XAPP1110 (v1.0) April 13, 2009

BFM Simulation of an EDK System Which Uses the PLBv46 Endpoint Bridge for PCI Express

Author: Lester Sanders, Mark Sasten

Abstract

This application note demonstrates how to run a simulation of an EDK system containing the PLBv46 Endpoint Bridge for PCI Express®. The simulation consists of a PCIe® Downstream Port Model communicating over a PCIe link to an EDK system containing the PLBv46 Endpoint Bridge for PCI Express. The PLBv46 Endpoint Bridge uses the Xilinx Block Plus Endpoint core for PCI Express in the Virtex®-5 FPGA. A Bus Functional Model (BFM) drives the EDK system.

Xilinx provides a simulation environment based on a Downstream Port Model which has a Test Program Interface (TPI) for test programs. The Downstream Port Model is built using the Xilinx Core Generator tool. Pre-written programs and Verilog tasks are used to generate Transaction Layer Packets (TLPs). The setup of the simulation and steps used to run the system simulation are provided as part of this application note. Example stimuli for root complex to endpoint and endpoint to root complex transactions test the PLBv46 Endpoint Bridge in the EDK system. The results from these tests are analyzed in the waveform viewer.

Included System

The project for the system simulation is available at:

https://secure.xilinx.com/webreg/clickthrough.do?cid=119465

In `xapp1110.zip`, the project directory is `ml505_bfl_plbv46_pcie_sim`, and the sub-directory `simulation` contains most of the files used in this application note.

Introduction

The PLBv46 Endpoint Bridge is a PCIe endpoint instantiated in a Xilinx FPGA. An endpoint normally communicates with a root complex. In this system simulation, the PLBv46 Endpoint Bridge in the EDK system is connected to a test environment based on a Downstream Port Model, which emulates the functionality of a root complex for test purposes. This application note shows how to run a system simulation of an EDK system which uses the PLBv46 Endpoint Bridge.The EDK system is based on a Base System Builder creation of a system using the Xilinx ML505 Embedded Development Platform.

After describing the system, the steps to setup and run a simulation are provided. The first step is to ensure that the EDK, ISE, and Smartmodel libraries are compiled and referenced by the EDK system properly. Within the EDK project, the structure and content of the simulation directory is discussed. The commands used to generate the EDK simulation model and the Downstream Port Model model are provided. The script used to run the simulation is given.

In the simulation, stimuli is generated from both the Downstream Port Model side and the PLBv46 side. The stimuli for the Downstream Port Model is provided in the `rc2ep.v` file. The Bus Functional Model stimuli is provided in the `ep2rc.bfl` file. The Xilinx PCIe simulation environment uses a Downstream Port Model which connects to test programs using at Test Program Interface (TPI). Xilinx provides several programs which use the TPI. The test programs use Verilog tasks to setup the simulation, configure and scan the Configuration Space Header, and generate memory and completion TLPs. The steps to use the Xilinx PCIe simulation environment and to write and use custom tests are provided.

Software Requirements

The software requirements for simulating this system are:

- Xilinx Platform Studio 10.1.03
- Xilinx Integrated Software Environment (ISE®) 10.1.03
- ModelSim 6.3c (Support of both VHDL and Verilog is required)
- Bus Functional Models (BFM) for IBM CoreConnect

The Bus Functional Models for PLBv46 are available from

http://www.xilinx.com/coreconnect

System Specifics

The PCIe Downstream Port Model and the EDK system are used in this simulation. The Downstream Port Model (DPM) is a set of Verilog files written using Coregen when the PCIe core is generated. Since the Verilog files are provided in the `simulation/dsport` directory, running Coregen is not necessary for this simulation. Since the EDK system is developed using EDK's Base System Builder, the Processor IP cores are defined in the `system.mhs` file.

Figure 1 shows a functional diagram of the system simulated.

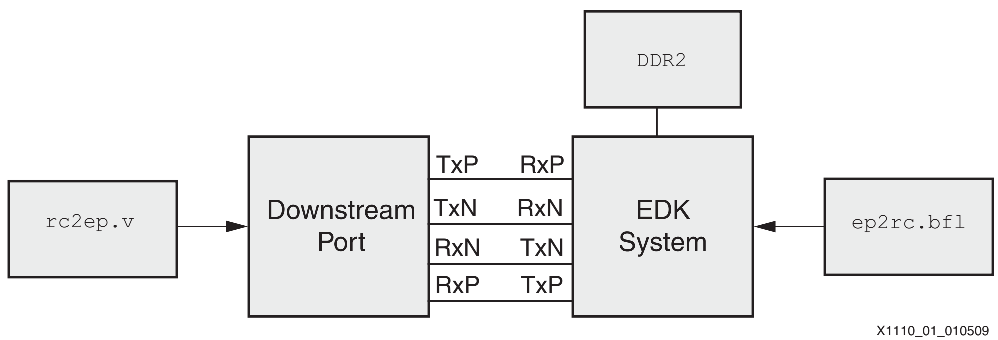


Figure 1: **System Simulation**

Since the PLBv46 transactions are done using the IBM Bus Functional Models and BFM cores, the MicroBlaze microprocessor is unused (commented) in the `system.mhs` file. To implement this system, uncomment the MicroBlaze processor and comment the three BFM cores in the `system.mhs` file.

Figure 2 is the block diagram of the EDK system. The EDK system includes the MPMC, XPS BRAM, XPS INTC, Clock Generator, XPS UART Lite, XPS Central DMA, and PLBv46 Endpoint Bridge cores. The EDK system also includes the PLBv46_Monitor_BFM, PLBv46_Master_BFM, and BFM_synch cores. The BFM cores are used to generate stimuli using IBM CoreConnect Bus Functional Language (BFL) commands, monitor the PLBv46, and synchronize transactions generated by the Downstream Port Model and the Bus Functional Language. In this system, the three BFM components replace the Microblaze and/or IBM PPC405 or IBM PPC440 hard microprocessors. An EDK simulation using a microprocessor executing C commands instead of BFMs is given in XAPP1111 application note. In both cases, the user is able to write and read registers and memory of the processor IP cores in the EDK system.

.

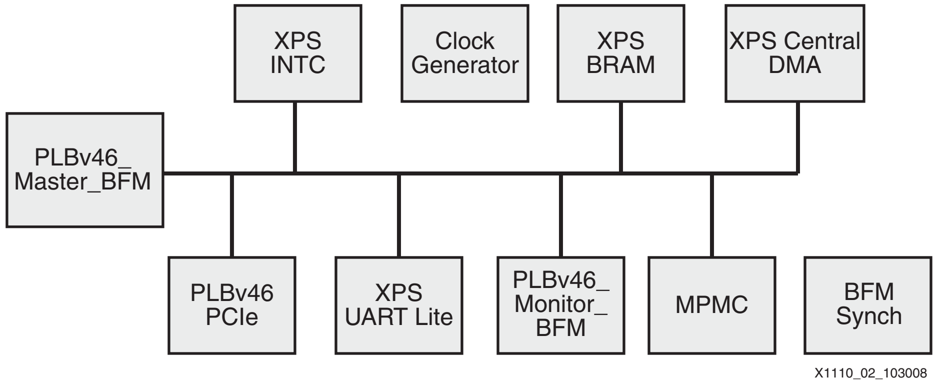


Figure 2: **EDK System**

Table 1 provides the address map of the system.

Table 1: **EDK System Address Map**

Peripheral	Instance	Base Address	High Address
XPS INTC	xps_intc_0	0x81800000	0x8180FFFF
Clock Generator	clock_generator_0	N/A	N/A
XPS BRAM CNTLR	xps_bram_if_cntlr_1	0x8AE10000	0x8AE1FFFF
XPS Central DMA	xps_central_dma_0	0x80200000	0x8020FFFF
PLBv46 Endpoint Bridge	PCIe_Bridge	0x85C00000	0x85C0FFFF
XPS Uartlite	RS232_Uart_1	0x84000000	0x8400FFFF
MPMC	DDR2_SDRAM	0x90000000	0x9FFFFFFF
PLBv46_Monitor_BFM	plbv46_monitor_bfm_0	N/A	N/A
PLBv46_Master_BFM	plbv46_master_bfm_0	N/A	N/A
BFM_Synch	bfm_synch_0	N/A	N/A

In EDK, double click on the PCIe_Bridge in the System Assembly View to invoke the PLBv46 _PCIe generics editor. The generics are used to configure the PLBv46 Endpoint Bridge. The Xilinx Device ID = `0x0505` and Vendor ID = `0x10EE` are displayed in many of the PCIe tests done in this application note.

Compiling Simulation Libraries

This section illustrates how to compile EDK, ISE, and Smartmodel simulation libraries.

Figure 3 shows the first step in compiling simulation libraries. Invoke EDK, and enter **File -> Open** to open an EDK project. Select **Simulation -> Compile Simulation Libraries**.

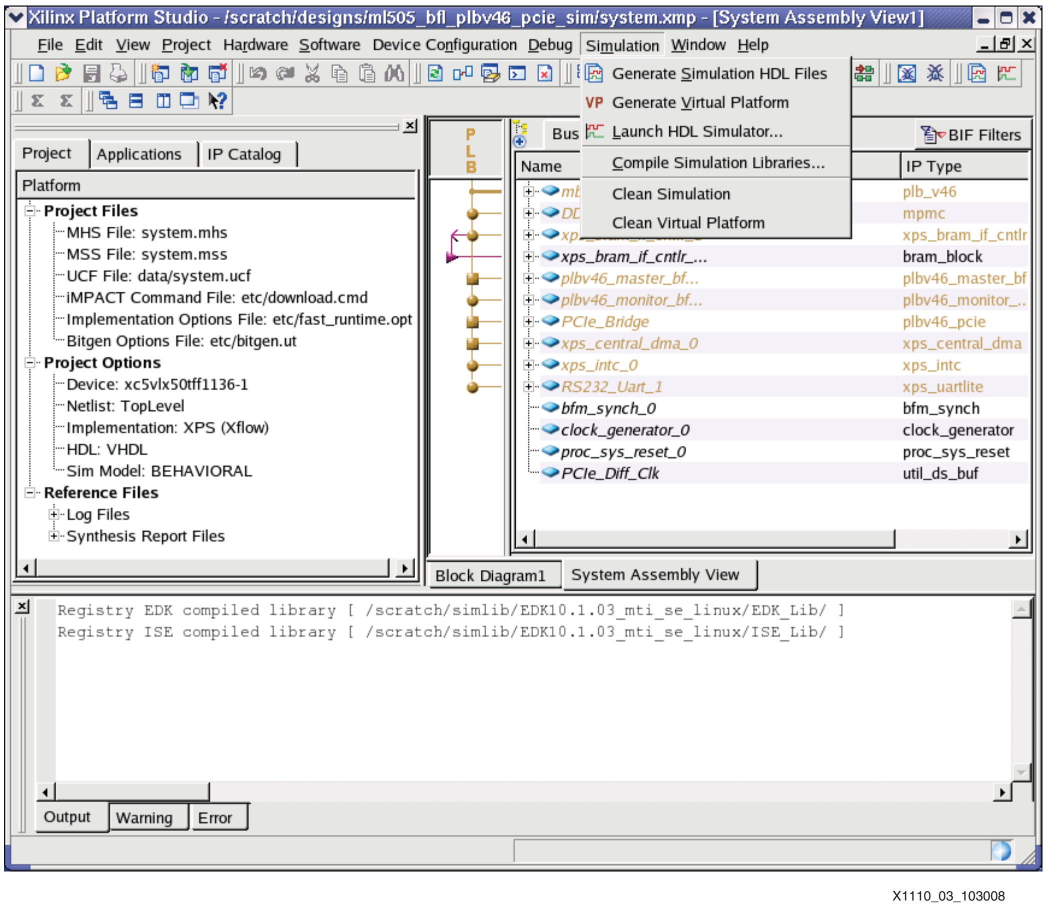


Figure 3: **Selecting Compiling Simulation Libraries**

Figure 4 shows the use of the Simulation Library Compilation wizard in compiling simulation libraries. Click **Next**.

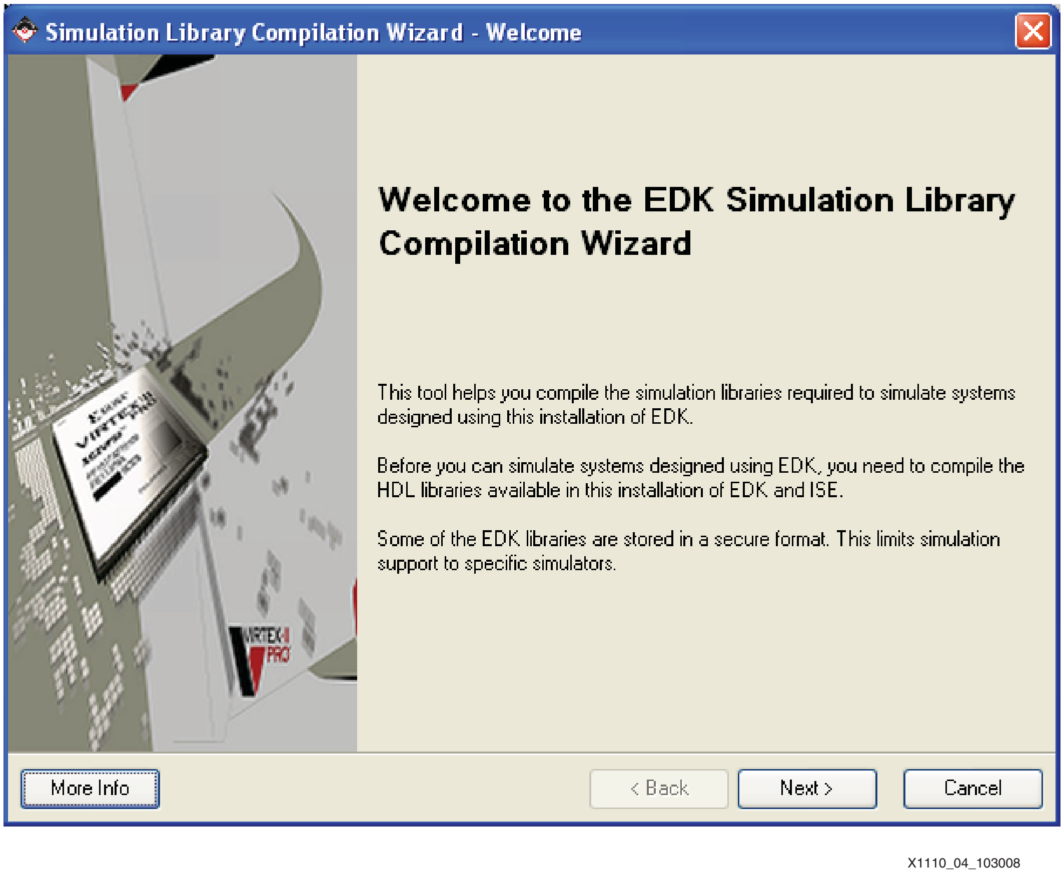
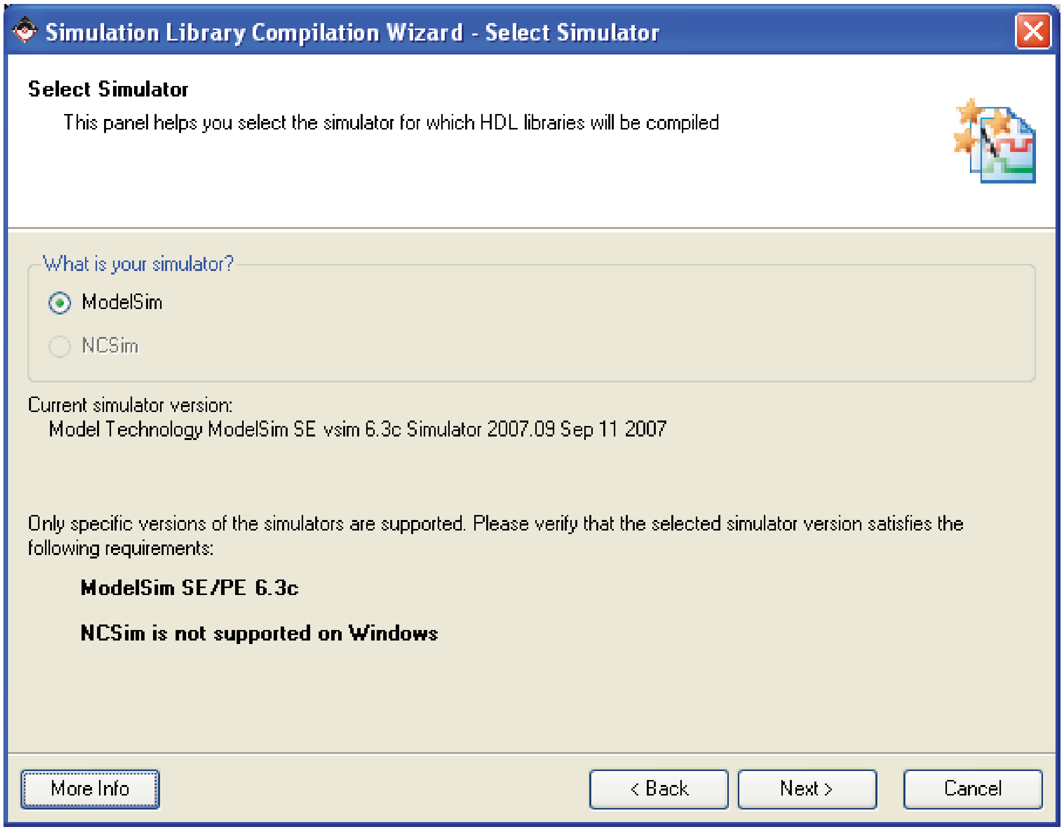


Figure 4: **Using the EDK Simulation Library Compilation Wizard**

Figure 5 shows the selection of ModelSim v6.3c in compiling simulation libraries. The Wizard displays the version of the ModelSim simulator that is detected. If this is grayed out, then the simulator is not properly set up. Click **Next**.



X1110_05_103008

Figure 5: **Selecting the ModelSim Simulator**

Figure 6 shows the selection of both the VHDL and Verilog. Click **Next**.

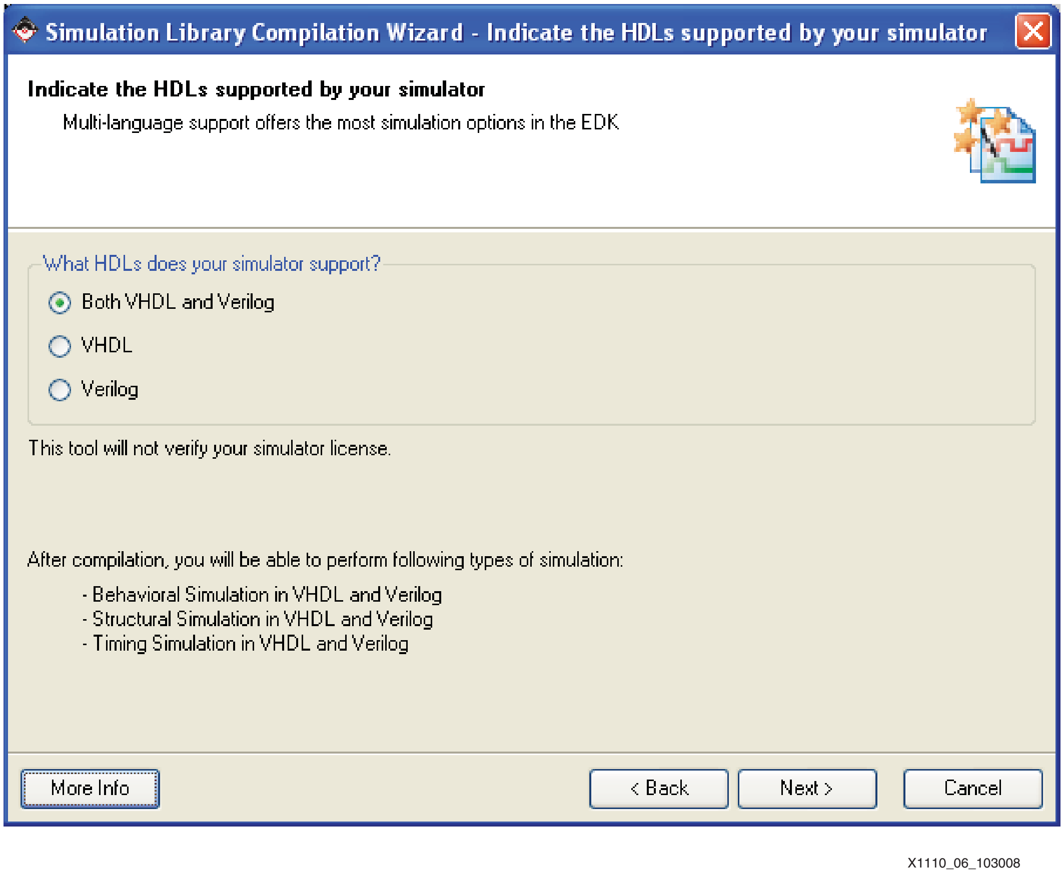


Figure 6: **Selecting both Verilog and VHDL for Mixed Simulation**

Figure 7 shows the specification of the directory for ISE library and Smartmodels. In general, the location of the path below `/scratch/simlib` will vary for each user. Click **Next**.

