



Lattice PCI Express Scatter-Gather DMA Demo Verilog Source Code

User's Guide

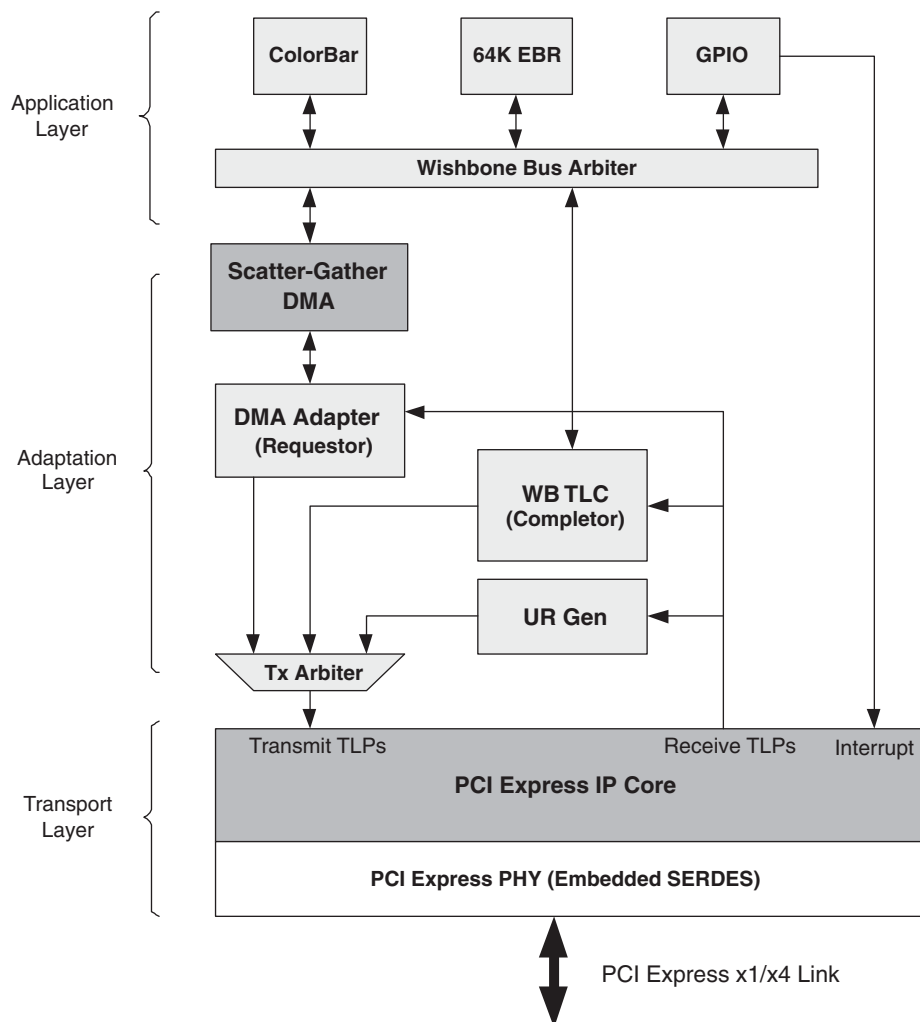
Introduction

This user's guide provides details of the Verilog code used for the Lattice PCI Express Scatter-Gather DMA Demo. A block diagram of the entire design is provided followed by a description for each module in the design. Instructions for building the demo design in ispLEVER® Project Navigator are provided as well as a review of the preference file used for the demo.

Top Level

Figure 1 provides a top-level diagram of the demo Verilog design.

Figure 1. Top Level Block Diagram



The block diagram is separated into three distinct functions. The Transport Layer is used to move TLPs to and from the PCI Express link. This set of features is supported by the Lattice PCI Express IP Core and embedded SERDES. Moving up the design stack, the next layer is the Adaptation Layer. This layer is responsible for converting PCI Express TLPs into useful data for the Application Layer. The Adaptation Layer uses Lattice's Scatter-Gather DMA IP core as well as several other soft-IP Verilog modules to extract received TLP contents and repackage transmit data into TLPs for transmission. The final level is the Application Layer. The Application Layer provides the demo capability that is utilized by the demo software. In this particular demo application a color bar and modified floating triangle are displayed on the monitor for the user.

