PCIe
Verification Component

User Guide for Release ????.??

By

Simon Southwell

simon.southwell@gmail.com

Table of Contents

1 Terminology : Upstream and Downstream

OSVVM PCIe VC and this document use the PCIe terminology, Upstream device and Downstream device. A PCIe Upstream device is a PCIe component whose link is 'upstream', nearer the root complex (e.g. the RC itself), whilst a PCIe downstream device is a PCIe component whose link is downstream, further away from the root complex (e.g. an endpoint). Further, a downlink is the TX output link from an upstream device (and thus an RX link for the downstream device), and an upstream link is the TX link from a downstream device (and an RX link of an upstream device).

2 Overview

The OSVVM PCIe Verification Component(VC) facilitates the interface and functionality of PCIe GEN1 and GEN2 devices [1] and is intended to for part of a structured test environment. It can support up to 16 lanes in a link, though configurable for 1, 2, 4 and 8 lane links as well.

The PCIe VC is a co-simulation component based on the *pcievhost* C/C++ model [2] and is integrated with the existing OSVVM co-simulation environment [3]. Like some other OSVVM VCs [4] the PCIe VC is a transaction based VC and uses the OSVVM Address Bus Model Independent Transaction interface [5]. The PCIe VC can act as an Upstream device interface or a Downstream device interface, depending on configuration. The PCIe VC can instigate transactions from a test sequencer program and receive arbitrary transactions and has internal memory models for both main memory and a configuration space, but these can be disabled and the received request transactions passed to a separate test sequencer to program arbitrary sequences of responses. The model supports all the types of PCIe TLPs:

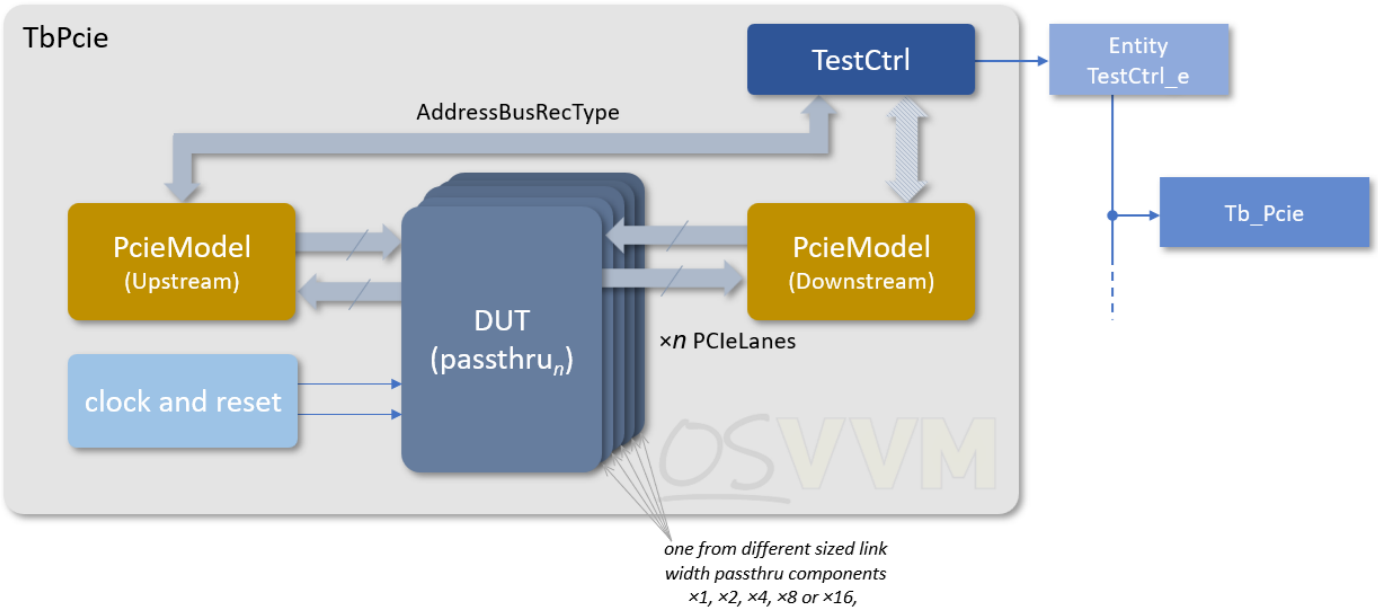
- **MRd** Memory read request (32 and64 bit address)
- **MRdLk** Memory read request-locked (32 and 64 bit address)
- **MWr** Memory write request (32 and 64 bit address)
- **IORd** I/O read request
- **IOWr** I/O write request
- **CfgRd0** Configuration read type 0
- **CfgWr0** Configuration write type 0
- **CfgRd1** Configuration read type 1
- **CfgWr1** Configuration write type 1
- **Msg** Message request
- **MsgD** Message request with data (
- **Cpl** Completion with no data. Used for all reads with error status and for I/O and configuration writes.
- **CplD** Completion with data. Used for all reads with good status.
- **CplLk** Completion for locked memory read with error status
- **CplDlk** Completion for locked memory read with good status.

The PCIe VC has many automatic features. The internal memories, already mentioned, are the default targets for memory and configurations space transactions, though these can be disabled. In addition, by default, all DLL packets for flow control, PHY Ordered Sets and PHY Training Sequences are automatically handled by the model. However, these can be bypassed in configuration and their pattern types

generated from a sequencer program and received in another sequencer program. The model provides facilities to do basic link initialisation, with the DLL initialisation built into the model for VC0, and a partial LTSSM compiled into the library, but separate from the model. This implements an LTSSM to go from electrical idle to the PHY Link Up state of L0 but does not yet implement the other states. It is extensible and the C source code for this is provided with the model.

3 OSVVM Testbench Architecture

3.1 Test Architecture Overview



3.2 The PcieModel VC HDL component

The `PcieModel` component is has a clock and reset signal inputs (`clk` and `nreset`), where the clock frequency needs to be set to the base symbol rate. E.g. PCIe GEN1 has a 2.5Gb/s rate, so for 10 bit symbols, this is 250MSym/s and a clock rate of 250 MHz. The active low reset signal must be asserted for a minimum of 1 cycle. The test control interface is via the `TransRec` port of type `AddressBusRecType` [5], and there are a pair of PCIe links, in output and input of type `LinkType` which carry the 10-bit encoded data or, if configured, the 9-bit pre-encoded PIPE data.

The entity declaration for the PcieModel VC is defined as shown below:

```
entity PcieModel is
generic (
  MODEL_ID_NAME      : string  := "" ;
  NODE_NUM           : integer := 8 ;
  ENDPOINT           : boolean := false ;
```

```
    REQ_ID            : integer := 0 ;
    EN_TLP_REQ_DIGEST : boolean := false ;
    PIPE              : boolean := false ;
    ENABLE_INIT_PHY   : boolean := true
  ) ;
  port (
    -- Globals
    Clk         : in   std_logic ;
    nReset      : in   std_logic ;

    -- Testbench Transaction Interface
    TransRec    : inout AddressBusRecType ;

    -- PCIe port Functional Interface
    PcieLinkOut : out  LinkType ;
    PcieLinkIn  : in   LinkType
  ) ;
```

3.2.1 Generics

The model has some generics to configure the model. The `MODEL_ID_NAME` can give an identify name for reports and assertions or can be left at the default, where is hierarchical instantiation name will be used.