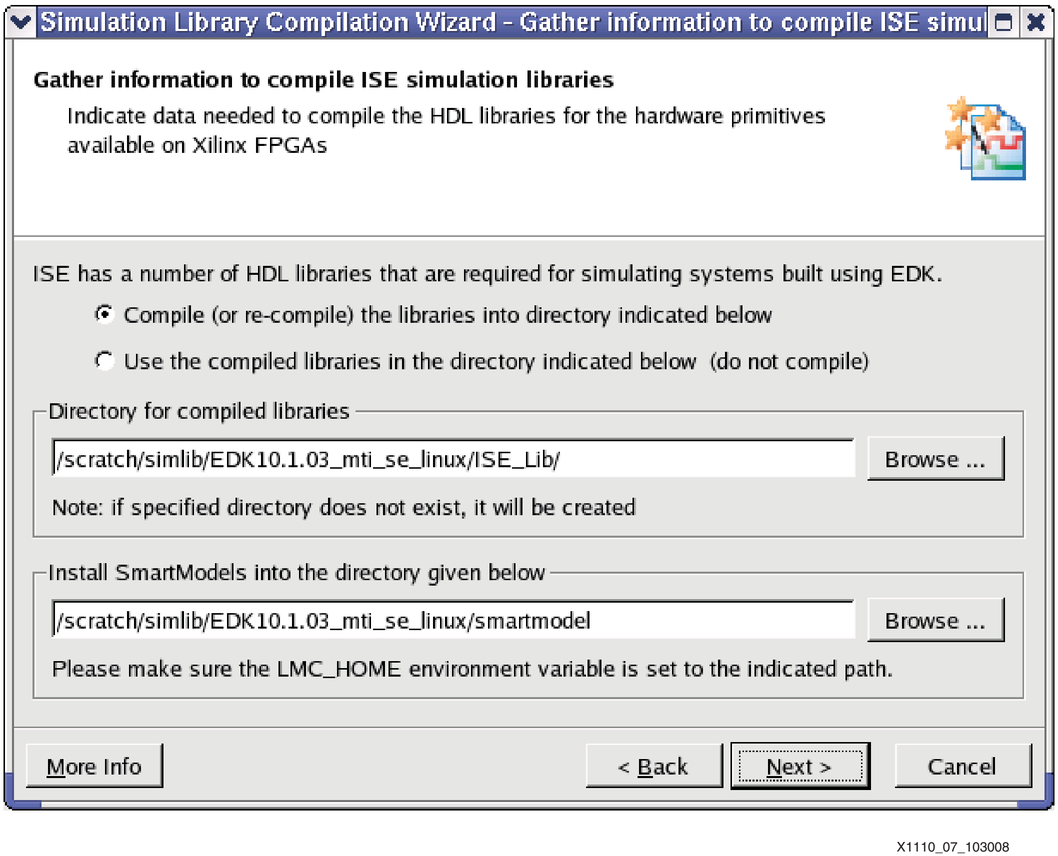
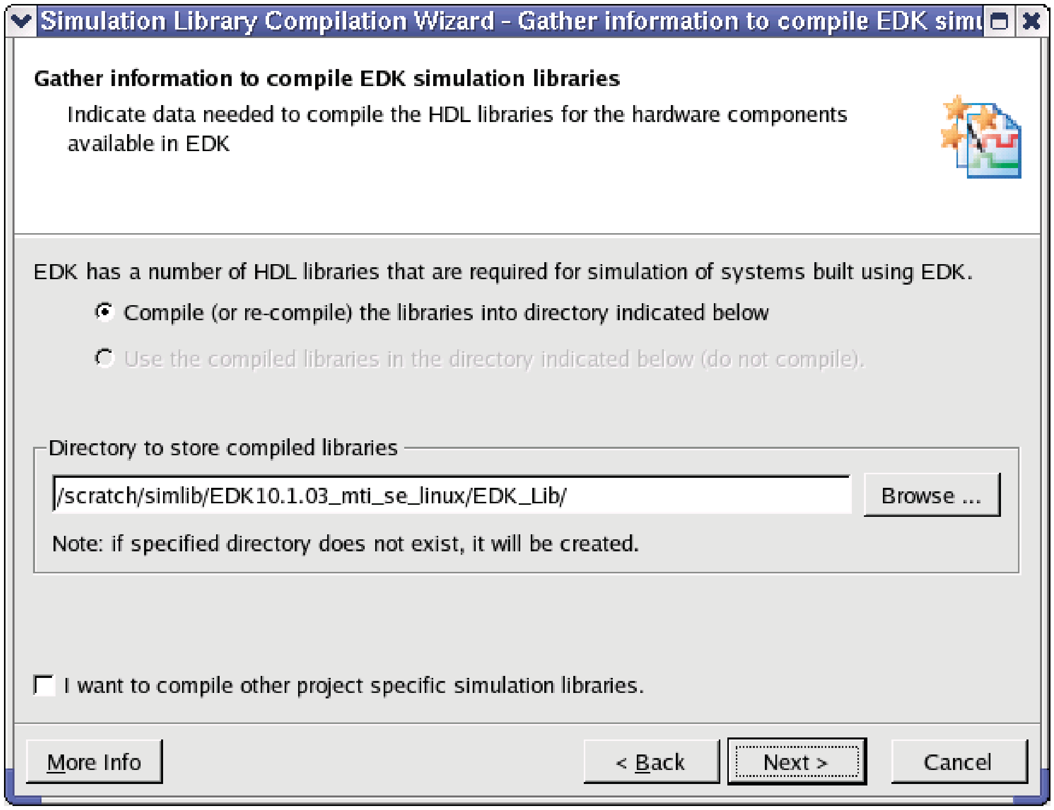


Figure 7: **Defining the Install Directories of ISE_Lib, Smartmodels**

Figure 8 shows the specification of the directory for the EDK libraries. Click **Next**.



X1110_08_103008

Figure 8: **Defining the Install Directory of EDK_Lib**

Set the environment variables below as appropriate for the simulation environment.

XILINX=/build/aqxfndry2/K.39.0/rtf; export XILINX

XILINX_EDK=/proj/abq_ip/edk_builds/EDK_K_SP3.6.1/rtf; export XILINX_EDK

MTI_LIBS=/scratch/simlib/EDK10.1.03_mti_se_linux; export MTI_LIBS

LMC_HOME=$MTI_LIBS/smartmodel; export LMC_HOME

LMC_CONFIG=$LMC_HOME/data/x86_linux.lib; export LMC_CONFIG

LMC_PATH=$LMC_HOME/foundry:$LMC_HOME/models; export LMC_PATH

LD_LIBRARY_PATH=$LMC_HOME/lib/linux.lib:$LD_LIBRARY_PATH

If Windows is used, make sure that %LMC_HOME%\lib\pcnt is in the Path under System Variables.

If the Linux operating system is used, set the following lines in the `modelsim.ini` file.

veriuser = $LMC_HOME/lib/linux.lib/swiftpli_mti.so

libsm = $MODEL_TECH/libsm.sl

libswift = $LMC_HOME/lib/linux.lib/libswift.so

If the Windows operating system is used, set the following lines in `modelsim.ini`.

veriuser = %LMC_HOME%/lib/pcnt.lib/swiftpli_mti.dll

libsm = %MODEL_TECH%/libsm.dll

libswift = %LMC_HOME%/lib/pcnt.lib/libswift.dll

Set the simulator resolution to ps and use a semicolon to comment the PathSeparator = / line.

To verify that SmartModels are set up correctly, enter the following command in the ModelSim command window:

`VSIM> vsim unisim.ppc405`

If there are no errors when loading, the SmartModels are set up correctly.

Simulation Directory Structure

The `xapp1110.zip` project directory, `ml505_bfl_plbv46_pcie_sim`, contains the `simulation` directory at the top level of the ml505_bfl_plbv46_pcie_sim project. The `simulation` directory contains the `scripts`, `ddr2`, `dsport`, and `testbench` sub-directories. When **`simgen`** is run, the `behavioral` sub-directory is created under the `simulation` directory.The Simgen tool can be run using the Generate HDL Simulation Files

Figure 9 shows the directory structure of the BFM simulation environment.

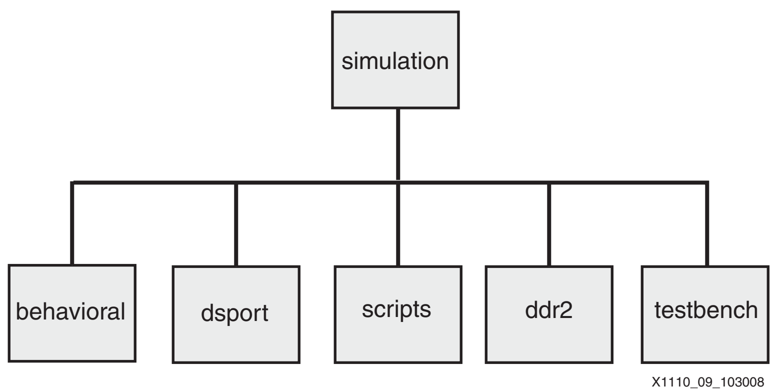


Figure 9: **Simulation Directory Structure**

The behavioral directory contains the wrappers which use core models in EDK_Lib to provide a model of the EDK system.

The `dsport` directory contains the verilog files which model the Downstream Port. The files are

- `board.v`
- `dsp_cfg.v` - Downstream Port PCIe configuration file
- `pcie_exp_1_lane_64b_dsport.v` - Downstream Port simulation model
- `pci_exp_usrapp_cfg.v`
- `pci_exp_usrapp_com.v` - Downstream Port Configuration Interface Controller
- `pci_exp_usrapp_rx.v` - Supports Endpoint to Downstream Port Model TLPs
- `pci_exp_usrapp_tx.v` - Verilog tasks for generating Downstream Port Model to Endpoint TLPs

- `xilinx_pci_exp_defines.v - Various parameter definitions`
- `xilinx_pci_exp_downstream_port.v` - top level Downstream Port Model file
- `xilinx_pci_exp_dsport.v - instantiates Downstream Port`

The `testbench` directory contains the `testbench.v` file. The `testbench.v` file instantiates the Downstream Port Model and the EDK system, which is the device under test (DUT). The `testbench.v` also defines the clocks and resets for both downstream and EDK systems. The `tests.v` file contains the names of the test program which drives the Downstream Port Model. It contains `rc2ep.v`, or a user-developed test program.

The `scripts` directory contains the BFL script used to generate PLBv46 transactions. The `scripts` directory also contains the files and commands used to compile the Downstream Port Model and testbench.

The `run.do` file compiles the `pcie_x1.f` file. The `pcie_x1.f` file is the list of files used in the Downstream Port Model. The `run.do` also invokes the Bus Functional Model Compiler to compile the commands in `ep2rc.bfl` into ModeSim do commands. The `run.do` runs the simulation for the time specified in the last line of the fun.do file.

Generating Simulation Models with Simgen

Before running **simgen**, edit the `simgen.opt` -X argument to specify the location of the ISE Lib compiled in the section "Compiling Simulation Libraries", and the -E argument to specify the location of the EDK_Lib. The `simgen.opt` file is located in the `ml505_bfl_plbv46_pcie_sim` directory.

Simgen can be run from EDK or a command prompt. Figure 10 shows how to run **simgen** from EDK. Select **Simulation -> Generate Simulation HDL Files.**

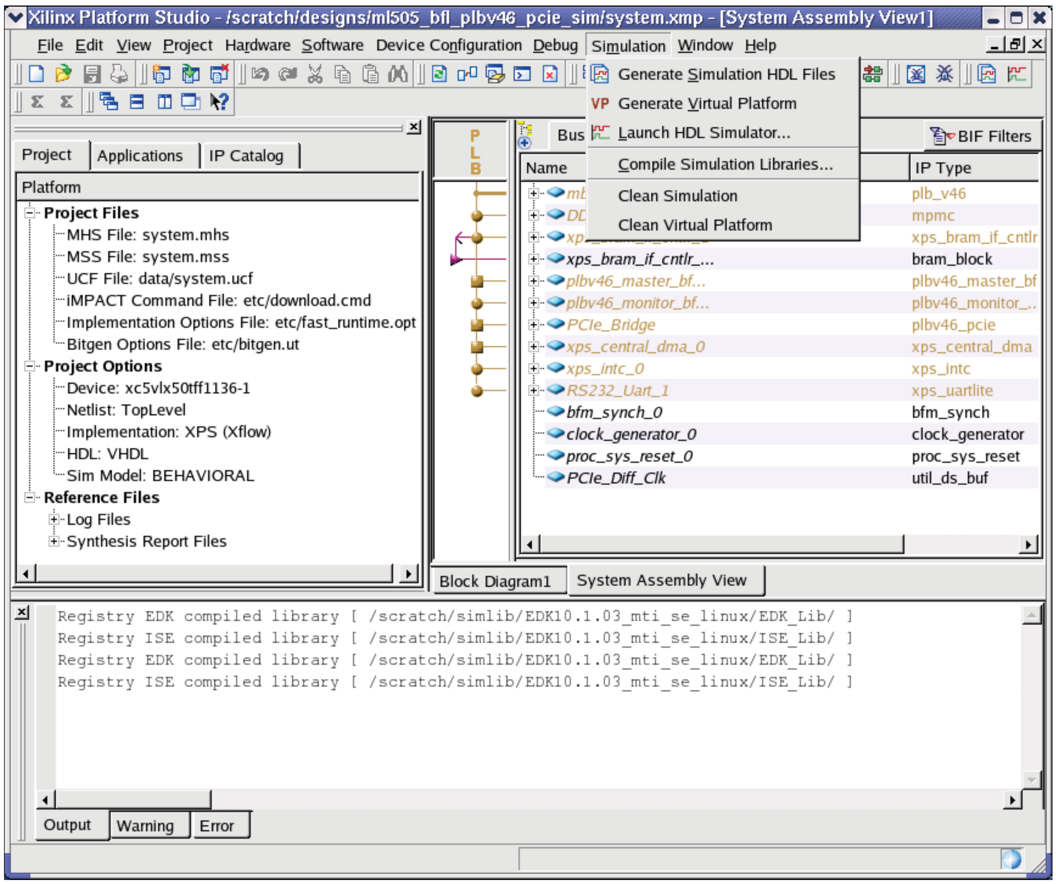


Figure 10: **Running Generate Simulation HDL Files from EDK**

To run **simgen** from the command line, edit or create a `simgen.opt` as shown in Figure 11.

```
system.mhs
-p
xc5vlx50tff1136-2
-lang
verilog
-mixed
yes
-s
mti
-X
/scratch/simlib/EDK10.1.03_mti_se_linux/ISE_Lib
-E
/scratch/simlib/EDK10.1.03_mti_se_linux/EDK_Lib
-m
beh
```

X1110_11_103008

Figure 11: **simgen.opt**

Run

simgen -f simgen.opt

If running simgen from the EDK GUI, modify library path values using

Edit -> Preferences -> Application Preferences -> Simulation Libraries Path

Running a Simulation

Do the following steps to run a simulation. From the project's root directory

cd simulation/behavioral

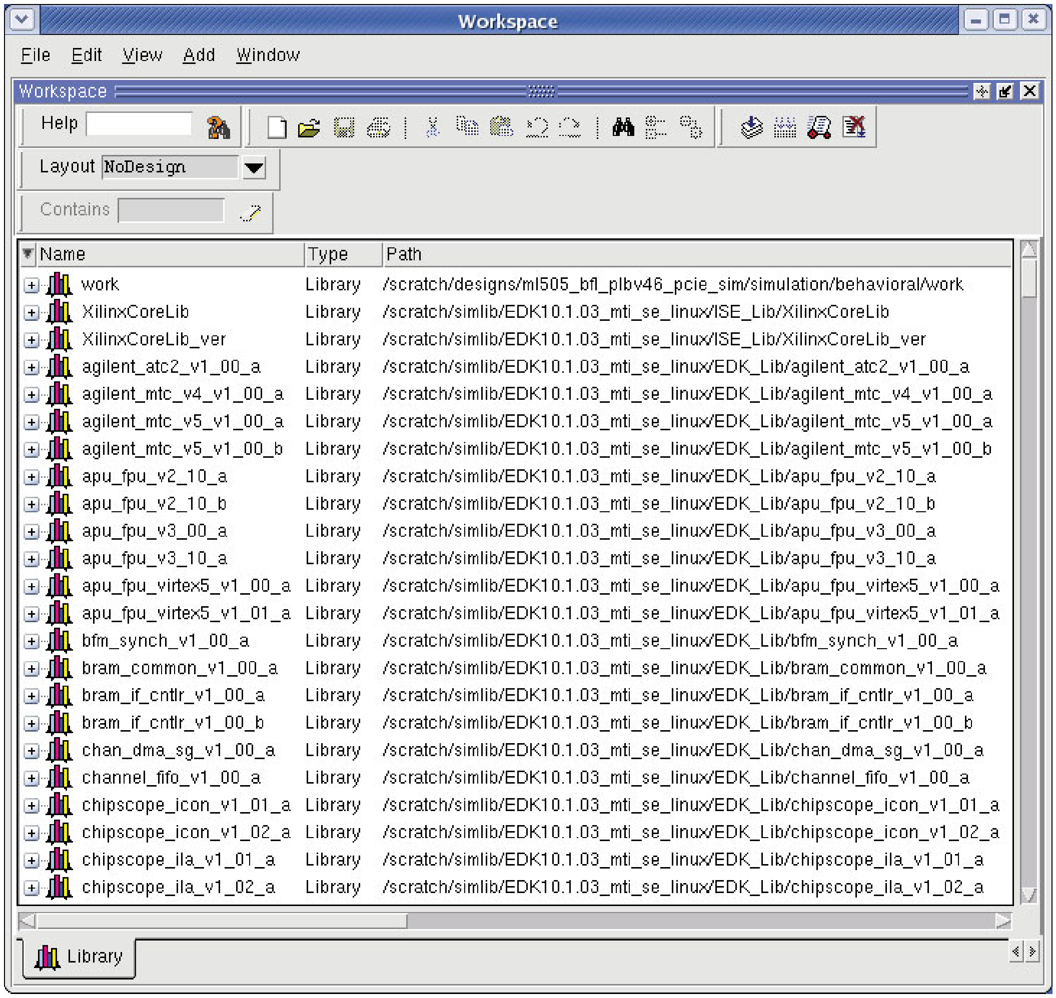
Invoke ModelSim using

vsim &

For added functionality, invoke ModelSim using the command

vsim -do system_setup.do

In ModelSim, select **View -> Workspace**. As shown in Figure 12, verify that the EDK and ISE libraries are correct and available. If the correct libraries are not displayed, copy the /scratch/simlib/EDK10.1_03_mti_se_linux/EDK_Lib/`modelsim.ini` to the project's simulation/behavioral directory and re-verify that the libraries are correct.



X1110_12_103008

Figure 12: **Verifying Libraries Using View -> Workspace**

Figure 13 shows the compilation the EDK system. Run

do system.do

in the transcript window.

