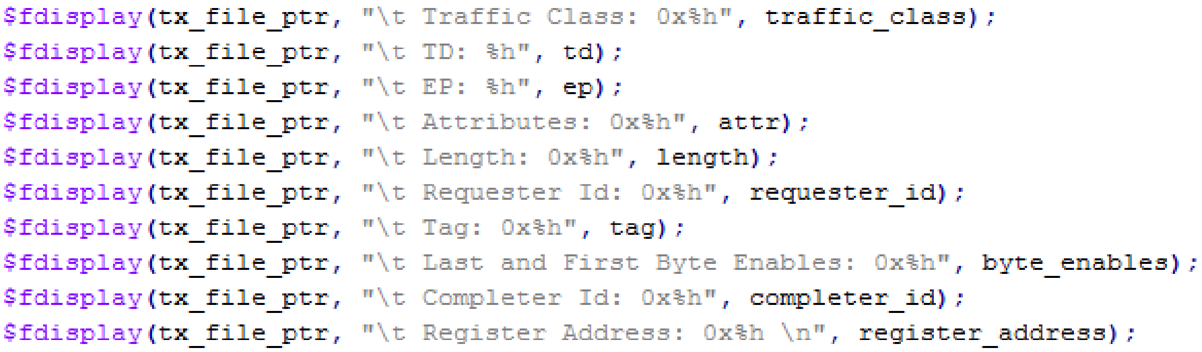
Figure 3 - Writing to a file using $fdisplay

An example output in the file for the above system calls is shown below.

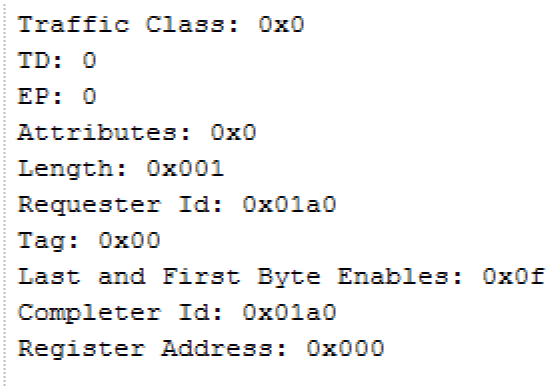


```
Traffic Class: 0x0
TD: 0
EP: 0
Attributes: 0x0
Length: 0x001
Completer Id: 0x0000
Completion Status: 0x0
Requester Id: 0x01a0
Tag: 0x00
```

Figure 4 shows the output for the trasmit data output log.



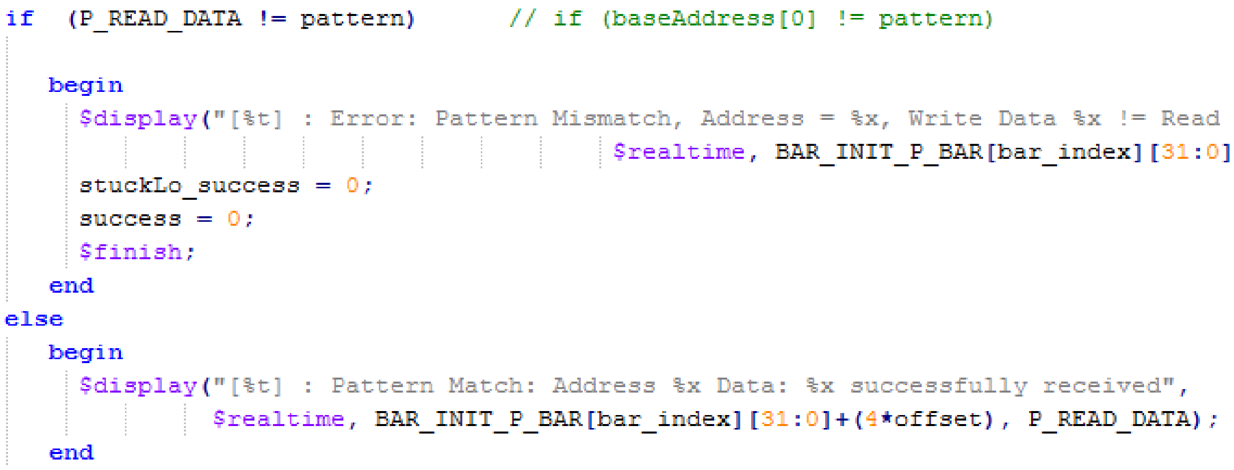
```
$fdisplay(tx_file_ptr, "\t Traffic Class: 0x%h", traffic_class);
$fdisplay(tx_file_ptr, "\t TD: %h", td);
$fdisplay(tx_file_ptr, "\t EP: %h", ep);
$fdisplay(tx_file_ptr, "\t Attributes: 0x%h", attr);
$fdisplay(tx_file_ptr, "\t Length: 0x%h", length);
$fdisplay(tx_file_ptr, "\t Requester Id: 0x%h", requester_id);
$fdisplay(tx_file_ptr, "\t Tag: 0x%h", tag);
$fdisplay(tx_file_ptr, "\t Last and First Byte Enables: 0x%h", byte_enables);
$fdisplay(tx_file_ptr, "\t Completer Id: 0x%h", completer_id);
$fdisplay(tx_file_ptr, "\t Register Address: 0x%h \n", register_address);
```



```
Traffic Class: 0x0
TD: 0
EP: 0
Attributes: 0x0
Length: 0x001
Requester Id: 0x01a0
Tag: 0x00
Last and First Byte Enables: 0x0f
Completer Id: 0x01a0
Register Address: 0x000
```

Figure 4 - Transmit Data Output Log

There are two system tasks for stopping and finishing a simulation. ***$stop*** is used to suspend the simulation and ***$finish*** system task is called to terminate the simulation. Figure 5 shows how ***$finish*** system task is called in a testbench.



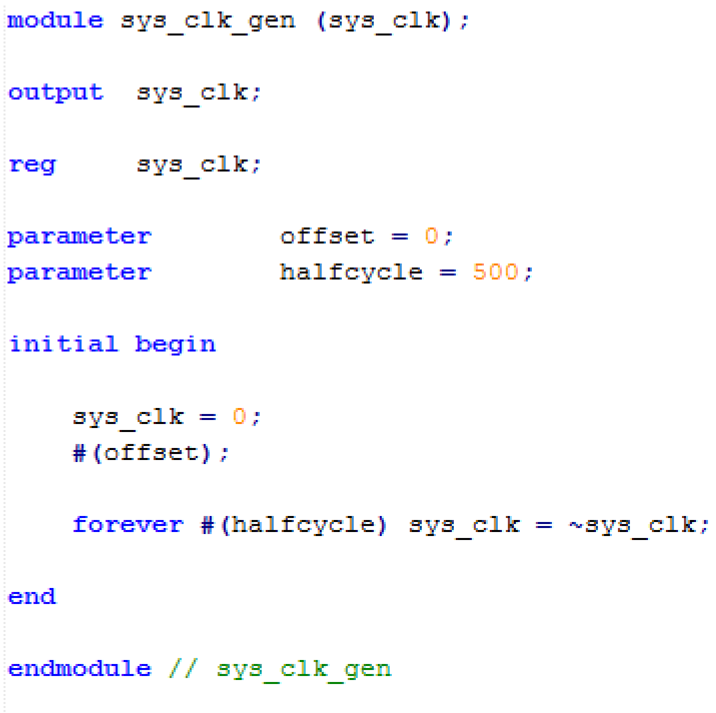
```
if  (P_READ_DATA != pattern)      // if (baseAddress[0] != pattern)

    begin
     $display("[%t] : Error: Pattern Mismatch, Address = %x, Write Data %x != Read
                                                 $realtime, BAR_INIT_P_BAR[bar_index][31:0]
     stuckLo_success = 0;
     success = 0;
     $finish;
    end
else
    begin
     $display("[%t] : Pattern Match: Address %x Data: %x successfully received",
              $realtime, BAR_INIT_P_BAR[bar_index][31:0]+(4*offset), P_READ_DATA);
    end
```

Figure 5 - $finish System Task in Verilog

Looping constructs

Verilog-2001 supports the following looping constructs: *forever, repeat, while,* and *for*. They are useful in Xilinx Integrated PCI Express Block and SRIO cores simulation typically for generating repetitive packets.

- ***forever*** is used to execute statements inside its begin and end block forever. There are no variables to control the loop. The loop exits only after the simulation session terminates.



```
module sys_clk_gen (sys_clk);

output  sys_clk;

reg     sys_clk;

parameter        offset = 0;
parameter        halfcycle = 500;

initial begin

    sys_clk = 0;
    #(offset);

    forever #(halfcycle) sys_clk = ~sys_clk;

end

endmodule // sys_clk_gen
```

- ***repeat*** executes the statements inside it for a fixed number of times.
- ***while*** executes the statements till its expression becomes false.
- ***for*** executes the statements based on the number of times its variable changes.

- ***do-while*** is similar to while loop except that the expression is calculated at the end of the loop.

Structured Procedures

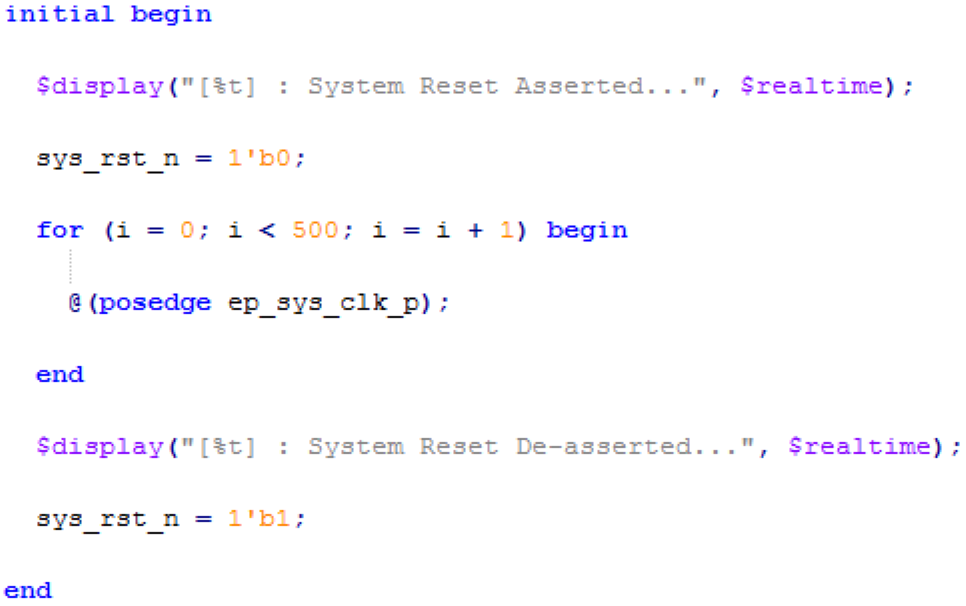
The following are four statements in Verilog that are used for creating structural HDL design. All procedures in Verilog designs are specified in one of the following:

- ***initial***
- ***always***
- ***Task***
- ***Function***

The main difference between *initial* and *always* is that *initial* executes only once i.e. it ceases after all the statements in the initial block have finished executing whereas *always* executes repeatedly and it ceases only when the simulation terminates.