SGDMA Project Directory Structure

Figure 2 provides the directory structure for the ispLEVER project. The LatticeECP2M™ and LatticeSC™ implementations use most of the same files. The only differences are the top-level modules and the IP cores. Specific files for each architecture family are located in their respective directories.

Figure 2. Directory Structure

```
<device>_PCIeSGDMA_<link width>/
  Source/
    wb_tlc/
    dma_adapter/
    gpio/
    ur_gen/
    64kebr/
    colorbar/
    wb_arb/
    <architecture>/
  ispLeverGenCore/
    <architecture>/
      pcie[x4,x4d1]/
      sgdma/
      pll/
  Implementation/
    <device>_PCIeSGDMA_<link width>/
```

Modules

This section discusses the details of each of the modules that make up the demo design. Each module listed will be followed by the filename of the Verilog file which includes this module. Verilog files can be found in the “src” directory of the demo package.

Top-Level – top_sgdma.v

The top-level module contains all of the blocks found in Figure 1. There is a different top-level file for the LatticeSC and the LatticeECP2M. The LatticeECP2M uses a PIPE interface to the embedded SERDES. The LatticeSC encapsulates the SERDES inside of the PCI Express IP core. The LatticeSC also uses the system bus to control the multi-channel alignment registers inside of the embedded PCS/SERDES. Other than these two differences, the top-level files are the same.

PCI Express IP Core – pcie_bb.v

This module is the Lattice PCI Express IP core. This module is an encrypted IP core which uses a Verilog black box model for synthesis and an obfuscated Verilog simulation model. IPexpress™ is used to create this module. In the ispLeverGenCore/<arch>/pcie[x4,x4d1] directory the file pcie.lpc is located. This file can be used to load the settings into IPexpress to recreate or modify the pcie module.

For the LatticeECP2M an additional file, pcie_top.v, is provided in the Source/ecp2m directory. This file provides a Verilog wrapper for the LatticeECP2M PCI Express core and LatticeECP2M PIPE interface.

x4 and x4d1

In the `ispLeverGenCore/<arch>/` directory there is either a `pciex4` or `pciex4d1` directory. The `pciex4` directory is used if the demo supports a native x4 PCI Express link. The `pciex4d1` is used if the demo supports a x1 link. The `pciex4d1` directory contains the Lattice x4 PCI Express IP core downgraded to a x1. This core is used instead of the native Lattice x1 PCI Express IP core to maintain the 64-bit interface required by the demo code. The native Lattice x1 PCI Express IP core uses a 16-bit interface, which can not be used with the demo code.

Embedded SERDES – pcs_pipe_bb.v (LatticeECP2M Only)

The LatticeECP2M uses a PIPE interface to the embedded SERDES. This module provides the PIPE interface and the SERDES interface. This file is created when creating the PCI Express IP core for the LatticeECP2M.

Tx Arbiter - ip_tx_arbiter.v

This module allows several clients to send TLPs to the PCI Express IP core. Using a round-robin scheduler, the tx arbiter waits for the current client to finish sending TLPs before switching to the next requesting client.

Rx Credit Processing – ip_rx_crpr.v

This module monitors receive TLPs coming from the PCI Express IP core and terminates credits for Posted and Non-Posted TLPs that are not handled by the `wb_tlc`. The number of credits used by each TLP is calculated and using the PCI Express IP core ports these credits are terminated. Terminated credits are stored until an UpdateFC DLLP can be sent by the PCI Express IP core informing the far-end that the credits have been freed.

Unsupported Request Generation – ur_gen.v

All Non-Posted TLPs and memory accesses to unsupported memory must provide a completion. This module will generate unsupported request completions informing the far-end device of these types of memory transactions.

This module receives input from both the PCI Express IP core `rx_us_req` signal as well as TLPs from the receiver. Whenever the `rx_us_req` signal indicates an unsupported request, this module will send an unsupported request completion to the PCI Express IP core.

This demo does not support I/O requests. Whenever an I/O request is made, this module will send an unsupported request completion to the PCI Express IP core.

This demo implements two BARs (BAR0 and BAR1). If a memory request is made to an address other than that serviced by BAR0 or BAR1, an unsupported request completion will be sent to the PCI Express IP core.