As the `PcieModel` VC is a co-simulation component it requires a node number, configured through the `NODE_NUM` generic. The model can exist within a test environment with other co-simulation components [3], including other `PcieModel` components, but each must have a unique node number in order to run the particular software for that node—this is the *pcievhost* C model in the case of `PcieModel`.

Unlike some other common other bus/interconnect protocols, a PCIe “subordinate”, knowns as an Endpoint, can initiate read and write type transactions, but not all possible transactions. It also has an associated Type 0 configurations space. The `ENDPOINT` generic configures the model to behave like and endpoint (though it is the same code inside. It enables an internal configuration space model (though this can be disabled through configuration) and has some effect on the link display features.

The `REQ_ID` generic sets the device’s requester ID that is added to each transaction request to identify the source. The default value is set to 0, but a recommended setting might be the device’s `NODE_NUM` value.

The `EN_TLP_REQ_DIGEST` generic enables the adding of an ECRC end-to-end cyclic redundancy check to each generated TLP. The inclusion of an ECRC is usually a function of whether extended PCIe capabilities are in the configurations space and whether enabled there. For testing of correct handling of these scenarios, this generic is included. If enabled, any auto-generated completion TLPs will include an ECRC in response to a TLP request that also has an ECRC.

By default, the VC will generate a 10-bit stream of scrambled and 8b10b encoded data. The PIPE generic, when set to true, bypasses the scrambling and 8b10 encoding (and the corresponding received data decoding) to produce a stream of 9-bit pre-encoded symbols, with 8 bits of data (bits 7 down to 0) and a control symbol indicator (bit 8), matching a PIPE interface's data paths. Note that a PIPE's data path ports can also be wider than a symbol, with 16, 32 and 64 bit widths supported with additional control bits for each of the extra bytes. This would require some wrapper logic to widen and down-clock the PcieModel output for each lane.

The ENABLE_INIT_PHY generic is used to indicate to the running software *if* configured to automatically generate initialisation sequences, to run the physical layer link training or not. Note that the link training is provided by some demonstration code ltssm.c in the PCIe/ltssm directory and is not part of the model proper, but uses the provide API to generate training sequence ordered sets. The provide code is limited to taking a link from an idle state all the way through the "link up" L0 state. Other low power states, or ancillary states are not supported, including the recovery state which is used when switching from a lower generation to a higher generation speed. PcieModel is capable of generating all the possible patterns required to fully implement the LTSSM and so test code can place a DUT into any state, but the provided ltssm.c code has limited functionality at this time. It has hooks for all the functionality and can be extended, which is why the source code is made available.

3.3 Address Bus Transaction Interface

The PcieModel Verification Component receives transactions from the test sequencer via a Transaction Interface. OSVVM implements the transaction interface as a record. The AddressBusRecType is used to connect the verification component to TestCtrl. AddressBusRecType, shown below, is defined in the Address Bus Model Independent Transaction package, AddressBusTransactionPkg.vhd, which is in the directory OsvvmLibraries/Common/src.

```
type AddressBusRecType is record
  -- Handshaking controls
  -- Used by RequestTransaction in the Transaction Procedures
  -- Used by WaitForTransaction in the Verification Component
  -- RequestTransaction and WaitForTransaction are in osvvm.TbUtilPkg
  Rdy : RdyType ;
  Ack : AckType ;
  -- Transaction Type
  Operation : AddressBusOperationType ;
  -- Address to verification component and its width
  -- Width may be smaller than Address
  Address : std_logic_vector_max_c ;
  AddrWidth : integer_max ;
  -- Data to and from the verification component and its width.
  -- Width will be smaller than Data for byte operations
  -- Width size requirements are enforced in the verification component
```

```
DataToModel : std_logic_vector_max_c ;
DataFromModel : std_logic_vector_max_c ;
DataWidth : integer_max ;
-- Burst FIFOs
WriteBurstFifo : ScoreboardIdType ;
ReadBurstFifo : ScoreboardIdType ;
-- StatusMsgOn provides transaction messaging override.
-- When true, print transaction messaging independent of
-- other verification based based controls.
 StatusMsgOn : boolean_max ;
 -- Verification Component Options Parameters - used by SetModelOptions
 IntToModel : integer_max ;
 IntFromModel : integer_max ;
 BoolToModel : boolean_max ;
 BoolFromModel : boolean_max ;
 -- Verification Component Options Type
 Options : integer_max ;
 end record AddressBusRecType ;
```

Note that Address, `DataToModel`, and `DataFromModel` are unconstrained. Hence, when they are used in a signal declaration they must be constrained. Address needs to be sized to match the maximum (`PCIE_ADDR_WIDTH`). `DataToModel` and `DataFromModel` need to be sized to match the maximum (`PCIE_DATA_WIDTH`).

The code below shows the declaration `UpstreamRec` (which connects a `PcieModel` RC to `TestCtrl`) and `DownstreamRec` (which connects the a `PcieModel` endpoint to `TestCtrl`).

```
signal UpstreamRec, DownstreamRec  : AddressBusRecType(
        Address      (PCIE_ADDR_WIDTH-1 downto 0),
        DataToModel  (PCIE_DATA_WIDTH-1 downto 0),
        DataFromModel(PCIE_DATA_WIDTH-1 downto 0)
      ) ;
```

3.4 PCIe Interface

3.4.1 PcieRecType

The PCIe interface is a record with `LinkOut` and `LinkIn` fields, each of which is of `LinkType`—where `LinkType` is an unbound array of unconstrained `std_logic_vector`. The definitions for these are shown below.

```
type LinkType is array (natural range <>) of std_logic_vector ;

type PcieRecType is record
  LinkOut       : LinkType ;
  LinkIn        : LinkType ;
end record PcieRecType;
```

The PcieModel PCIe interface supports up to 16 lanes, as defined by MAX_PCIE_LINK_WIDTH, with encoded symbol lane data width. The link width can be set for 1, 2, 4, 8 or 16 lanes.

The code below shows the declaration for PcieDnLink (which connects a PcieModel RC to a DUT) and PcieUpLink (which connects a PcieModel endpoint to a DUT).

```
-- PCIe Functional Interface
signal   PcieDnLink, PcieUpLink : PcieRecType (
  LinkOut (0 to MAX_PCIE_LINK_WIDTH-1)(PCIE_LANE_WIDTH-1 downto 0),
  LinkIn  (0 to MAX_PCIE_LINK_WIDTH-1)(PCIE_LANE_WIDTH-1 downto 0)
) ;
```

3.5 Connecting the HDL to the Co-simulation Model

Each instantiation of a PcieModel must be bound to the underlying PCIe C model that will run using the OSVVM co-simulation features. This requires a simple program to be provided and placed in a co-simulation test directory. The code will look like the following:

```
#include "pcieVcInterface.h"

extern "C" void VUserMain62 (int node)
{
    pcieVcInterface *vc = new pcieVcInterface(node);

    // Should not return
    vc->run();
}
```

This example code if for a PcieModel instantiated with NODE_NUM as 62, thus needs an entry point equivalent to "main" as VUserMain62 which must have C linkage. In this code the pcieVcInterface.h header file is included and a pointer to the model created in the main function. The model is then run using the run() method. For each instantiated model an equivalent VUserMain<n> function must be created. These can all reside in the same source file.