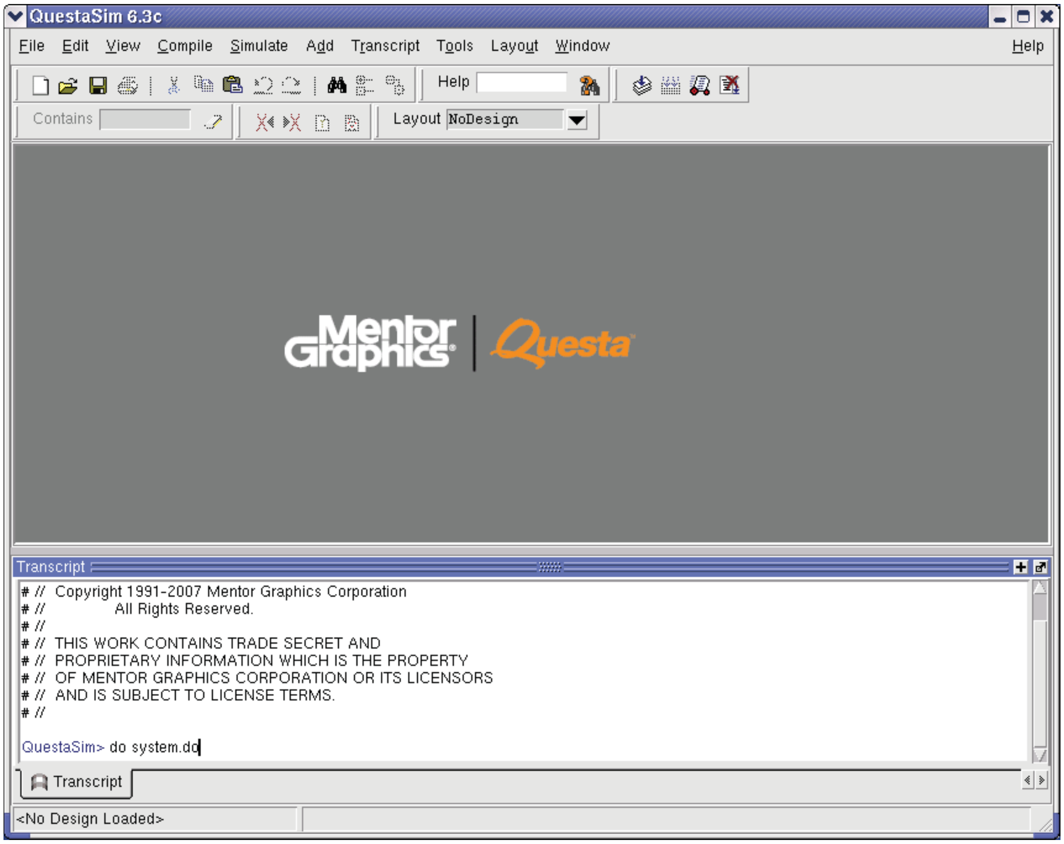
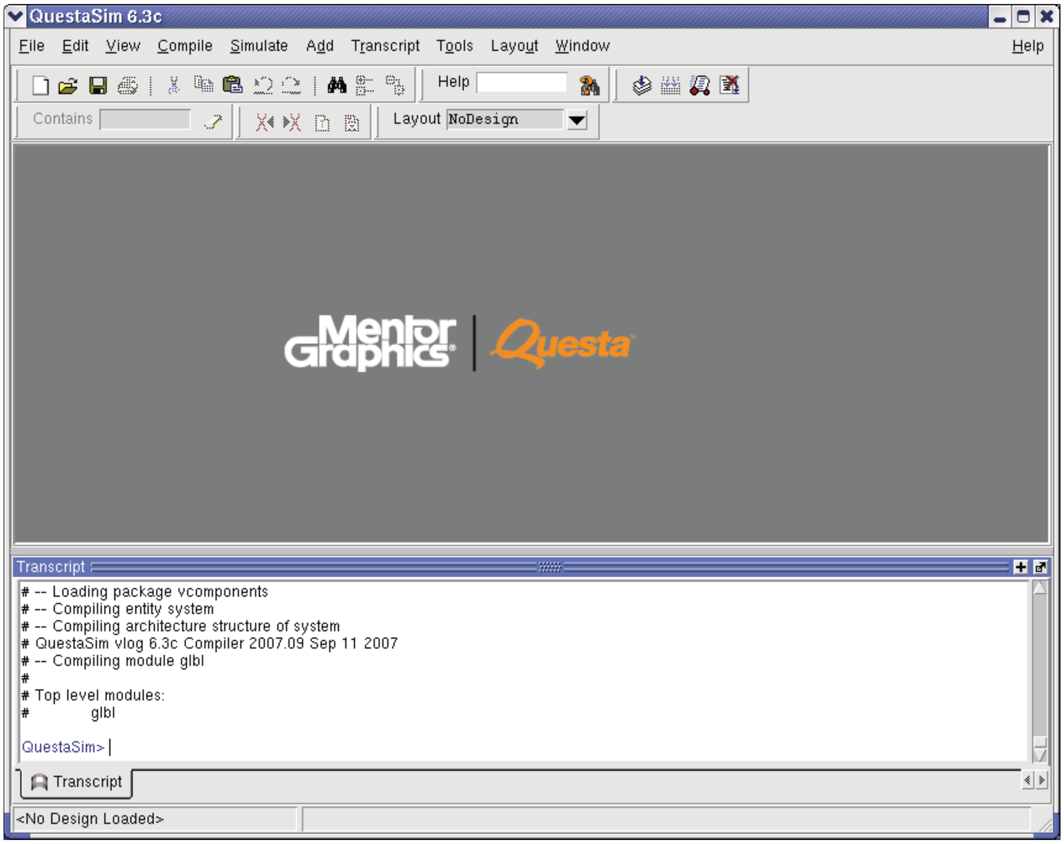


Figure 13: **Compiling the EDK System**

Figure 14 shows the expected results from compiling the EDK system. The Processor IP wrappers in the `behavioral` directory and the `system.v` file are compiled.



X1110_14_103008

Figure 14: **Compiled EDK System**

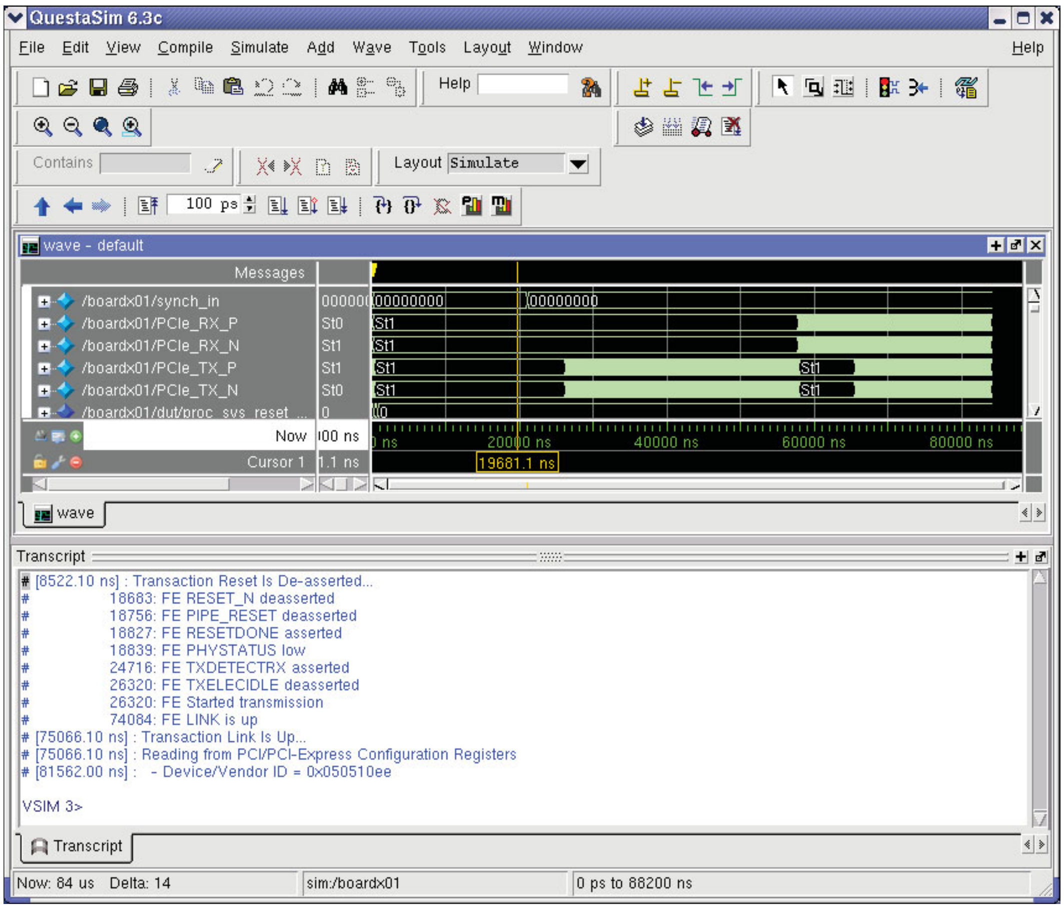
In the ModelSim transcript window run

do ../scripts/run.do

If there is a compilation error, verify that the **`-L XilinxCoreLib_ver`** argument is included in the vsim command in `run.do` If the compilation error persists and the NT operating system is used, add the -novopt argument to the **`vsim`** command.

Figure 15 shows the transcript window which displays the PLL lock and link training of the PCIe cores in the Downstream Port Model and the PLBv46 Endpoint Bridge. The resets are done at 5 us. FE is an abbreviation for Far End, referring to the Downstream Port Model. Link training starts at 26 us and is complete at 80 us.

If the simulation transcript window does not indicate that the PCIe link is trained, and the Downstream Port Model does not generate active signals on TxP, TxN, verify that the Smartmodels are set up correctly. While the Endpoint for PCI Express core and GTPs in the Virtex-5 FPGA use SecureIP models, the GT11 transceiver models in the Downstream Port Model require Smartmodels when using ISE 10.1.03 software.



X1110_15_103008

Figure 15: **Simulation Startup**

Figure 16 shows that the memory controller calibrated at 71135 ns.

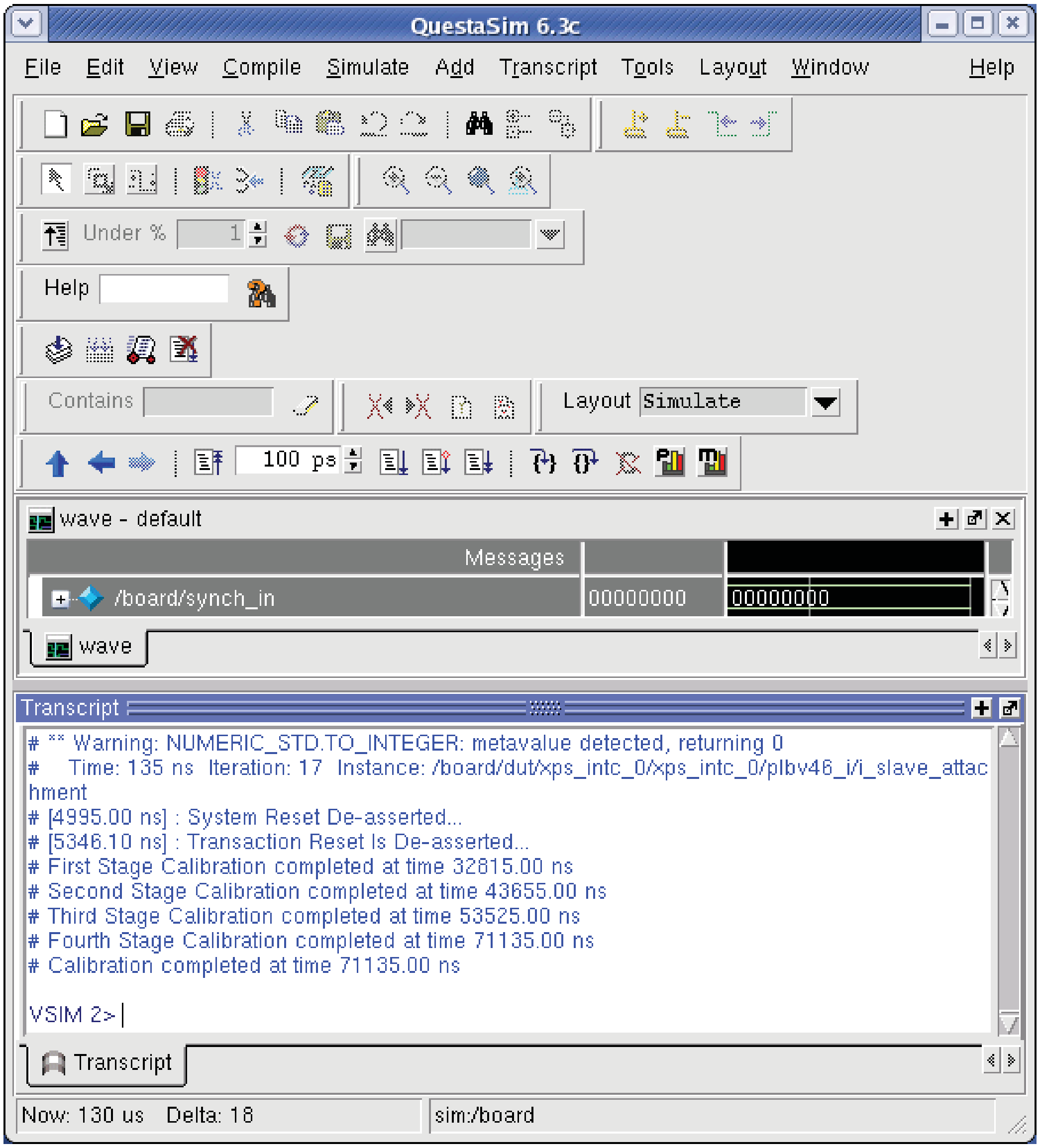
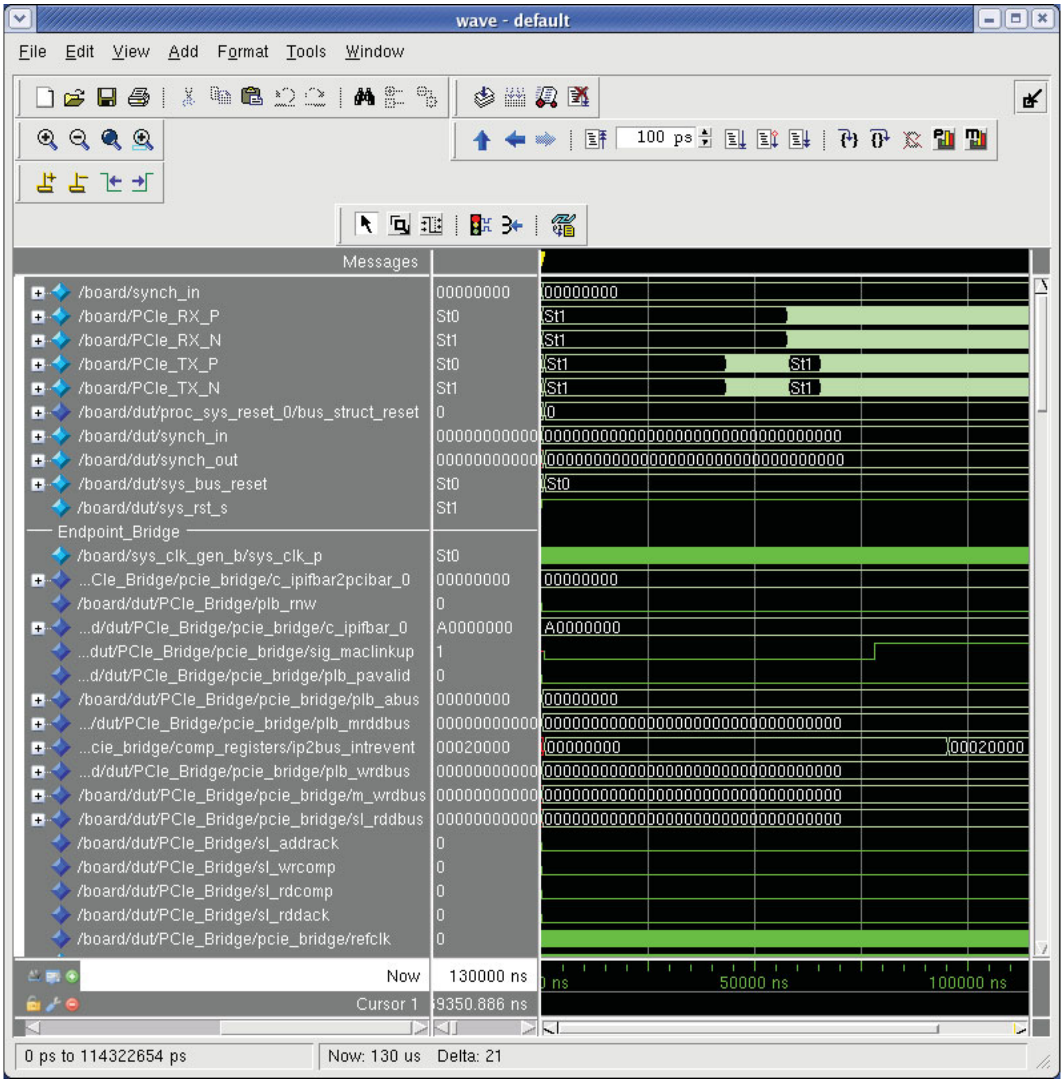


Figure 16: **Memory Controller Calibration**

Figure 17 shows the simulation startup in the ModelSim waveform viewer. The trn_reset_n signal is inactive at 5 us.The PCIe_Bridge (DUT) generates Training Sequence (TS) packets at 23 us. The pcie_master (Downstream Port Model) generates TS packets at 57 us. The trn_lnk_up_n is active at 80 us.

The waveform viewer contains dividers for the PCIe Bridge, Downstream Port Model, BRAM, DDR2, and Central DMA signals. In the waveform figures in this application note, only a limited number of signals are displayed. Additional signals need to be viewed to understand most simulations. This requires running the simulation and to scrolling through signals for each active IP in the waveform viewer.



X1110_17_103008

Figure 17: **Waveform of Simulation Startup**

Stimuli from the PCIe side

In this application note, Downstream Port Model to endpoint transactions are done using the `testbench/rc2ep.v` file.

The Xilinx simulation environment for PCI Express includes the Downstream Port Model tests listed in Table 2. These tests are provided in two files, `sample_smoke_tests.v` and `pio_tests.v`, in the `simulation/testbench` directory. These tests are listed for reference and are not discussed in this application note.

Table 2: **Downstream Port Model Tests**

Test Name	Description
sample_smoke_test0	**Issues a PCI Type 0 Configuration Read TLP, waits for Completion TLP, and compares the value with the Device/Vendor ID expected value.**
sample_smoke_test1	**Same as sample_smoke_test0 but uses parallel tests as defined in UG341**
pio_writeReadBack	**Transmits a 1 DWORD Write TLP followed by a 1 DWORD Read TLP, waits for Completion TLP, and verifies results.**
pio_testByteEnables_test0	**Issues four sequential Write TLPs enabling a unique byte enable, and then a Read TLP to verify the results**
pio_memTestDataBus	**Runs a walking 1s address test on BRAM**
pio_memTestAddrBus	**runs a walking 1s address test on BRAM**
pio_memTestDevice	**Runs an increment/decrement test on BRAM**
pio_timeoutFailureExpected	**Sends a MWr32TLP followed by a MRd32 TLP to an invalid address and waits for a CplD TLP**
pio_tlp_test0	**Example which issues a sequence of Read and Write TLPs to the RX interface**

Xilinx provides 40 Verilog tasks for use in the test program which connects to the Downstream Port Model through the TPI. The Verilog source for many of the tasks is given in the `simulation/dsport/pci_exp_usrapp_tx.v` file. The tasks accept input and write to and read from the DPM PCIe core's Local Link interface. Table B-4 in UG341 provides the inputs/arguments for the 40 pre-written Verilog tasks.

The `rc2ep.v` file calls the Type 0 Configuration Read and Configuration Write Verilog tasks and Memory Read and Memory Write tasks. The four Verilog tasks used in `rc2ep.v` are listed in Table 3.

Table 3: **Commonly Used Verilog Tasks**

Tasks	Arguments
TSK_TX_TYPE0_CONFIGURATION_READ	tag, address, first_dw_be
TSK_TX_TYPE0_CONFIGURATION_WRITE	tag, address, data, first_dw_be
TSK_TX_MEMORY_WRITE_32	tag, tc, length, address, last_dw_be, first_dw_be
TSK_TX_MEMORY_READ_32	tag, tc, length, address, last_dw_be, first_dw_be

PCIe to PLBv46 stimuli is provided in `rc2ep.v`. To provide an overview of `rc2ep.v`, most of `rc2ep.v` is shown in Figure 18. The `rc2ep.v` file begins with the TSK_SIMULATION_TIMEOUT and TSK_SYSTEM_INITIALIZATION tasks.

The `rc2ep.v` file then reads the Configuration Space Header (CSH) of the PLBv46 Endpoint Bridge using the TSK_TX_TYPE0_CONFIGURATION_READ task. In Figure 19, the Device ID/Vendor ID is read. In the actual `rc2ep.v`, the Command/Status, Class code/Revision ID and Header/Latency/Cache registers are also read. For reading and writing the Endpoint Configuration Space Header (CSH), the argument is the offset address of the register read or written.

This is followed by a configuration write of the Command/Status register, located at offset address x04, and then the Base Address Register 0 (BAR0), located at offset address `x10` in the CSH. The TSK_TX_TYPE0_CONFIGURATION_WRITE task is used. The second argument provides the CSH offset, and the third argument provides the data written.

The PCI BAR0 is actually a 64 bit BAR, and is written by two configuration writes to CSH offsets x10 and x14. Offset x10 is the least significant word and offset x14 is the most significant word, so the PCIBAR0 starting address is `0x0000000060000000`.

With BAR0 defined, memory writes are done to `0x60000000` using the TSK_TX_MEMORY_WRITE_32 task. To verify that the memory write is correct, this is followed by a memory read using the TSK_TX_MEMORY_READ_32 task.

Unlike TSK_TX_TYPE0_CONFIGURATION_WRITE, there is no data argument in the TSK_TX_TYPE_MEMORY_WRITE_32 task. The data written in the memory write task, TSK_TX_MEMORY_WRITE_32, is written into the DATA_STORE structure in `rc2ep.v`.