Wishbone Transaction Layer Completer (WB_TLC) – wb_tlc.v

The Wishbone Transaction Layer Completer (WB_TLC) is used as a control plane interface for the endpoint. This module accepts received TLPs from the PCI Express IP core. If they are memory transactions to either BAR0 or BAR1, the memory transaction is used by the completer. Otherwise, the TLP is dropped and is handled by the unsupported request module.

The WB_TLC is responsible for adapting 1DW TLP memory requests into wishbone transactions. 1 DW TLPs are only supported since this is a low throughput control plane interface which reduces logic. There are six modules underneath the WB_TLC top level.

TLP Decoder – wb_tlc_dec.v

The WB_TLC TLP decoder is responsible for decoding the type of TLP that enters the WB_TLC. The WB_TLC is only capable of supporting MRd and MWr TLPs that are accessing BAR0 or BAR1. All other TLPs are dropped and presumably handled by other modules in the design. After leaving the decoder, MRd and MWr TLPs are stored in a FIFO.

Memory Request TLP FIFO – `wb_tlc_req_fifo.v`

The request FIFO is used to store accepted TLPs until they can be converted into wishbone transactions on the wishbone bus. The request FIFO provides two clock domains (write and read), the 125 MHz PCI Express domain clock is used to write TLPs into the FIFO while the 105 MHz Wishbone domain clock is used to read TLPs from the FIFO.

TLPs are read from the FIFO when the FIFO is no longer empty under control of the wishbone interface module. This module terminates write credits when the TLPs are pulled from this FIFO. Read credits are terminated when the completion is sent.

Credit Processor - `wb_tlc_cr.v`

This module converts credits terminated in the wishbone clock domain to the PCI Express domain.

WB_TLC Wishbone Interface – `wb_intf.v`

The wishbone interface of the WB_TLC is responsible for reading TLPs from the request FIFO and creating wishbone transactions. When the request FIFO is not empty, the wishbone interface will read the TLP from the FIFO until the end of the TLP. As the TLP is read, the address and data will be converted into a wishbone transaction. The address is adjusted to use the least significant 18 bits.

For read transactions, the transaction ID is passed out of the module to be used by the completion generation module.

WB_TLC Completion Generation – `wb_tlc_cpId.v`

The WB_TLC completion generation module is responsible for accepting data from a read request on the wishbone bus and creating a CplD TLP. As data is returned from a read on the wishbone bus, the CplD TLP is filled with this data. The CplD also uses the transaction ID and length from the `wb_intf` module to fill the remaining fields in the CplD. Once the CplD TLP is created it is stored in the CplD FIFO.

WB_TLC Completion FIFO – `wb_tlc_cpId_fifo.v`

The completion FIFO stores the CplD TLPs from the completion generation module until the TLP can be sent to the PCI Express IP core.

Scatter-Gather DMA – `sgdma_top.v`

The Scatter-Gather DMA module is a wrapper to encapsulate the Scatter-Gather DMA IP core and the buffer descriptor memory used by the DMA core.

The Scatter-Gather DMA IP core (`sgdma_bb.v`) is an encrypted IP core which uses a Verilog black box model for synthesis and an obfuscated Verilog simulation model. IPexpress is used to create this module. In the `sgdma` directory the file `sgdma.lpc` is located. This file can be used to load the settings into IPexpress to recreate or modify the module.

The buffer descriptor memory is used to hold all of the buffer descriptors used by the SGDMA. A PMI-based Embedded Block RAM (EBR) is used for the buffer descriptor memory. This demo uses all 256 buffer descriptors as an example. This requires an EBR with a 10-bit address.

DMA Adapter – `dma_adapter.v`

The DMA Adapter is responsible for interfacing the SGDMA to the PCI Express IP core. Like the SGDMA, the DMA adapter uses channels. Channel 0 is used for writing data from the wishbone bus (via the SGDMA) to the PCI Express core. Channel 1 is used for reading data from the PCI Express core to the wishbone bus (via the SGDMA). The remaining channels of the SGDMA are not used by this adapter implementation.

