<trellisys>

>

>

lscc_pcie_x1_rc Bus Functional Model Users Manual

Change History

Date	Version	Comments
29. May. 2014	1.1	Type-1 configuration space accesses added Support for ECP5 x1
10. May. 2011	1.0	First general release
12. Feb. 2011	0.2	New VHDL and Verilog functions added
29. Sep. 2010	0.1	Preliminary Version derived from manual from previous (VHDL only) BFM

Table of Contents

Figures

1 Introduction

2 Description

2.1 BFM Overview

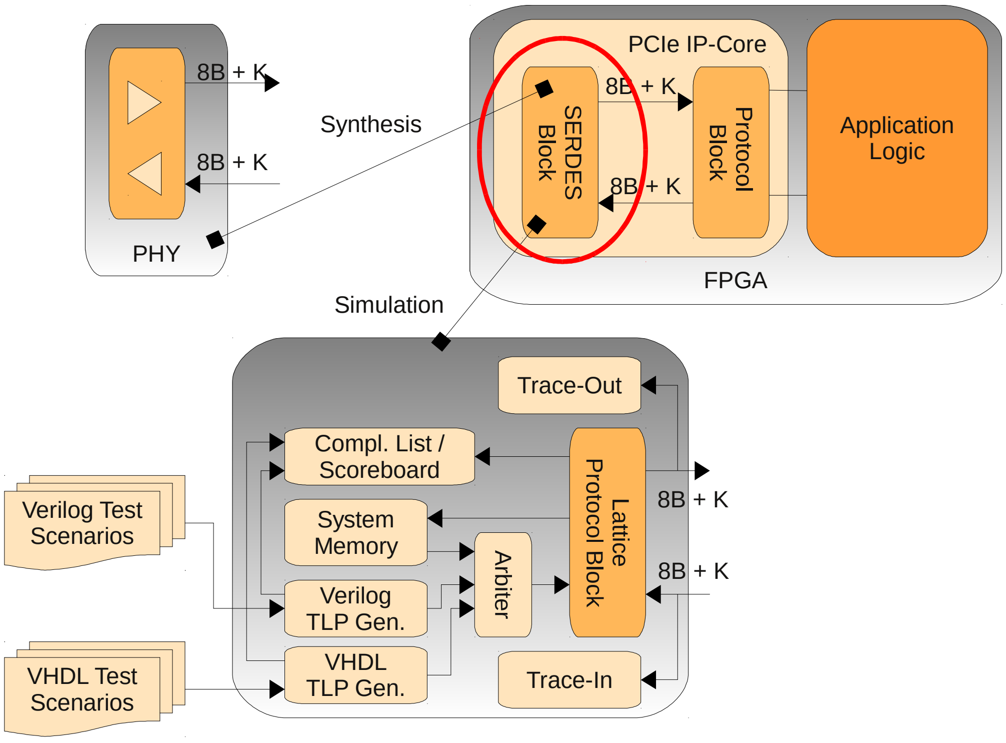

Figure 1: BFM Overview: Simulation vs. Synthesis

An overview of the PCI Express BFM and how it can be connected to the application logic in an ECP2M FPGA can be seen in 1. The BFM is designed to have exactly the same port connections as the SERDES unit from the Lattice ECP2M cell library. Communication between the BFM and the FPGA design takes place over the input and output PIPE interfaces.

The PCI Express BFM installation contains two modified versions of the top-level wrappers generated by the Lattice IPexpress tool. Both modified wrappers have the same port definitions which ensures that the rest of the design can use a single code base throughout the synthesis and simulation. The synthesis version of the modified wrapper is internally very similar to the original version output by IPexpress. The wrapper version used in simulation references the 'SERDES' block from the BFM library. Additionally, the wrapper contains a number of VHDL assertions which constantly monitor the PCI Express traffic generated by the user logic. Errors and warnings are written to the simulation console window.

The BFM is controlled by test scenarios written by the user. A test scenario is typically made up of one or more concurrent tasks which stimulate and monitor the device under test. The principal task in a test scenario simulates the behaviour of an upstream system such as a PC. This task, which must always be present, can model almost any PCI Express transaction layer command sequence which would be generated by a real system.

An additional optional task can be defined to monitor the system memory. This is most useful when the FPGA under test contains PCI Express requester functionality such as a DMA controller. It is possible, for instance, to model a scatter/gather memory buffer and check that the DMA controller does not write outside of the scatter/gather elements.

A further optional task can be defined to simulate the flow control (credits/ update flow-control packets) from the upstream port. This can be used to verify that the DUT behaves correctly when too few or no credits are available. If this task is not defined, the BFM automatically returns credits immediately.

2.1.1 How the BFM advertises Credits

In the default configuration, the BFM is initialised with minimum posted and non-posted credits but infinite completion credits. Infinite completion credits is a requirement on a PCI Express root complex or end-point. The BFM currently cannot be used to simulate the behaviour of a PCI Express switch.

The initial settings for posted and non-posted credits are one header credit and eight payload credits (corresponds to the smallest valid maximum payload size of 32 DWords or 128 bytes). These settings are of course only of importance if the BFM is the target of memory read or write requests generated by the DUT (i.e. the DUT has DMA functionality). Credits are automatically released by the BFM shortly after a packet has been received from the DUT.

The user has the option of modifying the default settings to enable higher performance or more exhaustive test scenarios. Using the svc_ca_np() procedure for non-posted credits (see section 3.1.23 on page 38) or the svc_ca_p() procedure for posted credits (see section 3.1.24 on page 39), the user can assign additional credits to the BFM which are automatically advertised to the DUT with the next credit-update packet from the data-link layer. Once additional credits are assigned they are automatically updated and returned to the DUT when the appropriate incoming packets to the BFM have been processed.

2.2 Using the PCI Express BFM

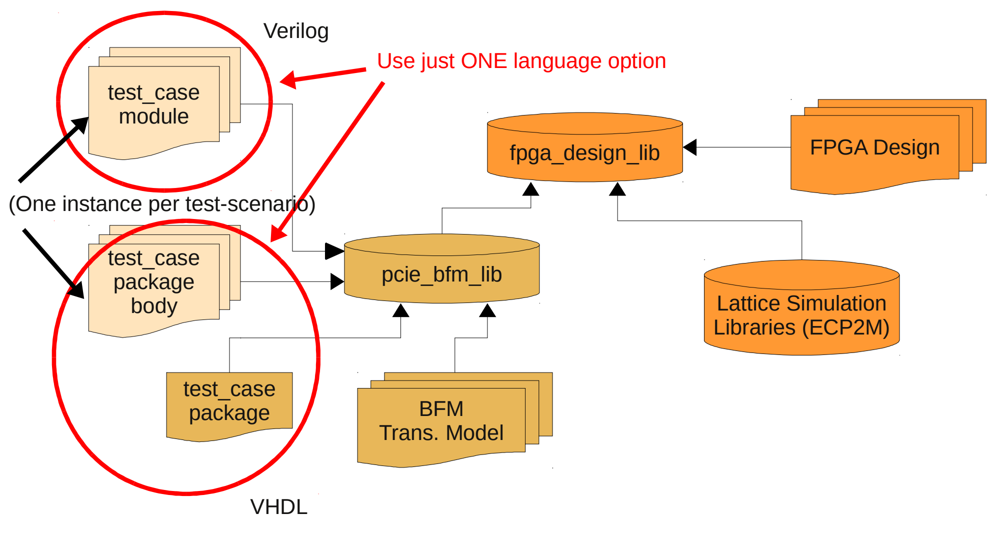

Figure 2: Simulation Library Overview

Figure 2 illustrates how the precompiled BFM library is incorporated into a verification environment. The BFM deliverables provide the 'pcie_bfm_lib' library. This library contains the BFM itself as well as VHDL package headers defining the entry points which can be implemented by a test case. Each entry point defines one of the concurrent tasks which can be executed by the BFM as described in the previous section.

Also as mentioned in the previous section, the BFM appears to the simulation environment as an implementation of the Lattice SERDES cell. The simulation version of the modified PCI-core top-level wrapper references the SERDES replacement module from the 'pcie_bfm_lib' library.

The FPGA design itself must be compiled to a different work library. The name of this library can be freely chosen by the user.