The TSK_TX_COMPLETION_DATA task sends a CplD TLP in response to a Memory Read TLP sent by the PLBv46 Endpoint Bridge in the EDK system.

```
TSK_SIMULATION_TIMEOUT(5050);

TSK_SYSTEM_INITIALIZATION;

$display("[%t] : - Device/Vendor ID = 0x%08x", $realtime, P_READ_DATA);
TSK_TX_TYPE0_CONFIGURATION_READ(DEFAULT_TAG, 12'h000, 4'hf);

$display("[%t] : - Command/Status Register = 0x%08x", $realtime);
TSK_TX_TYPE0_CONFIGURATION_WRITE(DEFAULT_TAG, 12'h004, 32'hFFFFFFFF, 4'Hf);

$display("[%t] : - BAR = 0x%08x", $realtime);
TSK_TX_TYPE0_CONFIGURATION_WRITE(DEFAULT_TAG, 12'h10, 32'h60000000, 4'hf);
TSK_TX_TYPE0_CONFIGURATION_WRITE(DEFAULT_TAG, 12'h14, 32'h00000000, 4'hf);

$display("[%t] : RC to EP Single Tests", $realtime);

DATA_STORE[0] = 8'h78;
DATA_STORE[1] = 8'h56;
DATA_STORE[2] = 8'h34;
DATA_STORE[3] = 8'h12;

TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 'h60000000, 4'h0, 4'hF, 1'b0)

TSK_TX_MEMORY_READ_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 'h60000000, 4'h0, 4'hF)

TSK_WAIT_FOR_READ_DATA;

#20000
TSK_TX_COMPLETION_DATA(DEFAULT_TAG, DEFAULT_TC, 10'd1, 12'd4, 7'b0000000, 3'h0, 1'b0)
```

X1110_18_120708

Figure 18: **Display of the rc2ep.v file**

The `rc2ep.v` file can be edited to perform a variety of tests. In the remainder of this section, single and burst PCIe to PLBv46 transactions are done. Besides standard configuration writes/reads and memory writes/reads, `rc2ep.v` can be edited to cause the Endpoint Bridge in the EDK system to transmit TLPs which cause interrupts to be generated. These are referred to as abnormal transactions in the Xilinx PLBv46 Endpoint Bridge documentation. Several examples are given in "Abnormal PCIe to PLBv46 Transactions".

In most applications, the functionality of IP cores in the EDK system is controlled by the either the PowerPC or MicroBlaze microprocessor. As an alternative, `rc2ep.v` can control the functionality of IP cores in the EDK system over the PCIe link. An example is given in "Controlling EDK Functions from the PCIe Side".

The `rc2ep.v` file provided in `xapp1110.zip` cannot do all of the tests provided in this application note. Most of the tests are included in `rc2ep.v` as comments.

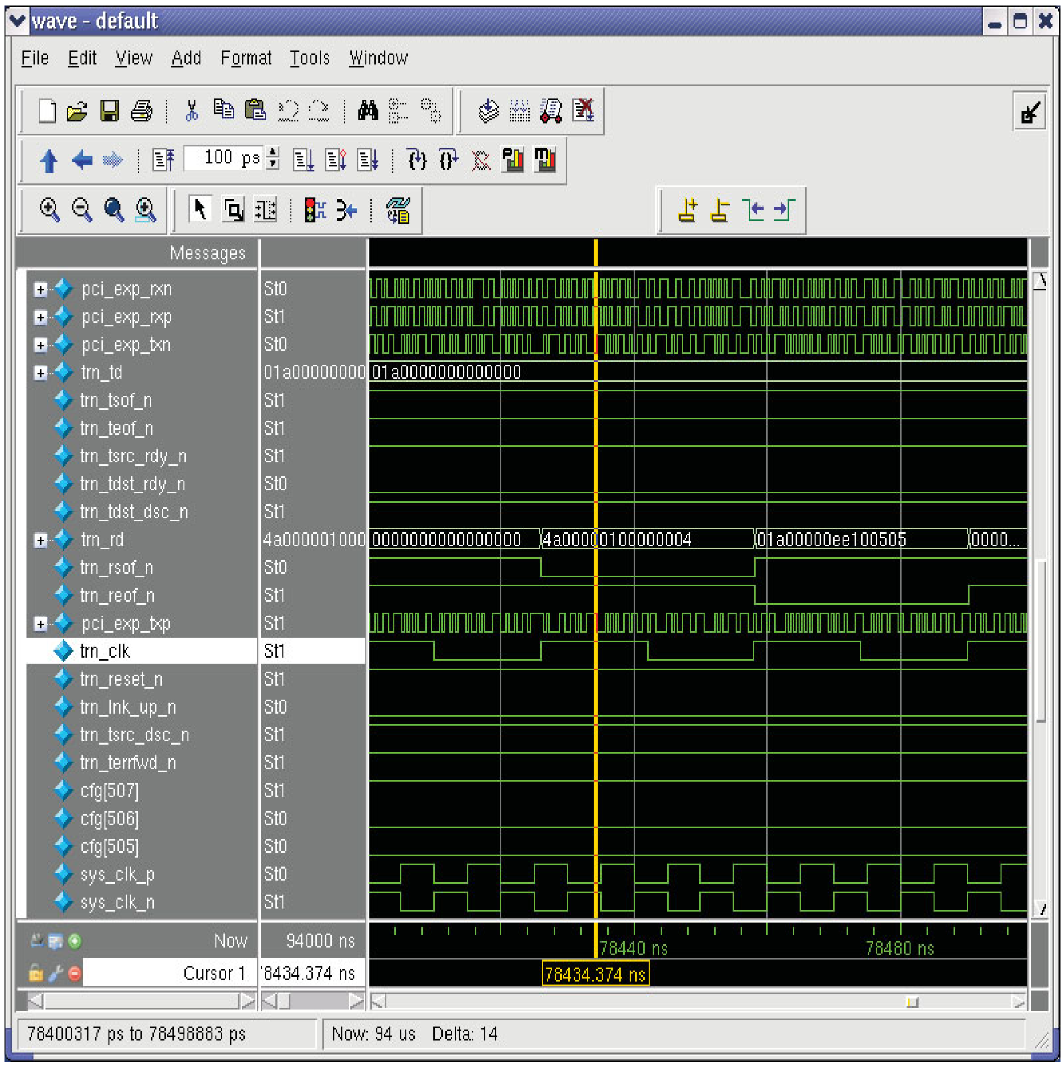
Understanding transactions in the Downstream Port Model

The DPM is a simulation model which acts as a pseudo Root Complex interface to the PLBv46 Endpoint Bridge. The DPM is not a complete system. It does not have memory. The principal interfaces in the DPM are the transmit local link interface trn_td and receive local link interface trn_rd. This section provides training needed to understand PCIe transactions in the DPM. The Xilinx Live e-Learning course *Designing a LogiCore PCI Express System* provides examples and a lab on using the PCIe local link interface.

Figure 19 shows a configuration read from the `rc2ep.v` file. The Xilinx Device ID/Vendor ID is provided at the local link trn_rd signal at 78480 ns. The following line in `rc2ep.v` is the stimuli for reading the PLBv46 Endpoint Bridge CSH.

```
TSK_TX_TYPE0_CONFIGURATION_READ(DEFAULT_TAG, 12'h000, 4'hF);
```

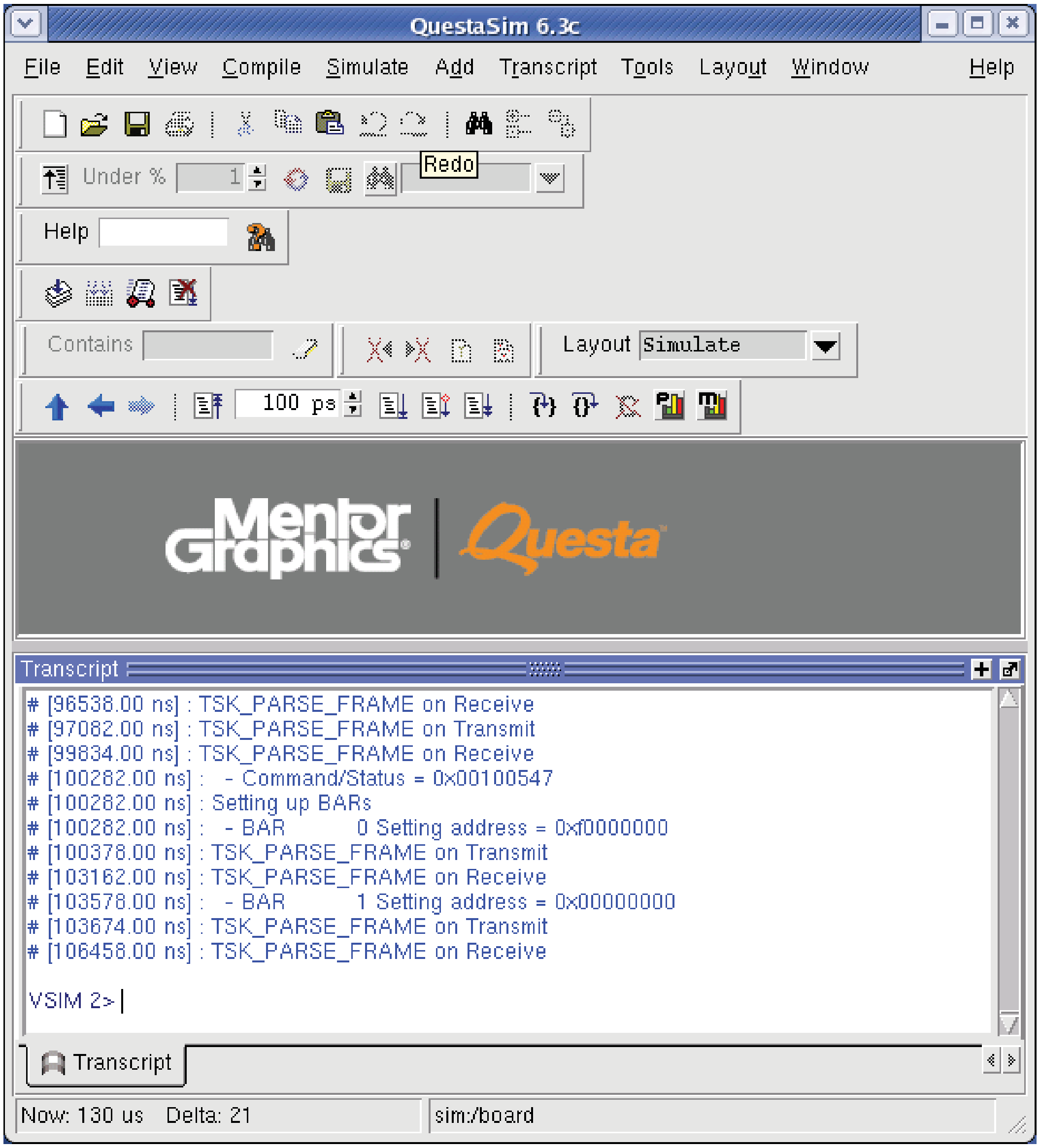
In the waveform, the trn_rd link contains the CplD TLP. The **4A** defines the TLP as a CplD packet. The returned data, `ee100505`, indicates the Device ID of the endpoint is `0505` and the Vendor ID is `10EE`. The next pages provide a description of how to read TLPs in the DPM.



X1110_19_120708

Figure 19: **Configuration Read from the Downstream Port Model**

Figure 20 shows the transcript window indicating the time of the configuration transactions in the DPM. The TSK_PARSE_FRAME task in `dsport/pci_exp_usrapp_com.v` displays in the transcript window the time a TLP is generated or received in the DPM.



X1110_20_120708

Figure 20: **Transcript of Configuration Reads**

After noting the TSK_PARSE_FRAME output in the transcript window, there are two ways of understanding the TLPs transmitted and received in the DPM. The `tx.dat` and `rx.dat` log files in the `simulation/behavioral` directory provide information on DPM transactions. The second method, illustrated briefly in Figure 19, is to analyze the trn_td and trn_rd signals in the ModelSim waveform viewer. The next figures show excerpts from the log files and show how to identify common TLP transactions transmitted and received by the DPM.

The Downstream Port Model uses a Local Link interface. Table 4 provides format information for reading data on the trn_rd and trn_td local link interfaces. The last column in this table provides the type of transaction transmitted or received by the Downstream Port Model. As an example, the trn_rd signals in Figure 19 have a value of `0x4A000001` when trn_rsof_n becomes active. From Table 4, the **4A** indicates that a Completion with Data (CPLD) TLP is received.

Table 4: **Downstream Port Model Local Link Commands**

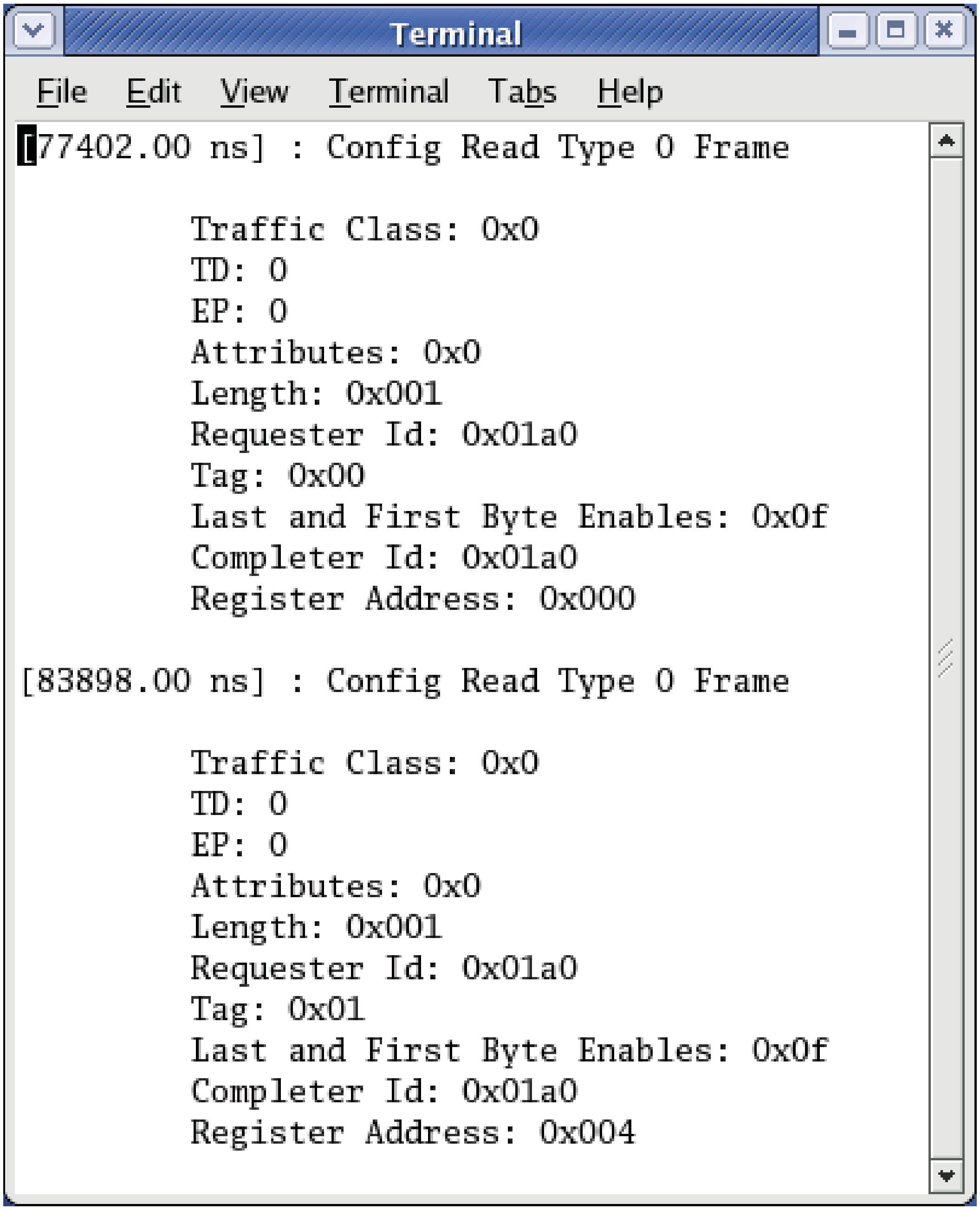
Type	Format (1:0)	Type(4:0)	Description	trn_td[63:56] trn_rd[63:56]
MRd	00 01	0 0000	Memory Read Request	00
MWr	10 11	0 0000	Memory Write Request	40
CfgRd0	00	0 0100	Type 0 Configuration Read	04
CfgWr0	01	0 0100	Type 0 Configuration Write	44
Cpl	00	0 1010	Completion without Data	0A
CplD	10	0 1010	Completion with Data	4A

Table 5 lists the Format field. This shows the 32 and 64 bit TLPs in the previous table for MRd and MWr TLPs.

Table 5: **Format Field**

FMT[10]	TLP Format
00	3 DW Header, No Data
01	4 DW Header, No Data
10	3 DW Header, With Data
11	4 DW Header, With Data

Figure 21 shows an excerpt from the DPM transmit log file, `tx.dat`, after running a simulation with `rc2ep.v`. The transmit TLPs are the write tasks defined in `testbench/rc2ep.v`. The `tx.dat` file shows a configuration read of the Device/Vendor ID and the Command/Status registers in the Configuration Space Header, at 77402 and 83898 ns. To get the result of the two reads, open `rx.dat` and look for CplD TLPs sometime following the Memory Read request times in `tx.dat`.

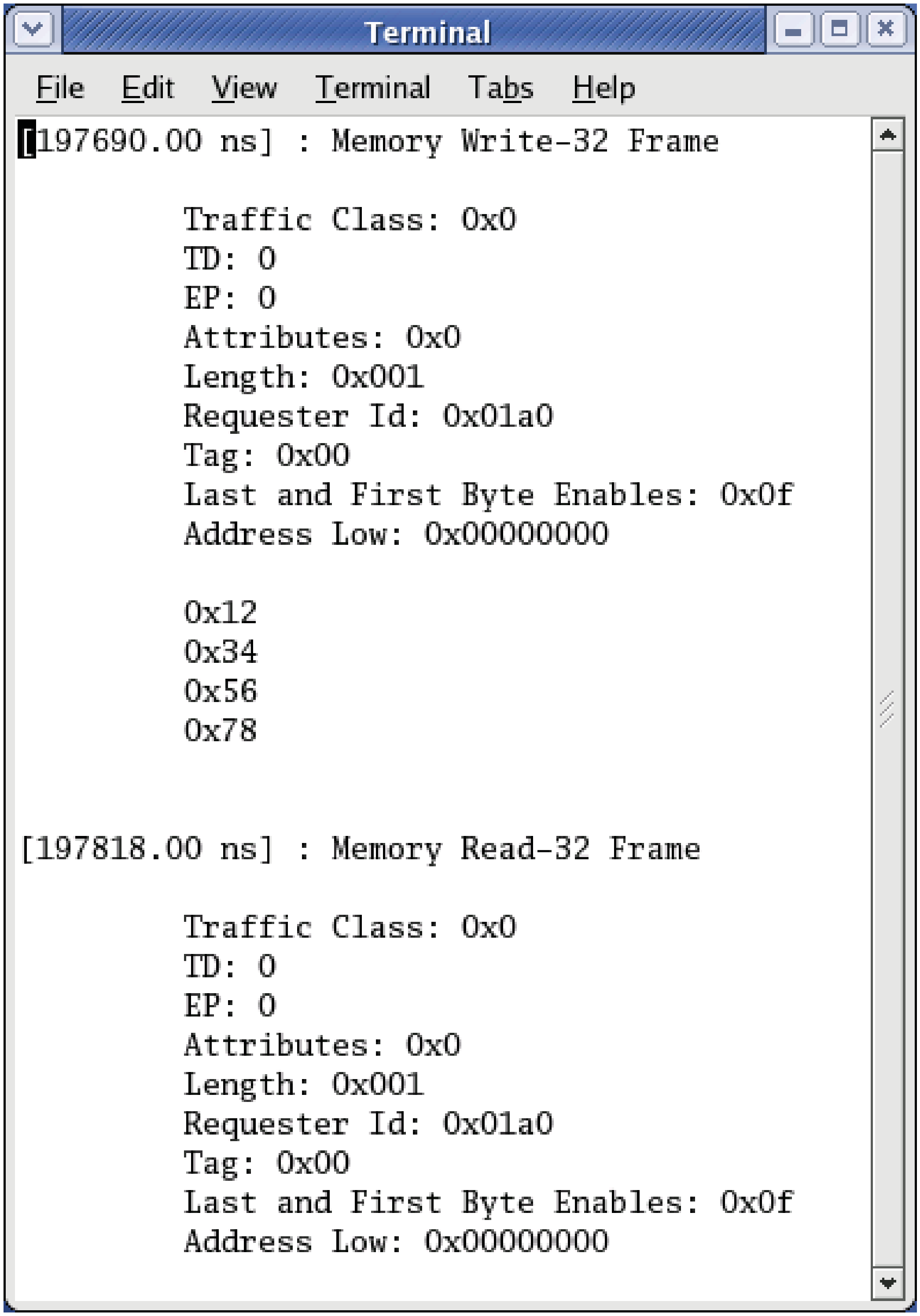



```
[77402.00 ns] : Config Read Type 0 Frame

        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

[83898.00 ns] : Config Read Type 0 Frame

        Traffic Class: 0x0
        TD: 0
        EP: 0
        Attributes: 0x0
        Length: 0x001
        Requester Id: 0x01a0
        Tag: 0x01
        Last and First Byte Enables: 0x0f
        Completer Id: 0x01a0
        Register Address: 0x004
```