When the SGDMA does a write operation (from the wishbone to the PCI Express) channel 0 is used to write data into the adapter. The burst length of the wishbone transaction will be no larger than 128 bytes (the largest size supported by the demo software). Using the burst length information from the SGDMA a TLP is created and stored in the transmit FIFO. When this FIFO is not empty the adapter requests to send this TLP to the PCI Express IP core. First the number of available Posted credits is checked and then the TLP is requested to be sent and finally sent.

When the SGDMA does a read operation (from the PCI Express to the wishbone) channel 1 is used to terminate a read transaction from the SGDMA. The adapter will immediately issue a wishbone retry informing the SGDMA that this transaction will take some time and to wait until a request comes back from the adapter to gather the data.

The adapter then converts the wishbone read transaction into a memory read TLP to be sent to the PCI Express IP core. This TLP is stored in the transmit FIFO. The TLP is read out of the FIFO when there are enough Non-Posted credits available to send the TLP. A request is made to send the TLP followed by the actual TLP.

A memory read TLP may come back with several CplD TLPs until all of the data has been returned. The adapter will wait and store data until all of the data has been returned. Once all of the data has been returned the adapter will issue a `dma_req` to the SGDMA alerting the SGDMA that all of the data for that channel has been collected. The SGDMA will then issue a read to gather all of the data. As the data has been read from the FIFO to the SGDMA the credits for these CplDs are released back to the PCI Express IP core.

The DMA adapter uses several modules to provide the functionality described.

Wishbone Slave – `dma_wbs.v`

The wishbone slave interface of the dma adapter is a wishbone slave which talks directly with the SGDMA master interface. This module converts the wishbone transaction into a memory TLP.

Transmit FIFO – `dma_tx_fifo.v`

The transmit FIFO is used to store both the memory write and memory read transactions before sending them to the PCI Express IP core.

Credit Available – `dma_ca.v`

This module is used to compare if enough credits are available to send a TLP to the PCI Express IP core interface. The number of credits of the next TLP to be sent is compared to the number of credits available from the PCI Express IP core. If enough credits are not available to send a TLP, then the TLP request is not made until the credits become available.

Receive FIFO – `dma_rx_fifo.v`

The receive FIFO module is used to store CplD TLPs from the PCI Express core. These CplD TLPs must match the tag of the outstanding request otherwise the CplD is dropped. The receive FIFO will then store all of the CplD data until all of the bytes have been received and then issue a `dma_req`. When the CplD data is read from the FIFO the credits are released to the PCI Express IP core.

Control – `dma_ctrl.v`

The DMA adapter control module is responsible for controlling the read of the transmit FIFO and sending TLPs to the PCI Express IP core. It is responsible for making sure that the credits are checked before sending and handling the requests and ready indications from the core.

Wishbone Arbiter – `wb_arb.v`

The wishbone arbiter is responsible for arbitrating between the WB_TLC and the SGDMA for access to the wishbone bus. It is also responsible for the slave select based on an address decode from the master selected. To account for the demo applications features, the arbiter does not support all masters transacting with all slaves. The SGDMA can only request data from the colorbar and 64k EBR. The WB_TLC on the other hand can request data from all slaves on the wishbone bus.

GPIO – `wbs_gpio.v`

The General Purpose Input Output (GPIO) module is responsible for several housekeeping functions. It provides an ID register used by the software to identify the feature set and version of the design. It also provides access to

control the 16-segment LED on the board. There is a section dedicated to interrupt control logic as well as other maintenance type functions. Table 1 is a memory map for the GPIO module.

Table 1. GPIO Module Memory Map

Address	Bits	Description
0x0	[0:31]	ID register
0x4	[0:31]	Scratch pad
0x8	[0:15]	DIP switch value
	[16:31]	16 segment LED
0xc		Generic down counter control
	[0]	Counter run
	[1]	Counter reload
0x10	[0:31]	Counter value
0x14	[0:31]	Counter reload value
0x18		SGDMA Control and Status
	[0:4]	DMA Request (per channel)
	[5:9]	DMA Ack (per channel)
0x1c	[0:31]	DMA Write Counter - The number of clock cycles from the DMA request to the DMA ack of channel 0
0x20	[0:31]	DMA Read Counter - The number of clock cycles from the DMA request to the DMA ack of channel 1
0x24	[0:15]	Root complex Non-Posted Buffer size
	[16:31]	Root complex Posted Buffer size
0x28	[0:31]	EBR Filter value used for triangle manipulation
0x2c		Not used
0x30	[0]	ColorBar reset
0x100	[0:31]	Interrupt Controller ID
0x104	[0]	Current status of interrupt
	[1]	Test mode
	[2]	Interrupt enable to pass interrupt onto the PCI Express IP core
	[3:7]	Not used
	[8:15]	Interrupt Test value 0
	[16:23]	Interrupt Test value 1
	[24:31]	Not used
0x108	[0:15]	If in normal mode: [0:4] dma_ack [5] down counter = 0 If in test mode: [0:7] Test value 0 [8:15] Test value 1
0x10c	[0:31]	Interrupt mask for all sources