Initial

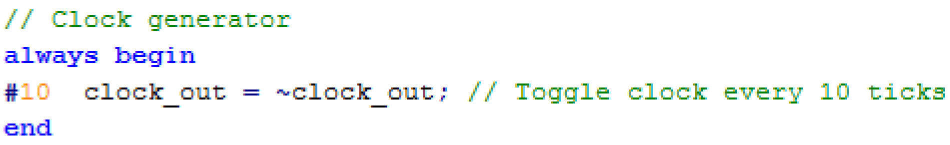
As mentioned above, *initial* block executes only once. If there is more than one *initial* block in the design, it will be executed at the same time independent of each other. *initial* block is a powerful tool in simulation. The test cases provided in this document later use *initial* block to inject packets into core interfaces using the *force* statement and release them later using the *release* statement.



```
initial begin
  $display("[%t] : System Reset Asserted...", $realtime);
  sys_rst_n = 1'b0;
  for (i = 0; i < 500; i = i + 1) begin
    @(posedge ep_sys_clk_p);
  end
  $display("[%t] : System Reset De-asserted...", $realtime);
  sys_rst_n = 1'b1;
end
```

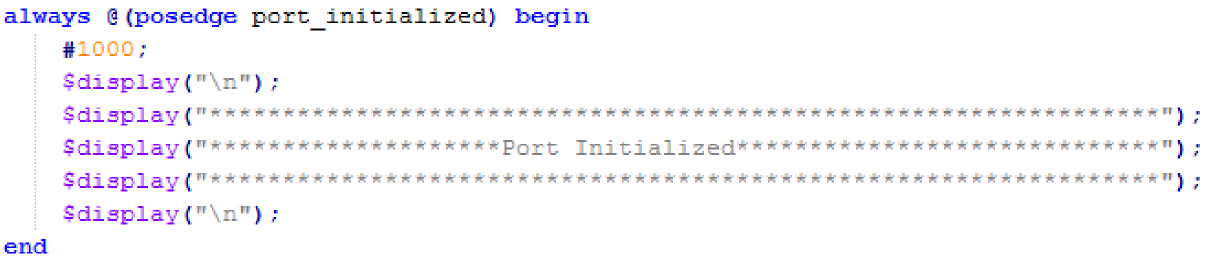
Always

An *always* block executes repeatedly unless the simulation terminates or is stopped by $finish or $stop. An *always* block executes repeatedly unless the simulation terminates or is stopped by $finish or $stop. *always* is used in a design to model a block of activity that is repeated in a looping fashion. In simulation, it is used mostly for generating clock as shown in Figure 6.



```
// Clock generator
always begin
#10  clock_out = ~clock_out; // Toggle clock every 10 ticks
end
```

Figure 6 - Clock Generation Using 'always' statement



```
always @(posedge port_initialized) begin
    #1000;
    $display("\n");
    $display("******************************************************************");
    $display("********************Port Initialized******************************");
    $display("******************************************************************");
    $display("\n");
end
```

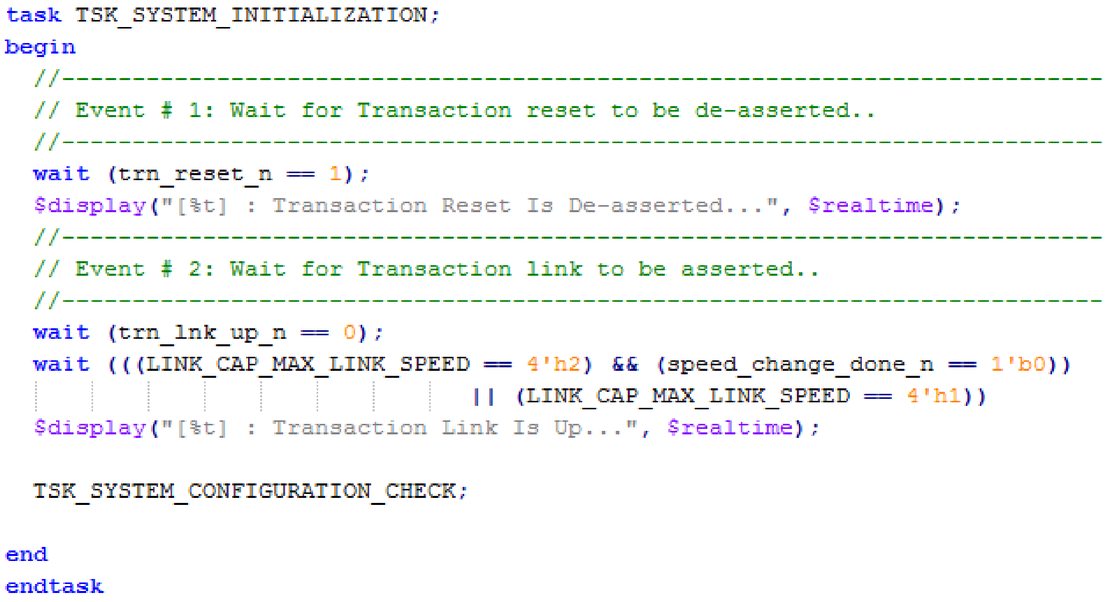
Figure 7 – Message Display on the Output Console after port_initialized assertion in SRIO Simulation

Figure 7 shows an example of *always* block usage in SRIO simulation where the message shown is displayed after the assertion of *port_initialized* is asserted.

Task

Tasks are used to reduce code repetition in a Verilog design. They are called in the main body of the design with necessary input and output parameters. Tasks can also be defined in a separate file instead of embedding into the main code. If defined separately, it is included in the design by using compile directive ***'include***.

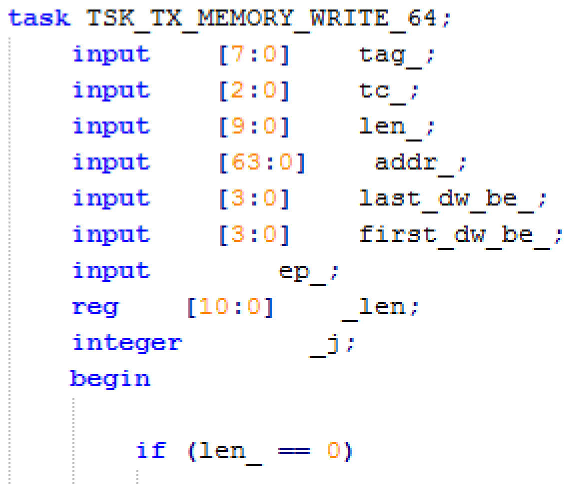
Figure 8 shows a system initialization task in Xilinx Integrated PCI Express core testbench. There is another task called TSK_SYSTEM_CONFIGURATION_CHECK which is separately defined and is called within the definition of TSK_SYSTEM_INITIALIZATION. There are no inputs or outputs in this task.



```
task TSK_SYSTEM_INITIALIZATION;
begin
  //--------------------------------------------------------------------------
  // Event # 1: Wait for Transaction reset to be de-asserted..
  //--------------------------------------------------------------------------
  wait (trn_reset_n == 1);
  $display("[%t] : Transaction Reset Is De-asserted...", $realtime);
  //--------------------------------------------------------------------------
  // Event # 2: Wait for Transaction link to be asserted..
  //--------------------------------------------------------------------------
  wait (trn_lnk_up_n == 0);
  wait (((LINK_CAP_MAX_LINK_SPEED == 4'h2) && (speed_change_done_n == 1'b0))
                                    || (LINK_CAP_MAX_LINK_SPEED == 4'h1))
  $display("[%t] : Transaction Link Is Up...", $realtime);

  TSK_SYSTEM_CONFIGURATION_CHECK;

end
endtask
```

Figure 8 - System Initialization Task in Xilinx Integrated PCI Express Block Example Design Testbench

Figure 9, from Xilinx PCI Express core example design testbench, shows definition of a task to do 64 bits memory write to the downstream endpoint core. It shows a list of inputs this task takes. The output is the correct toggling of the core interface signals to generate 64-bit Memory Write packet.



```
task TSK_TX_MEMORY_WRITE_64;
    input    [7:0]    tag_;
    input    [2:0]    tc_;
    input    [9:0]    len_;
    input    [63:0]    addr_;
    input    [3:0]    last_dw_be_;
    input    [3:0]    first_dw_be_;
    input          ep_;
    reg    [10:0]    _len;
    integer        _j;
    begin

        if (len_ == 0)
```

Figure 9 - 64-bits Memory Write Task in Xilinx Integrated PCI Express Block Example Design Testbench

function

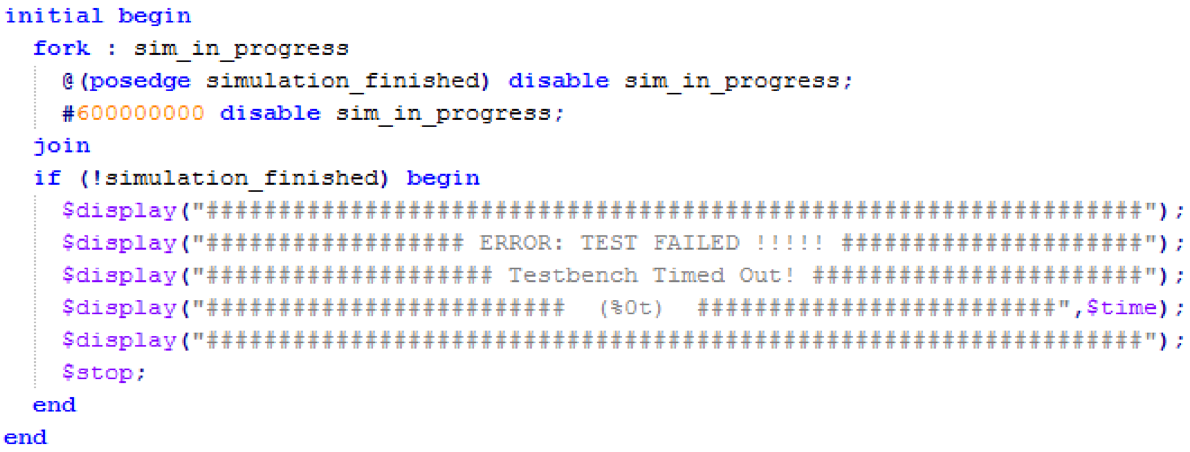
function is similar to task but there are few differences between the two. The major differences to take into account during simulation are listed below:

- *function* executes in Zero simulation time whereas *task* can contain time control statements such as @(posedge.) and delay operator (#).
- *task* doesn't return a value. However, it can have output arguments. This is in contrast to *function* where it returns a value or optionally can be voided.

Fork-join

The *fork-join* pair is used to create parallel processes. All the statements between *fork* and *join* are executed simultaneously as soon as the control hits the *fork* statement. It will exit from a *fork-join* block only after the completion of the longest running statement or block that is defined inside the *fork-join* block. Figure 10 is from the Serial RapidIO Gen2 example design testbench. Here, the *initial* block kicks in at simulation time 0. Both statements defined in the *fork-join* block are executed simultaneously. The simulation will stop in one of two scenarios: *simulation_finished* is asserted i.e. the simulation completed without any issue or after 600000 ns. 600000 ns is the time out value. If *simulation_finished* is not asserted even after 600000 ns, the simulation will time out and exits the *fork-join* block and then displays the error message shown.

disable is a Verilog feature to abort the execution of a task or a block of code. Figure 10 uses *disable* statement to abort the execution of *sim_in_progress* block after *simulation_finished* is asserted or after 600000 ns.



```
initial begin
  fork : sim_in_progress
    @(posedge simulation_finished) disable sim_in_progress;
    #600000000 disable sim_in_progress;
  join
  if (!simulation_finished) begin
    $display("#################################################################");
    $display("################## ERROR: TEST FAILED !!!!! #####################");
    $display("#################### Testbench Timed Out! #######################");
    $display("#########################  (%0t)  #########################",$time);
    $display("#################################################################");
    $stop;
  end
end
```

Figure 10 - fork-join Statement in Verilog

Xilinx IP Simulation

When generating Xilinx Integrated PCI Express Block and Serial Rapid IO cores in Coregen or Vivado, there are two HDL options: VHDL and Verilog. Verilog is more powerful for simulation due to different simulation specific constructs that are integrated with the language. Both Coregen and Vivado allow generating the cores in both languages. If Verilog is selected, the output files including the wrapper and example design files will all be in Verilog.

The simulation of Xilinx Integrated PCI Express and Serial Rapid IO cores are straight forward as all necessary setup to get a user started with the simulation is provided with the generation of the cores. When generating these cores in Coregen or Vivado, the tools generate all required files for the core and also generate example design with a top level wrapper that instantiates both example user application and the core top level wrapper module. This design is synthesizable. Users can quickly implement this design and download it on their board to check the general functionality of the cores. Apart from the example design, a test bench is also generated to simulate the entire design. The users can run simulation in one of the supported simulators by just running the provided script which is generated along with the generation of the cores.

Xilinx Integrated PCI Express Block Simulation

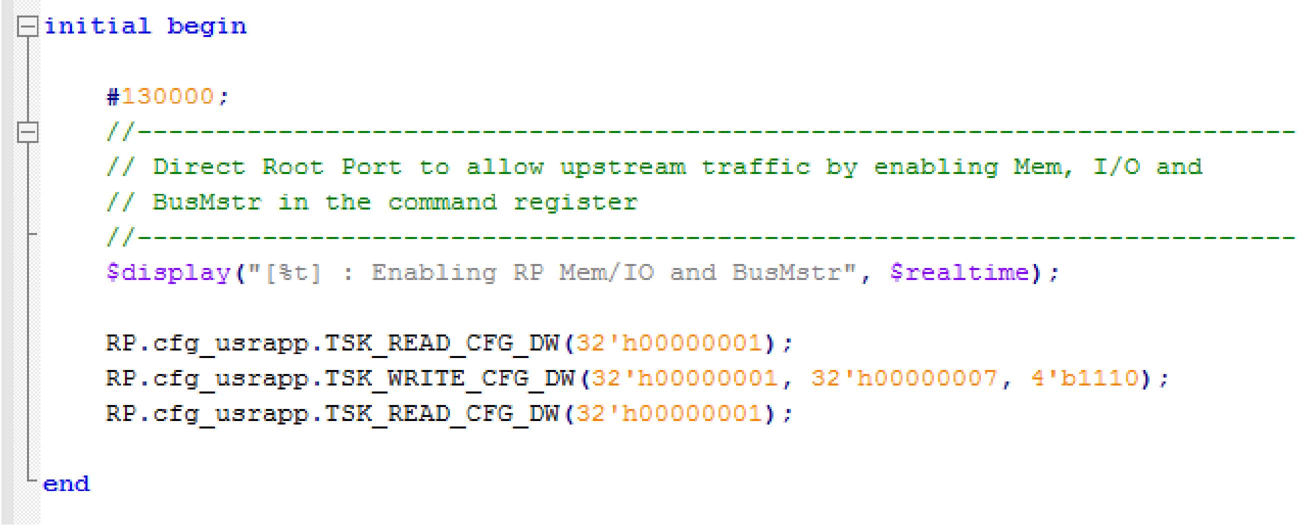
This section describes techniques to reproduce specific test cases with the Xilinx Integrated PCI Express Block example design simulation. As mentioned above, this is done by using *force-release* feature in Verilog. With this technique, users can reproduce different test cases such as generation of MSI interrupt, generation of MWr/MRd packets, malformed packets from Endpoint to the Root Port etc.