To run a test-case, the user first compiles a test-case file to the 'pcie_bfm_lib' library. Each test case file defines the package bodies (comparable to a call-back implementation) for the user definable tasks provided by the BFM. When running longer test sequences such as regression tests, there is no need to recompile the design each time. Only the package bodies must be recompiled, an action which merely takes a few seconds to perform.

3 VHDL Programming Interface

3.1 Main Test Task

The main test task must always be implemented as a body to the package 'pcie_vhdl_test_case_pkg' and be compiled to the 'pcie_bfm_lib' library. The entry point must be called 'run_test' and have the parameter foot-print as shown below.

The IEEE libraries 'std_logic_1164' and 'numeric_std' are already referenced by the pre-compiled package header and therefore need not be additionally declared again before the user-defined package body.

Main Test Declaration and Entry-Point

```
Package Body pcie_vhdl_test_case_pkg is
   procedure run_test (signal clk   : in    std_logic;
                       signal sv    : inout t_bfm_stim;
                       signal rv    : in    t_bfm_resp;
                              id    : in    natural := 0) is

         -- Test specific signal or variable declarations go here
         -- Also, test specific functions or procedures
   begin
         -- Test Content goes here

   end;
End Package;
```

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
id	natural	variable	in	Identifier allowing multiple BFM instances to be instantiated. The run_test procedure would contain a separate section for each instance.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Referenced by most routines.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Referenced by most routines

3.1.1 Type-0 Configuration Space Read Sequence

```
cfgrd0(clk, sv, rv, addr, data,
       be_first := b”1111”, cpl_wait := true, cpl_sta := c_cpl_sta_sc,
       no_compare := false);
```

Description

Generate a type 0 configuration read cycle. If the cpl_wait parameter is set to 'true' the call will wait until a completion packet has been returned by the DUT. The expected completion response can also be specified (successful, unsupport, completer abort)

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the address of the configuration space register. PCI compatible configuration space registers lie in the range 0 to 0xFF. PCIexpress extended configuration space registers are located in the range 0 to 0xFFF. Larger values are silently truncated to twelve bits.
be_first	std_logic_vector	variable	in	Specifies the valid bytes in the expected data. Typically used to specify byte / word / dword accesses.
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
cpl_sta	std_logic_vector	variable	in	Optionally specifies the expected completion response status. The default value is successful (c_cpl_sta_sc). Other possible values are unsupported request (c_cpl_sta_ur) , completer abort (c_cpl_sta_ca) or configuration request retry (c_cpl_sta_crs).
cpl_wait	boolean	variable	in	Specifies whether program flow waits until the DUT has returned a completion packet. Default is true
data	std_logic_vector	variable	in	Specifies the expected value of the DWord to be returned by DUT.
no_compare	boolean	variable	in	Specifies whether the BFM scoreboard checks the returned dword for correctness. The default setting is true, which always checks the incoming dword. Setting this variable to false can be useful for polling a configuration space register during simulation
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

Example

```
Package Body pcie_vhdl_test_case_pkg is
   procedure run_test (signal clk   : in    std_logic;
                       signal sv    : inout t_stim_desc;
                       signal rv    : in    t_resp_desc;
                              id    : in    natural := 0) is
   begin
      cfgrd0(clk, sv, rv, c_csreg_vend_id, X"47110815", cpl_wait => false);
   end;
end package;
```

The above example illustrates reading the vendor ID / Device ID and comparing with the expected value of 0x47110815. If the expected value is not returned, an error message will be logged to standard output.

3.1.2 Type-0 Configuration Space Write Sequence

```
cfgwr0(clk, sv, rv, addr, data,
       be_first := b"1111", cpl_wait := false, cpl_sta := c_cpl_sta_sc);
```

Description

Generate a type 0 configuration write cycle. If the cpl_wait parameter is set to 'true' the call will wait until a completion packet has been returned by the DUT. The DUT is required to return a 'cpl' packet, i.e. completion without payload. This is checked by the BFM.

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the address of the configuration space register.
be_first	std_logic_vector	variable	in	Specifies the valid bytes in the data sent to the DUT. Typically used to specify byte / word / dword accesses.
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
cpl_sta	std_logic_vector	variable	in	Optionally specifies the expected completion response status. The default value is successful (c_cpl_sta_sc). Other possible values are unsupported request (c_cpl_sta_ur) , completer abort (c_cpl_sta_ca) or configuration request retry (c_cpl_sta_crs).
cpl_wait	boolean	variable	in	Specifies whether program flow waits until the DUT has returned a completion packet. Default is false
data	std_logic_vector	variable	in	Specifies the DWord to be written to the DUT
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

Example

```
cfgwr0(clk, sv, rv, c_csreg_bar0, X"FFFFFFFF");
cfgrd0(clk, sv, rv, c_csreg_bar0, X"FFFF0008");
cfgwr0(clk, sv, rv, c_csreg_bar0, X"80860000");
cfgrd0(clk, sv, rv, c_csreg_bar0, X"80860008");
```

The above example illustrates how base address register (BAR) initialisation can be simulated. The register is first written with all ones. The second line tests for the expected window size of 64K bytes in prefetchable memory space. The third line sets the resource base address to 0x80860000. The final line checks the value written to the base address register.

3.1.3 Type-1 Configuration Space Read Sequence

```
cfgrd1(clk, sv, rv, addr, data,
       be_first := b”1111”, cpl_wait := true, cpl_sta := c_cpl_sta_sc,
       no_compare := false);
```

Description

Generate a type 1 configuration read cycle. If the cpl_wait parameter is set to 'true' the call will wait until a completion packet has been returned by the DUT. The expected completion response can also be specified (successful, unsupport, completer abort)

Note that it is a protocol error if a PCI Express endpoint is the target of a type-1 configraiton space access. Type-1 configuration space accesses should only be directed at a PCI or PCI Express bridge. An end-point which is the target of a type-1 configuration space access should always return a status of 'unsupported request'

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the address of the configuration space register. PCI compatible configuration space registers lie in the range 0 to 0xFF. PCIexpress extended configuration space registers are located in the range 0 to 0xFFF. Larger values are silently truncated to twelve bits.
be_first	std_logic_vector	variable	in	Specifies the valid bytes in the expected data. Typically used to specify byte / word / dword accesses.
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
cpl_sta	std_logic_vector	variable	in	Optionally specifies the expected completion response status. The default value is successful (c_cpl_sta_sc). Other possible values are unsupported request (c_cpl_sta_ur) , completer abort (c_cpl_sta_ca) or configuration request retry (c_cpl_sta_crs).
cpl_wait	boolean	variable	in	Specifies whether program flow waits until the DUT has returned a completion packet. Default is true
data	std_logic_vector	variable	in	Specifies the expected value of the DWord to be returned by DUT.
no_compare	boolean	variable	in	Specifies whether the BFM scoreboard checks the returned dword for correctness. The default setting is true, which always checks the incoming dword. Setting this variable to false can be useful for polling a configuration space register during simulation
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

Example

```
Package Body pcie_vhdl_test_case_pkg is
   procedure run_test (signal clk   : in    std_logic;
                       signal sv    : inout t_stim_desc;
                       signal rv    : in    t_resp_desc;
                              id    : in    natural := 0) is
   begin
      cfgrd1(clk, sv, rv, c_csreg_vend_id, X"47110815", cpl_wait => false);
   end;
end package;
```

The above example illustrates reading the vendor ID / Device ID and comparing with the expected value of 0x47110815. If the expected value is not returned, an error message will be logged to standard output.

3.1.4 Type-1 Configuration Space Write Sequence

```
cfgwr1(clk, sv, rv, addr, data,
       be_first := b”1111”, cpl_wait := false, cpl_sta := c_cpl_sta_sc);
```

Description

Generate a type-1 configuration write cycle. If the cpl_wait parameter is set to 'true' the call will wait until a completion packet has been returned by the DUT. The DUT is required to return a 'cpl' packet, i.e. completion without payload. This is checked by the BFM.