64K EBR – wbs_64kebr.v

The 64K EBR is used to store data on the wishbone bus. The wishbone slave is 64-bit wide and supports burst operations on the bus.

ColorBar – wbs_colorbar.v

This wishbone slave is used to generate the color bar display pattern used in the demo. Each time the colorbar is read the pixel information is incremented ultimately producing the colorbar pattern.

System Bus – sysbus.v (LatticeSC Only)

The system bus is an embedded block of the LatticeSC that provides access to the memory map for the PCS/SERDES block. This is required in the LatticeSC implementation of the PCI Express for multi lane links.

uML Controller – uml.v (LatticeSC Only)

The uML controls the PCS Multi Channel aligner registers of the LatticeSC PCS to account for the lane width determined during LTSSM training. This module receives information from the PCI Express IP core and then writes registers in the PCS to control the multi channel aligner. More information on the uML can be found in the PCI Express IP core user's guide.

LED Status – led_status.v

This module provides the control of the LEDs on the demo board for the LTSSM states.

PLL – pll.v

LatticeECP2M

In the LatticeECP2M design the FPGA PLL is used to create the wishbone clock domain (105 MHz).

LatticeSC

In the LatticeSC design the FPGA PLL is used to create two clock domains. The LatticeSC SERDES does not support a 25x multiplier, so the 100 MHz PCI Express clock is converted into a 250 MHz refclk for the SERDES. The SERDES then uses a 10x multiplier to create the 2.5 GHz clock for the PCI Express link. The PLL is also used to create the wishbone clock domain (105 MHz).

Building the Design in Project Navigator

This section describes how to open an existing project or create a new project and build a bitstream for the demo design.

Opening an Existing Project

Project Navigator uses .syn project files. Simply open Project Navigator and open the .syn file delivered with the demo design. To build the project, double-click on the **Generate Bitstream Data** process on right-side pane.

Note: This project enables the use of the IP Hardware Timer. If the user does not have a license for any of the IP used in this design, the IP Hardware Timer will hold the FPGA in global reset after approximately four hours of operation. Once a valid license is installed and the project rebuilt, the Hardware Timer will no longer be used.

Creating a New Project for the LatticeECP2M

To create a new project the user will need to select a device, import all of the HDL files, create the .lpf file, set the search paths, and copy all of the autoconfig files to the project directory. Below is a list of steps that need to be completed in creating a new project.

1. Open Project Navigator
 2. Select **File->New Project**
 3. Provide a project name and a location
 4. Select the design entry type **Verilog HDL** and the synthesis tool **Synplify**
 5. Select **Next**
 6. On the **Select Device** page select the **LatticeECP2M family, LFE2M50E device, -6 speed grade**, and a **FPBGA672 package**.
 7. Select **Next**
-

8. Add all of the source files described in the previous section.
9. Additionally, add the `pmi_def.v` file. This design uses PMI modules and this file provides the module definition for these modules.
10. Select **Next** and **Finish**.
11. Once inside Project Navigator, select **top_sgdma.v** as the top-level design.
12. The Build Database process needs to be able to find all of the IP cores used in this design. Under **Build Properties** go to **macro search path**. The following directories should be added to the search path.

```
ispLeverGenCore\ecp2m\pcie[x4,x4d1]
ispLeverGenCore\ecp2m\sgdma
```

Note: In the macro search path the string delimiter is a “-p”. The string that needs to be entered in the text box is:

```
..\..\ispLeverGenCore\ecp2m\pcie[x4,x4d1] -p..\..\ispLeverGenCore\ecp2m\sgdma
```