MSI Interrupt Simulation

The Xilinx Integrated PCI Express Block example design simulation does not contain MSI generation. How to generate an MSI from Endpoint to the Root Port is described in the respective product guides. This section describes how to generate an MSI in the example design simulation for the core generated for Kintex-7.

To generate an MSI, follow the steps below:

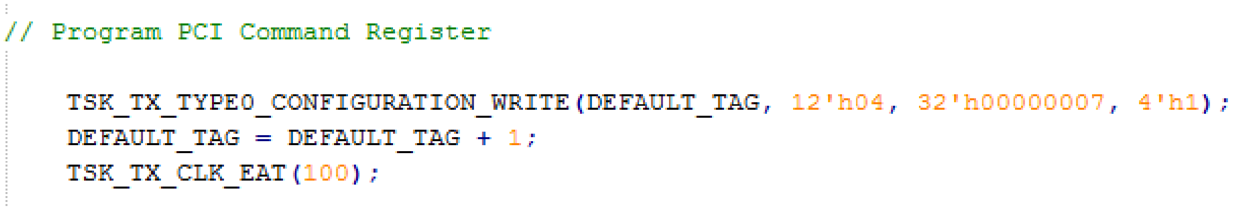
1. Enable ‘Bus Master Enable’ bit in the Root Port by adding the code shown in Figure 11, in *board.v*.



```
initial begin

    #130000;
    //-----------------------------------------------------------------------------
    // Direct Root Port to allow upstream traffic by enabling Mem, I/O and
    // BusMstr in the command register
    //-----------------------------------------------------------------------------
    $display("[%t] : Enabling RP Mem/IO and BusMstr", $realtime);

    RP.cfg_usrapp.TSK_READ_CFG_DW(32'h00000001);
    RP.cfg_usrapp.TSK_WRITE_CFG_DW(32'h00000001, 32'h00000007, 4'b1110);
    RP.cfg_usrapp.TSK_READ_CFG_DW(32'h00000001);

end
```

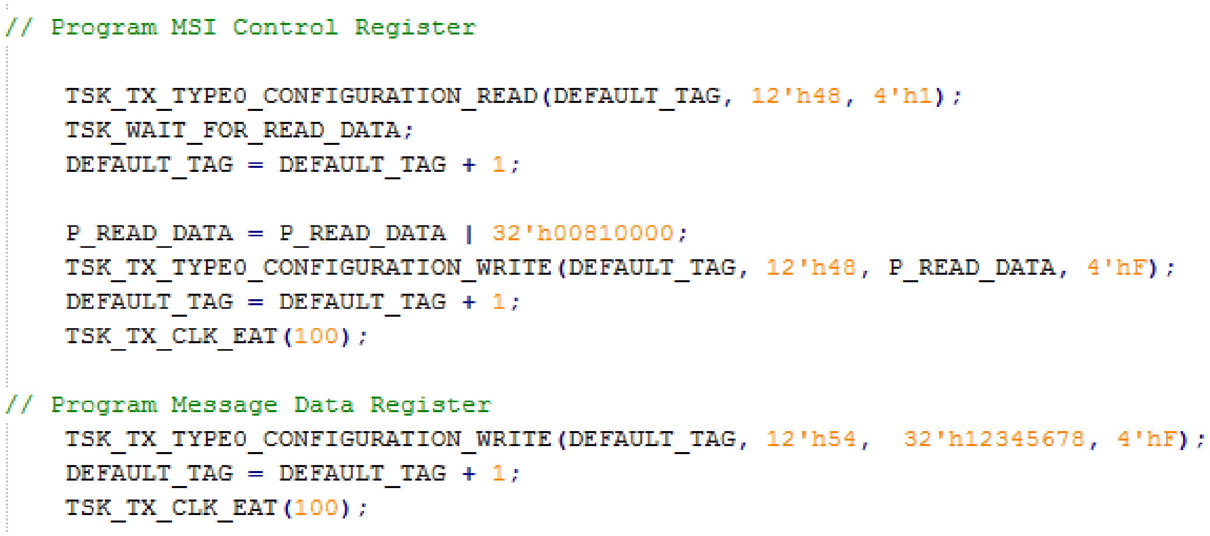
Figure 11 – Enabling ‘Bus Master Enable’ bit in Root Port Command Register

2. In *dsport/pci_exp_usrapp_tx.v*, do the following modification to enable Bus Master Enable bit in the Endpoint Command Register.



```
// Program PCI Command Register

    TSK_TX_TYPE0_CONFIGURATION_WRITE(DEFAULT_TAG, 12'h04, 32'h00000007, 4'h1);
    DEFAULT_TAG = DEFAULT_TAG + 1;
    TSK_TX_CLK_EAT(100);
```

3. Add the following in *dsport/pci_exp_usrapp_tx.v* to configure the MSI registers.



```
// Program MSI Control Register

    TSK_TX_TYPE0_CONFIGURATION_READ(DEFAULT_TAG, 12'h48, 4'h1);
    TSK_WAIT_FOR_READ_DATA;
    DEFAULT_TAG = DEFAULT_TAG + 1;

    P_READ_DATA = P_READ_DATA | 32'h00810000;
    TSK_TX_TYPE0_CONFIGURATION_WRITE(DEFAULT_TAG, 12'h48, P_READ_DATA, 4'hF);
    DEFAULT_TAG = DEFAULT_TAG + 1;
    TSK_TX_CLK_EAT(100);

// Program Message Data Register
    TSK_TX_TYPE0_CONFIGURATION_WRITE(DEFAULT_TAG, 12'h54,  32'h12345678, 4'hF);
    DEFAULT_TAG = DEFAULT_TAG + 1;
    TSK_TX_CLK_EAT(100);
```

4. Figure 12 shows interface waveform for requesting interrupt service from the user application to the core. Users should assert *cfg_interrupt* and provide the Message data on *cfg_interrupt_di* as shown in the waveform. A detailed information on generating an MSI interrupt request can be found in PG054 [1].

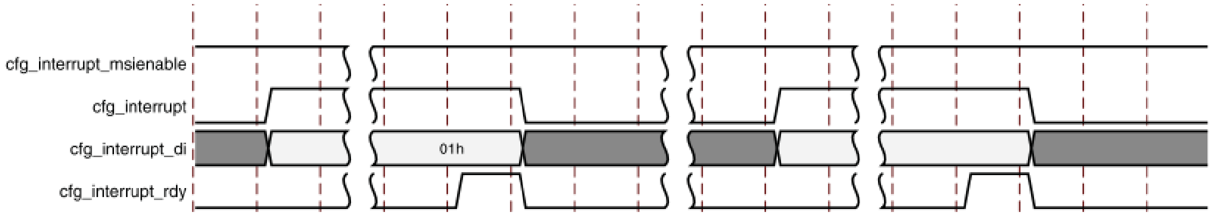
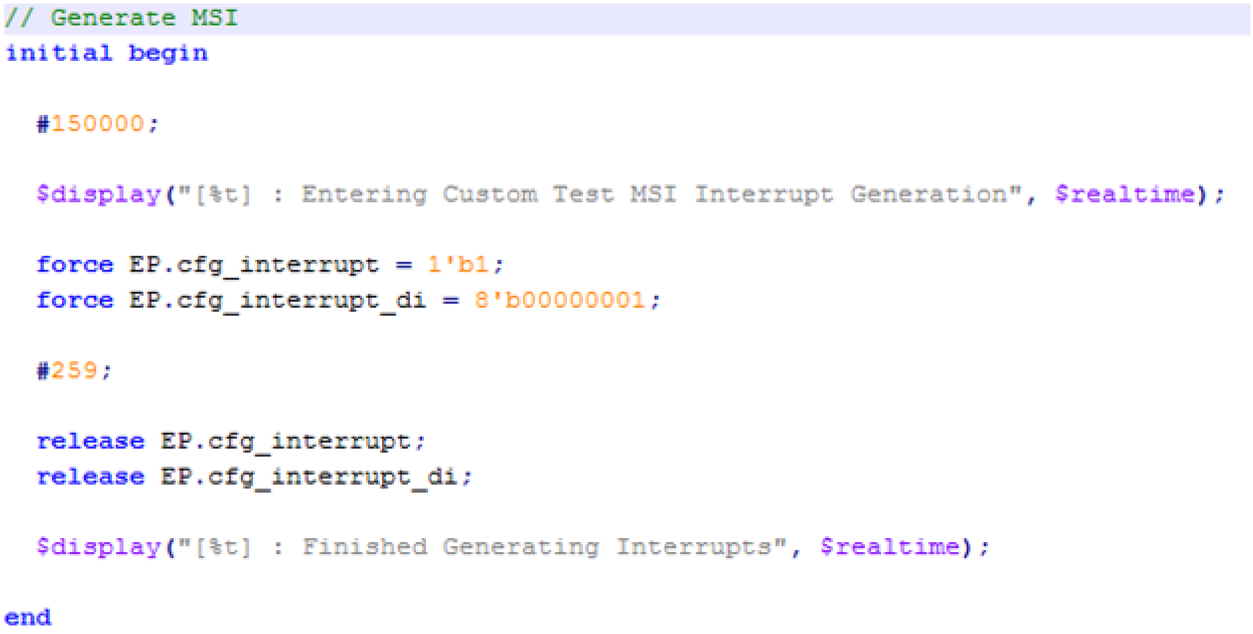


Figure 12 – Requesting Interrupt Service

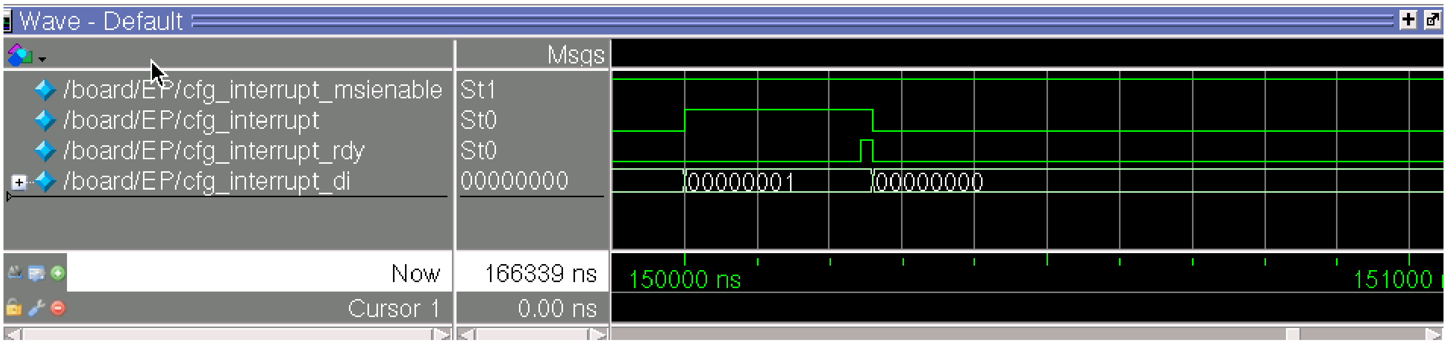
In order to generate an MSI, assert *cfg_interrupt* and provide Message data on *cfg_interrupt_di*. This can be done by using verilog *force-release* statements as shown in Figure 13. Add the following code snippet to board.v.



```
// Generate MSI
initial begin

  #150000;

  $display("[%t] : Entering Custom Test MSI Interrupt Generation", $realtime);

  force EP.cfg_interrupt = 1'b1;
  force EP.cfg_interrupt_di = 8'b00000001;

  #259;

  release EP.cfg_interrupt;
  release EP.cfg_interrupt_di;

  $display("[%t] : Finished Generating Interrupts", $realtime);

end
```