Note that it is a protocol error if a PCI Express endpoint is the target of a type-1 configraiton space access. Type-1 configuration space accesses should only be directed at a PCI or PCI Express bridge. An end-point which is the target of a type-1 configuration space access should always return a status of 'unsupported request'

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the address of the configuration space register.
be_first	std_logic_vector	variable	in	Specifies the valid bytes in the data sent to the DUT. Typically used to specify byte / word / dword accesses.
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
cpl_sta	std_logic_vector	variable	in	Optionally specifies the expected completion response status. The default value is successful (c_cpl_sta_sc). Other possible values are unsupported request (c_cpl_sta_ur) , completer abort (c_cpl_sta_ca) or configuration request retry (c_cpl_sta_crs).
cpl_wait	boolean	variable	in	Specifies whether program flow waits until the DUT has returned a completion packet. Default is false
data	std_logic_vector	variable	in	Specifies the DWord to be written to the DUT
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

Example

```
cfgwr1(clk, sv, rv, c_csreg_bar0, X”FFFFFFFF”);
```

3.1.5 Completion without Payload

```
cpl(clk, sv, rv, req_id := X”0080”, tag := X”oo”);
```

Description

Generate a completion sequence without payload. The transcation tag and requestor-id can optionally be specified. This method is intended for testing the DUT response to an unexpected completion. Completion packets in response to a non-posted request from the DUT (e.g. DMA memory read) are automatically generated by the BFM.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
req_id	std_logic_vector	variable	in	Transaction requestor ID. This input parameter will automatically be resized to sixteen bits. Bits [15:8] are the bus number, bits [7:3] the device number and bits [2:0] the function number.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
tag	std_logic_vector	variable	in	Transaction Tag. Specifies the tag value to use in the transaction response packet

Examples

Generate a completion packet with tag value of 0x03.

```
cpl(clk, sv, rv, tag => X”03”);
```

Generate a completion packet using the default tag value.

```
cpl(clk, sv, rv);
```

3.1.6 Completion with Payload

```
cpld(clk, sv, rv, data, req_id := X”0008”, tag := X”00”);

cpld(clk, sv, rv, pload, req_id := X”0008”, tag := X”00”);
```

Description

Generate a completion sequence with payload. The payload can be a single DWord or an array of DWords up to a maximum length of 1024. The transcation tag can optionally be specified. This method is intended for testing the DUT response to an unexpected completion. Completion packets in response to a non-posted request from the DUT (e.g. DMA memory read) are automatically generated by the BFM.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
data	std_logic_vector	variable	in	Optional parameter. Specifies a single DWord as payload in the completion response packet
pload	t_tlm_payload	variable	in	Optional parameter. Specifies an array of DWords as completion response payload
req_id	std_logic_vector	variable	in	Transaction requestor ID. This input parameter will automatically be resized to sixteen bits. Bits [15:8] are the bus number, bits [7:3] the device number and bits [2:0] the function number.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
tag	std_logic_vector	variable	in	Transaction Tag. Specifies the tag value to use in the transaction response packet

Examples

Generate a completion packet with a single DWord and a tag value of 0x03.

```
cpld(clk, sv, rv, X”DEAD_BEEF”, req_id => X”FF00”, tag => X”03”);
```

Generate a completion packet with a payload of two DWords using the default tag and requestor-id values. .

```
cpld(clk, sv, rv, (X”0123_4567”, X”89AB_CDEF”));
```

3.1.7 I/O Space Read Sequence

```
iord(clk, sv, rv, addr, data,
     be_first := b"1111", cpl_wait := true, cpl_sta := c_cpl_sta_sc,
     no_compare := false);
```

Description

Generate a read sequence to PCI Express I/O space. The expected completion status can be specified and has a default value of c_cpl_sta_sc for successful completion (SC). This enables verification of expected unsupported requests when addressing unmapped or unsupported I/O space regions.

The first form generates an I/O space read sequence whereby the data returned are considered don't care. No checking is performed. The PCI Express specification does not permit bursting (payload size larger than one) to or from I/O space.

The second form generates a single DWord read request. The expected value of the DWord returned by the DUT is specified in the second parameter. If the check fails an error message is printed to the output console.

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the target address of the I/O space read transaction.
be_first	std_logic_vector	variable	in	Specifies the valid bytes in the expected data. Typically used to specify byte / word / dword accesses.
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
cpl_sta	std_logic_vector	variable	in	Optionally specifies the expected completion response status. The default value is successful (c_cpl_sta_sc). Other possible values are unsupported request (c_cpl_sta_ur) , completer abort (c_cpl_sta_ca) or configuration request retry (c_cpl_sta_crs).
cpl_wait	boolean	variable	in	Specifies whether program flow waits until the DUT has returned a completion packet. Default is true
data	std_logic_vector	variable	in	Specifies the expected value of the DWord to be returned by DUT
no_compare	boolean	variable	in	Specifies whether the BFM scoreboard checks the returned dword for correctness. The default setting is true, which always checks the incoming dword. Setting this variable to false can be useful for polling an I/O space address during simulation
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

Examples

Read a single DWord from the specified location in PCI Express I/O space.

```
iord(clk, sv, rv, X”80860000”, X”DEAD_BEEF”);
```

Perform a read access to an unmapped I/O space region, expecting a response of 'unsupported request' (UR)

```
iord(clk, sv, rv, X”80860000”, cpl_sta => c_cpl_sta_ur);
```

3.1.8 I/O Space Write Sequence

```
iowr(clk, sv, rv, addr, data,
     be_first := b”1111”, cpl_wait := false, cpl_sta := c_cpl_sta_sc);
```

Description

Generate a write sequence to PCI Express I/O space. The expected completion status can be specified and has a default value of c_cpl_sta_sc for successful completion (SC). This enables verification of expected unsupported requests when addressing unmapped or unsupported I/O space regions.

The first form generates an I/O space write sequence whereby the data written are considered don't care. Simple incrementing bytes (0x00010203) are used. The PCI Express specification does not permit bursting (payload size larger than one) to or from I/O space.

The second form generates a single DWord write request. The DWord to be written is specified in the second parameter.

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the target address of the I/O space write transaction.
be_first	std_logic_vector	variable	in	Specifies the valid bytes in the data sent to the DUT. Typically used to specify byte / word / dword accesses.
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
cpl_sta	std_logic_vector	variable	in	Optionally specifies the expected completion response status. The default value is successful (c_cpl_sta_sc). Other possible values are unsupported request (c_cpl_sta_ur) , completer abort (c_cpl_sta_ca) or configuration request retry (c_cpl_sta_crs).
cpl_wait	boolean	variable	in	Specifies whether program flow waits until the DUT has returned a completion packet. Default is false
data	std_logic_vector	variable	in	Specifies the DWord to be written to the DUT
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

Examples

Write a single DWord to the specified location in PCI Express I/O space.

```
iowr(clk, sv, rv, X”80860000”, X”DEAD_BEEF”);
```

Perform a write access to an unmapped I/O space region, expecting a response of 'unsupported request' (UR)

```
iowr(clk, sv, rv, X”80860000”, X”DEAD_BEEF”, cpl_sta => c_cpl_sta_ur);
```

3.1.9 Memory Read Sequence

```
memrd(clk, sv, rv, addr, data,
      be_first := b"1111", cpl_wait := true, cpl_sta := c_cpl_sta_sc,
      no_compare := false);

memrd(clk, sv, rv, addr, pload,
      be_first := b"1111", be_last := b"1111",
      cpl_wait := true, cpl_sta := c_cpl_sta_sc, no_compare := false);
```

Description

Generate a memory read sequence. The expected completion status can be defined and has a default value of c_cpl_sta_sc for completion successful (SC). This enables verification of expected unsupported requests when addressing unmapped memory areas.

The first form generates a single DWord read request. The expected value of the DWord returned by the DUT is specified in the data parameter. If the optional check fails, an error message is printed to the output console.

The second form is used to read multiple DWords from the DUT. The pload parameter specifies an array of std_logic_vector elements. The array size determines the request size of the resulting TLP.