X1110_21_120708

Figure 21: **Downstream Port Model tx.dat file**

Figure 22 shows excerpts of the `rx.dat` file for the DPM. The `rx.dat` log file is the result of DPM read tasks in `rc2ep.v` and/or read or write commands across the PCIe link from PLBv46 Endpoint Bridge in the EDK system, which is driven by the `ep2rc.bfl`. file. The TLPs in the figure are the result of memory write and memory read commands in `ep2rc.bfl`.




```
[197690.00 ns] : Memory Write-32 Frame

         Traffic Class: 0x0
         TD: 0
         EP: 0
         Attributes: 0x0
         Length: 0x001
         Requester Id: 0x01a0
         Tag: 0x00
         Last and First Byte Enables: 0x0f
         Address Low: 0x00000000

         0x12
         0x34
         0x56
         0x78


[197818.00 ns] : Memory Read-32 Frame

         Traffic Class: 0x0
         TD: 0
         EP: 0
         Attributes: 0x0
         Length: 0x001
         Requester Id: 0x01a0
         Tag: 0x00
         Last and First Byte Enables: 0x0f
         Address Low: 0x00000000
```

X1110_22_120708

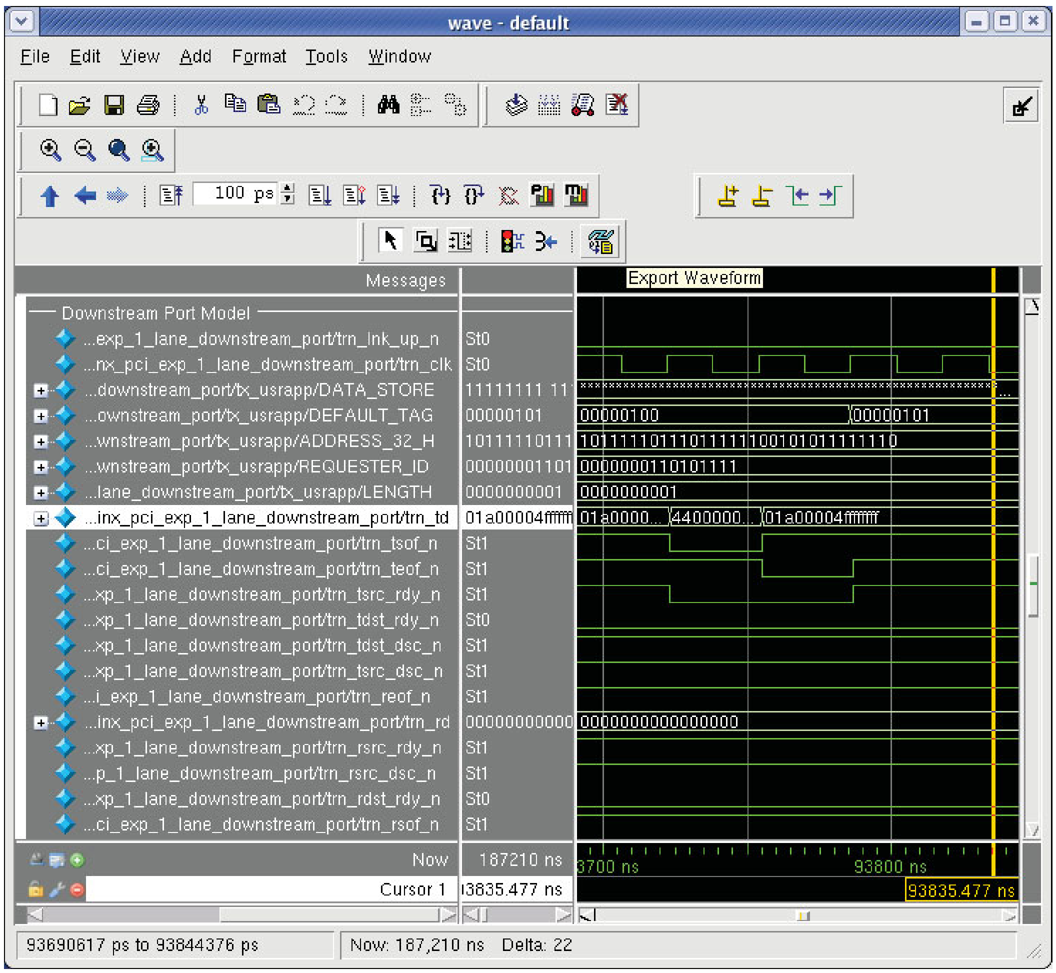
Figure 22: **Downstream Port Model rx.dat file**

Figure 23 shows the Configuration Write TLP on the DPM trn_td local link. The task in `rc2ep.v` is:

```
TSK_TX_TYPE0_CONFIGURATION_WRITE(DEFAULT_TAG, 12'h04, 32'hFFFFFFFF, 4'hF)
```

In this task, the 12'h04 offset is the Command Status Register in the Configuration Space Header. The third argument is the data written. All Fs are written to the Command/Status register to avoid looking up each bit.

The trn_td signal shows **44** as the type Configuration Write as defined in Table 4, followed by the `0xFFFFFFFF` payload.



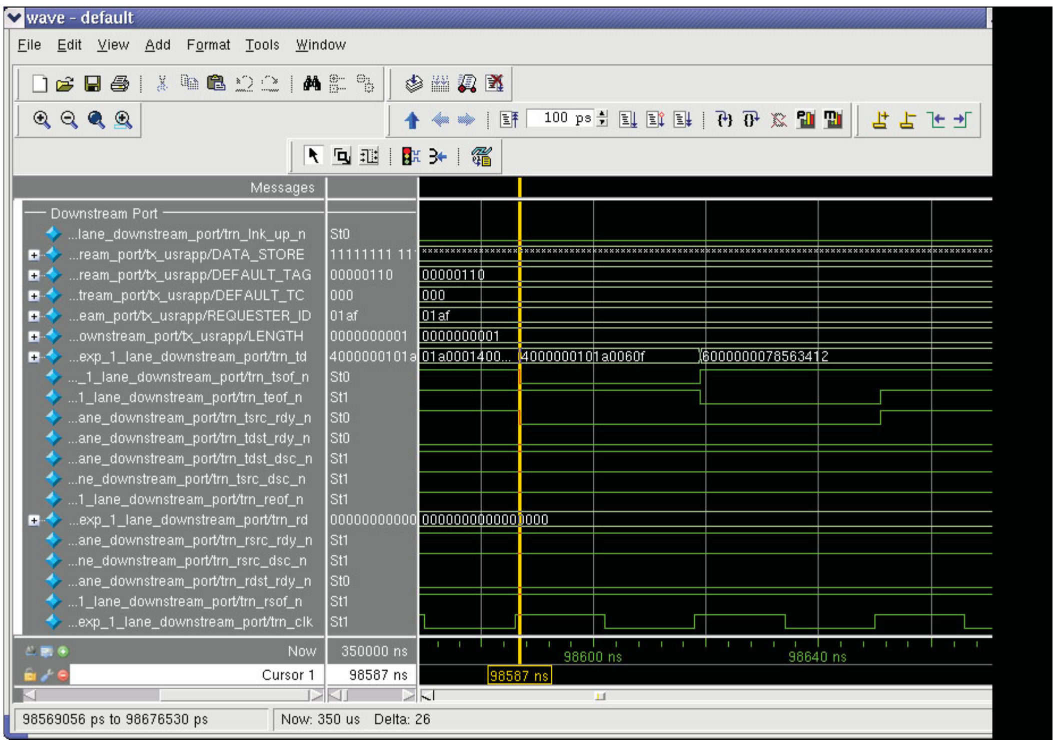
X1110_23_120708

Figure 23: **Downstream Port Model Configuration Write TLP**

Figure 24 shows DPM memory write TLP on trn_td. The task in `rc2ep.v` is:

```
TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 32'h60000000, 4'h0,
4'hF, 1'b0)
```

Setting the third argument, the length field, to 10'd1, causes a single DWORD write to be generated. A burst transaction is generated when the length field is > 1. The forth argument, the address, is `32'h60000000`. Since the `32'h60000000` address is the same address written in the configuration write statements in `rc2ep.v` to define BAR0 (offset `x10`, `x14`), this TLP will be recognized by the PLBv46 Endpoint Bridge.



X1110_24_120708

Figure 24: **Memory Write TLP**

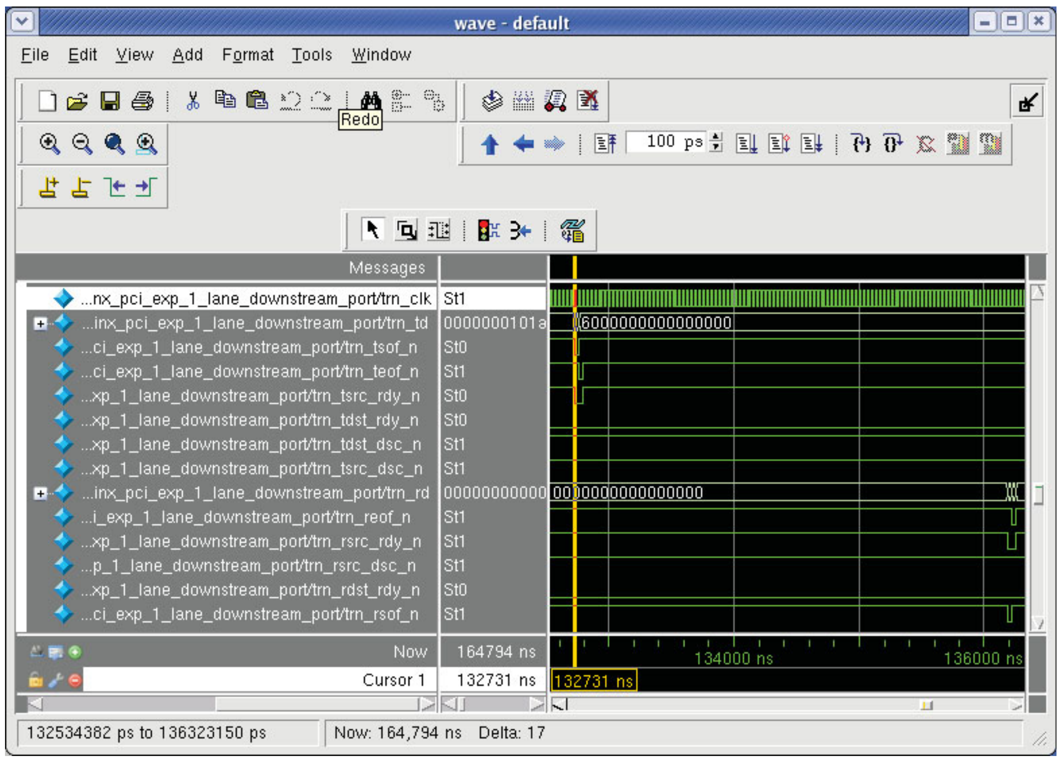
Figure 25 shows the Downstream Port Model Memory Read TLP on trn_td. The task in `rc2ep.v` which generates the read is:

```
TSK_TX_MEMORY_READ_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 32'h60000000, 4'h0,
4'hF)
```

Setting the third argument, the length field, to 10'd1, causes a single DWORD read to be generated. A burst transaction is generated when the length field is > 1. The address is `32'h60000000`, the address written in the configuration read statements in the `rc2ep.v` file.

Note: The code cited in this application note may or may not be in the rc2ep.v and ep2rc.bfl files provided in xapp1110.zip. This application note describes simulations for a number of different tests. To run a specific test, the rc2ep.v and ep2rc.bfl files generally need to be edited.

The memory read operation begins with a transmission of a type **00** Memory Read TLP on trn_td. After a relatively long cycle, a (type **4A**) CplD TLP is received on trn_rd.



X1110_25_120708

Figure 25: **Downstream Port Model Memory Read TLP**

Abnormal PCIe to PLBv46 Transactions

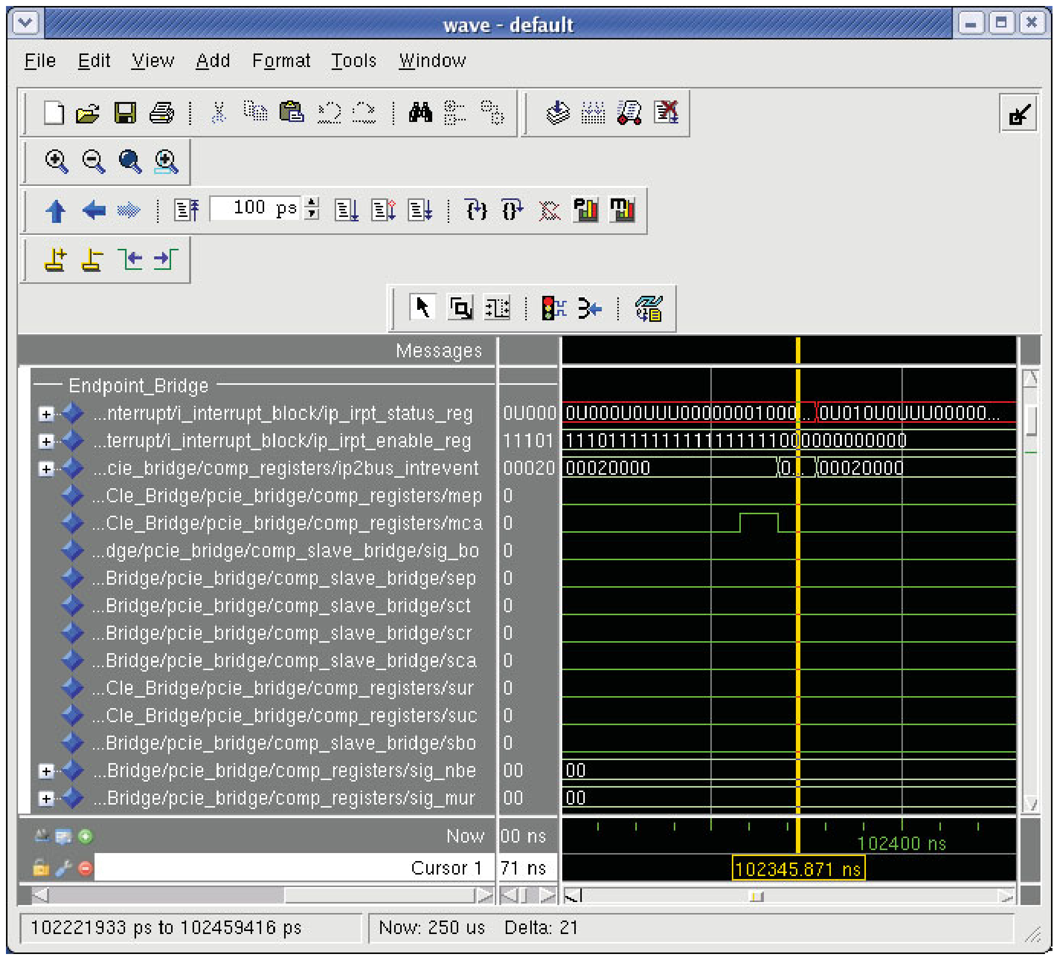
The next figures show how to generate abnormal conditions. Abnormal conditions are not desirable. The motivation for learning about abnormal conditions is to facilitate quick debug of incorrect behavior.

The Master Completion Abort (MCA) is asserted when the master side of the Endpoint Bridge receives an Abort from the PLB. To cause a MCA interrupt, change the C_PCIBAR2IPIFBAR_0 generic in `system.mhs` from `0x90000000`, the address of DDR2, to `0x60000000`, an address without a target. Re-run the **`simgen`** command.

The Global Interrupt Enable and Bridge Interrupt Enable registers are written in `ep2rc.bfl`. The `rc2ep.v` write command does not change:

```
TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 32'h60000000, 4'h0,
4'hF, 1'b0)
```

Figure 26 shows the Master Completer Abort interrupt.



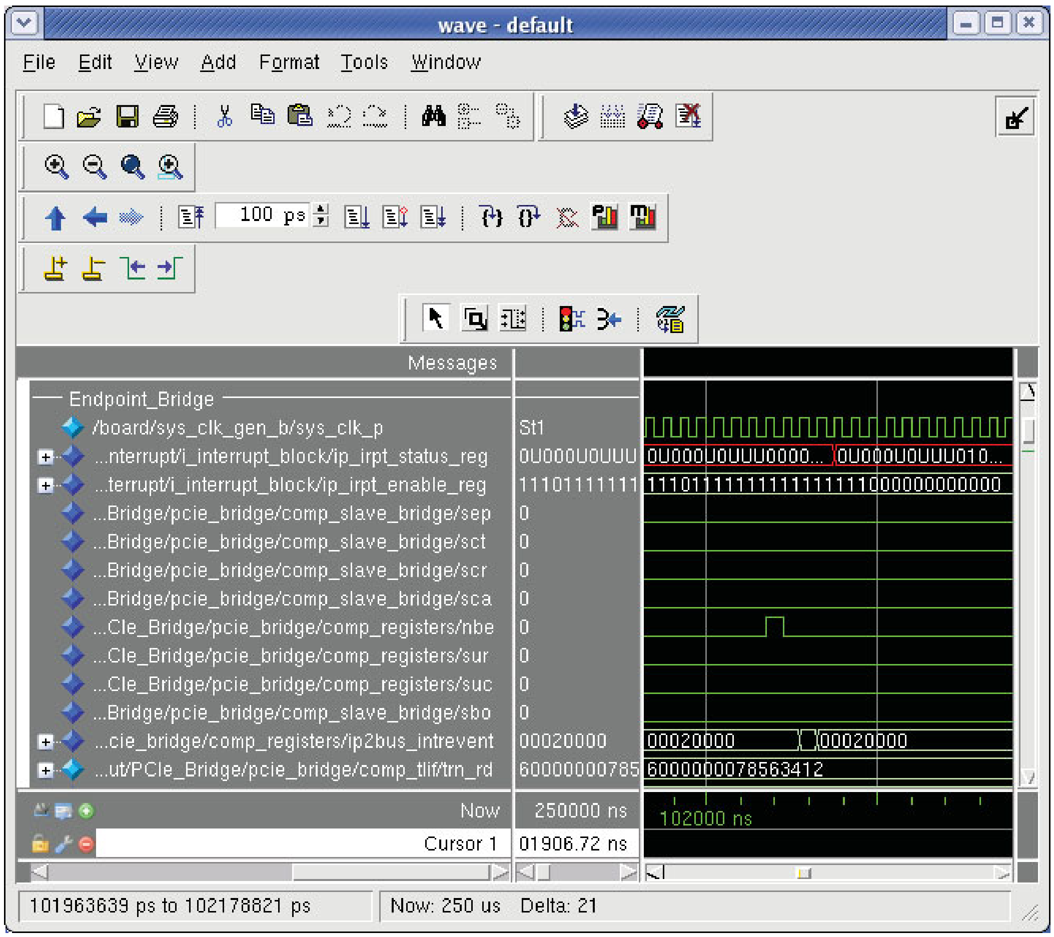
X1110_26_120708

Figure 26: **Master Completer Abort Interrupt**

To generate a Non-Contiguous Byte Enable interrupt, the original `system.mhs` is used. The sixth argument in the memory write task in the `rc2ep.v file`, First Byte Enable, is changed from `0xF` to `0xA`.

```
TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 32'h60000000, 4'h0,
4'hA, 1'b0)
```

Figure 27 shows the interrupt due to non-contiguous byte enables.