Figure 13 – MSI Generation using *force/release* Statement

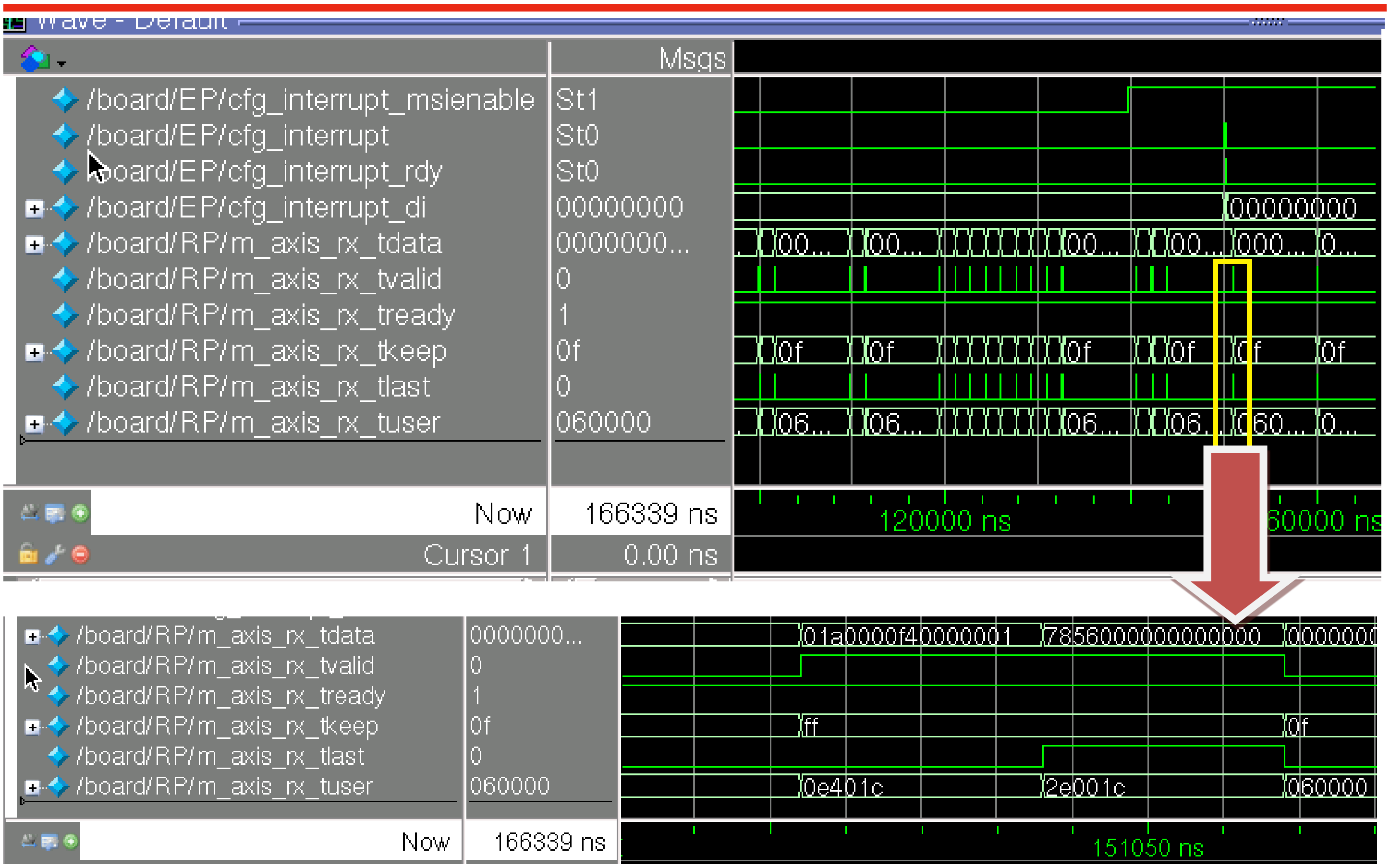
Figure 14 shows the interrupt configuration interface after adding the above code.



```
/board/EP/cfg_interrupt_msienable   St1
/board/EP/cfg_interrupt             St0
/board/EP/cfg_interrupt_rdy         St0
/board/EP/cfg_interrupt_di          00000000      00000001      00000000

Now        166339 ns     150000 ns    151000
Cursor 1   0.00 ns
```

Figure 14 – Requesting MSI in Simulation

Figure 15 shows MSI at the receive side of the Root Port.



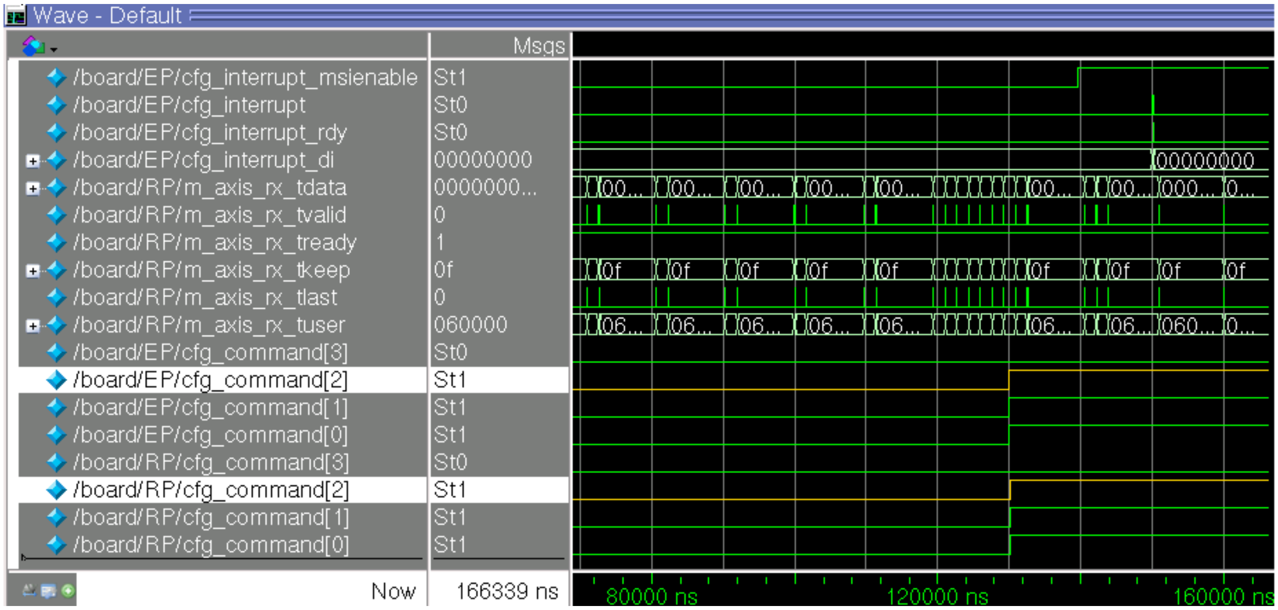
Wave - Default

	Msgs
/board/EP/cfg_interrupt_msienable	St1
/board/EP/cfg_interrupt	St0
/board/EP/cfg_interrupt_rdy	St0
/board/EP/cfg_interrupt_di	00000000
/board/RP/m_axis_rx_tdata	0000000...
/board/RP/m_axis_rx_tvalid	0
/board/RP/m_axis_rx_tready	1
/board/RP/m_axis_rx_tkeep	0f
/board/RP/m_axis_rx_tlast	0
/board/RP/m_axis_rx_tuser	060000
Now	166339 ns
Cursor 1	0.00 ns

/board/RP/m_axis_rx_tdata	0000000...
/board/RP/m_axis_rx_tvalid	0
/board/RP/m_axis_rx_tready	1
/board/RP/m_axis_rx_tkeep	0f
/board/RP/m_axis_rx_tlast	0
/board/RP/m_axis_rx_tuser	060000
Now	166339 ns

Figure 15 – MSI TLP at Root Port

While simulating MSI or debugging MSI issues in hardware, make sure ‘Bus Master Enable’ bit in both Endpoint and Root Port are asserted. You could do so by checking *cfg_command[2]* as shown in Figure 16:



Wave - Default

	Msgs
/board/EP/cfg_interrupt_msienable	St1
/board/EP/cfg_interrupt	St0
/board/EP/cfg_interrupt_rdy	St0
/board/EP/cfg_interrupt_di	00000000
/board/RP/m_axis_rx_tdata	0000000...
/board/RP/m_axis_rx_tvalid	0
/board/RP/m_axis_rx_tready	1
/board/RP/m_axis_rx_tkeep	0f
/board/RP/m_axis_rx_tlast	0
/board/RP/m_axis_rx_tuser	060000
/board/EP/cfg_command[3]	St0
/board/EP/cfg_command[2]	St1
/board/EP/cfg_command[1]	St1
/board/EP/cfg_command[0]	St1
/board/RP/cfg_command[3]	St0
/board/RP/cfg_command[2]	St1
/board/RP/cfg_command[1]	St1
/board/RP/cfg_command[0]	St1
Now	166339 ns

Figure 16 – ‘Bus Master Bit’ assertion in both Endpoint and Root Port

Upstream Memory Write TLP Simulation

The example design doesn't have a feature to send a TLP upstream from the Endpoint to the Root Port. One could add this feature in the user application or just inject the packet into the TX user interface using *force-release* to send the required packet (MWr or MRd) upstream.

Figure 17 shows the Endpoint transmit user interface in the default example design simulation. In the default simulation, you would see only one packet i.e. a completion packet (yellow circle in waveform shown below) on the Endpoint TX user interface for an incoming MRd from the Root Port.

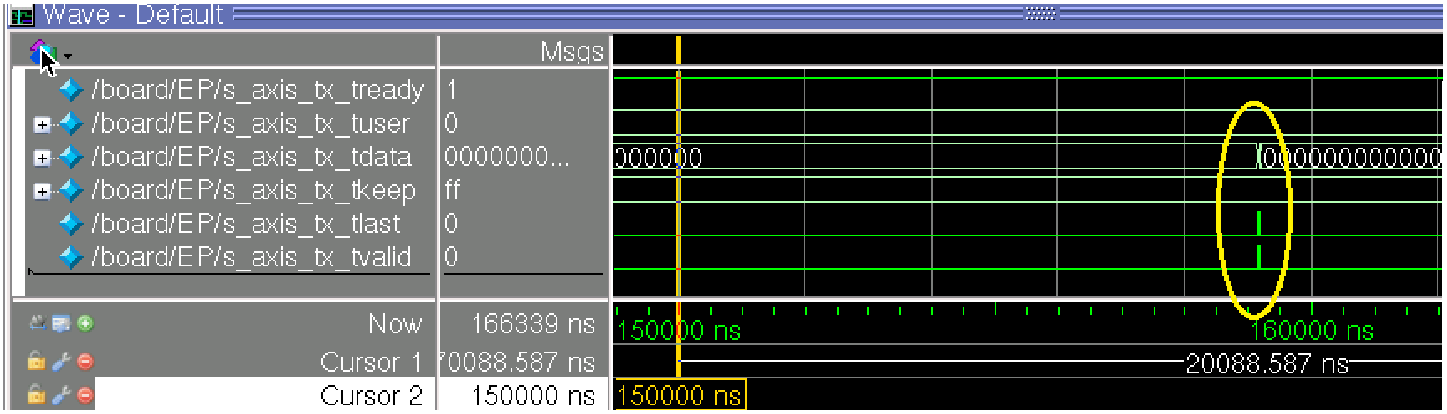
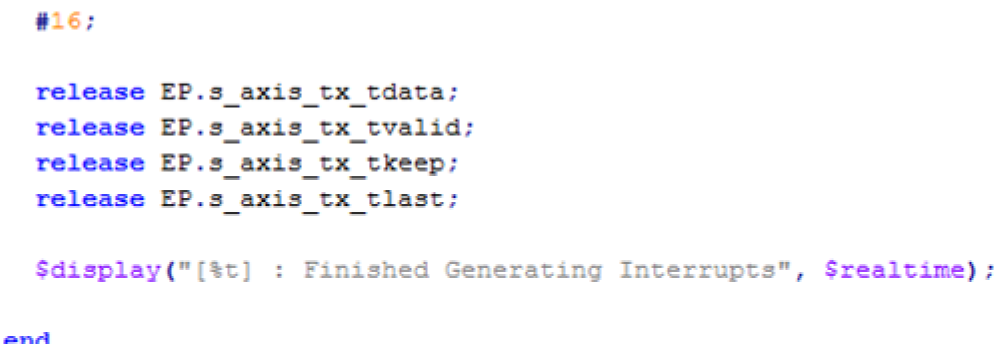


Figure 17 – Completion Packet at Endpoint Transmit Interface

Add the code below, in *board.v,* to inject MWr packet at the cursor shown in the waveform in Figure 17.

```
// Generate TLP,  from EP to RP
initial begin

  #150000;

  $display("[%t] : Entering Custom Test TLP Generation from EP to RP", $realtime);

  force EP.s_axis_tx_tdata = 64'h01a0090f40000001;
  force EP.s_axis_tx_tvalid = 1'b1;
  force EP.s_axis_tx_tkeep = 8'hFF;
  force EP.s_axis_tx_tlast = 1'b0;

  #16;

  force EP.s_axis_tx_tdata = 64'h0403020100000010;
  force EP.s_axis_tx_tvalid = 1'b1;
  force EP.s_axis_tx_tkeep = 8'hFF;
  force EP.s_axis_tx_tlast = 1'b1;
```



```
  #16;

  release EP.s_axis_tx_tdata;
  release EP.s_axis_tx_tvalid;
  release EP.s_axis_tx_tkeep;
  release EP.s_axis_tx_tlast;

  $display("[%t] : Finished Generating Interrupts", $realtime);

end
```

Figure 18 – Generating MWr TLP from Endpoint to Root Port using force/release Statement

The waveform in Figure 19 shows the TX Endpoint interface simulation after the above modification of the *board.v* file.