To compile the model code the test script should look something like the following:

```
library    osvvm_TbPcie
ChangeWorkingDirectory ../tests

MkVproc    vc
TestName   CoSim_pcie
simulate Tb_PCIe
```

This script is adapted from the `PCIe/testbench/vc.pro` file. The source code files are in a `PCIe/tests/vc` directory, and they are all compiled with the `MkVproc vc` command.

The `simulate Tb_PCIe [CoSim]` command then runs the model.

4 Configuration Options for the PcieModel VC

4.1 SetModelOptions and Configuring the Model

The `PcieModel` is highly configurable with the VHDL component having several generic settings. In addition to these fixed parameters, the test code can configure the model via the `SetModelOptions` procedure. The options can have an argument value, or do not require an argument, and the `SetModelOptions` value parameter can be set to `NULLOPTVALUE` for these cases. E.g.

```
SetModelOptions(TransactionRec, CONFIG_ENABLE_FC, NULLOPTVALUE);
```

The type of options for the `PcieModel` available fall into two categories:

- Those that are sent to the underlying *pcievhost* C model
- Those that configure the VC, e.g. for signalling

For the first category details can be found in the *pcievhost* documentation [2], but the table below summarises the available options

TYPE	VALUE?	UNITS	Description
Configuration setting values for `pcievhost model`			
CONFIG_FC_HDR_RATE	yes	cycles	Rx Header consumption rate (default 4)
CONFIG_FC_DATA_RATE	yes	cycles	Rx Data consumption rate (default 4)
CONFIG_ENABLE_FC	no		Enable auto flow control (default)
CONFIG_DISABLE_FC	no		Disable auto flow control
CONFIG_ENABLE_ACK	yes	cycles	Enable auto acknowledges with processing rate (default rate 1)
CONFIG_DISABLE_ACK	no		Disable auto acknowledges
CONFIG_ENABLE_MEM	no		Enable internal memory (default)
CONFIG_DISABLE_MEM	no		Disable internal memory
CONFIG_ENABLE_SKIPS	yes	cycles	Enable regular Skip ordered sets, with interval (default interval 1180)

CONFIG_DISABLE_SKIPS	no		Disable regular Skip ordered sets
CONFIG_DISABLE_SCRAMBLING	no		Disable data scrambling
CONFIG_ENABLE_SCRAMBLING	no		Enable data scrambling (default)
CONFIG_DISABLE_8B10B	no		Disable 8b10b encoding and decoding
CONFIG_ENABLE_8B10B	no		Enable 8b10b encoding and decoding (default)
CONFIG_DISABLE_ECRC_CMPL	no		Disable ECRC auto-generation on completions for requests with ECRCs
CONFIG_ENABLE_ECRC_CMPL	no		Enable ECRC auto-generation on completions for requests with ECRCs (default)
CONFIG_ENABLE_CRC_CHK	no		Enable CRC checks (default)
CONFIG_DISABLE_CRC_CHK	no		Disable CRC checks
CONFIG_ENABLE_UR_CPL	no		Enable auto unsupported request completions (default)
CONFIG_DISABLE_UR_CPL	no		Disable auto unsupported request completions
CONFIG_ENABLE_INTERNAL_MEM	no		Enable internal memory (default)
CONFIG_DISABLE_INTERNAL_MEM	no		Disable internal memory (packets passed to user callback if registered)
CONFIG_ENABLE_DISPLINK_COLOUR	no		Enable colour formatting of link display output (default)
CONFIG_DISABLE_DISPLINK_COLOUR	no		Disable colour formatting of link display output
CONFIG_BCK_NODE_NUM	yes	node#	Set the displink display node number for back completing node (default this node# ^ 1)
CONFIG_POST_HDR_CR†	yes	credits	Initial advertised posted header credits (default 32)
CONFIG_POST_DATA_CR†	yes	credits	Initial advertised posted data credits (default 1K)
CONFIG_NONPOST_HDR_CR†	yes	credits	Initial advertised non-posted header credits (default 32)
CONFIG_NONPOST_DATA_CR†	yes	credits	Initial advertised non-posted data credits (default 1)
CONFIG_CPL_HDR_CR†	yes	credits	Initial advertised completion header credits (default ∞)
CONFIG_CPL_DATA_CR†	yes	credits	Initial advertised non-posted data credits (default ∞)
CONFIG_CPL_DELAY_RATE†	yes	cycles	Auto completion delay rate (default 0)
CONFIG_CPL_DELAY_SPREAD†	yes	cycles	Auto completion delay randomised spread (default 0)
Configuration setting values for ltssm.c if used			
CONFIG_LTSSM_LINKNUM††	yes	integer	Training sequence advertised link number (default 0)
CONFIG_LTSSM_N_FTS††	yes	integer	Training sequence number of fast training sequences (default 255)
CONFIG_LTSSM_TS_CTL††	yes	integer	Five bit TS control field (default 0)
CONFIG_LTSSM_DETECT_QUIET_TO††	yes	cycles	Detect quite timeout (default 1500/6M, depending if LTSSM_ABBREVIATED defined or not)
CONFIG_LTSSM_POLL_ACTIVE_TO_COUNT††	yes	cycles	Polling active TX count (default 16/1024, depending if LTSSM_ABBREVIATED defined or not)
CONFIG_LTSSM_DISABLE_DISP_STATE††	yes	integer	Disable display of link state (default 0 = false)

† Call before generating transactions to take effect from time 0
†† Call before calling initialising link to take effect in training sequences.

The second group of settings configure the VC itself, independent of the *pcievhost* model. These are used for things such as setting values to be used in transaction fields or some simulation control.

4.1.1 Setting the Transaction Mode

Unlike some other bus or interconnect interfaces PCIe can generate a range of different types of transactions though, fundamentally, these still send a transaction over the link and either get a response (like a read), or don't (like a write). However, to generate the desired transaction type, the model must first be placed in a mode to generate that type. This is done with the SETTRANSMODE setting. There are six valid values for this configuration:

value	Description
MEM_TRANS	This mode is for sending memory read requests or memory write requests. These can have variable sized read data requests or variable sized write data payloads. Reads receive a completion (non-posted), whilst writes do not (posted).
IO_TRANS	This model is for I/O read and write transactions. These transactions always receive a completion, including writes (non-posted), and are limited to 32-bit data.
CFG_SPC_TRANS	This mode is for sending configuration read and write transactions. These transactions always receive a completion, including writes (non-posted), and are limited to 32-bit data.
MSG_TRANS	This mode is for sending messages. These transactions do not receive a completion (posted).
CPL_TRANS	This mode is for sending a full completion to a non-posted request. The transaction may or may not have associated return data.
PART_CPL_TRANS	This mode is for sending a part completion to a non-posted request. The transaction must have data which is only a part of the requested amount.