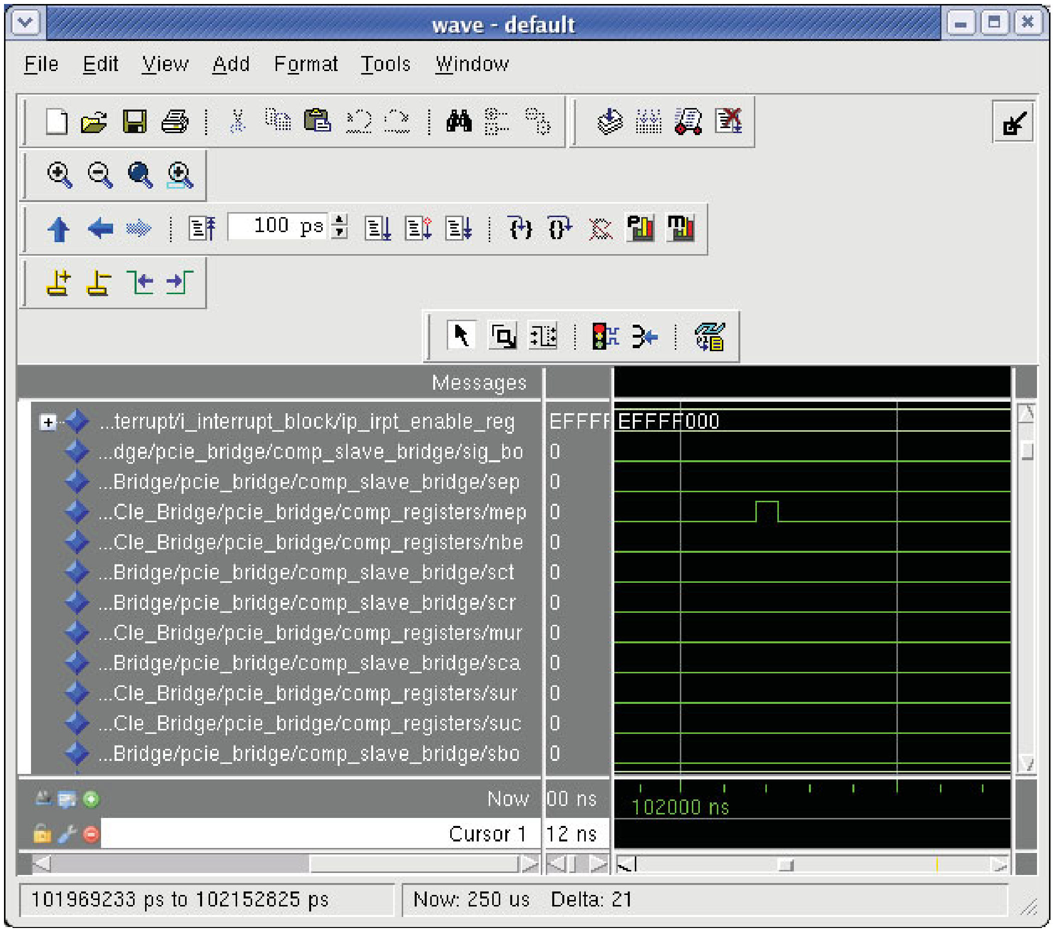
X1110_27_120708

Figure 27: **Interrupt Due to Non-contiguous Byte Enable(s)**

The interrupt due to a poisoned payload is generated by changing the last field in the memory write task in the `rc2ep.v` file from 0 to 1.

```
TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 32'h60000000, 4'h0,
4'hF, 1'b1)
```

Figure 28 shows the interrupt due to a Poison bit.



X1110_28_120708

Figure 28: **Poison Bit Interrupt**

Controlling EDK Functions from the PCIe Side

The `rc2ep.v` file can be edited so the DPM controls operations within the EDK system from the PCIe side. Typical applications include controlling the SPI or IIC cores in the EDK system.

Another application is to control the DMA Controller in the EDK system from the PCIe side. This may address the performance issue of PCIe to PLBv46 read operations. Write transactions are faster than read transactions. To transfer data from the EDK system memory to the DPM, a DMA operation controlled by the DPM may increase throughput.

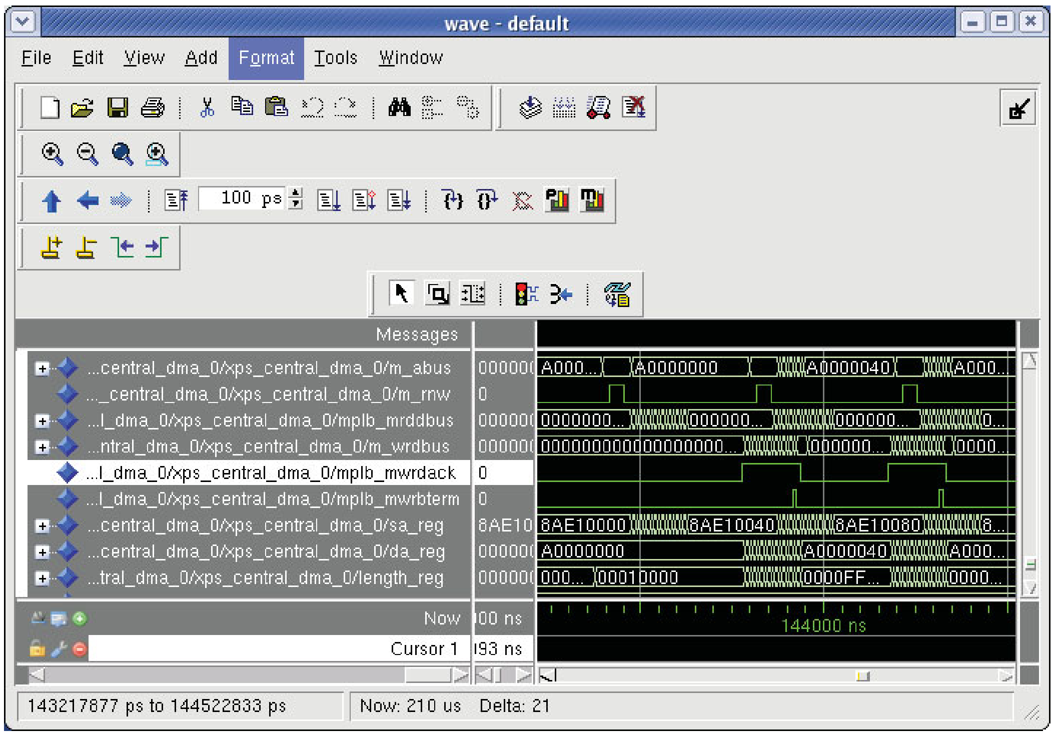
To initiate a DMA operation, the `rc2ep.v` file controls the XPS Central DMA Controller in the EDK System. The use of the DMA Controller is defined in the section "Stimuli from the PLBv46 side". To control the DMA controller with the DPM, set C_PCIBAR2IPIFBAR = x80200000, the base address of the XPS DMA Controller in the `system.mhs` file, and re-run **simgen**.

The `rc2ep.v` code below shows the control of the DMA controller from PCIe side. Four memory write tasks write to the control, source address, destination address, and length registers in the DMA controller. The data written to these registers is stored in the DATA_STORE array. The data is set for each write task.

```
-- Write DMA Controller Control Register

DATA_STORE[0] = 8'hC0;
DATA_STORE[1] = 8'h00;
DATA_STORE[2] = 8'h00;
DATA_STORE[3] = 8'h04;

TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 32'h80200004, 4'h0,
4'hF, 1'b1)

-- Write DMA Controller Source Address

DATA_STORE[0] = 8'h90;
DATA_STORE[1] = 8'h00;
DATA_STORE[2] = 8'h00;
DATA_STORE[3] = 8'h00;

TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 32'h80200008, 4'h0,
4'hF, 1'b1)

-- Write DMA Controller Destination Address

DATA_STORE[0] = 8'hA0;
DATA_STORE[1] = 8'h00;
DATA_STORE[2] = 8'h00;
DATA_STORE[3] = 8'h00;

TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 32'h8020000C, 4'h0,
4'hF, 1'b1)

-- Write DMA Controller Length

DATA_STORE[0] = 8'h00;
DATA_STORE[1] = 8'h00;
DATA_STORE[2] = 8'h10;
DATA_STORE[3] = 8'h00;

TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, 32'h80200010, 4'h0,
4'hF, 1'b1)
```

The tasks in the `rc2ep.v` file cause the DMA Controller in the EDK system to transfer data from XPS BRAM across the PCIe link. This transaction involves operations from the DPM, BRAM, PLBv46 Endpoint Bridge, and XPS Central DMA Controller, so analysis requires running a simulation and scrolling through the waveform viewer signals.

Figure 29 shows the XPS Central DMA in the EDK system reading BRAM at address `0x8AE10000` and writing the data to the C_IPIFBAR0 address `0xA0000000`. This address is translated across the PCIe link to the DPM.



X1110_29_120908

Figure 29: **DMA Controller Controlled from PCIe Side**

Stimuli from the PLBv46 side

The `ep2rc.bfl` is used to write and read EDK peripherals, both registers and memory. The Bus Functional Language (BFL) commands in `ep2rc.bfl` test the BRAM and MPMC/DDR2, and then set up the PLBv46 PCIe Bridge and XPS Central DMA Controller.

To do Endpoint to Root Complex transactions over the PCIe link, commands in `ep2rc.bfl` write to the C_IPIFBAR0 address range. There are two ways to write/read across the PCIe link. In one, a BFL write is done directly to the C_IPIFBAR0 address. In the second, the DMA Controller Source and/or Destination address is in the C_IPIFBAR0 range.

The `ep2rc.bfl` is compiled into a MTI do file when do `../scripts/run.do` is run. To provide an overview of the `ep2rc.bfl` file functionality, most of the `ep2rc.bfl` file is shown in Figure 30. A brief description of the file is given below.

The **`set_device`** command selects the model to initialize. The path argument specifies the path to the PLBv46 master. The device type argument specifies the type of model initialized.

The **configure** command allows the user to configure different PLBv46 model attributes. The msize=01 configures the master bus size as 64 bits.

The **wait(level = START)** command waits for the beginning of the simulation. The generation of the START is defined and controlled in testbench.v. Since the EDK system contains DDR2, this allows the BFL to wait for the DDR2 to initialize before generating stimuli.

The statements following the comment “Write/Read the Bridge Control Register” write a value of 0x003F0107 to the Bridge Control Register in the PLBv46 Endpoint Bridge, located at 0x85C010E0.

The read command verifies that the write is done correctly. This form of the write and read commands uses the aliases defined at the beginning of the ep2rc.bfl file. The aliases are not shown in the figure. It is not necessary to use aliases.

The section following the comment “Write/Read the Management Interface” reads the Management Interface in the PLBv46 Endpoint Bridge, which is located at 0x85C00000 + 0x2000.

The two sections following the comments “Test BRAM” and “Test DDR2” test that BRAM and DDR2 function as expected. In the section following the comment “Test DMA from BRAM to DDR2”, a DMA transaction from BRAM to DDR2 is done.

In the section following the comment “Test single write/read to Downstream Port Model”, a single write/read to the Downsteam Port Model is done. This uses a write command to an address in the C_IPIFBAR region (0xA0000000).

In the section following the comment “Test DMA to Downstream Port Model”, a DMA from BRAM to the Downstream Port Model across the PCIe link is done. In this case, the destination address in the DMA controller is in the C_IPIFBAR_0 region.

```
set_device(path /boardx/dut/plbv46_master_bfm_0/plbv46_master_bfm_0/master, device_type = plb_master)
configure(msize = 01)

wait(level=START);

-- Write/Read Bridge Control Register
mem_update(addr = 0x85C001E0, data = 003F0107)
write(addr = 0x85C001E0,  size = SINGLE_NORMAL, be = WORD0)
read (addr = 0x85C001E0,  size = SINGLE_NORMAL, be = WORD0)

-- Write/Read Management Interface
wait(level = NEXT)
mem_update(addr = 0x85C02000, data = EE100505)
read (addr = 0x85C02000, size = 0000, be = 1111_0000)

-- Test BRAM
wait(level = NEXT)
mem_update(addr = 0x8AE10000, data = 00000000)
write(addr = 0x8AE10000, size = WORD_BURST, be = FBURST16)
read(addr = 0x8AE10000, size = WORD_BURST, be = FBURST16)

-- Test DDR2
wait(level=NEXT)
mem_update(addr = 0x90000000, data = 00000000)
write(addr = 0x90000000, size = WORD_BURST, be = FBURST16)
read(addr = 0x90000000, size = WORD_BURST, be = FBURST16)

-- Test DMA from BRAM to DDR2
wait(level= NEXT)
mem_update(addr = 0x80200004, data = C0000004)
mem_update(addr = 0x80200008, data = 8AE10000)
mem_update(addr = 0x8020000C, data = 90001000)
mem_update(addr = 0x80200010, data = 00000010)
write(addr = 0x80200004, size = 0000, be = 0000_1111)
write(addr = 0x80200008, size = 0000, be = 1111_0000)
write(addr = 0x8020000C, size = 0000, be = 0000_1111)
write(addr = 0x80200010, size = 0000, be = 1111_0000)

-- Test single write/read to Downstream Port Model
wait(level=NEXT)
mem_update(addr = 0xA0000000, data = 12345678)
write(addr = 0xA0000000, size = 0000, be = 1111_0000)
read(addr = 0xA0000000, size = 0000, be = 1111_0000)

-- Test DMA to Downstream Port Model
wait(level=NEXT)
mem_update(addr = 0x80200004, data = C0000004)
mem_update(addr = 0x80200008, data = 8AE10000)
mem_update(addr = 0x8020000C, data = A0000000)
mem_update(addr = 0x80200010, data = 00000010)
write(addr = 0x80200004, size = 0000, be = 0000_1111)
write(addr = 0x80200008, size = 0000, be = 1111_0000)
write(addr = 0x8020000C, size = 0000, be = 0000_1111)
write(addr = 0x80200010, size = 0000, be = 1111_0000)
```

X1110_30_120708

Figure 30: **Excerpts of ep2rc.bfl file**

Figure 31 shows the waveform resulting from the write and read of the Bridge Control Register of the PLBv46 Endpoint Bridge. This write operation shows a value of `0x85C001e0` on PLB_ABus when PLB_PAvalid is high and a value of `0x003F0107` on PLB_Wrdbus when sl_wrcomp is asserted. The read operation shows a value of `0x003F0107` when sl_rdcomp is asserted.

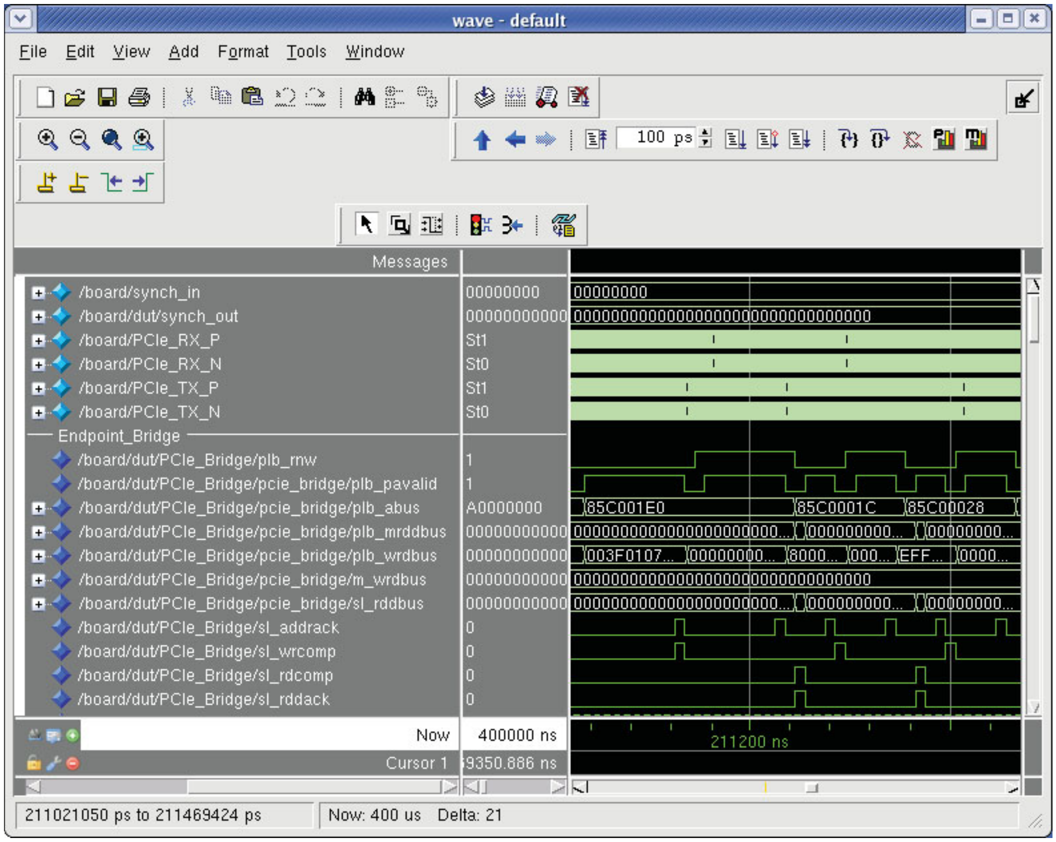


Figure 31: **Write and Read of Bridge Control Register**

The `ep2rc.bfl` file contains other setup commands, including enabling the interrupt registers.

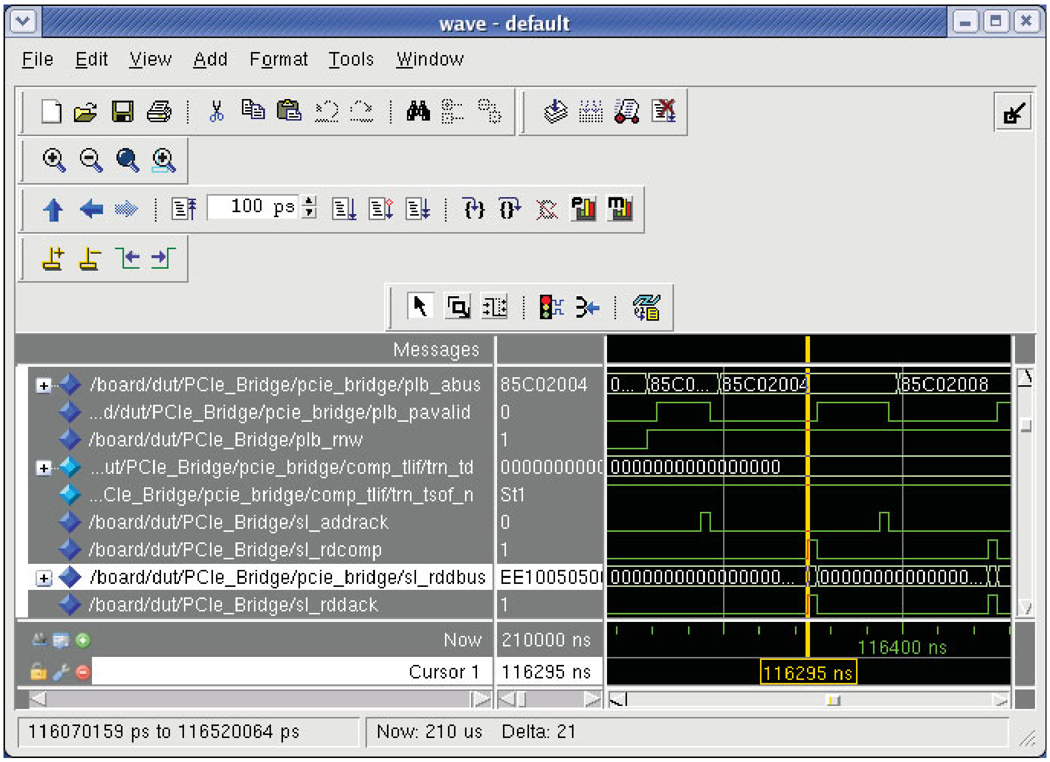
After the link is trained, the Request Control Register and the Status Register are read to determine max payload size, max read request size, link status, and link width:

```
-- Reading MPS, RRS
mem_update(addr = 0x85C001EC, data = 00000002)
read (addr= 0x85C001EC, size = 0000, be = 1111_0000)
-- Reading Link Width, Link Status
mem_update(addr = 0x85C001F0, data = 00000060)
read (addr= 0x85C001F0, size = 0000, be = 1111_0000)
```

Additional status registers from the PCIe core management interface (MI) are provided at the beginning Endpoint Bridge address C_BASEADDR + 0x2000. The content of these registers are defined in UG197 Virtex-5 Integrated Endpoint Block for PCI Express Designs User Guide. An excerpt of `ep2rc.bfl` which reads MI registers is given below.

```
-- Reading PLBv46 PCIe Management Interface Registers
mem_update(addr = 0x85C02000, data = EE100505)
mem_update(addr = 0x85C02004, data = 47051000)
mem_update(addr = 0x85C02008, data = 00008005)
read (addr = 0x85C02000, size = 0000, be = 1111_0000)
read (addr = 0x85C02004, size = 0000, be = 0000_1111)
read (addr = 0x85C02008, size = 0000, be = 1111_0000)
```

Figure 32 shows the reading the PCIe MI registers located at `0x85C00000 + x2000`.