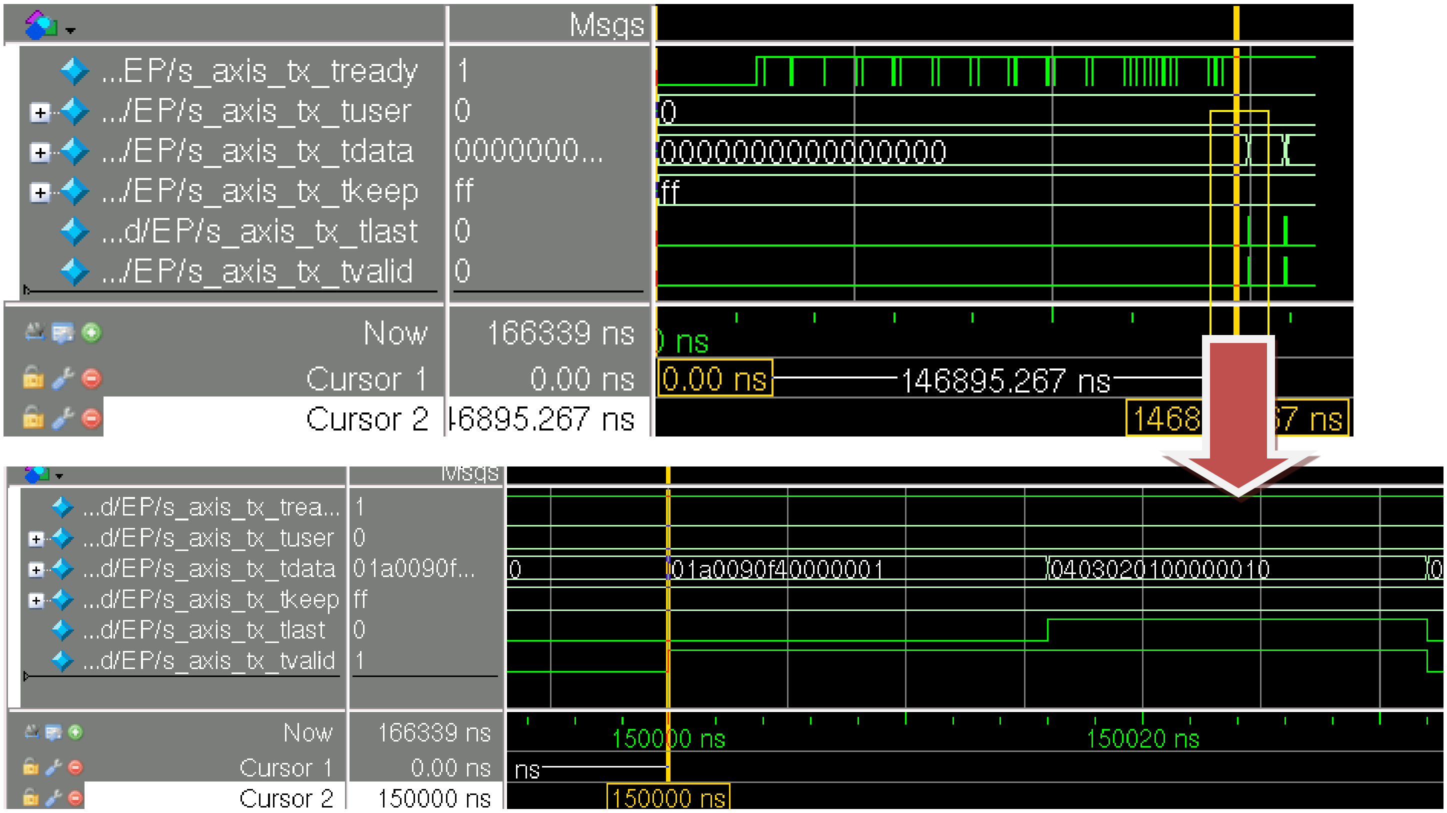


Figure 19 – MWr TLP on Endpoint Transmit Interface

Figure 20 shows the MWr packet at the receive side of the Root Port.

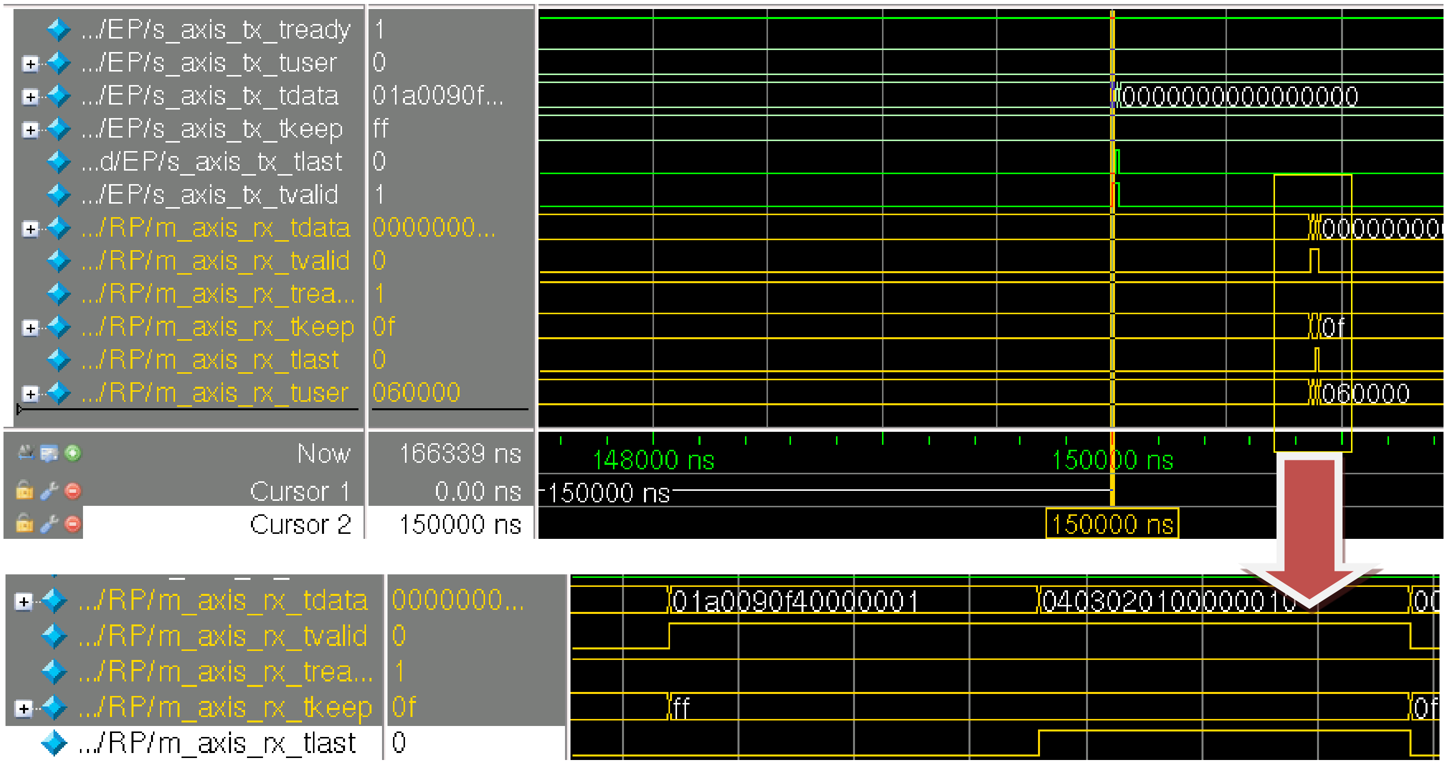
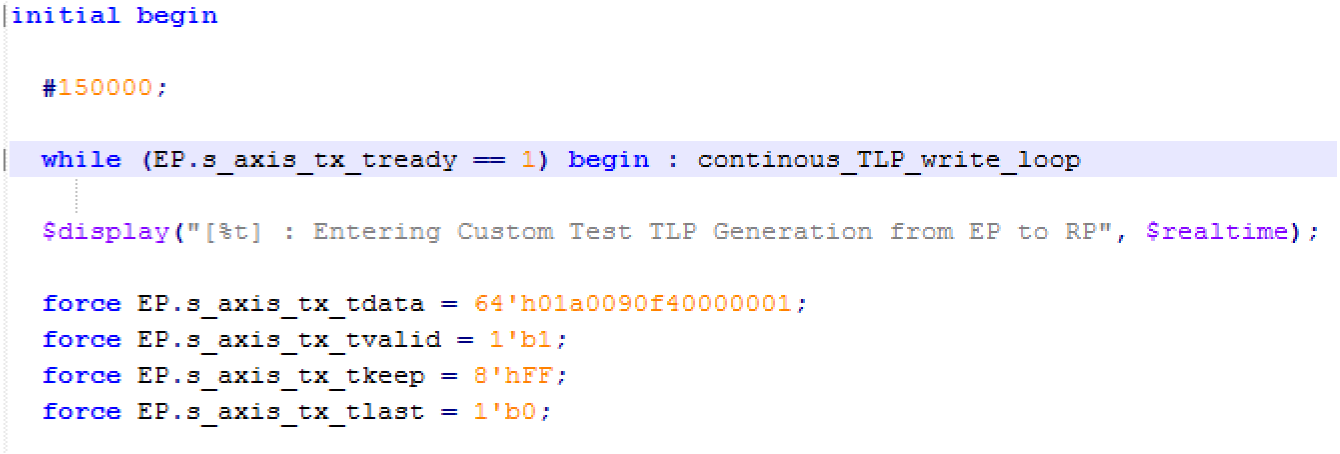


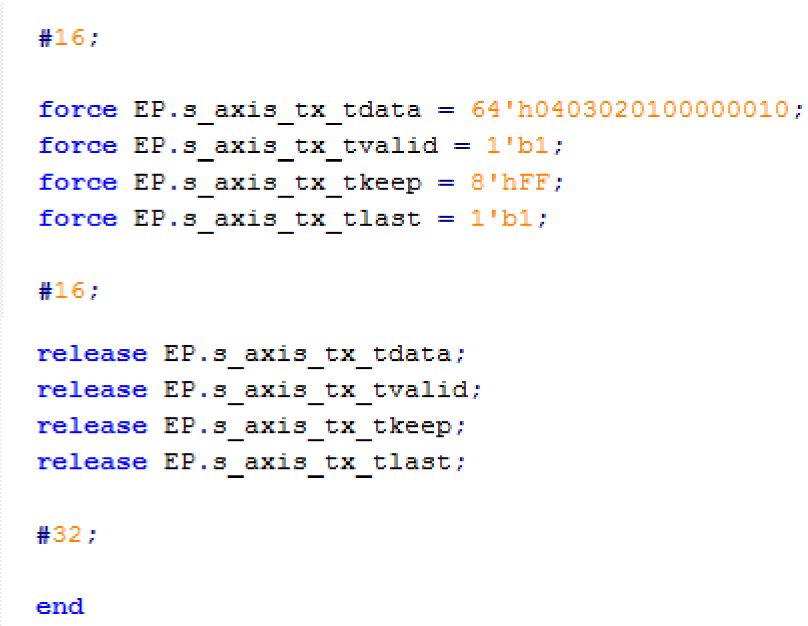
Figure 20 - MWr TLP on Root Port Receive Interface

To simulate upstream MWr/MRd, make sure to enable 'Bus Master Enable' bit in both Root Port and Endpoint command register as described in 'MSI Simulation' section above.