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the start address of the memory read transaction.
be_first	std_logic_vector	variable	in	Optionally specifies the byte-enable mask for the first or only DWord
be_last	std_logic_vector	variable	in	Optionally specifies the byte-enable mask for the last DWord
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
cpl_sta	std_logic_vector	variable	in	Optionally specifies the expected completion status. Currently, the values c_cpl_sta_sc (successful) and c_cpl_sta_ur (unsupported request) are supported
cpl_wait	boolean	variable	in	Specifies whether program flow waits until the DUT has returned a completion packet. Default is true
data	std_logic_vector	variable	in	Optional parameter. Specifies the expected value of a single DWord returned from the DUT
no_compare	boolean	variable	in	Specifies whether the BFM scoreboard checks the returned data for correctness. The default setting is true, which always checks the incoming data. Setting this variable to false can be useful for polling one or more memory space locations during simulation
pload	t_tlm_payload	variable	in	Optional parameter. Specifies an array of DWords as expected values
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

Examples

Read a single DWord from the specified location.

```
memrd(clk, sv, rv, X”80860000”, X”DEAD_BEEF”);
```

Read a single DWord from a location specified by a local variable and four DWords from an initial address offset by a further 0x100 bytes.

```
Package Body pcie_x1_tlm_test_case_pkg is
   procedure run_test (signal clk   : in    std_logic;
                       signal sv    : inout t_bfm_stim;
                       signal rv    : in    t_bfm_resp;
                              id    : in    natural := 0) is
      variable v_target_addr  : std_logic_vector(31 downto 0);
   begin
         --Initialisation Sequence (BARs etc.) omitted.

      v_target_addr := X“8086_0000”;
      memrd(clk, sv, rv, v_target_addr, X"01234567");

      memrd(clk, sv, rv,
            std_logic_vector(unsigned(v_target_addr) + 16#100#),
            (X”7654_3210”, X”FEDCBA98”, X”10111213”, X”2425_2627”));

         – Expect an unsupport request (UR) response from address 0xFFFFFFFF
      memrd(clk, sv, rv, X”FFFF_FFFF”, X”F0F0F0F0”,
            cpl_sta => c_cpl_sta_ur);
   end;
end package;
```

Notes

The user does not have to be concerned with either the max_read_request_size setting in the BFM (see section Fehler: Referenz nicht gefunden on page Fehler: Referenz nicht gefunden) or the 4 KByte address boundary which a PCI Express memory request is not permitted to cross. Both of these restrictions are automatically handled by the BFM. A single call to the memrd() procedure may result in multiple PCI Express transactions being sent to the DUT.

3.1.10 Locked Memory Read Sequence

```
memrd_lk(clk, sv, rv, addr, data,
         be_first := b”1111”, cpl_wait := true, cpl_sta := c_cpl_sta_sc,
         no_compare := false);

memrd_lk(clk, sv, rv, addr, pload,
         be_first := b”1111”, be_last := b”1111”,
         cpl_wait := true, cpl_sta := c_cpl_sta_sc, no_compare := false);
```

Description

Generate a locked memory read sequence. The expected completion status can be defined and has a default value of c_cpl_sta_sc for completion successful (SC). This enables verification of expected unsupported requests when addressing unmapped memory areas.

The first form generates a single DWord locked read request. The expected value of the DWord returned by the DUT is specified in the data parameter. If the optional check fails, an error message is printed to the output console.

The second form is used to read multiple DWords from the DUT with the locked condition set. The pload parameter specifies an array of std_logic_vector elements. The array size determines the request size of the resulting TLP.

Note that locked read is only supported by legacy end-points. The PCI Express specification advises against using locked accesses for new developments. If successful, the locked status persists until the root-complex issues an unlock message.

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the start address of the memory read transaction.
be_first	std_logic_vector	variable	in	Optionally specifies the byte-enable mask for the first or only DWord
be_last	std_logic_vector	variable	in	Optionally specifies the byte-enable mask for the last DWord
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
cpl_sta	std_logic_vector	variable	in	Optionally specifies the expected completion status. Currently, the values c_cpl_sta_sc (successful) and c_cpl_sta_ur (unsupported request) are supported
cpl_wait	boolean	variable	in	Specifies whether program flow waits until the DUT has returned a completion packet. Default is true
data	std_logic_vector	variable	in	Optional parameter. Specifies the expected value of a single DWord returned from the DUT
no_compare	boolean	variable	in	Specifies whether the BFM scoreboard checks the returned data for correctness. The default setting is true, which always checks the incoming data. Setting this variable to false can be useful for polling one or more memory space locations during simulation
pload	t_tlm_payload	variable	in	Optional parameter. Specifies an array of DWords as expected values
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

Examples

Read a single DWord from the specified location.

```
memrd_lk(clk, sv, rv, X”80860000”, X”DEAD_BEEF”);
```

Read a single DWord from a location specified by a local variable and four DWords from an initial address offset by a further 0x100 bytes.

```
Package Body pcie_x1_tlm_test_case_pkg is
   procedure run_test (signal clk   : in    std_logic;
                       signal sv    : inout t_bfm_stim;
                       signal rv    : in    t_bfm_resp;
                              id    : in    natural := 0) is
      variable v_target_addr  : std_logic_vector(31 downto 0);
   begin
         --Initialisation Sequence (BARs etc.) omitted.

      v_target_addr := X“8086_0000”;
      memrd_lk(clk, sv, rv, v_target_addr, X"01234567");

      memrd_lk(clk, sv, rv,
               std_logic_vector(unsigned(v_target_addr) + 16#100#),
               (X”7654_3210”, X”FEDCBA98”, X”10111213”, X”2425_2627”));

      memwr(clk, sv, rv, X”80870000”, X”DEAD_BEEF”);

         – Expect an unsupport request (UR) response from address 0xFFFFFFFF
      memrd_lk(clk, sv, rv, X”FFFF_FFFF”, X”F0F0F0F0”,
               cpl_sta => c_cpl_sta_ur);

      msg_unlock(clk, sv, rv);
   end;
end package;
```

Notes

A locked condition persists from the first successful locked read (completion status SC) up to the unlock message generated by the root-complex. During the locked state, all read accesses must use the locked read format whereas all memory write transactions use the standard memory write format. Locking affects the complete path from the root complex to the end-point. When a locked status is active all other devices are prevented from accessing any device in the locked bus segments.

3.1.11 Memory Write Sequence

```
memwr(clk, sv, rv, addr, data, be_first := B“1111”);
memwr(clk, sv, rv, addr, pload, be_first := B“1111”, be_last := B”1111”);
```

Description

A PCI Express memory write transaction is a posted request and never receives a completion packet.

The first form generates a single DWord write request. The DWord to be written is specified in the data parameter.

The second form is used to write multiple DWords to the DUT. The pload parameter specifies an array of std_logic_vector elements. The array size determines the request size of the resulting TLP.

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the start address of the memory write transaction.
be_first	std_logic_vector	variable	in	Specifies the byte-enable mask for the first or only DWord
be_last	std_logic_vector	variable	in	Specifies the byte-enable mask for the last DWord
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
data	std_logic_vector	variable	in	Optional parameter. Specifies the value to be written to the DUT.
pload	t_tlp_payload	variable	in	Optional parameter. Specifies an array of DWords to be written to the DUT
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

Examples

Write a single DWord to the specified location.

```
memwr(X"80860000", X"DEAD_BEEF", clk, tlp, sv, rv);
```

Write blocks of increasing size from 1 to 64 contiguously to the target area. Read back and verify the written values. The constant c_membg_4kb_0 is the first of sixteen pre-defined 4 KByte blocks of random data precompiled into the test-case package header.

```
Package Body pcie_x1_tlm_test_case_pkg is
   procedure run_test (signal clk   : in    std_logic;
                       signal sv    : inout t_bfm_stim;
                       signal rv    : in    t_bfm_resp;
                              id    : in    natural := 0) is
      variable v_target_addr  : std_logic_vector(31 downto 0);
   begin
         --Initialisation Sequence (BARs etc.) omitted.

          --Write blocks of increasing size
      v_target_addr := X"8086_0000";
      for i in 0 to 63
         memwr(clk, sv, rv, v_target_addr,
               c_membg_4kb_0(0 to 0 + i));

         v_target_addr  := std_logic_vector(unsigned(v_target_addr) +
                                            ((i + 1) * 4));
      end loop;

              --Read back
      v_target_addr := X"8086_0000";
      for i in 0 to 63
         memrd(clk, sv, rv, v_target_addr,
               c_membg_4kb_0(0 to 0 + i));

         v_target_addr  := std_logic_vector(unsigned(v_target_addr) +
                                            ((i + 1) * 4));
      end loop;
   end;
end package;
```

Notes

The user does not have to be concerned with either the max_payload_size setting in the BFM (see section Fehler: Referenz nicht gefunden on page Fehler: Referenz nicht gefunden) or the 4 KByte address boundary which a PCI Express request is not permitted to cross. Both of these restrictions are automatically handled by the BFM. A single call to the memwr() procedure may result in multiple PCI Express transactions being sent to the the DUT.