An example configuration setting might be:

```
SetModelOptions(UpstreamRec, SETTRANSMODE, IO_TRANS) ;
```

4.1.2 Setting Transaction Fields

option	Valid transactions	Description
SETRDLCK	Memory reads, completions, part completions	Generates the RdLck variants of memory reads and completions (either full or part).
SETCMPLRID	Completions, part completions	Sets the requester ID for a [part] completion, as sent in the request that is being responded to.
SETCMPLCID	Completions, part completions	Sets the completer ID for a [part] completion. This is set for a device on configuration write transactions and consists of a bus (8 bits), device (5 bits) and function number (3 bits), take make a 16 -bit CID value.
SETCMPLRLEN	Part completions	Sets the remaining length field of a part completion, where this indicates the amount of data still to be completed, including the current completion.
SETCMPLTAG	Completions, part completions	Sets the tag value for the completion. This is an incrementing number for each transaction sent from the device.
SETREQTAG	All but completions, or part completions	Set the requester tag for a request transaction. This is an incrementing number for each transaction sent from the device. The model can automatically generate a tag if the value is set to TLP_TAG_AUTO. It will then keep incrementing from the last set value.

4.1.3 Setting the Configuration Space

If a model is configured as an endpoint (via the ENDPOINT generic) and has its internal memories enabled (model option CONFIG_ENABLE_MEM) the value in the configuration space can be set with the following options:

option	Description
SETCFGSPCOFFSET	Set the offset into the configuration space that will be updated by a SETCFGSPC or SETCFGSPCMASK
SETCFGSPC	Set the register indexed by last SETCFGSPCOFFSET call to value
SETCFGSPCMASK	Set the register mask indexed by last SETCFGSPCOFFSET call to value

The SETCFGSPCOFFSET option is set first to select the offset into the configuration space that a word is to be written. To update the word at the pre-programmed offset the SETCFGSPC option is used. If any of the bits in the word are read-only, then the SETCFGSPCMASK can be used to mask those bits, with a 1 indicating the read-only bits.

4.1.4 Initialising the Link

A PCIe interface needs initialising if starting from a powered down or reset state. The PHY layer requires link training and once it is in the "link up" (L0) state, then the data link layer (DLL) needs to initialise flow control for virtual channel zero (VC0) before any data can be transferred over the link. The PcieModel can operate without these initialisation steps if a target DUT has test modes that allow it to do so as well, saving in simulation time. In order to do so the model must use the CONFIG_DISABLE_FC model configuration setting. To optionally go through the DLL and/or the PHY initialisation, the following configurations settings can be used:

option	Description
INITPHY	Initiate link training from electrical idle to L0
INITDLL	Initiate DLL flow control initialisation from VC0

4.2 GetModelOptions and Transaction Status

All received completions return a status and the tag number of the received request which the completion is associated with. When processing a completion these two values can be accessed with GetModelOptions. The table below shows the option to use for each of these.

option	Description
GETLASTCMPLSTATUS	Get the completion status for the last received completion packet. This can be one of CPL_SUCESS, CPL_UNSUPPORTED, CPL_CRS or CPL_ABORT.
GETLASTRXREQTAG	Get the returned tag value for the last received completion.

5 Generating Transactions

The options described in the previous section are used in association with the standard Read, ReadAddress, ReadData, ReadBurst, Write, WriteAddressAsync and WriteBurst procedures. If

sending data whose width is greater than set for the `DataToModel` width of the `AddressBusRecType` signal, then the burst procedures are used and the `WriteBurstFifo` and `ReadBurstFifo` are used.

5.1 Write Transactions

5.1.1 Memory word writes

For memory writes, the transaction mode needs to be set for memory transactions, and a tag value supplied or auto-tagging used. The example shows a 16-bit word write.

```
SetModelOptions(TransactionRec, SETTRANSMODE, MEM_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_TAG_AUTO) ;
Write(TransactionRec, X"00000106", X"CAFE") ;
```

5.1.2 Memory Burst Writes

For burst writes, data bytes are pushed onto the write fifo, then set up is the same as for word writes. `WriteBurst` is than called with an address and a byte count. The example below shows a memory write transfer of 27 bytes to a given address.

```
for i in 0 to 26 loop
  Push(TransactionRec.WriteBurstFifo, to_slv(i, 8)) ;
end loop ;
SetModelOptions(TransactionRec, SETTRANSMODE, MEM_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_TAG_AUTO) ;
WriteBurst(TransactionRec, X"00000210", 27) ;
```

5.1.3 Configuration Space Writes

For configuration writes, the transaction mode is set for configurations and the tag set. A normal write is then performed, but the “address” argument is in two parts. The lower 16 bits define the register index into the configuration space as a byte offset aligned to 32 bits, whilst the upper 16-bits constitute the bus, device and function number that the device must use as its CID for subsequent completions.

As all configuration accesses get a returned completion, the status is fetched after the write transaction returns. E.g.

```
SetModelOptions(TransactionRec, SETTRANSMODE, CFG_SPC_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_TAG_AUTO) ;
Write(TransactionRec, X"0200_0010", X"00010000") ;
GetModelOptions(TransactionRec, GETLASTCMPLSTATUS, Status) ;
```

As configurations accesses are all 32-bit, there are no burst transactions.

5.1.4 I/O Writes

I/O write are similar to configuration writes, in that they are limited to 32 bit accesses and get a completion. The mode is set of I/O and a tag or auto-tag provided. The address is a true address value. E.g.

```
SetModelOptions(TransactionRec, SETTRANSMODE, IO_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_TAG_AUTO) ;
Write(TransactionRec, X"12345678", X"87654321") ;
GetModelOptions(TransactionRec, GETLASTCMPLSTATUS, Status) ;
```

As I/O accesses are all 32-bit, there are no burst transactions.

5.1.5 Message Writes

Messages are similar to memory writes in that they don't receive a completion. The "address" for messages have a different meaning and some messages also don't have a payload, determined by its type. Here the "address" will be used to specify the message type, and this can be one of the following values:

MSG_ASSERT_INTA	MSG_ASSERT_INTB	MSG_ASSERT_INTC	MSG_ASSERT_INTD
MSG_DEASSERT_INTA	MSG_DEASSERT_INTB	MSG_DEASSERT_INTB	MSG_DEASSERT_INTD
MSG_PM_ACTIVE_STATE_NAK	MSG_PME	MSG_PME_TURN_OFF	MSG_PME_TO_ACK
MSG_ERR_CORR	MSG_ERR_NON_FATAL	MSG_ERR_FATAL	MSG_SET_PWR_LIMIT
MSG_UNLOCK	MSG_VENDOR_0	MSG_VENDOR_1	

The example below is for a message with no data, and `WriteAddressAsync` is used as there is no data payload:

```
SetModelOptions(TransactionRec, SETTRANSMODE, MSG_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_AUTO_TAG) ;
WriteAddressAsync(TransactionRec, MSG_ERR_NON_FATAL) ;
```

For a message requiring data the setup is the same bit the `Write` procedure is used.

```
SetModelOptions(TransactionRec, SETTRANSMODE, MSG_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_TAG_AUTO) ;
Write(TransactionRec, MSG_SET_PWR_LIMIT, X"20251015") ;
```