Continuous MWr packets upstream can be simulated by using a while loop as shown in Figure 21.



```
initial begin

  #150000;

  while (EP.s_axis_tx_tready == 1) begin : continous_TLP_write_loop

  $display("[%t] : Entering Custom Test TLP Generation from EP to RP", $realtime);

  force EP.s_axis_tx_tdata = 64'h01a0090f40000001;
  force EP.s_axis_tx_tvalid = 1'b1;
  force EP.s_axis_tx_tkeep = 8'hFF;
  force EP.s_axis_tx_tlast = 1'b0;
```



```
#16;

force EP.s_axis_tx_tdata = 64'h0403020100000010;
force EP.s_axis_tx_tvalid = 1'b1;
force EP.s_axis_tx_tkeep = 8'hFF;
force EP.s_axis_tx_tlast = 1'b1;

#16;

release EP.s_axis_tx_tdata;
release EP.s_axis_tx_tvalid;
release EP.s_axis_tx_tkeep;
release EP.s_axis_tx_tlast;

#32;

end
```

Figure 21 – Simulating Continuous MWr TLPs from Endpoint to Rootport

Figure 22 shows multiple MWr packets heading upstream on the transmit side of the user interface (interface between the user application and the core).

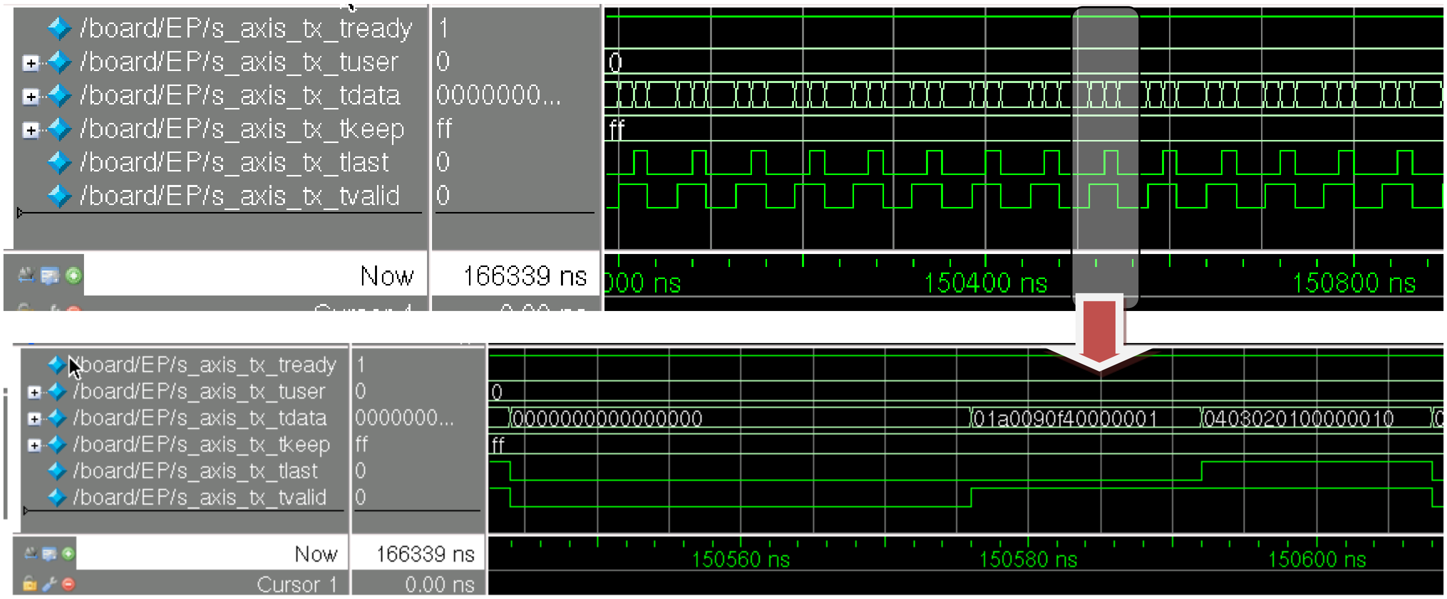


Figure 22 – Multiple MWr TLPs on Endpoint Transmit Interface

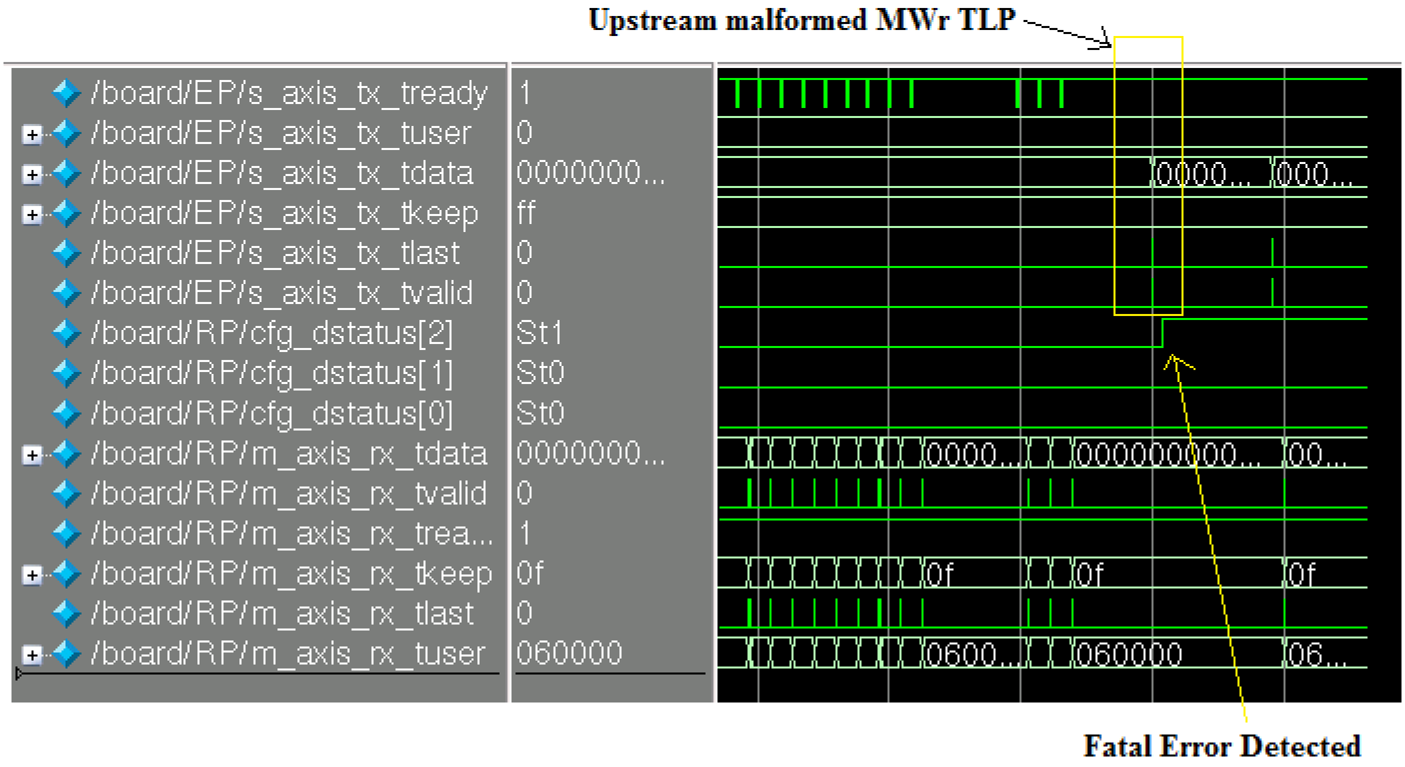
Malformed TLP Simulation

According to the PCI Express Base Specification, if the length field doesn't match the actual payload in the packet it results in a malformed TLP which is a fatal error. This is one of the causes that results in the generation of a malformed TLP. In simulation and in hardware, this can be checked by monitoring *cfg_dstatus[2]*. The code in Figure 23 has been modified to assign length field to 3DW. The actual length of the data in the MWr packet is 1DW.

```
force EP.s_axis_tx_tdata = 64'h01a0090f40000003;
force EP.s_axis_tx_tvalid = 1'b1;
force EP.s_axis_tx_tkeep = 8'hFF;
force EP.s_axis_tx_tlast = 1'b0;
```

Figure 23 – MWr TLP Modification to generate a Malformed TLP

As this TLP travels through to the Root Port, the dsport model asserts *cfg_dstatus[2],* indicating a receipt of a malformed TLP. This is shown in Figure 24:



Upstream malformed MWr TLP

/board/EP/s_axis_tx_tready	1
/board/EP/s_axis_tx_tuser	0
/board/EP/s_axis_tx_tdata	0000000...
/board/EP/s_axis_tx_tkeep	ff
/board/EP/s_axis_tx_tlast	0
/board/EP/s_axis_tx_tvalid	0
/board/RP/cfg_dstatus[2]	St1
/board/RP/cfg_dstatus[1]	St0
/board/RP/cfg_dstatus[0]	St0
/board/RP/m_axis_rx_tdata	0000000...
/board/RP/m_axis_rx_tvalid	0
/board/RP/m_axis_rx_trea...	1
/board/RP/m_axis_rx_tkeep	0f
/board/RP/m_axis_rx_tlast	0
/board/RP/m_axis_rx_tuser	060000

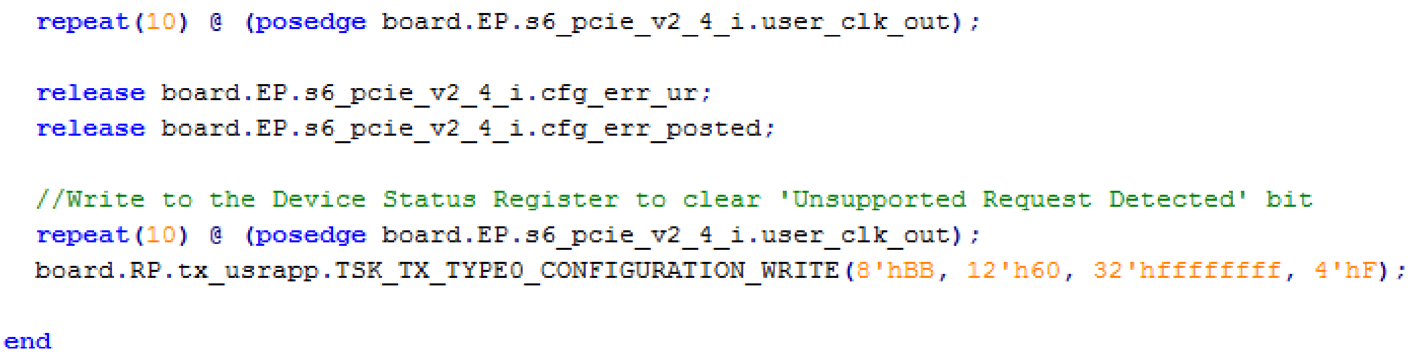
Fatal Error Detected

Figure 24 – Malformed TLP from Endpoint to Root Port and Fatal Error detection at the Root Port

Spartan-6 'Unsupported Request Detected' Simulation

This section describes Spartan-6 PCI Express Block 'Unsupported Request Detected' simulation. Add the following code (Figure 25) in *simulation/tests.v* of the example design project files generated during the generation of the core.

```
initial begin

  repeat(3700) @ (posedge board.EP.s6_pcie_v2_4_i.user_clk_out);

  //Flag unsupported request detected to the PCIe hard block
  force board.EP.s6_pcie_v2_4_i.cfg_err_ur = 1'b1;
  force board.EP.s6_pcie_v2_4_i.cfg_err_posted = 1'b1;
```

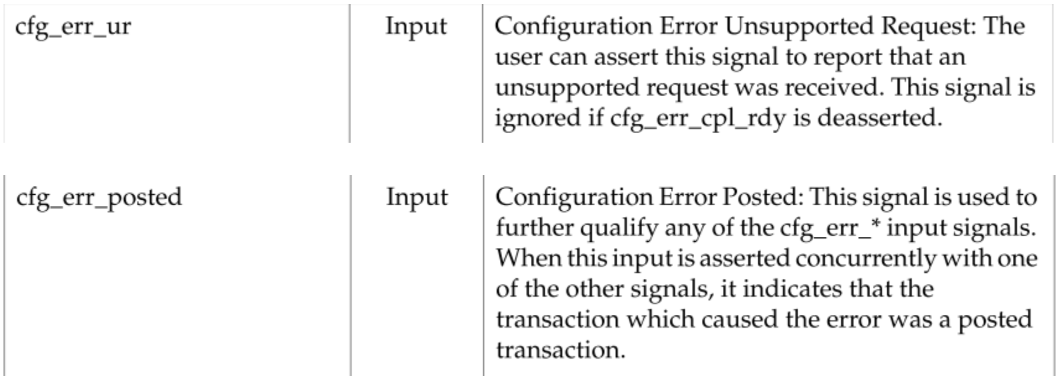


```
  repeat(10) @ (posedge board.EP.s6_pcie_v2_4_i.user_clk_out);

  release board.EP.s6_pcie_v2_4_i.cfg_err_ur;
  release board.EP.s6_pcie_v2_4_i.cfg_err_posted;

  //Write to the Device Status Register to clear 'Unsupported Request Detected' bit
  repeat(10) @ (posedge board.EP.s6_pcie_v2_4_i.user_clk_out);
  board.RP.tx_usrapp.TSK_TX_TYPE0_CONFIGURATION_WRITE(8'hBB, 12'h60, 32'hffffffff, 4'hF);

end
```