3.1.12 Message Request without Payload

```
msg(clk, sv, rv, msg_code, routing := c_route_local_rcv_term,
    hdr_dw2 := X”0000_0000”, hdr_dw3 := X”0000_0000”);
```

Description

Generates a PCI Express message transaction. The transaction is of type Msg, message without payload. For normal use the predefined messages described in sections 3.1.13 to 3.1.16 should be used. The msg() method described here can be used to simulate the insertion of invalid messages.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
hdr_dw2	std_logic_vector	variable	in	Specifies the second DWord in the PCI Express message header. For details please consult the PCI Express literature
hdr_dw3	std_logic_vector	variable	in	Specifies the third DWord in the PCI Express message header. For details please consult the PCI Express literature
msg_code	std_logic_vector	variable	in	Specifies the eight bit PCI Express message code. The input parameter will be resized as required
routing	std_logic_vector	variable	in	Specifies the three bit routing code. The input parameter is resized as required. It is recommended to use the pre-defined constants c_route_broadcast_from_rc, c_route_by_address, c_route_by_id, c_route_forward_ro_rc, c_route_gathered_to_rc, c_route_local_rcv_term
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.

Example

```
msg(clk, sv, rv, c_msg_pme_to, routing => c_route_broadcast_from_rc);
```

3.1.13 Power Management Active State NAK Message

```
msg_pm_active_state_nak(clk, sv, rv);
```

Description

Generates a PCI Express PM_Active_State_Nak message. This message is issued by a root complex if an end-points request to enter the L1 power down state is rejected

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.

Example

```
msg_active_state_nak(clk, sv, rv);
```

3.1.14 Power Management Event Turn Off Message

```
msg_pme_turn_off(clk, sv, rv);
```

Description

Generates a PCI Express Power Management Event turn-off message. This message is issued by a root complex when a system is being powered down to disable power management event messages from downstream components

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.

Example

```
msg_pme_turn_off(clk, sv, rv);
```

3.1.15 Unlock Message

```
msg_unlock(clk, sv, rv);
```

Description

Generates a PCI Express unlock message. This message is issued by a root complex to terminate a locked transaction sequence with a legacy end-point.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.

Example

```
msg_unlock(clk, sv, rv);
```

3.1.16 Vendor Defined Message without Payload

```
msg_vendor_defined(clk, sv, rv,vend_id, vend_dw := X"0000_0000",
                   routing := c_route_local_rcv_term, type_id := 1);
```

Description

Generates a vendor defined PCI Express message without payload. The message type can be specified as type-0 or type-1. A type 0 vendor-defined message must be handled as ERR_CORR or ERR_NONFATAL if it is not supported by the receiver. An unsupported type1 message is silently dropped.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
routing	std_logic_vector	variable	in	Specifies the three bit routing code. The input parameter is resized as required. It is recommended to use the pre-defined constants c_route_broadcast_from_rc, c_route_by_address, c_route_by_id, c_route_forward_ro_rc, c_route_gathered_to_rc, c_route_local_rcv_term
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
type_id	natural	variable	in	Specifies the message type. Valid values are 0 and 1.
vend_dw	std_logic_vector	variable	in	Specifies a user-specific fourth DWord for the message header
vend_id	std_logic_vector	variable	in	Specifies the sixteen-bit vendor id. The input parameter is resized as required.

Example

```
msg_vendor_defined(clk, sv, rv, vend_id => X"8086", type=id => 1);
```

3.1.17 Message Request with Payload

```
msgd(clk, sv, rv, msg_code, data, routing := c_route_local_rcv_term,
     hdr_dw2 := X"0000_0000", hdr_dw3 := X"0000_0000");

msgd(clk, sv, rv, msg_code, payload, routing := c_route_local_rcv_term,
     hdr_dw2 := X"0000_0000", hdr_dw3 := X"0000_0000");
```

Description

Generates a PCI Express message transaction. The transaction is of type MsgD, message without payload. For normal use the predefined messages described in sections 3.1.18 to 3.1.19 should be used. The msgd() method described here can be used to simulate the insertion of invalid messages.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
data	std_logic_vector	variable	in	Optional parameter. Specifies a single DWord value to be written to the DUT.
hdr_dw2	std_logic_vector	variable	in	Specifies the second DWord in the PCI Express message header. For details please consult the PCI Express literature
hdr_dw3	std_logic_vector	variable	in	Specifies the third DWord in the PCI Express message header. For details please consult the PCI Express literature
msg_code	std_logic_vector	variable	in	Specifies the eight bit PCI Express message code. The input parameter will be resized as required
pload	t_tlp_payload	variable	in	Optional parameter. Specifies an array of DWords to be written to the DUT
routing	std_logic_vector	variable	in	Specifies the three bit routing code. The input parameter is resized as required. It is recommended to use the pre-defined constants c_route_broadcast_from_rc, c_route_by_address, c_route_by_id, c_route_forward_ro_rc, c_route_gathered_to_rc, c_route_local_rcv_term
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.

Example

```
msgd(clk, sv, rv, c_msg_set_slot_pwr_limit, routing => c_route_local_rcv_term);
```

3.1.18 Set-Slot-Power-Limit (SSPL) Message

```
msgd_set_slot_power_limit(clk, sv, rv, value := “00”, scale := “0”);
```

Description

A common end-point design error is incorrect handling of the set-slot-power limit message. Most endpoints ignore this message (unsupported request) but the posted header and payload credits must be returned. An end-point must not generate a completion packet in response to this message.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
scale	std_logic_vector	variable	in	Optional parameter. Specifies the scaling factor to apply to the value field
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
svalue	std_logic_vector	variable	in	Optional parameter. Specifies slot power limit value

Example

```
msgd_set_slot_power_limit(clk, sv, rv);
```

3.1.19 Vendor Defined Message with Payload

```
msgd_vendor_defined(clk, sv, rv,vend_id, data, vend_dw := X”0000_0000”,
                    routing := c_route_local_rcv_term, type_id := 1);

msgd_vendor_defined(clk, sv, rv, vend_id, pload, vend_dw := X”0000_0000”,
                    routing := c_route_local_rcv_term, type_id := 1);
```

Description

Generates a vendor defined PCI Express message with payload. The message type can be specified as type-0 or type-1. A type 0 vendor-defined message must be handled as ERR_CORR or ERR_NONFATAL if it is not supported by the receiver. An unsupported type1 message is silently dropped.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
data	std_logic_vector	variable	in	Optional parameter. Specifies a single DWord value to be written to the DUT.
pload	t_tlp_payload	variable	in	Optional parameter. Specifies an array of DWords to be written to the DUT
routing	std_logic_vector	variable	in	Specifies the three bit routing code. The input parameter is resized as required. It is recommended to use the pre-defined constants c_route_broadcast_from_rc, c_route_by_address, c_route_by_id, c_route_forward_ro_rc, c_route_gathered_to_rc, c_route_local_rcv_term
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
type_id	natural	variable	in	Specifies the message type. Valid values are 0 and 1.
vend_dw	std_logic_vector	variable	in	Specifies a user-specific fourth DWord for the message header
vend_id	std_logic_vector	variable	in	Specifies the sixteen-bit vendor id. The input parameter is resized as required.

Example

```
msgd_vendor_defined(clk, sv, rv, X”8086”, X”0102_0304”, type=id => 1);
```

3.1.20 Reading from BFM System Memory

```
sys_memrd(clk, sv, rv, addr, data,
          be_first := b”1111”, no_compare := false);

sys_memrd(clk, sv, rv, addr, pload,
          be_first := b”1111”, be_last := b”1111”,no_compare := false);
```

Description

Generate a read sequence to the BFM system memory. BFM system memory emulates the memory region external to the DUT. It is addressed by the DUT when the DUT performs DMA accesses. The test-case can also read and write to BFM memory allowing communication between the test-case and the DUT.

The first form generates a single DWord read request. The expected value of the DWord returned by the BFM is specified in the data parameter. If the optional check fails, an error message is printed to the output console.