X1110_32_120908

Figure 32: **Reading PCIe Management Interface Registers**

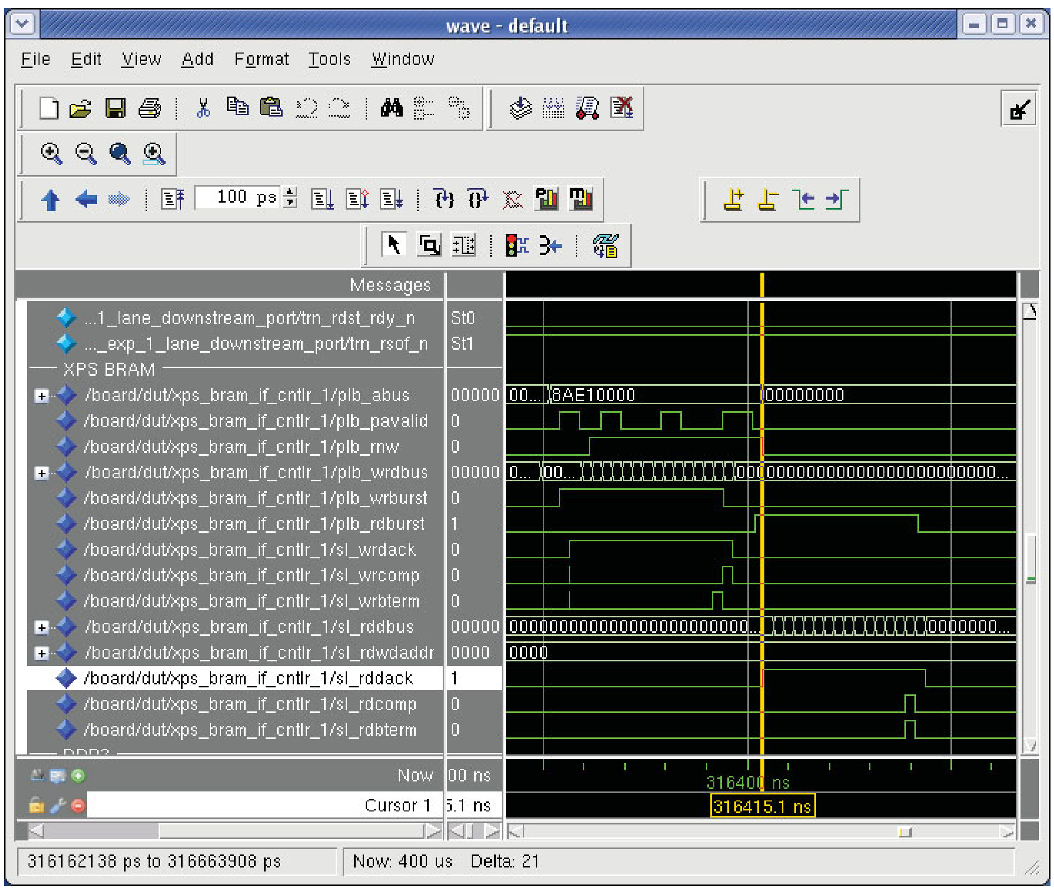
A common PCIe function is to transfer data from one memory across the PCIe link to a memory at the other end of the link. Memory addresses and address translations are used at each end of the PCIe link. Other PCIe use cases can often be modeled as subsets of a memory to memory transfer across the PCIe link. In the next figures, writing and reading stimuli to BRAM, DDR2, and the Central DMA controller is discussed. The initial transactions are to verify that BRAM, DDR2, and the Central DMA controller in the EDK system is functionally correct. After these tests, transfers across the PCIe link are tested.

Figure 33 shows writing and reading BRAM. The C_BASEADDR for BRAM is `0x8AE1000`.

Excerpts of `ep2rc.bfl` are

```
mem_update(addr = 0x8AE10000, data = 00000000
..
mem_update(addr = 0x8AE1003C, data = 0000000F
write(addr = 0x8AE10000, size = WORD_BURST, be = FBURST16)
read(addr = 0x8AE10000, size = WORD_BURST, be = FBURST16)
```

In the `ep2rc.bfl` file, WORD_BURST is aliased as 1010, and FBURST16 is aliased as 11110000.

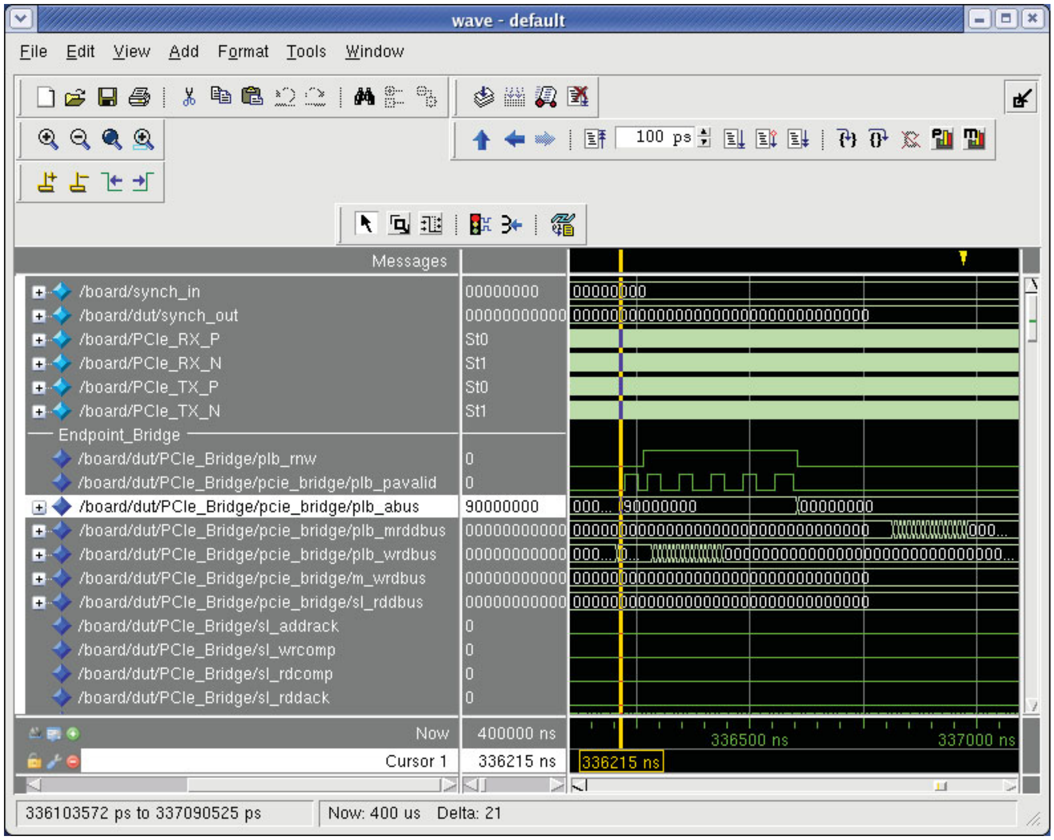


X1110_33_120708

Figure 33: **Writing and Reading BRAM**

DDR2 provides a much larger memory than BRAM. The `ep2rc.bfl` commands for testing DDR2 are similar to those provided for BRAM. DDR2 uses the Micron Memory model in the `simulation/ddr2` directory. In the simulation, the memory controller initialization time affects the startup of the simulation.

Figure 34 shows writing and reading DDR2.



X1110_34_120708

Figure 34: **Writing/Reading DDR2**

Memory to memory transactions are usually done by a DMA controller. Four write commands to the DMA controller registers defined in Table 6 generate a DMA operation.

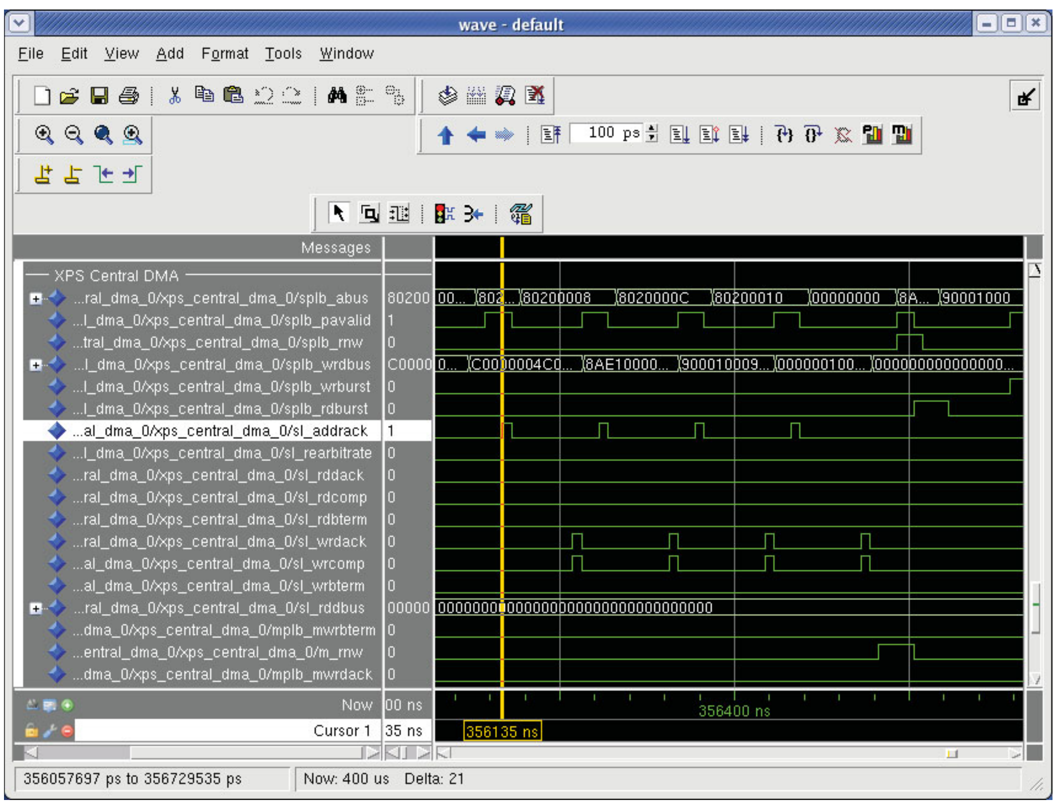
Table 6: **XPS Central DMA Controller Register**

DMA Register	Address
Control	`C_BASEADDR + 0x04`
Source Address	`C_BASEADDR + 0x08`
Destination Address	`C_BASEADDR + 0x0C`
Length	`C_BASEADDR + 0x10`

After writing to the DMA Controller control register, the source address (SA) and destination address (DA) register are written. The transfer is then initiated by a write to the length register. The stimuli in `ep2rc.bfl` initially verifies DMA controller operation with a DMA transfer from BRAM to DDR2.

A transaction over the PCIe link occurs when a bus master writes an address which resides in th C_IPIFBAR region, which is in the range C_IPIFBAR: C_IPIFBAR_HIGHADDRESS. To read across the PCIe link, write an address in the C_IPIFBAR region to the DMA controller Source Address register. To write across the PCIe link, write an address in the C_IPIFBAR region to the DMA controller Destination Address register.

Figure 35 shows the DMA Controller operation. The splb_wrdbus has values written to the DMA controller control, SA, DA, and length registers when sl_wrcomp is asserted.



X1110_35_120708

Figure 35: **DMA Operations**

PCIe write transactions are posted.

The IPIFBAR used for the PLB to PCIe transactions is defined in `system.mhs` as C_IPIFBAR_0 = `0xA0000000`.

The excerpt of the `ep2rc.bfl` file used to generate a write transaction to the PCIe link is

```
mem_update(addr = 0xA0000000, data = 12345678)
write(addr = 0xA0000000, size = 0000, be = 1111_0000)
```

Figure 36 shows a PLB to PCIe single write operation. The value written is `0x12345678`.

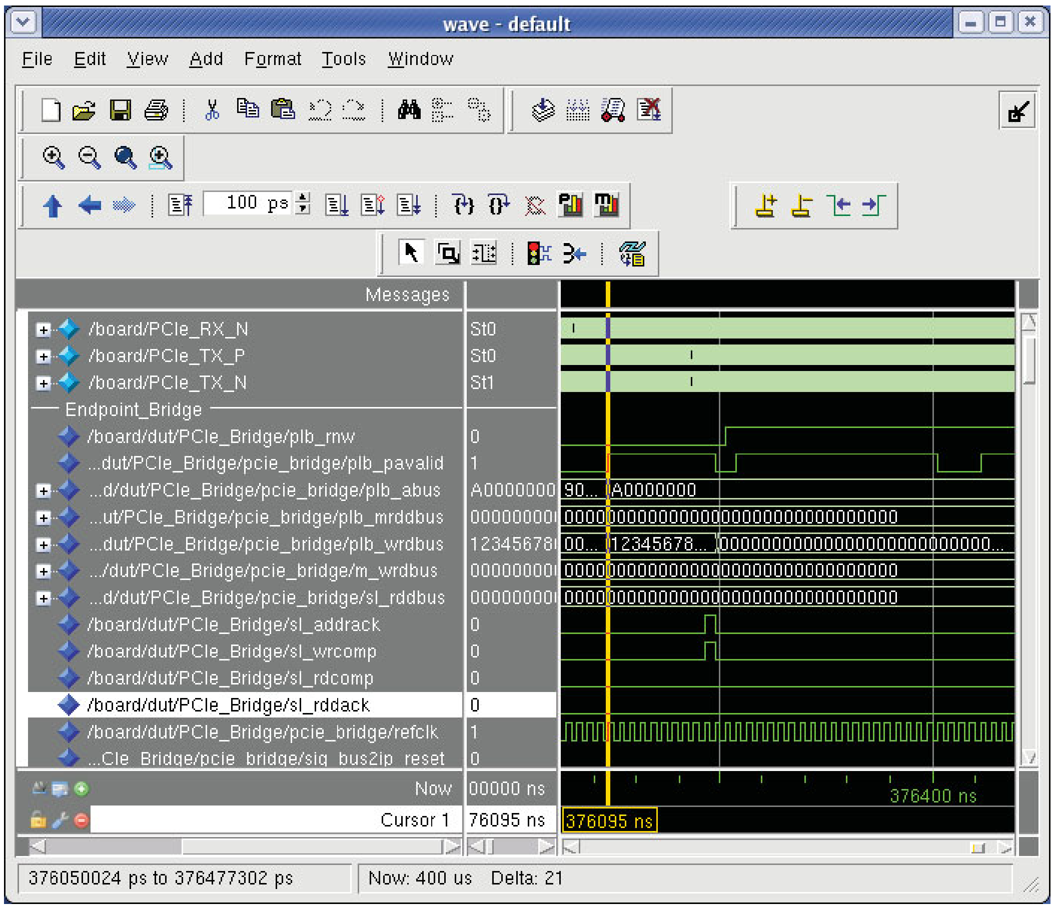


Figure 36: **Single Write Operation**

PCIe read transactions are non-posted, meaning that a response is required.

The IPIFBAR used for the PLB to PCIe transactions is defined in `system.mhs` as C_IPIFBAR_0 = `0xA0000000`.

The excerpt of `ep2rc.bfl` used to generate the read transaction is

```
mem_update(addr = 0xA0000000, data = 12345678)
read (addr = 0xA0000000, size = 0000, be = 1111_0000)
```

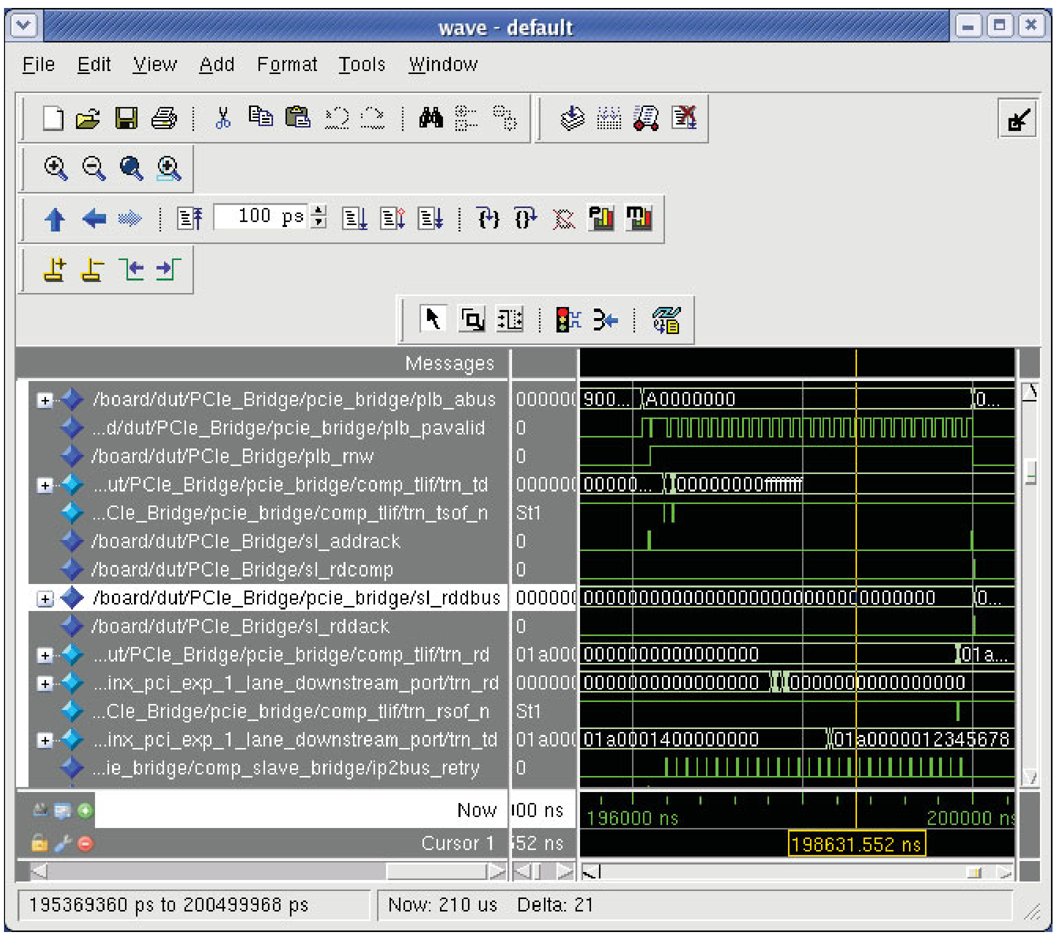
After receiving the MemRd TLP on trn_rd, the DPM writes a completion with data TLP (CplD). The following task in `rc2ep.v` generates the CplD TLP.

```
TSK_TX_COMPLETION_DATA(0, DEFAULT_TC, 10'd1, 12'd4, 7'b0000000, 3'h0,
1'b0)
```

To match the request tag, the first argument in TSK_TX_COMPLETION_DATA is changed from DEFAULT_TAG to 0. The data in the completion packet is defined in the DATA_STORE array as `0x12345678`.

Read transactions are complex, and it is constructive to read the DPM `rx.dat` and `tx.dat` files to understand the timing.

Figure 37 shows the overall PLB to PCIe single Read operation to `0xA0000000`. The signals in the figure are in the PLBv46 Endpoint Bridge. To understand the read operation, the signals at the receive end (DPM) must also be analyzed.



X1110_37_120908

Figure 37: **Complete PLBv46 to PCIe Read Operation**

Read transactions start with the master read request on the PLB. The Endpoint Bridge then generates a MemRd TLP request across the PCIe link. At the receiver, the data is normally fetched from memory. The DPM does not have memory. In the `rc2ep.v` file, the DPM transmits the CplD TLP back to the requesting Endpoint Bridge, and the Endpoint Bridge completes the read on the PLB.

Figure 38 shows the start of the PLBv46 to PCIe read operation from the EDK system.