Figure 25 – Setting and Clearing ‘Unsupported Request Detected’ bit in Device Status Register

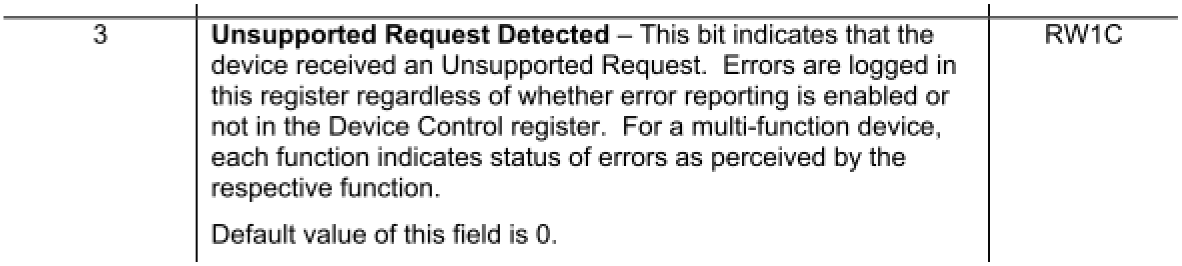
The code does the following:

1. Assert *cfg_err_ur* and *cfg_err_posted* to inform the core that the user application has detected unsupported request.



cfg_err_ur	Input	Configuration Error Unsupported Request: The user can assert this signal to report that an unsupported request was received. This signal is ignored if cfg_err_cpl_rdy is deasserted.
cfg_err_posted	Input	Configuration Error Posted: This signal is used to further qualify any of the cfg_err_* input signals. When this input is asserted concurrently with one of the other signals, it indicates that the transaction which caused the error was a posted transaction.

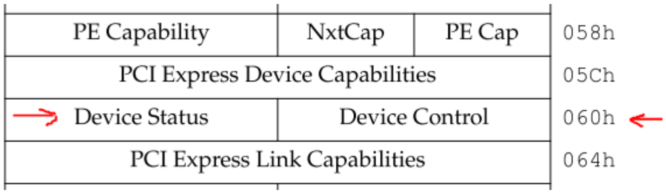
As soon as these signals are asserted, the unsupported request detected bit in the Device Status Register is set as described in the spec.



3	**Unsupported Request Detected** – This bit indicates that the device received an Unsupported Request. Errors are logged in this register regardless of whether error reporting is enabled or not in the Device Control register. For a multi-function device, each function indicates status of errors as perceived by the respective function. Default value of this field is 0.	RW1C

There is a signal called *dbg_reg_detected_unsupported* in the core.This signal is a mirror of the ‘Unsupported Request Detected’ in the Device Status Register. This bit is cleared by writing to this bit from the root complex.

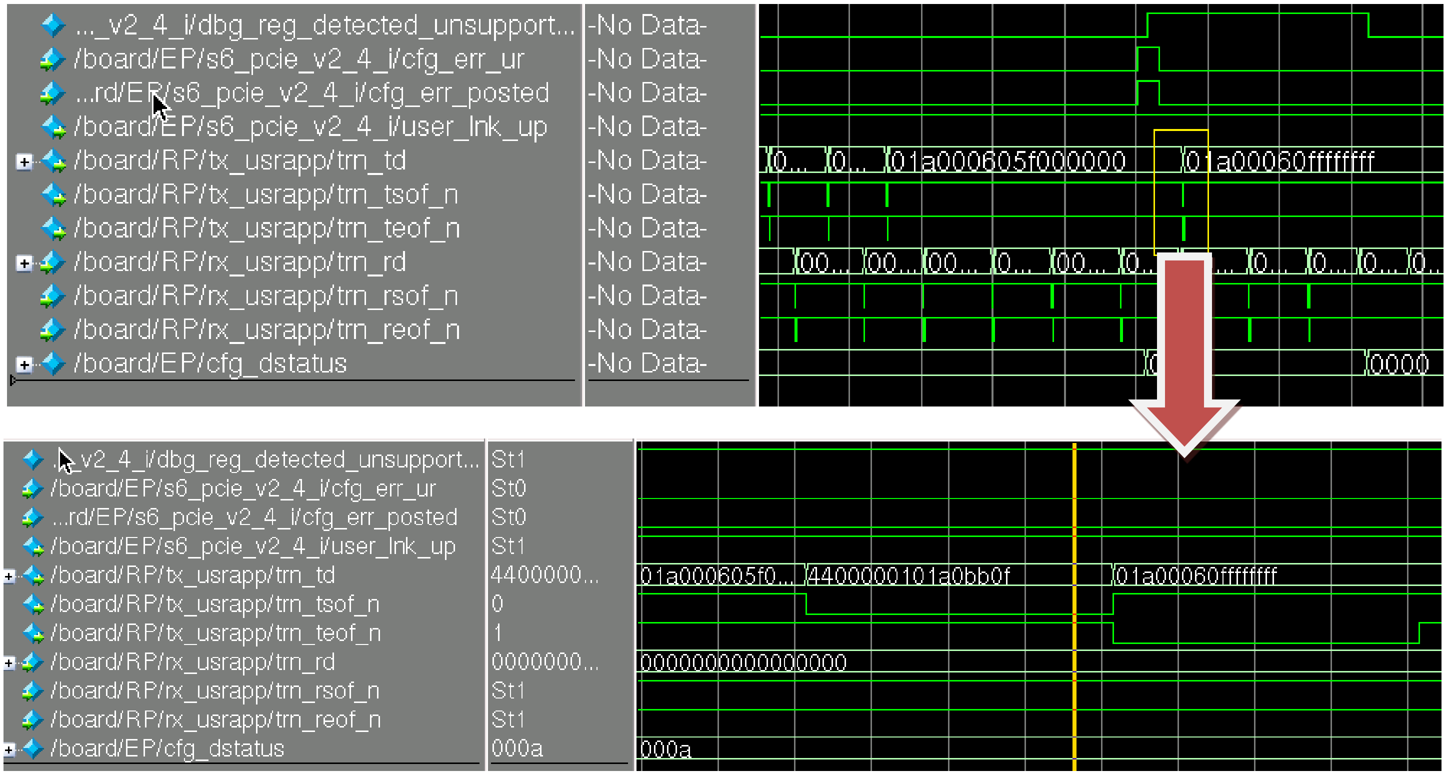
2. Write to the Device Status register at address 12h’60 to clear ‘*Unsupported Request Detected’* bit.



PE Capability	NxtCap	PE Cap	058h
PCI Express Device Capabilities			05Ch
Device Status	Device Control		060h
PCI Express Link Capabilities			064h

Figure 26 – PCI Express Configuration Space

Figure 27 shows the assertion of *dbg_reg_detected_unsupported* signal as a result of asserting *cfg_err_ur* and *cfg_err_posted* from the user sider. Both these signals are input to the core. The box in yellow shows the Configuration Memory Write transaction issued from the Root Port to write to the Device Status Register to clear the 'Unsupported Request Detected' bit.



..._v2_4_i/dbg_reg_detected_unsupport...	-No Data-
/board/EP/s6_pcie_v2_4_i/cfg_err_ur	-No Data-
...rd/EP/s6_pcie_v2_4_i/cfg_err_posted	-No Data-
/board/EP/s6_pcie_v2_4_i/user_lnk_up	-No Data-
/board/RP/tx_usrapp/trn_td	-No Data-
/board/RP/tx_usrapp/trn_tsof_n	-No Data-
/board/RP/tx_usrapp/trn_teof_n	-No Data-
/board/RP/rx_usrapp/trn_rd	-No Data-
/board/RP/rx_usrapp/trn_rsof_n	-No Data-
/board/RP/rx_usrapp/trn_reof_n	-No Data-
/board/EP/cfg_dstatus	-No Data-

..v2_4_i/dbg_reg_detected_unsupport...	St1
/board/EP/s6_pcie_v2_4_i/cfg_err_ur	St0
..rd/EP/s6_pcie_v2_4_i/cfg_err_posted	St0
/board/EP/s6_pcie_v2_4_i/user_lnk_up	St1
/board/RP/tx_usrapp/trn_td	4400000...
/board/RP/tx_usrapp/trn_tsof_n	0
/board/RP/tx_usrapp/trn_teof_n	1
/board/RP/rx_usrapp/trn_rd	0000000...
/board/RP/rx_usrapp/trn_rsof_n	St1
/board/RP/rx_usrapp/trn_reof_n	St1
/board/EP/cfg_dstatus	000a

Figure 27 – Assertion and Deassertion of dgb_reg_detected_unsupported

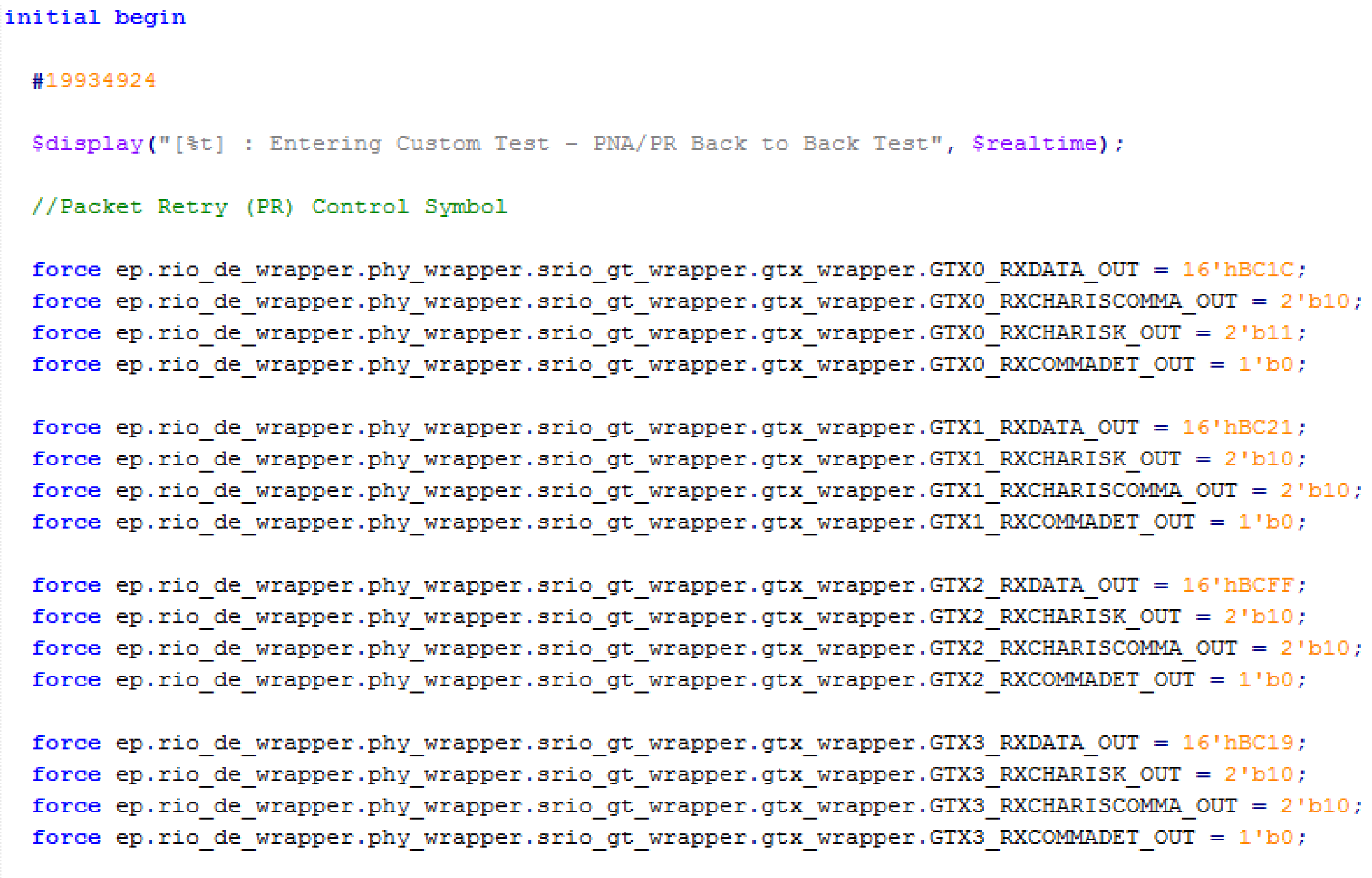
Serial Rapid IO Simulation

Sometimes in SRIO systems, the core may react unexpectedly due to certain data pattern received from the link partner. This section describes how to reproduce such issues in simulation.