The second form is used to read multiple DWords from the BFM. The pload parameter specifies an array of std_logic_vector elements. The array size determines the request size of the resulting TLP.

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the start address of the BFM system memory region.
be_first	std_logic_vector	variable	in	Optionally specifies the byte-enable mask for the first or only DWord
be_last	std_logic_vector	variable	in	Optionally specifies the byte-enable mask for the last DWord
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
data	std_logic_vector	variable	in	Optional parameter. Specifies the expected value of a single DWord returned from the DUT
no_compare	boolean	variable	in	Specifies whether the data returned by the BFM memory should be checked against the values provided in the data or payload parameters. The default setting is true. Setting this variable to false can be useful for polling one or more memory space locations during simulation
pload	t_tlm_payload	variable	in	Optional parameter. Specifies an array of DWords as expected values
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

3.1.21 Writing to BFM System Memory

```
sys_memwr(clk, sv, rv, addr, data, be_first := b”1111”);
sys_memwr(clk, sv, rv, addr, pload, be_first := b”1111”, be_last := b”1111”);
```

Description

Generate a read sequence to the BFM system memory. BFM system memory emulates the memory region external to the DUT. It is addressed by the DUT when the DUT performs DMA accesses. The test-case can also read and write to BFM memory allowing communication between the test-case and the DUT.

The first form generates a single DWord read request. The expected value of the DWord returned by the BFM is specified in the data parameter. If the optional check fails, an error message is printed to the output console.

The second form is used to read multiple DWords from the BFM. The pload parameter specifies an array of std_logic_vector elements. The array size determines the request size of the resulting TLP.

Parameter Description

Parameter	Type	Mode	Dir.	Description
addr	std_logic_vector	variable	in	Specifies the start address of the BFM system memory region.
be_first	std_logic_vector	variable	in	Optionally specifies the byte-enable mask for the first or only DWord
be_last	std_logic_vector	variable	in	Optionally specifies the byte-enable mask for the last DWord
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
data	std_logic_vector	variable	in	Optional parameter. Specifies the dword value which is written to BFM system memory
pload	t_tlm_payload	variable	in	Optional parameter. Specifies an array of DWords which are written to BFM system memory.
sv	t_bfm_stim	signal	inout	Global handle to control signals in BFM component. Connect to corresponding parameter of parent routine.
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Connect to corresponding parameter of parent routine.

3.1.22 Assigning Additional Completion Credits to the BFM

```
svc_ca_cplx(clk, sv, hdr_credits, data_credits);
```

Description

In the default configuration after reset, the BFM advertises the minimum number of completion credits allowed by the PCI Express specification. These are one completion header and eight completion payload (data) credits. The eight completion credits indicate the capacity to accept one completion with payload having a size of 128 bytes. The user can assign additional completion credits to the BFM using the svc_ca_cplx() command.

Note that only a switch is permitted to advertise non-infinite completion credits. An endpoint or root-complex must always advertise infinite credits. A consequence of this rule is that a root-complex or end-point can only issue a request when local resources are avilable to accept any resulting completions. Wih regard to completion credits, the BFM is playing the role of a possible switch between root complex and end-point.

The command can be issued any time (or multiple times) during a test-case. The user must ensure that in total no more than 127 header credits or 2047 payload credits are issued. This is a requirement of the PCI Express specification. Issuing larger values can lead to packet corruption.

Parameter Description

Parameter	Type	Mode	Dir.	Description
hdr_credits	natural	variable	in	Specifies the number of additional completion header credits assigned to the BFM. This field can be set to zero
data_credits	natural	variable	in	Specifies the number of additional completion payload credits assigned to the BFM. This field can be set to zero
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
sv	t_bfm_stim	signal	inout	Global handle to test-bench control signals. Connect to corresponding parameter of parent routine.

Example

Assign four additional completion header credits and 32 additional completion payload credits to the BFM.

```
svc_ca_cplx(clk, sv, 4, 32);
```

3.1.23 Assigning Additional Non-Posted Credits to the BFM

```
svc_ca_np(clk, sv, hdr_credits, data_credits);
```

Description

In the default configuration after reset, the BFM advertises the minimum number of non-posted credits allowed by the PCI Express specification. These are one non-posted header and eight non-posted payload (data) credits. The eight non-posted credits indicate the capacity to accept one non-posted payload having a size of 128 bytes. The user can assign additional non-posted credits to the BFM using the svc_ca_np() command.

Note that non-posted payloads are only associated with configuration or I/O space write transactions. These are limited to a length of one DWord by the PCI Express specification.

The command can be issued any time (or multiple times) during a test-case. The user must ensure that in total no more than 127 header credits or 2047 payload credits are issued. This is a requirement of the PCI Express specification. Issuing larger values can lead to packet corruption.

Parameter Description

Parameter	Type	Mode	Dir.	Description
hdr_credits	natural	variable	in	Specifies the number of additional non-posted header credits assigned to the BFM. This field can be set to zero
data_credits	natural	variable	in	Specifies the number of additional non-posted payload credits assigned to the BFM. This field can be set to zero
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
sv	t_bfm_stim	signal	inout	Global handle to test-bench control signals. Connect to corresponding parameter of parent routine.

Example

Assign four additional non-posted header credits and 32 additional non-posted payload credits to the BFM.

```
svc_ca_np(clk, sv, 4, 32);
```

3.1.24 Assigning Additional Posted Credits to the BFM

```
svc_ca_p(clk, sv, hdr_credits, data_credits);
```

Description

In the default configuration after reset, the BFM advertises the minimum number of posted credits allowed by the PCI Express specification. These are one posted header and eight posted payload (data) credits. The eight posted credits indicate the capacity to accept one posted payload having a size of 128 bytes. A posted payload is only associated with a memory write transaction. If the DUT performs DMA writes to the BFM, the minimum settings will generally not be sufficient for reasonable performance. The user can assign additional posted credits to the BFM using the svc_ca_p() command.

This command can be issued any time (or multiple times) during a test-case. The user must ensure that in total no more than 127 header credits or 2047 payload credits are issued. This is a requirement of the PCI Express specification. Issuing larger values can lead to packet corruption.

Parameter Description

Parameter	Type	Mode	Dir.	Description
hdr_credits	natural	variable	in	Specifies the number of additional posted header credits assigned to the BFM. This field can be set to zero
data_credits	natural	variable	in	Specifies the number of additional posted payload credits assigned to the BFM. This field can be set to zero
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
sv	t_bfm_stim	signal	inout	Global handle to test-bench control signals. Connect to corresponding parameter of parent routine.

Example

Assign eight additional posted header credits and 24 additional posted payload credits to the BFM.

```
svc_ca_p(clk, sv, 8, 24);
```

3.1.25 Assigning a Unique ID (Bus No. / Function No. / Device No.) to the DUT

```
set_dut_id(sv, bus_no, dev_no, func_no);
```

Description

The set_dut_id() procedure assigns a unique ID to the DUT. The unique ID consists of a bus number, device number and function number and is transferred to the DUT with the next type-0 configuration space write cfgwr0() call. The bus number is an eight bit field, the device number a five bit field and the function number a three bit field. Larger parameter values are automatically truncated without warning. During truncation, the upper bits are lost and the lower bits retained.

The DUT will use the unique ID as the completer ID in all successive completion packets. Typically this is one of the first procedure calls in a test-case and should be issued before sending any PCI Express request to the DUT.

Note that the function number should always be set to zero for a non multi-function device.

Parameter Description

Parameter	Type	Mode	Dir.	Description
bus_no	natural	variable	in	Specifies the PCI Express bus number assigned to the DUT. The bus number can be in the range 0 to 255. Larger values are automatically truncated without warning
dev_no	natural	variable	in	Specifies the PCI Express device number assigned to the DUT. The device number can be in the range 0 to 31. Larger values are automatically truncated without warning.
func_no	natural	variable	in	Specifies the PCI Express function number assigned to the DUT. The function number can be in the range 0 to 7. Larger values are automatically truncated without warning.
sv	t_bfm_stim	signal	inout	Global handle to test-bench control signals. Connect to corresponding parameter of parent routine.