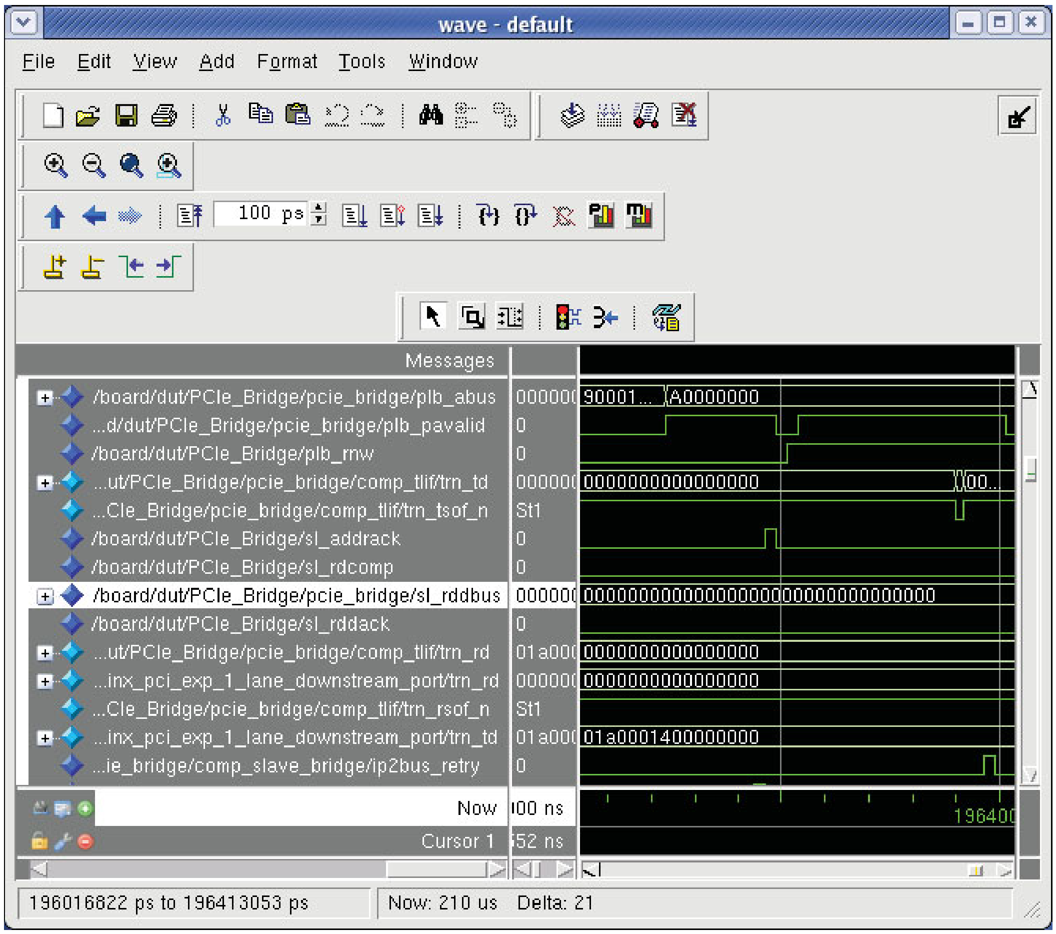
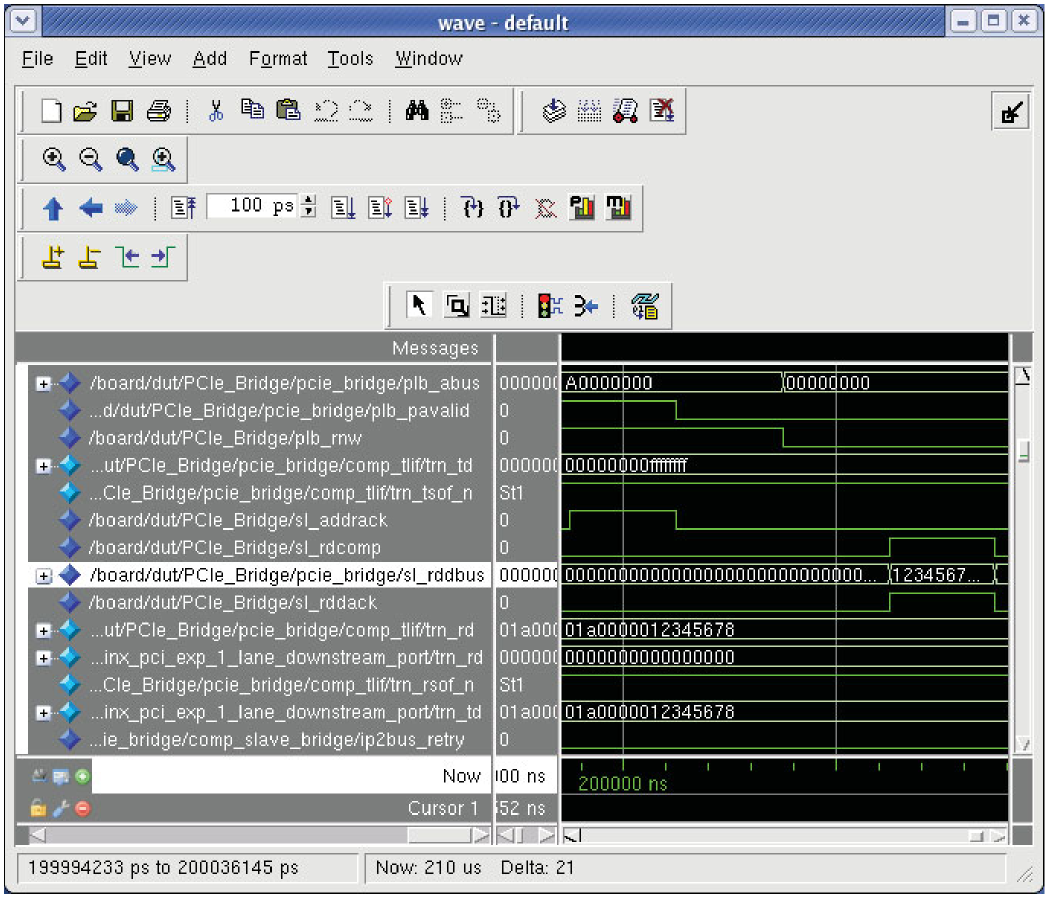


Figure 38: **Start of PLBv46 to PCIe Read Operation**

Figure 39 shows the end of the PLBv46 to PCIe read operation for the EDK system. The sl_rdcomp is asserted at 200.4 us. The address is `0xA0000000`.



X1110_39_120908

Figure 39: **Completion of the PLBv46 to PCIe Read Operation**

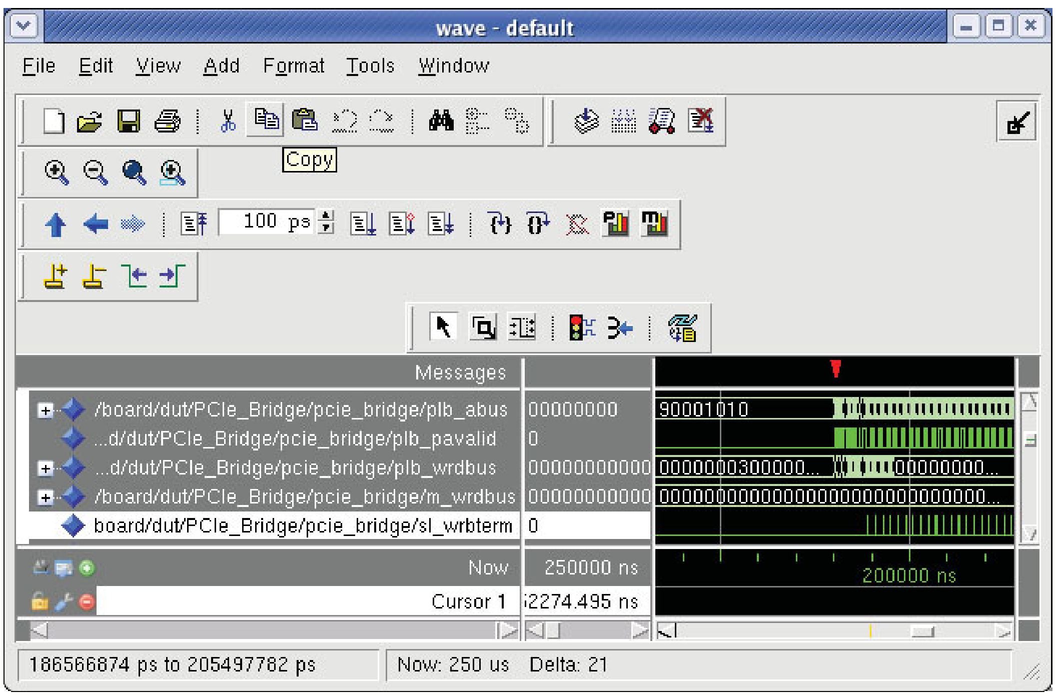
In the next pages, example PLB to PCIe abnormal transactions are given.

The maximum payload size in the Endpoint Bridge, given in the Request Control register, is 128. When the requested payload size of a TLP exceeds the MPS, the Endpoint Bridge asserts Sl_wrBterm.

The following commands in `ep2rc.bfl` use the DMA Controller to write a packet which exceeds the MPS across the PCIe link:

```
-- Exceed MPS: Verify sl_WrBterm asserted
mem_update(addr = 0x80200008, data = 8AE10000)
mem_update(addr = 0x8020000C, data = A0000000)
mem_update(addr = 0x80200010, data = 00001000)
write(addr = 0x80200004, size = 0000, be = 0000_1111)
write(addr = 0x80200008, size = 0000, be = 1111_0000)
write(addr = 0x8020000C, size = 0000, be = 0000_1111)
write(addr = 0x80200010, size = 0000, be = 1111_0000)
```

Figure 40 shows the Endpoint Bridge assertion of slave burst terminate when the payload size exceeds maximum payload size.



X1110_40_120708

Figure 40: **Burst Termination - Sl_WrBterm**

The Endpoint Bridge generates a Slave Bar Overrun interrupt when a PLB master requests an address outside the IPIFBAR address range. The following values for generics are defined In `system.mhs`

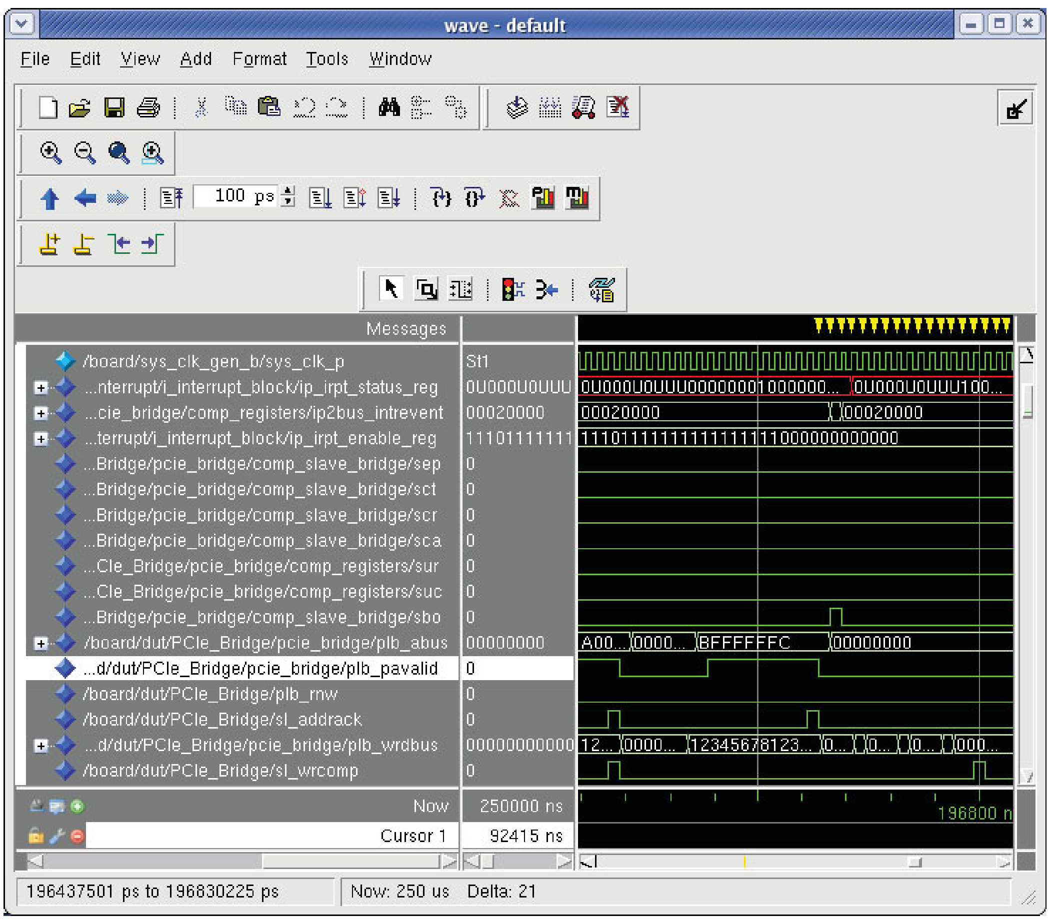
C_IPIFBAR_0 = `0xA0000000`

C_IPIFBAR_HIGHADDRESS_0 = `0xBFFFFFFF`

The following `ep2rc.bfl` excerpt sets the Destination Address register in the DMA controller close to C_IPIFBAR_HIGHADDRESS_0, and writes a length in the Length register which causes a write outside the IPIFBAR upper boundary.

```
-- Generate SBO
mem_update(addr = 0x80200008, data = 8AE10000)
mem_update(addr = 0x8020000C, data = BFFFFFF0)
mem_update(addr = 0x80200010, data = 00000100)
write(addr = 0x80200004, size = 0000, be = 0000_1111)
write(addr = 0x80200008, size = 0000, be = 1111_0000)
write(addr = 0x8020000C, size = 0000, be = 0000_1111)
write(addr = 0x80200010, size = 0000, be = 1111_0000)
```

Figure 41 shows the slave bar overrun interrupt SBO. The address on PLB_ABus is `0xBFFFFFFC`.



X1110_41_120708

Figure 41: **Overrun Error**

Reset Functionality

The Endpoint Bridge can be reset using a software reset, PERSTN, or a PLB reset. The Endpoint Bridge consists of the Block Plus LogiCORE, PLB Master and Slave interfaces, and Bridge functions. A simulation can show the reset behavior of each section.

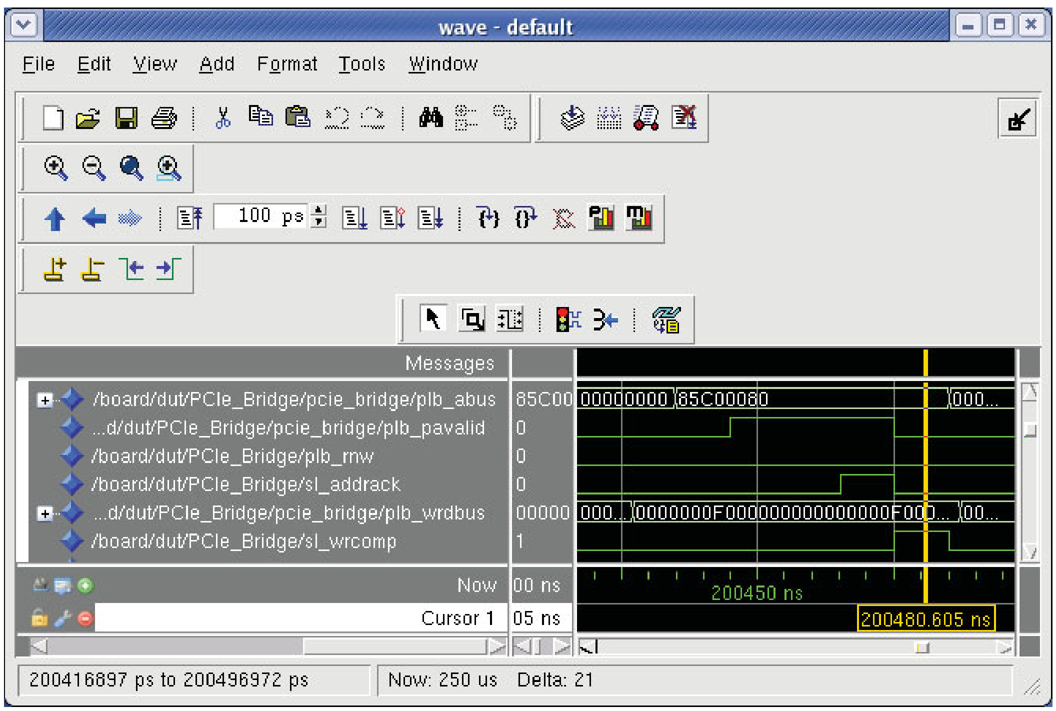
To generate a soft reset, the following command is used in `ep2rc.bfl`:

```
mem_update(addr = 0x85C00080, data = 0000000F)
write(addr = 0x85C00080, size = 0000, be = 1111_0000)
```

To generate a reset on PLB or PERSTN, edit `testbench/testbench.v`:

```
initial begin
 $timeformat(-9,2," ns", 1);
  sys_reset = 1;
  synch_in = 0;
  #1000;
  sys_reset = 1;
```

Figure 42 shows a write of `0x0000000F` to address `0x85C00080` on plb_abus to generate a software reset. Add registers whose reset behavior is of interest to the waveform viewer.



X1110_42_120708

Figure 42: **Reset Functionality**

Synchronizing Stimuli from the PLBv46 side

Although simultaneous transactions can be done, most of the tests in rc2ep.v and ep2rc.bfl are point tests, meant to occur independent of other tests. To focus on a single test at a time, synchronization operations are used. The BFM Synch module in the EDK project provides a method of synchronizing operations between the Downstream Port Model and the PLBv46 Master BFM transactions. The synch bus is used. Synchronization transactions are in the `ep2rc.bfl` and in `testbench.v`. Timing in `rc2ep.v` is controlled by the Verilog time delay:

```
#2000
```

The START, STOP, and NEXT aliases in `ep2rc.bfl` are used to control timing. To divide the timing of individual tests in `ep2rc.bfl`, the wait command is used:

```
wait(level=NEXT)
```

The SendSynch task is defined at the bottom of `testbench.v`. Typical code in `testbench.v` to delineate a test is:

```
$display("[%t] : Testing BFL Endpoint to DPM transfers ...", $realtime);
SendSynch(SYNC_NEXT);
#2000
```

In this case, the BFL transfers control to the testbench only for the testbench to display information on the test being done. In other cases, the BFL transfers control so that `testbench.v` or `rc2ep.v` generates or checks stimuli.

Figure 43 shows the synchronization signals in the waveform viewer.

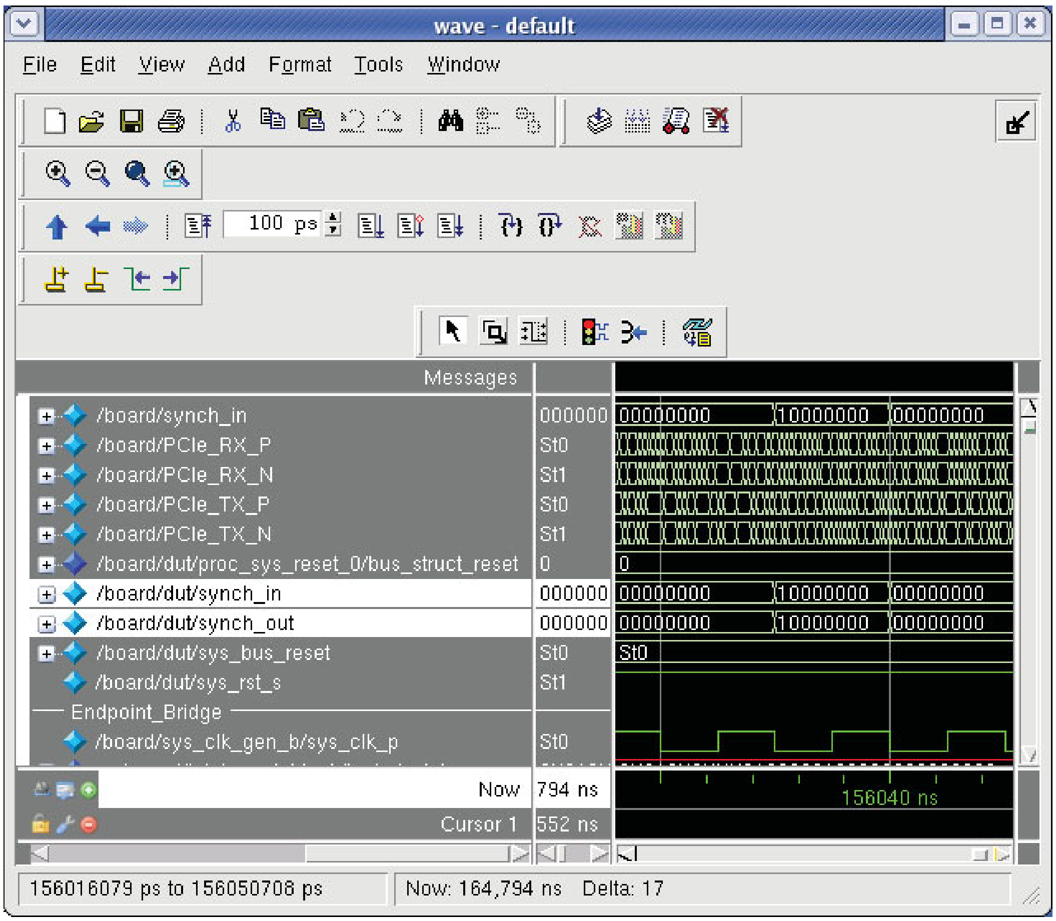


Figure 43: **Synchronization Signals**

References

1. UG197 Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs User Guide
2. UG341 LogiCORE Endpoint Block Plus v1.7 for PCI Express User Guide - April 25, 2008
3. XAPP1030 Reference System: PLBv46 PCIe in the ML505 Embedded Development Platform
4. XAPP1111 C Simulation of an EDK System which Uses the PLBv46 Endpoint Bridge for PCI Express
5. XAPP1000 Reference System: PLBv46 PCIe in the ML555 PCI/PCIE

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
4/13/09	1.0	Initial release.

Notice of Disclaimer