5.2 Read Transactions

5.2.1 Memory Word Reads

For memory reads, the transaction mode needs to be set for memory transactions, and a tag value supplied or auto-tagging used. The example shows a 16-bit word read and fetching the returned completion status.

```
SetModelOptions(TransactionRec, SETTRANSMODE, MEM_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_TAG_AUTO) ;
Read(TransactionRec, X"00000106", Data_slv) ;
GetModelOptions(TransactionRec, GETLASTCMPLSTATUS, Status_int) ;
```

5.2.2 Memory Burst Reads

For burst reads the set up is the same as for word writes with the addition of setting the read lock bit as well. `ReadBurst` is then called with an address and a byte count. The status of the returned completion can then be fetched for inspection and the data popped from the read fifo. The example below shows a memory read transfer of 27 bytes to a given address.

```
SetModelOptions(TransactionRec, SETTRANSMODE, MEM_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_TAG_AUTO) ;
SetModelOptions(TransactionRec, SETRDLCK,     0) ;
ReadBurst(TransactionRec, X"00000210", 27) ;
GetModelOptions(TransactionRec, GETLASTCMPLSTATUS, Status_int) ;
for i in 0 to 26 loop
  Pop(TransactionRec.ReadBurstFifo, to_slv(i, 8)) ;
end loop ;
```

5.2.3 Configuration Space Reads

For configuration reads, the transaction mode is set for configurations and the tag set. A normal read is then performed, but the “address” argument defines the register index into the configuration space as a byte offset aligned to 32 bits. The status can then be fetched after the read transaction returns and inspected. E.g.

```
SetModelOptions(TransactionRec, SETTRANSMODE, CFG_SPC_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_TAG_AUTO) ;
Read(TransactionRec, X"00000010", Data_slv) ;
GetModelOptions(TransactionRec, GETLASTCMPLSTATUS, Status) ;
```

As configurations accesses are all 32-bit, there are no burst transactions.

5.2.4 I/O Reads

I/O reads are similar to configuration reads, in that they are limited to 32 bit accesses and get a completion. The mode is set of I/O and a tag or auto-tag provided. The address is a true address value. E.g.

```
SetModelOptions(TransactionRec, SETTRANSMODE, IO_TRANS) ;
SetModelOptions(TransactionRec, SETREQTAG,    TLP_TAG_AUTO) ;
Read(TransactionRec, X"12345678", Data_slv) ;
GetModelOptions(TransactionRec, GETLASTCMPLSTATUS, Status) ;
```

As I/O accesses are all 32-bit, there are no burst transactions.

5.3 Full Completion Transactions

5.3.1 Word Completions

Completions require a few more options. As well as setting the transaction type to `CPL_TRANS`, a requester ID and a completer ID must also be set. There is no auto-tag mode as the tag must be set to the received request's tag that the for which completion is the response. A locked completion should also be the response to a locked memory read request. Since a completion is a data push, requiring no response, it is considered a posted write like memory writes, and so the standard `Write` procedure is used. The address for the call to `Write` needs only be the 7 lower address bits of a memory read request, or 0 in completions for other transaction request types. E.g.

```
SetModelOptions(TransactionRec, SETTRANSMODE, CPL_TRANS) ;
SetModelOptions(TransactionRec, SETCMPLRID,   X"3e") ;
SetModelOptions(TransactionRec, SETCMPLCID,   X"0123") ;
SetModelOptions(TransactionRec, SETCMPLTAG,   12) ;
SetModelOptions(TransactionRec, SETRDLCK,     0) ;

Write(TransactionRec, X"63", "0badf00d") ;
```

5.3.2 No Data Payload Completions

If a completion does not require a data payload, then the `WriteAddressAsync` procedure is used in place of Write, though the setup is the same. E.g.

```
SetModelOptions(TransactionRec, SETTRANSMODE, CPL_TRANS) ;
SetModelOptions(TransactionRec, SETCMPLRID,   X"3e") ;
SetModelOptions(TransactionRec, SETCMPLCID,   X"0123") ;
SetModelOptions(TransactionRec, SETCMPLTAG,   12) ;
SetModelOptions(TransactionRec, SETRDLCK,     0) ;
```

```
WriteAddressAsync(TransactionRec, X"63") ;
```

5.3.3 Burst Completions

.As well as setting the transaction type to CPL_TRANS, a requester ID and a completer ID must also be set. There is no auto-tag mode as the tag must be set to the received request's tag that the for which completion is the response. A locked completion should also be the response to a locked memory read request. Since a completion is a data push, requiring no response, it is considered a posted write like memory writes, and so the standard WriteBurst procedure is used. The address for the call to WriteBurst needs only be the 7 lower address bits of a memory read request. Completions for other types of transaction require either no data or a single word payload, and so this burst method is limited to use in completing for read requests only. E.g.

```
for i in 0 to 15 loop
  Push(TransactionRec.WriteBurstFifo, to_slv(i, 8)) ;
end loop ;

SetModelOptions(TransactionRec, SETTRANSMODE, CPL_TRANS) ;
SetModelOptions(TransactionRec, SETCMPLRID,   X"3e") ;
SetModelOptions(TransactionRec, SETCMPLCID,   X"0123") ;
SetModelOptions(TransactionRec, SETCMPLTAG,   12) ;
SetModelOptions(TransactionRec, SETRDLCK,     0) ;

WriteBurst(TransactionRec, X"00010080", 16) ;
```

5.4 Part Completions

Part completions are only associated with returning a partial set of data to a burst read request, and so only needs a burst method to instigate. The set up is then same as for a burst completion but, in addition, must set up the remaining length value. E.g.

```
for i in 0 to 15 loop
  Push(TransactionRec.WriteBurstFifo, to_slv(i, 8)) ;
end loop ;

SetModelOptions(TransactionRec, SETTRANSMODE, CPL_TRANS) ;
SetModelOptions(TransactionRec, SETCMPLRID,   X"3e") ;
SetModelOptions(TransactionRec, SETCMPLCID,   X"0123") ;
SetModelOptions(TransactionRec, SETCMPLRLEN,  X"20") ;
SetModelOptions(TransactionRec, SETCMPLTAG,   12) ;
SetModelOptions(TransactionRec, SETRDLCK,     0) ;

WriteBurst(TransactionRec, X"00010080", 16) ;
```

5.5 The PCIe specific Wrapper Procedures

The PcieInterfacePkg provides a set of PCIe specific wrapper procedures to abstract away the details of the setting and getting of options as described in the last subsections, and to present a more coherent set of arguments. Below are shown the available procedures.