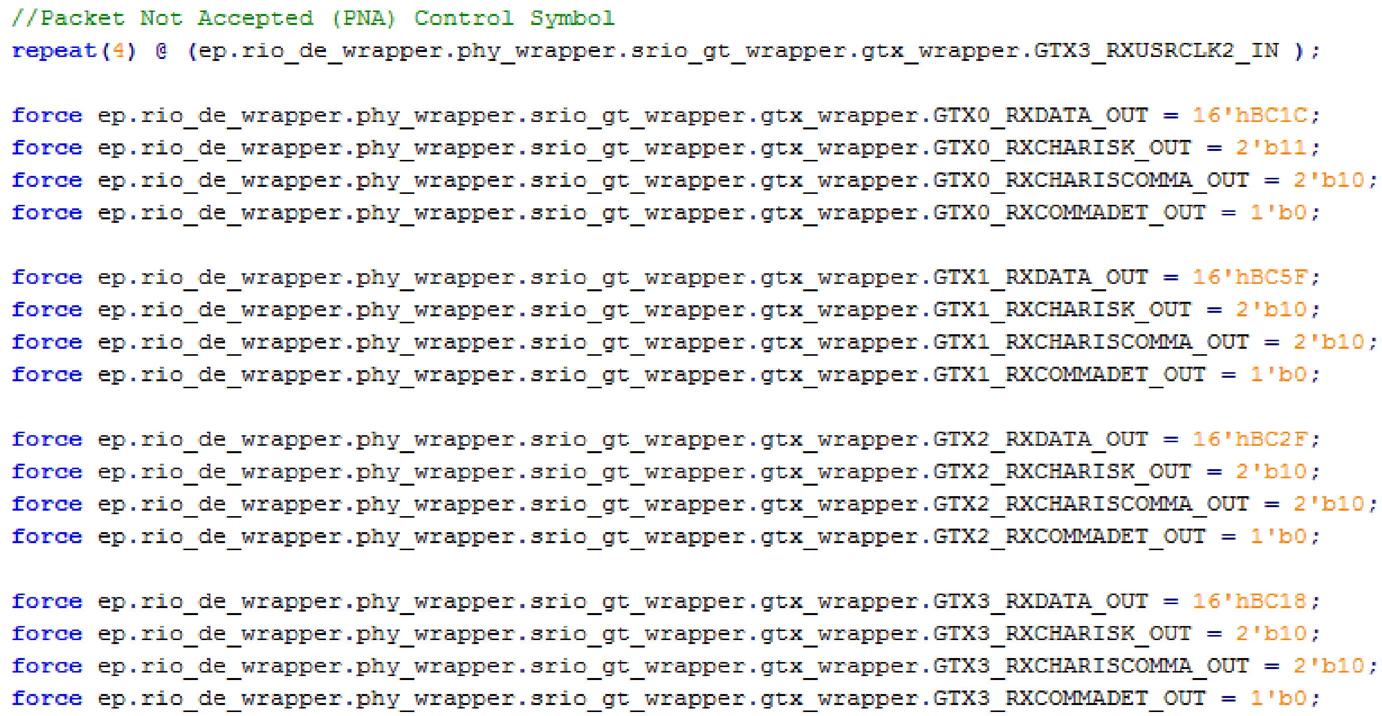
Back to Back Packet Retry (PR) and Packet Not Accepted Simulation

The specific example here is a hypothetical case to reproduce a scenario where Packet Retry (PR) and Packet Not Accepted (PNA) control symbols are received at the same time. For comprehensive Serial Rapid IO packet analysis and to understand how to decode packets, refer Xilinx Answer 50166 [3].

Generate Serial Rapid IO gen1 core in Coregen and add the following code in *ep_tb.v* file.



```
initial begin

  #19934924

  $display("[%t] : Entering Custom Test - PNA/PR Back to Back Test", $realtime);

  //Packet Retry (PR) Control Symbol

  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXDATA_OUT = 16'hBC1C;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXCHARISCOMMA_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXCHARISK_OUT = 2'b11;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXCOMMADET_OUT = 1'b0;

  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXDATA_OUT = 16'hBC21;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXCHARISK_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXCHARISCOMMA_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXCOMMADET_OUT = 1'b0;

  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXDATA_OUT = 16'hBCFF;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXCHARISK_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXCHARISCOMMA_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXCOMMADET_OUT = 1'b0;

  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXDATA_OUT = 16'hBC19;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXCHARISK_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXCHARISCOMMA_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXCOMMADET_OUT = 1'b0;
```



```
  //Packet Not Accepted (PNA) Control Symbol
  repeat(4) @ (ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXUSRCLK2_IN );

  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXDATA_OUT = 16'hBC1C;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXCHARISK_OUT = 2'b11;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXCHARISCOMMA_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXCOMMADET_OUT = 1'b0;

  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXDATA_OUT = 16'hBC5F;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXCHARISK_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXCHARISCOMMA_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXCOMMADET_OUT = 1'b0;

  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXDATA_OUT = 16'hBC2F;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXCHARISK_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXCHARISCOMMA_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXCOMMADET_OUT = 1'b0;

  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXDATA_OUT = 16'hBC18;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXCHARISK_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXCHARISCOMMA_OUT = 2'b10;
  force ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXCOMMADET_OUT = 1'b0;
```

```
   repeat(4) @ (ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXUSRCLK2_IN );

   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXDATA_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXCHARISK_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXCHARISCOMMA_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX0_RXCOMMADET_OUT;

   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXDATA_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXCHARISK_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXCHARISCOMMA_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX1_RXCOMMADET_OUT;

   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXDATA_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXCHARISK_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXCHARISCOMMA_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX2_RXCOMMADET_OUT;

   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXDATA_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXCHARISK_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXCHARISCOMMA_OUT;
   release ep.rio_de_wrapper.phy_wrapper.srio_gt_wrapper.gtx_wrapper.GTX3_RXCOMMADET_OUT;

end
```

The waveform in

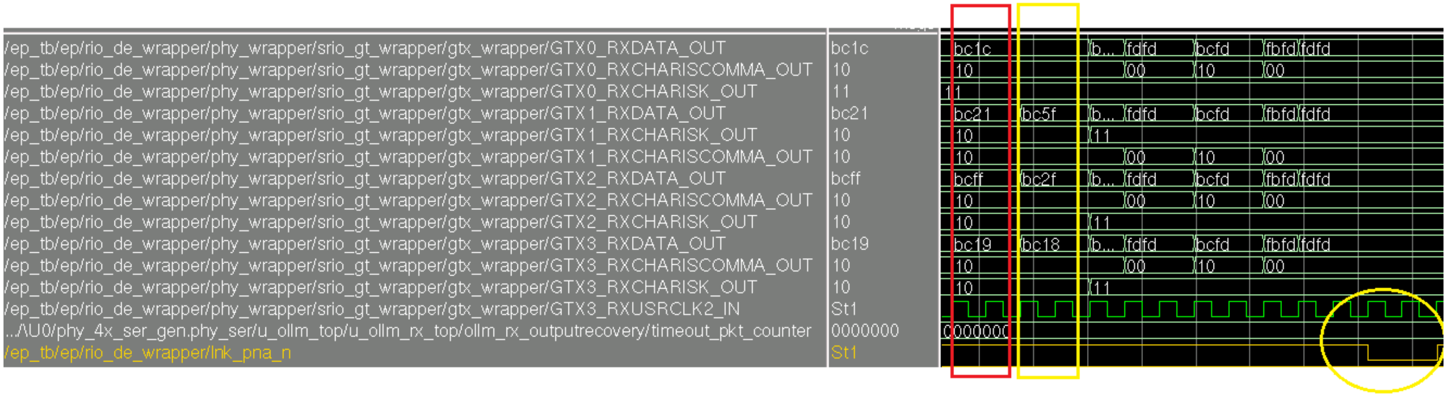
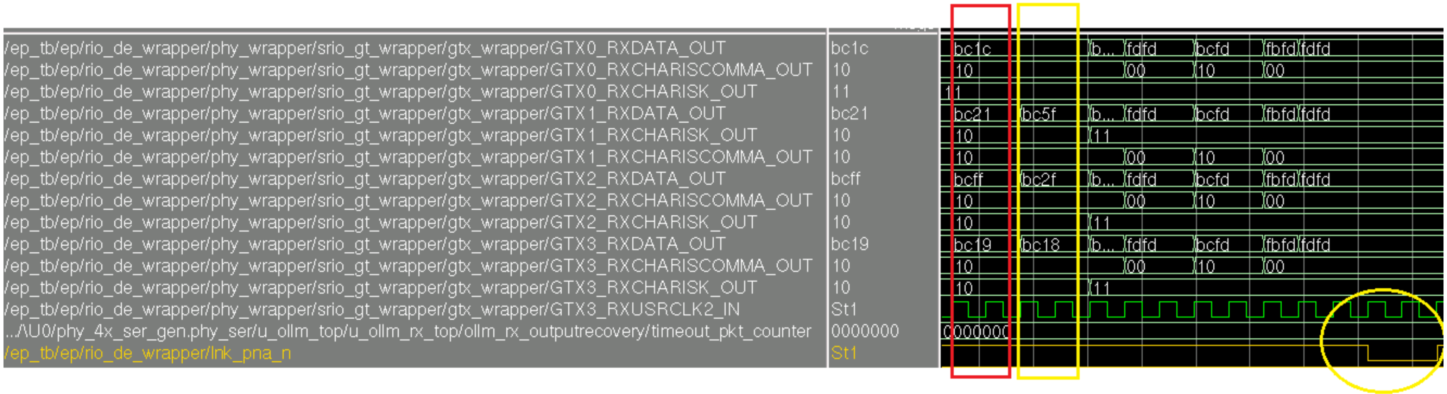


Figure 28 shows the Packet Retry (PR) control symbol (Yellow Box) followed by the Packet Not Accepted (PNA) control symbol (Red Box) on the receive side of the GT interface. As the result of receiving a PNA, the core asserts lnk_pna_n as shown in yellow circle in the waveform.



```
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX0_RXDATA_OUT            bc1c
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX0_RXCHARISCOMMA_OUT     10
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX0_RXCHARISK_OUT         11
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX1_RXDATA_OUT            bc21
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX1_RXCHARISK_OUT         10
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX1_RXCHARISCOMMA_OUT     10
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX2_RXDATA_OUT            bcff
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX2_RXCHARISCOMMA_OUT     10
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX2_RXCHARISK_OUT         10
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX3_RXDATA_OUT            bc19
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX3_RXCHARISCOMMA_OUT     10
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX3_RXCHARISK_OUT         10
/ep_tb/ep/rio_de_wrapper/phy_wrapper/srio_gt_wrapper/gtx_wrapper/GTX3_RXUSRCLK2_IN          St1
.../U0/phy_4x_ser_gen.phy_ser/u_ollm_top/u_ollm_rx_top/ollm_rx_outputrecovery/timeout_pkt_counter 0000000
/ep_tb/ep/rio_de_wrapper/lnk_pna_n                                                          St1
```

Figure 28 – Back to back PR and PNA simulation

Conclusion

This document described techniques for injecting packets in simulation of Xilinx Integrated PCI Express Block and Serial RapidIO cores using force-release statement in Verilog. In corner case scenarios, it becomes quite complex and time consuming to generate packets by writing code in test bench. In such situations, the techniques described in this document could be used to easily inject desired packets on any interface of the core or in the user's design.

References

1. PG054, 7 Series FPGAs Integrated Block for PCI Express Product Guide
2. UG503, LogiCORE IP Serial RapidIO v5.6, User Guide
3. LogiCORE IP Serial RapidIO Gen2 - Debugging and Packet Analysis Guide
4. Virtex-5 Integrated PCI Express Block Plus - Debugging Guide for Link Training Issues
5. Virtex-6 Integrated PCIe Block Wrapper - Debugging and Packet Analysis Guide
6. Virtex-5 Endpoint Block Plus for PCI Express - Debugging and Packet Analysis Guide with Downstream Port Model and PIO Example Design

Revision History

25/06/2013 – Initial Release