Example

Assign a completer ID to the DUT

```
set_dut_id(sv, 16#42#, 5, 0);
```

3.1.26 Querying the Credits Advertised by the DUT

```
show_credits(rv);
```

Description

This task can be used to display the status of the credits displayed by the DUT. This task is useful for verifying that all incoming requests, including unsupported requests, have been correctly handled and all credits consumed by the BFM have been re-advertised. If this task is called at the end of simulation, the values displayed should match the values configured for the DUT in the IPexpress mask.

Parameter Description

Parameter	Type	Mode	Dir.	Description
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Referenced by most routines

Example

```
show_credits(rv);
```

3.1.27 Waiting for all outstanding Completions

```
wait_all_cplx_pending(clk, rv);
```

Description

The wait_all_cplx_pending procedure suspends the test-case program flow until all outstanding non-posted requests (memrd(), I/O-space read or write, configuration space read or write) have been completed by the DUT. It is typically used to synchronise the BFM to the DUT after a test section.

If there are no outstanding requests, the procedure will return immediately allowing the test-case program flow to continue normally.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
rv	t_bfm_resp	signal	in	Global handle to status signals in BFM component. Referenced by most routines

Example

Issue two memory read requests to the DUT and wait until all outstanding completions have been returned.

```
memrd(clk, sv, rv, X”1234_0000”, c_membg_4kb_0(0 to 31), cpl_wait => false);
memrd(clk, sv, rv, X”9876_0000”, c_membg_4kb_1(0 to 63), cpl_wait => false);
wait_all_cplx_pending(clk, rv);
```

3.1.28 Accessing the Contents of the Last Completion Packet

```
get_cpl_buffer(rv, index := 0) return std_logic_vector;
```

Description

The get_cpl_buffer() function can be used to access the contents of the last completion packet returned by the DUT. The index parameter selects the DWord from the completion packet. This call is typically used when DUT memory region.

Note that this is a function call which returns a std_logic_vector parameter.

Parameter Description

Parameter	Type	Mode	Dir.	Description
index	natural	variable	in	Index selecting a single DWord from the last completion packet
rv	t_bfm_resp	variable	in	Global handle to BFM status signals.
return	std_logic_vector	variable	out	The return value form the function call is a thirty-two bit std_logic_vector

Example

Read the first DWord from the PCI Express capability structure and verify that the structure ID is correct. The constant c_csreg_pcie_cap evaluates to the configuration space address 0x090, which is the location of the PCI Express capability structure in the Lattice PCI Express core.

```
variable v_test_reg : std_logic_vector(31 downto 0);
    .  .  . .
begin
    .  .  .  .
cfgrd0(c_csreg_pcie_cap, clk, tlp, sv, rv, cple, c_wait_cpl);
v_test_reg := get_cpl_buffer(rv);
assert (v_test_reg(7 downto 0) = X"10")
       report "PCIexpress Capability Structure not found"
       severity error;
```

3.1.29 Accessing the Contents of the BFM Memory Read Buffer

```
get_mem_buffer(rv, index := 0) return std_logic_vector;
```

Description

The get_mem_buffer() function can be used to access the contents of buffer filled by the last syss_memrd() call. The index parameter selects the DWord from the BFM buffer.

Note that this is a function call which returns a std_logic_vector parameter.

Parameter Description

Parameter	Type	Mode	Dir.	Description
index	natural	variable	in	Index selecting a single DWord from the buffer filled by the last read access from BFM system memory.
rv	t_bfm_resp	variable	in	Global handle to BFM status signals.
return	std_logic_vector	variable	out	The return value form the function call is a thirty-two bit std_logic_vector

Example

Read the seventh DWord from the PCI Express capability structure and verify that the low byte has the value X”42”.

```
variable v_test_reg : std_logic_vector(31 downto 0);
    .  .  . .
begin
    .  .  .  .
sys_memrd(clk, sv, rv, X”FEDC_7200”, X”00000000”, no_compare => true);
v_test_reg := get_mem_buffer(rv, 7);
assert (v_test_reg(7 downto 0) = X"42")
       report "Memory Value Incorrect"
       severity error;
```

3.1.30 Idle Task

```
idle(clk, count := 1);
```

Description

The idle() task can be used to insert a defined number of wait states (clock cycles) in the test program flow.

Parameter Description

Parameter	Type	Mode	Dir.	Description
clk	std_logic	signal	in	Global clock. Connect to corresponding parameter of parent routine
count	natural	variable	in	Specifies the number of idle cycles