5.5.1 Memory Transactions

```
------------------------------------------------------------
procedure PcieMemWrite (
-- do PCIe Memory Write Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iData          : In    std_logic_vector ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemWrite (
-- do PCIe Burst Write Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iByteCount     : In    integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemRead (
-- do PCIe Memory Read Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           oData          : Out   std_logic_vector ;
           oStatus        : Out   integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;
```

```
------------------------------------------------------------
procedure PcieMemRead (
-- do PCIe Read Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iByteCount     : In    integer ;
           oStatus        : Out   integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemReadLock (
-- do PCIe Memory Read Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           oData          : Out   std_logic_vector ;
           oStatus        : Out   integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemReadLock (
-- do PCIe Read Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iByteCount     : In    integer ;
           oStatus        : Out   integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemReadAddress (
-- do PCIe Read Address Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iByteCount     : In    integer ;
           iLock          : In    boolean := false ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;
```

```
------------------------------------------------------------
procedure PcieMemReadAddress (
-- do PCIe Read Address Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iByteCount     : In    integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemReadLockAddress (
-- do PCIe Read Address Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iByteCount     : In    integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemReadData (
-- do PCIe Read Address Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           oData          : Out   std_logic_vector ;
           oStatus        : Out   integer ;
           oTag           : Out   TagType
) ;
```

5.5.2 Configuration Space Transactions

```
------------------------------------------------------------
procedure PcieCfgSpaceWrite (
-- do PCIe Configuration Space Write Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iCid           : In    std_logic_vector ;
           iData          : In    std_logic_vector ;
           oStatus        : Out   integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;
```

```
------------------------------------------------------------
procedure PcieCfgSpaceRead (
-- do PCIe Configuration Space Read Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           oData          : Out   std_logic_vector ;
           oStatus        : Out   integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;
```

5.5.3 I/O Transactions

```
------------------------------------------------------------
procedure PcieIoWrite (
-- do PCIe I/O Space Write Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iData          : In    std_logic_vector ;
           oStatus        : Out   integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieIoRead (
-- do PCIe I/O Space Read Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           oData          : Out   std_logic_vector ;
           oStatus        : Out   integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;
```

5.5.4 Message Transactions

```
------------------------------------------------------------
procedure PcieMessageWrite (
-- do PCIe message (no data) Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iMsgType       : In    std_logic_vector ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMessageWrite (
-- do PCIe message (with data) Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iMsgType       : In    std_logic_vector ;
           iMsgData       : In    std_logic_vector ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;
```

5.5.5 Completion Transactions

```
------------------------------------------------------------
procedure PcieCompletion (
-- do PCIe completion (with data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iData          : In    std_logic_vector ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType
) ;
```

```
------------------------------------------------------------
procedure PcieCompletion (
-- do PCIe completion (with data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iByteCount     : In    integer ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType
) ;

------------------------------------------------------------
procedure PcieCompletionLock (
-- do PCIe locked completion (with data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iData          : In    std_logic_vector ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType
) ;

------------------------------------------------------------
procedure PcieCompletionLock (
-- do PCIe locked completion (with data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iByteCount     : In    integer ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType
) ;
```

```
------------------------------------------------------------
procedure PcieNoDataCompletion (
-- do PCIe completion (with no data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iData          : In    std_logic_vector ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType
) ;
```

5.5.6 Part Completions

```
------------------------------------------------------------
procedure PciePartCompletion (
-- do PCIe completion (with burst data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iByteCount     : In    integer ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iRemainingLen  : In    std_logic_vector ;
         iTag           : In    TagType
) ;
------------------------------------------------------------
procedure PciePartCompletionLock (
-- do PCIe locked completion (with burst data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iByteCount     : In    integer ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iRemainingLen  : In    std_logic_vector ;
         iTag           : In    TagType
) ;
```

6 The PCIe Link Display

The PCIe C model has abilities to display the traffic on the link in a formatted way. The level of detail displayed, and other characteristics is controllable from a file read at run-time, and the display can be turned on or off at specified cycles.

To control the display a file must be present in the directory from which the simulation is run as `./hex/ContDisps.hex`. If the file is not present, then the simulation will run without any link display output. An example file is shown below:

```
// Example ContDisps.hex file
// Copy to the appropriate <test dir>/hex directory, and edit as necessary.

//                      +8              +4              +2              +1
//  ,---> 11 - 8:    DispSwNoColour     DispSwEnIfEp   DispSwEnIfRc    DispSwTx
//  |,-->  7 - 4:    DispRawSym         DispPL         DispDL          DispTL
//  ||,->  3 - 0:    Unused             DispStop       DispFinish      DispAll
//  ||| ,-> Time (clock cycles, decimal)
//  ||| |
    D70 000000000000
    002 009999999999
```

In this file there are specified a series of number pairs. The first of the pair is a 12 bit ***hexadecimal*** value to control the display and the second is a ***decimal*** value (note the difference) for which clock cycle count the control is activated.

The first nibble of the control has some simulation control and a “display all” override. Bit 0, when set, will override individual output control of the other bits in the word. Bit 1 will cause the simulation to finish and exit at the specified cycle, and bit 2 will cause the simulation to stop. Bit 3 is unused, but all unused bits should be set to 0 to allow for future expansion.

The second nibble controls what is displayed as output. This matches the three layers of PCIe, with control for TLP, DLLP and PHY traffic, or even the unformatted raw data on the link, though this produces a lot of output. Bit 4, when set, enables formatted TLP output, bit 5 DLLP formatted output and bit 6 PHY Ordered Set output. All these bits can be used in isolation or together. If a lower level (starting from PHY) is used with a higher level, then the higher level output is indented for ease of interpretation.

By default, the `PcieModel` displays only received data. If two models were being used back-to-back then all traffic would be display over a given link. If the model is being used in solitude to drive a DUT, then bit 8 can turn on the display of traffic that is being transmitted by the model as well as that being received, so all traffic on the link is displayed for this case. As the model can be configured as an endpoint (downstream) or root-complex (upstream) device bits 9 and 10 enable display individually for models configured as RCs or EPs (as defined by the ENDPOINT generic setting).

Finally, the model displays colour output by default, but this can be disabled by bit 11 being set. Though extremely useful to have colour output for discriminating between up- and down-link traffic, the data that is logged for the simulation or displayed on some simulator’s console, does not decode the escape sequences and pollutes the output with literal additional characters. Not that the colour display can also

be disabled via the CONFIG_DISABLE_DISPLINK_COLOUR model option, which will override the settings in the ContDisps.hex file.

The diagram below shows an example output fragment for the ContDisps.hex file example above, with all three layers active from time 0 (but not raw output), with the EP doing both RX and TX display and the RC disabled and colour output disabled.

```
PCIEU63: {SDP
PCIEU63: 00 00 00 0f dc fd
PCIEU63: END}
PCIEU63: ...DL Ack seq 15
PCIEU63: ...DL Good DLLP CRC (dcfd)
PCIEU63: {SDP
PCIEU63: 80 09 43 f4 6e 02
PCIEU63: END}
PCIEU63: ...DL UpdateFC-P   VC0  HdrFC=37 DataFC=1012
PCIEU63: ...DL Good DLLP CRC (6e02)
PCIED62: {STP
PCIED62: 00 10 00 00 00 20 00 3e 8a 7e 00 01 02 00 14 e2 2c d5
PCIED62: END}
PCIED62: ...DL Sequence number=16
PCIED62: ......TL Mem read req Addr=00010200 (32) RID=003e TAG=8a FBE=1110 LBE=0111 Len=020
PCIED62: ......Traffic Class=0, Strong ordering (PCI)
PCIED62: ......TL No ECRC
PCIED62: ...DL Good LCRC (14e22cd5)
PCIEU63: {STP
PCIEU63: 00 09 4a 00 00 20 02 00 00 7e 00 3e 8a 01 00 00 01 02 03 04 05 06
PCIEU63: 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c
PCIEU63: 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32
PCIEU63: 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f 40 41 42 43 44 45 46 47 48
PCIEU63: 49 4a 4b 4c 4d 4e 4f 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e
PCIEU63: 5f 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74
PCIEU63: 75 76 77 78 79 7a 7b 7c 7d 00 d2 4c 4a b4
PCIEU63: END}
PCIEU63: ...DL Sequence number=9
PCIEU63: ......TL Completion with data Successful CID=0200 BCM=0 Byte Count=07e RID=003e TAG=8a Lower Addr=01
PCIEU63: ......Traffic Class=0, Strong ordering (PCI), Payload Length=0x020 DW
PCIEU63: ......00000102 03040506 0708090a 0b0c0d0e 0f101112 13141516 1718191a 1b1c1d1e
PCIEU63: ......1f202122 23242526 2728292a 2b2c2d2e 2f303132 33343536 3738393a 3b3c3d3e
PCIEU63: ......3f404142 43444546 4748494a 4b4c4d4e 4f505152 53545556 5758595a 5b5c5d5e
PCIEU63: ......5f606162 63646566 6768696a 6b6c6d6e 6f707172 73747576 7778797a 7b7c7d00
PCIEU63: ......TL No ECRC
PCIEU63: ...DL Good LCRC (d24c4ab4)
```