13. Next, the autoconfig text files need to be copied to the project directory. The autoconfig text files contain lines to program the hard macros in the design. The file **pcs_pipe_8b_X4.txt** file should be copied into the project directory. If the design is targeting a x1 link (using a PCI Express x4 downgraded x1 core) then the unused channels will need to be disabled by editing the `pcs_pipe_8b_x4.txt` file. See the Lattice *PCI Express User's Guide* for more information on modifying the file.
14. Next, the .lpf file needs to be modified. First add the following SYSCONFIG preference.

```
SYSCONFIG COMPRESS_CONFIG=ON MCCLK_FREQ=26 ;
```

This preference will compress the bitstream as well set the master CCLK frequency to nominally 26MHz.

15. Next, the location of the PCS/SERDES needs to be set.

```
LOCATE COMP "pcie/u1_pcs_pipe/pcs_top_0/pcs_inst_0" SITE "URPCS" ;
```

This preference locates the PCS/SERDES to the upper right PCS location.

16. Next, the frequency needs to be set for each clock in the design.

Internal PCI Express core clocks:

```
FREQUENCY NET "pcie/u1_pcs_pipe/ff_rx_fclk_0" 250.000000 MHz ;
FREQUENCY NET "pcie/u1_pcs_pipe/ff_rx_fclk_1" 250.000000 MHz ;
FREQUENCY NET "pcie/u1_pcs_pipe/ff_rx_fclk_2" 250.000000 MHz ;
FREQUENCY NET "pcie/u1_pcs_pipe/ff_rx_fclk_3" 250.000000 MHz ;
FREQUENCY NET "pcie/pclk" 250.000000 MHz ;
```

User clock:

```
FREQUENCY NET "clk_125" 125.000000 MHz ;
FREQUENCY NET "wb_clk" 105.000000 MHz ;
```

17. Next, add relaxation preferences to relax timing analysis on paths that do not need to be covered.

Internal PCI Express core constraints:

```
BLOCK PATH FROM CELL "*ctc_reset_chx*";
BLOCK NET "pcie/u1_pcs_pipe/sync_rst";
BLOCK NET "pcie/core_rst_n";
```

```

BLOCK NET "pcie/*rxp_status_ln0_2";
MULTICYCLE FROM CELL "*lbk_sloopback*" TO CELL "*cs_reqdet_sm*" 2 X;
MULTICYCLE FROM CELL "*lbk_sloopback*" TO CELL "*cnt_st*" 2 X;
MULTICYCLE FROM CELL "*lbk_sloopback*" TO CELL "*ffc_pcie_det_en*" 2 X;
MULTICYCLE FROM CELL "*lbk_sloopback*" TO CELL "*det_result*" 2 X;
MULTICYCLE FROM CELL "*nfts_rx_skp_cnt*" TO CELL "*cnt_done_nfts_rx*" 2 X;
MULTICYCLE FROM CELL "*nfts_rx_skp_cnt*" TO CELL "*ltssm_nfts_rx_skp*" 2 X;

```

User constraint:

```

BLOCK PATH FROM PORT "rstn"

```

18. Next, add bank preferences to set the voltage for each I/O bank.

```

BANK 0 VCCIO 3.3 V;
BANK 1 VCCIO 2.5 V;
BANK 2 VCCIO 2.5 V;
BANK 3 VCCIO 2.5 V;
BANK 4 VCCIO 3.3 V;
BANK 5 VCCIO 1.8 V;
BANK 6 VCCIO 3.3 V;
BANK 7 VCCIO 2.5 V;