If using the ltssm.c code for link training, then it will display progress on state transitions. This can be disabled using the CONFIG_LTSSM_DISABLE_DISP_STATE model option.

More details of the link display capabilities can be found in the *pcievhost* documentation [2].

7 The Automatic Features of the Model

The *pcievhost* C model integrated into OSVVM was originally conceived as a means to generate and decode all the various type of PCIe traffic from the viewpoint of a host machine (i.e. a root complex) in order to drive an Endpoint or other downstream device. From this perspective, all higher layer features existed outside of the model, including the LTSSM, configuration space or main memory etc. It generated just the traffic for the three layers and a program driving the model needed to add these features if necessary. Since that time additional features have been added that cover some of these functional components.

7.1 PHY Layer Automatic Features

7.1.1 LTSSM

All ordered set types, including training sequence ordered sets, can be generated from a test program via the API as detailed in this document. Included with the model, but not part of it, are the `ltssm.c` and `ltssm.h` files located in the `Pcie/ltssm` directory. This will generate training sequences to automatically take a link from electrical idle to the L0 "link up" state. It is incomplete, however, and does not support the low power states (L1, L2 and L0s) or any of the ancillary states (Disabled, Hot Reset, Loopback or Recovery). The hooks are present in the code to add this functionality, and the source code is available to extend as necessary. Or the test code itself can send the appropriate OSs as needed.

By default, the model does not instigate a PHY link training sequence, but this can be initiated with a command:

```
SetModelOptions(UpstreamRec, INITPHY, NULLOPTVALUE) ;
```

The model will still function without the link training which can be skipped, if the DUT has some test model to also skip this step, to save on simulation time where link training is not the focus of the test.

The LTSSM can be configured via `SetModelOptions`. The control symbols in the training sequences can be set using the following options:

- `CONFIG_LTSSM_LINKNUM`
- `CONFIG_LTSSM_N_FTS`
- `CONFIG_LTSSM_TS_CTRL`

These program the values at the positions 1, 3 and 5 respectively to set the link number, the number of fast training sequence from L0s to L0, and the CTL control symbol value for generated Training Sequence ordered sets.

In order to speed up simulations a couple of timeout counts can be re-programmed using the CONFIG_LTSSM_DETECT_QUIET_TO and CONFIG_LTSSM_POLL_ACTIVE_TO_COUNT. For normal operations these would never timeout but to generate a timeout at these points they can be set to be much lower values using these options.

7.1.2 Skip Ordered Sets

By default, the model will automatically schedule skip ordered sets at a period of 1180 cycles. This rate can be changed in configuration:

```
SetModelOptions(TransactionRec, CONFIG_ENABLE_SKIPS, 1200) ;
```

If required, automatic generation of skip ordered sets can be disabled:

```
SetModelOptions(TransactionRec, CONFIG_DISABLE_SKIPS, NULLOPTVALUE) ;
```

Configuration of skip ordered set generation should be done before any link initialisation and transaction generation in order to be active from time 0.

7.2 Automatic DLL Packets

7.2.1 Flow Control

The model has automatic flow control features with the ability to configure the apparent size of its receive buffers, initialise the link and automatically generate flow control update DLLPs. To configure the size of the buffers, in units of “credits”, a set of model options for setting each of the six types is available:

- CONFIG_POST_HDR_CR
- CONFIG_POST_DATA_CR
- CONFIG_NONPOST_HDR_CR
- CONFIG_NONPOST_DATA_CR
- CONFIG_CPL_HDR_CR
- CONFIG_CPL_HDR_CR

E.g. `SetModelOptions(TransactionRec, CONFIG_POST_DATA_CR, 1024) ;`

The rate at which DLL headers and data packets are consumed can be separately configure using the CONFIG_FC_HDR_RATE and CONFIG_FC_DATA_RATE model options, both of which have a 4 cycle default value. This allows for emulating latency in response with update DLLPs. E.g:

```
SetModelOptions(TransactionRec, CONFIG_FC_HDR_RATE, 2) ;
SetModelOptions(TransactionRec, CONFIG_FC_DATA_RATE, 10) ;
```

In addition, the rate at which auto-generated completions are generated can be controlled with the CONFIG_CPL_DELAY_RATE and the CONFIG_CPL_DELAY_SPREAD model option. This first sets a cycle delay before responding with a completion to a processed request (default 0), and the second sets a randomised spread around the programmed delay (default 0). E.g.:

```
SetModelOptions(TransactionRec, CONFIG_CPL_DELAY_RATE, 20) ;
SetModelOptions(TransactionRec, CONFIG_CPL_DELAY_SPREAD, 10) ;
```

Flow control can be disabled, meaning that, effectively, all credit values are 0, which is equivalent to infinite buffer space, and no flow control. E.g.:

```
SetModelOptions(TransactionRec, DISABLE_FLOW_CONTROL, NULLOPTVALUE) ;
```

Built into the model is code to initialise flow control for VC0, though the model does not require this step to function, but then the flow control must be disabled before any transactions generated. To initiate VC0 flow control initialisation execute:

```
SetModelOptions(TransactionRec, INIT_DLL, NULLOPTVALUE) ;
```

This must be done after any PHY link initialisation but before any transactions are generated.

7.2.2 Packet Acknowledges

The model can automatically generate ACK packets on reception of a TLP (with good LCRC). By default, this is enabled with a “processing rate” of 1. I.e., it will send an ACK on reception of each packet. This can be altered to receive multiple packets before acknowledging them in a single ACK. E.g.:

```
SetModelOptions(TransactionRec, ENABLE_ACK, 3) ;
```

This can be updated throughout a simulation. Automatic generation of ACKs can be also disabled:

```
SetModelOptions(TransactionRec, DISABLE_ACK, NULLOPTVALUE) ;
```

7.3 Internal Memory Models and Automatic Completions

7.3.1 Main Memory Model

Built into the PCIe C model is a sparse memory model that can model a full 64-bit address space. When the model receives memory requests it will handle these requests internally and access the memory model to load or store data as appropriate for reads or writes. For memory reads, it will automatically generate a completion packet to return the data. If the model is configured as an endpoint (via the ENDPOINT generic) and internal memory is enabled, then the incoming memory requests will be checked against the active BAR registers and, if outside a valid range, the model will automatically

return an unsupported request completion. This feature can be disabled with the CONFIG_DISABLE_UR_CPL model option.