```

19. Next, add pin locations and types for all of the top-level ports in the design.

```

IOBUF PORT "pll_lk" SITE=W2 IO_TYPE=LVCMS33 ;
IOBUF PORT "poll" SITE=U5 IO_TYPE=LVCMS33 ;
IOBUF PORT "l0" SITE=U2 IO_TYPE=LVCMS33 ;
IOBUF PORT "dl_up" SITE=U3 IO_TYPE=LVCMS33 ;
IOBUF PORT "usr0" SITE=U6 IO_TYPE=LVCMS33 ;
IOBUF PORT "usr1" SITE=V2 IO_TYPE=LVCMS33 ;
IOBUF PORT "usr2" SITE=V1 IO_TYPE=LVCMS33 ;
IOBUF PORT "usr3" SITE=U4 IO_TYPE=LVCMS33 ;

IOBUF PORT "na_pll_lk" SITE=V7 IO_TYPE=LVCMS33 ;
IOBUF PORT "na_poll" SITE=U8 IO_TYPE=LVCMS33 ;
IOBUF PORT "na_l0" SITE=Y3 IO_TYPE=LVCMS33 ;
IOBUF PORT "na_dl_up" SITE=W5 IO_TYPE=LVCMS33 ;
IOBUF PORT "na_usr0" SITE=W6 IO_TYPE=LVCMS33 ;
IOBUF PORT "na_usr1" SITE=Y5 IO_TYPE=LVCMS33 ;
IOBUF PORT "na_usr2" SITE=AA3 IO_TYPE=LVCMS33 ;
IOBUF PORT "na_usr3" SITE=Y4 IO_TYPE=LVCMS33 ;

IOBUF PORT "led_out_15" SITE=A6 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_14" SITE=D8 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_13" SITE=C8 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_12" SITE=D6 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_11" SITE=D9 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_10" SITE=B8 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_9" SITE=G13 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_8" SITE=F12 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_7" SITE=B9 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_6" SITE=C10 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_5" SITE=H14 IO_TYPE=LVCMS33 ;
IOBUF PORT "led_out_4" SITE=F13 IO_TYPE=LVCMS33 ;

```

```

IOBUF PORT "led_out_3" SITE=G14 IO_TYPE=LVC MOS33 ;
IOBUF PORT "led_out_2" SITE=B10 IO_TYPE=LVC MOS33 ;
IOBUF PORT "led_out_1" SITE=H15 IO_TYPE=LVC MOS33 ;
IOBUF PORT "led_out_0" SITE=H16 IO_TYPE=LVC MOS33 ;
IOBUF PORT "dp" SITE=A4 IO_TYPE=LVC MOS33 ;
BLOCK NET "led_out*" ;
BLOCK NET "dp*" ;
IOBUF PORT "dip_switch_0" SITE=T3 IO_TYPE=LVC MOS33 ;
IOBUF PORT "dip_switch_1" SITE=T4 IO_TYPE=LVC MOS33 ;
IOBUF PORT "dip_switch_2" SITE=P8 IO_TYPE=LVC MOS33 ;
IOBUF PORT "dip_switch_3" SITE=R6 IO_TYPE=LVC MOS33 ;
IOBUF PORT "dip_switch_4" SITE=T1 IO_TYPE=LVC MOS33 ;
IOBUF PORT "dip_switch_5" SITE=U1 IO_TYPE=LVC MOS33 ;
IOBUF PORT "dip_switch_6" SITE=R7 IO_TYPE=LVC MOS33 ;
IOBUF PORT "dip_switch_7" SITE=T5 IO_TYPE=LVC MOS33 ;

```

20. Under Place and Route properties set the number of Placement Iterations to 10. This will allow place and route to try 10 different placements while attempting to satisfy all of the timing constraints.

21. The design is now ready to be built. Double-click on the **Generate Bitstream Data** process to create the bitstream.

Note: This project enables the use of the IP Hardware Timer. If the user does not have a license for any of the IP used in this design, the IP Hardware Timer will hold the FPGA in global reset after approximately four hours of operation. Once a valid license is installed and the project rebuilt, the Hardware Timer will no longer be used.

Conclusion

This user's guide provides a description for the PCI Express SGDMA Demo design. With the help of this guide, a user can rebuild the bitstream used for the demo and begin to modify the design to achieve their design goals.

Technical Support Assistance

Hotline: 1-800-LATTICE (North America)
+1-503-268-8001 (Outside North America)
e-mail: techsupport@latticesemi.com
Internet: www.latticesemi.com

Revision History

Date	Version	Change Summary
January 2008	01.0	Initial release.
July 2008	01.1	Document title change from "Lattice PCI Express x4 Scatter-Gather DMA Demo Verilog Source Code User's Guide" to "Lattice PCI Express Scatter-Gather DMA Demo Verilog Source Code User's Guide."
		Updates to support PCI Express IP core version 3.3.
		Updates to support solution kit directory structure.
		Updates to support wishbone clock domain change.