Internal memory can be disabled through a model option:

```
SetModelOptions(TransactionRec, DISABLE_MEM, NULLOPTVALUE) ;
```

7.3.2 Configuration Space

Built into the model is a 4K word Type 0 configuration space. By default, this acts just like a memory and can be written and read using configuration space writes and reads if the model is configured as an endpoint (via the ENDPOINT generic) and internal memory is enabled. When active, completions are automatically generated for both writes and reads (as configurations accesses are non-posted). If configuration writes or reads are sent to the model when not an endpoint the model will automatically generate an unsupported request completion, unless disabled (via the CONFIG_DISABLE_UR_CPL model option). Just as for the main memory model, the internal configuration space can be disabled with the model DISABLE_MEM option.

The internal configurations space, when active, can be programmed to contain valid Type 0 configuration space values and to include a mask for each word to indicate if a bit in the register word is read-only or not, with a mask bit of 1 indicating read-only. To program a configuration space register three model options are available. The option SETCFGSPCOFFSET is used first to set which register is active for an update—a byte offset aligned to 32-bits— and is then programmed with the SETCFGSPC option. A read-only mask can then be set as well, if necessary, with the SETCFGSPCMASK option. The example below shows setting the BAR0 register to be a 32-bit prefetchable space of 4Kbytes:

```
SetModelOptions(TransactionRec, CFGSPCOFFSET, 16#10#) ;
SetModelOptions(TransactionRec, CFGSPC,       16#00000008#) ;
SetModelOptions(TransactionRec, CFGSPCMASK,   16#00000fff#) ;
```

7.3.3 I/O Space

The model does not currently support an internal I/O space and will automatically send an unsupported request completion on reception of I/O requests, unless disabled with the CONFIG_DISABLE_UR_CPL model option.

8 Writing Tests in Co-simulation Software

In the previous sections, configuration and control of the model was done through standard OSVVM VHDL procedure calls. OSVVM's co-simulation capabilities map these calls (and others) to the C or C++ domains [3] and so the model can be driven from software in this way, though this is true of all the OSVVM VCs.

However, the PCIe model is a true co-simulation component and can be driven directly by software using the C++ API that the underlying *pcievhost* model provides [2]. Here the VUserMain<n> function

does not call the `pcieVcInterface run()` method but creates a `pcieModelClass` API object and makes calls directly to this interface. A template program might look like the following:

```
#include <stdio.h>
#include <stdlib.h>

#include "OsvvmCosim.h"
#include "pcieVcInterface.h"
#include "ltssm.h"

//------------------------------------------------------------
static void VUserInput (pPkt_t pkt, int status, void* usrptr) {
    // handle received packets here

    // Once packet is finished with, the allocated space *must* be freed.
    // All input packets have their own memory space to avoid overwrites
    // which shared buffers.
    DISCARD_PACKET(pkt);
}

//------------------------------------------------------------
extern "C" void VUserMain63 (int node) {
    // Create an API object for this node
    pcieModelClass* pcie = new pcieModelClass(node);

    // Initialise PCIe VHost, with input callback function & pass in node number.
    pcie->initialisePcie(VUserInput, &node);

    /********** CONFIGURE MODEL*************/
    /**** DO ANY LINK/DLL INITIALISATION ***/
    /*********** DO ANY TESTS **************/

    // Send out idles forever
    while(true) pcie->sendIdle(10000);

}
```

This program does not use the transaction interface by drives the PCIe link directly. If the model is configured as an endpoint with memory enabled, then the code just needs to configure the model and run any PHY and/or DLL initialisations. It can then bypass the need to make calls to the API methods to generate transactions and simply act as a target for an upstream device, automatically responding to TLP requests. The code mustn't return, and so it simply loops over the methods to generate idles to advance simulation time. The model will internally then process any received transactions. The code in `PCIe/tests/vc/VUserMainEP.cpp` is of this type of usage.

Details of the `PcieModelClass` API can be found in the *pcievhost* documentation [2], but there are some low level API calls that are useful to get access to the settings of the HDL model. To access these the `VRead` function is called which has a prototype:

```
VRead(uint32_t value_type, uint32_t *value, int delta, int node);
```

The type is just an offset where the value to be read is available but comes with predefined values. The most useful are shown below:

NODENUMADDR	Fetch the node number generic's value
LANESADDR	Fetch the number of lanes for the connected link
EP_ADDR	Fetch the ENDPOINT generic's setting
CLK_COUNT	Fetch the model's current clock count
RESET_STATE	Fetch the current state of the reset input port (1 = being reset)

When accessing these types, the value is returned in the `value` pointer. The delta argument is to control whether the access takes a cycle or not of simulation time. It is usually the case for these value that they should be "delta" cycles, taking no simulation time, and the argument is usually set to `pcieVcInterface::DELTACYCLE`. Lastly the node number is passed in to send the access to the correct instantiation.

9 About OSVVM

The OSVVM utility and verification component libraries were developed and are maintained by Jim Lewis of SynthWorks VHDL Training and the co-simulation libraries/features were developed and are maintained by Simon Southwell. These libraries evolved from methodology and packages developed for SynthWorks' VHDL Testbenches and verification class.

Please support OSVVM by purchasing your VHDL training from SynthWorks.

10 About the Authors and Contributors

10.1 About the PCIe VC Contributor – Simon Southwell

Simon Southwell has over thirty five years of embedded software, ASIC and FPGA design experience in fields that include high performance computing, wireless, cellular modem (LTE) and processor system modelling. Mr. Southwell has developed and successfully deployed co-simulation techniques at a variety of companies using his own open-source IP.

Mr. Southwell is currently developing open source IP in areas such as RISC-V and co-simulation, is actively mentoring undergraduate and new graduate engineers in digital systems design and is also collaborating on the development of OSVVM.

If you find bugs the co-simulation packages, the PCIe VC, or would like to request enhancements, Mr. Southwell can be reached at simon.southwell@gmail.com.

10.2 About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has thirty plus years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr. Lewis has done ASIC and FPGA design, custom model development, and consulting.

Mr. Lewis is chair of the IEEE 1076 VHDL Working Group (VASG) and is the primary developer of the Open Source VHDL Verification Methodology (OSVVM.org) packages. Neither of these activities generate revenue. Please support our volunteer efforts by buying your VHDL training from SynthWorks.

If you find bugs these packages or would like to request enhancements, you can reach me at jim@synthworks.com.

11 References

[1] Simon Southwell, PCIe Primer
[2] Simon Southwell, The *pcievhost* model and documentation
[3] Simon Southwell, OSVVM Co-simulation Framework
[4] Jim Lewis, OSVVM AXI4 Verification Components
[5] Jim Lewis, OSVVM Address Bus Model Independent Transaction User Guide