Example

```
idle(clk, 127);
```

3.1.31 Predefined Constants

The following text block lists the predefined constants available in VHDL mode.

```
   -- PCI Config. Space: Status / Command Register Bits
constant c_bsel_bus_mst_en            : std_logic_vector(31 downto 0) := X"00000004";
constant c_bsel_cap_list_present      : std_logic_vector(31 downto 0) := X"00100000";
constant c_bsel_intx_disable          : std_logic_vector(31 downto 0) := X"00000400";
constant c_bsel_intx_pending          : std_logic_vector(31 downto 0) := X"00080000";
constant c_bsel_io_space_en           : std_logic_vector(31 downto 0) := X"00000001";
constant c_bsel_mem_space_en          : std_logic_vector(31 downto 0) := X"00000002";

   -- PCI Config. Space: MSI Capability Structure.
   --                    Enable / Config settings
constant c_bsel_msi_en                : std_logic_vector(31 downto 0) := X"00010000"
constant c_bsel_msi_mm_1              : std_logic_vector(31 downto 0) := X"00000000"
constant c_bsel_msi_mm_2              : std_logic_vector(31 downto 0) := X"00100000"
constant c_bsel_msi_mm_4              : std_logic_vector(31 downto 0) := X"00200000"
constant c_bsel_msi_mm_8              : std_logic_vector(31 downto 0) := X"00300000"

   -- PCI Express Completion Status Codes
constant c_cpl_sta_ca                 : std_logic_vector(2 downto 0) := "100";
constant c_cpl_sta_crs                : std_logic_vector(2 downto 0) := "010";
constant c_cpl_sta_sc                 : std_logic_vector(2 downto 0) := "000";
constant c_cpl_sta_ur                 : std_logic_vector(2 downto 0) := "001";

   -- PCI Express Configuration Space Register Addresses
constant c_csreg_vend_id              : std_logic_vector(11 downto 0) := X"000";
constant c_csreg_dev_id               : std_logic_vector(11 downto 0) := X"002";
constant c_csreg_command              : std_logic_vector(11 downto 0) := X"004";
constant c_csreg_status               : std_logic_vector(11 downto 0) := X"006";
constant c_csreg_rev_id               : std_logic_vector(11 downto 0) := X"008";
constant c_csreg_class_code_high      : std_logic_vector(11 downto 0) := X"00A";
constant c_csreg_bar0                 : std_logic_vector(11 downto 0) := X"010";
constant c_csreg_bar1                 : std_logic_vector(11 downto 0) := X"014";
constant c_csreg_bar2                 : std_logic_vector(11 downto 0) := X"018";
constant c_csreg_bar3                 : std_logic_vector(11 downto 0) := X"01C";
constant c_csreg_bar4                 : std_logic_vector(11 downto 0) := X"020";
constant c_csreg_bar5                 : std_logic_vector(11 downto 0) := X"024";
constant c_csreg_cis_ptr              : std_logic_vector(11 downto 0) := X"028";
constant c_csreg_subs_vend_id         : std_logic_vector(11 downto 0) := X"02C";
constant c_csreg_subs_dev_id          : std_logic_vector(11 downto 0) := X"02E";
constant c_csreg_expansion_rom        : std_logic_vector(11 downto 0) := X"030";
constant c_csreg_cap_ptr              : std_logic_vector(11 downto 0) := X"034";
   -- The following will vary for different PCIe Cores
constant c_csreg_pmcsr                : std_logic_vector(11 downto 0) := X"054";
constant c_csreg_msi_cap              : std_logic_vector(11 downto 0) := X"070";
constant c_csreg_msi_control          : std_logic_vector(11 downto 0) := X"072";
constant c_csreg_msi_addr_low         : std_logic_vector(11 downto 0) := X"074";
constant c_csreg_msi_addr_high        : std_logic_vector(11 downto 0) := X"078";
constant c_csreg_msi_msg_data         : std_logic_vector(11 downto 0) := X"07C";
constant c_csreg_pcie_cap             : std_logic_vector(11 downto 0) := X"090";
constant c_csreg_pcie_dev_ctrl        : std_logic_vector(11 downto 0) := X"098";
constant c_csreg_pcie_dev_stat        : std_logic_vector(11 downto 0) := X"09A";
constant c_csreg_pcie_link_ctrl       : std_logic_vector(11 downto 0) := X"0A0";
constant c_csreg_pcie_link_stat       : std_logic_vector(11 downto 0) := X"0A2";

   -- PCI Express permissible values for Max. Payload Size
constant c_max_payload_128            : std_logic_vector(2 downto 0) := "000";
constant c_max_payload_256            : std_logic_vector(2 downto 0) := "001";
constant c_max_payload_512            : std_logic_vector(2 downto 0) := "010";
constant c_max_payload_1024           : std_logic_vector(2 downto 0) := "011";
constant c_max_payload_2048         : std_logic_vector(2 downto 0) := "100";
constant c_max_payload_4096         : std_logic_vector(2 downto 0) := "101";

   -- PCI Express permissible values for Max. Read-Request Size
constant c_max_read_req_128         : std_logic_vector(2 downto 0) := "000";
constant c_max_read_req_256         : std_logic_vector(2 downto 0) := "001";
constant c_max_read_req_512         : std_logic_vector(2 downto 0) := "010";
constant c_max_read_req_1024        : std_logic_vector(2 downto 0) := "011";
constant c_max_read_req_2048        : std_logic_vector(2 downto 0) := "100";
constant c_max_read_req_4096        : std_logic_vector(2 downto 0) := "101";

   -- PCI Express Pre-defined Message Codes
constant c_msg_err_corr             : std_logic_vector(7 downto 0) := X"30";
constant c_msg_err_fatal            : std_logic_vector(7 downto 0) := X"33";
constant c_msg_err_non_fatal        : std_logic_vector(7 downto 0) := X"31";
constant c_msg_inta_assert          : std_logic_vector(7 downto 0) := X"20";
constant c_msg_inta_deassert        : std_logic_vector(7 downto 0) := X"24";
constant c_msg_intb_assert          : std_logic_vector(7 downto 0) := X"21";
constant c_msg_intb_deassert        : std_logic_vector(7 downto 0) := X"25";
constant c_msg_intc_assert          : std_logic_vector(7 downto 0) := X"22";
constant c_msg_intc_deassert        : std_logic_vector(7 downto 0) := X"26";
constant c_msg_intd_assert          : std_logic_vector(7 downto 0) := X"23";
constant c_msg_intd_deassert        : std_logic_vector(7 downto 0) := X"27";
constant c_msg_pm_as_nak            : std_logic_vector(7 downto 0) := X"14";
constant c_msg_pm_pme               : std_logic_vector(7 downto 0) := X"18";
constant c_msg_pm_pme_to            : std_logic_vector(7 downto 0) := X"19";
constant c_msg_pm_pme_to_ack        : std_logic_vector(7 downto 0) := X"1B";
constant c_msg_set_slot_pwr_limit   : std_logic_vector(7 downto 0) := X"50";
constant c_msg_unlock               : std_logic_vector(7 downto 0) := X"00";
constant c_msg_vendor_defined_0     : std_logic_vector(7 downto 0) := X"7E";
constant c_msg_vendor_defined_1     : std_logic_vector(7 downto 0) := X"7F";

      -- Routing Method Specifiers
constant c_route_broadcast_from_rc  : std_logic_vector(7 downto 0) := X"03";
constant c_route_by_address         : std_logic_vector(7 downto 0) := X"01";
constant c_route_by_id              : std_logic_vector(7 downto 0) := X"02";
constant c_route_forward_to_rc      : std_logic_vector(7 downto 0) := X"00";
constant c_route_gathered_to_rc     : std_logic_vector(7 downto 0) := X"05";
constant c_route_local_rcv_term     : std_logic_vector(7 downto 0) := x"04";

   -- PCI Express Commands
constant c_tlp_mrd32                : std_logic_vector(7 downto 0) := X"00";
constant c_tlp_mrd64                : std_logic_vector(7 downto 0) := X"20";
constant c_tlp_mrdlk32              : std_logic_vector(7 downto 0) := X"01";
constant c_tlp_mrdlk64              : std_logic_vector(7 downto 0) := X"21";
constant c_tlp_mwr32                : std_logic_vector(7 downto 0) := X"40";
constant c_tlp_mwr64                : std_logic_vector(7 downto 0) := X"60";
constant c_tlp_iord                 : std_logic_vector(7 downto 0) := X"02";
constant c_tlp_iowr                 : std_logic_vector(7 downto 0) := X"42";
constant c_tlp_cfgrd0               : std_logic_vector(7 downto 0) := X"04";
constant c_tlp_cfgwr0               : std_logic_vector(7 downto 0) := X"44";
constant c_tlp_cfgrd1               : std_logic_vector(7 downto 0) := X"05";
constant c_tlp_cfgwr1               : std_logic_vector(7 downto 0) := X"45";
constant c_tlp_msg                  : std_logic_vector(7 downto 0) := X"30";
constant c_tlp_msgd                 : std_logic_vector(7 downto 0) := X"70";
constant c_tlp_cpl                  : std_logic_vector(7 downto 0) := X"0A";
constant c_tlp_cpld                 : std_logic_vector(7 downto 0) := X"4A";
constant c_tlp_cpllk                : std_logic_vector(7 downto 0) := X"0B";
constant c_tlp_cpldlk               : std_logic_vector(7 downto 0) := X"4B";
```

4 Verilog Programming Interface

4.1 Main Test Task

The main test task must always be implemented as a 'portless' module named 'pcie_vlog_test_case'. As in the case of the VHDL interface described in chapter 3, a test case must be compiled into the 'pcie_bfm_lib' library. The test case is typically contained in a verilog 'initial' task.

The test case must reference the BFM API by including the 'bfm_lspcie_rc_tlm_lib.v' library. This library in turn includes the 'bfm_lspcie_rc_constants.v' file which contains useful constants particularly for accessing configuration space registers

Main Test Declaration

```
`timescale 1ns / 1 ps
module pcie_vlog_test_case;

   `include "bfm_lspcie_rc_tlm_lib.v"

   initial begin

         // Assign additional posted credits to the BFM
         // default is 1 posted Header + 8 posted data
      svc_ca_p(7, 56);

         // PCI Spec. First access must be write
      cfgwr0(c_csreg_bar0, 'hffffffff, c_be_all, c_wait_cpl, c_cpl_sta_sc);
      cfgwr0(c_csreg_bar2, 'hffffffff, c_be_all, c_wait_cpl, c_cpl_sta_sc);

         // Get Resource requirements
      cfgrd0(c_csreg_bar0, 'hfff8000c, c_be_all, c_wait_cpl, c_cpl_sta_sc);
      cfgrd0(c_csreg_bar2, 'hfff80004, c_be_all, c_wait_cpl, c_cpl_sta_sc);

           . . . . . . .           . . . . .

         // Disable legacy interrupts and enable/configure MSI
      cfgwr0(c_csreg_command,
             c_bsel_bus_mst_en | c_bsel_mem_space_en | c_bsel_intx_disable,
             c_be_all, c_wait_cpl, c_cpl_sta_sc);
      cfgwr0(c_csreg_msi_addr_low, 'hffffedac,
             c_be_all, c_wait_cpl, c_cpl_sta_sc);
      cfgwr0(c_csreg_msi_msg_data, 'h12304560,
             c_be_all, c_wait_cpl, c_cpl_sta_sc);
      cfgwr0(c_csreg_msi_control, c_bsel_msi_mm_8 | c_bsel_msi_en,
             c_be_all, c_no_wait_cpl, c_cpl_sta_sc);
      wait_all_cplx_pending();

         // Memory write and read
      memwr('h45657ff8, {32'hdeadbeef, 32'80804711}, 2, c_be_all, c_be_all);

      memrd('h45657ff8, {32'hdeadbeef, 32'80804711}, 2,
            c_be_all, c_be_all, c_wait_cpl, c_cpl_sta_sc);
      idle(1024);
      $finish;
   end
endmodule
```

4.1.1 Verilog Path to Test-Case Module

The BFM should be instantiated in place of the Serdes unit in the PCIexpress core when used with a Lattice FPGA. Typically, this requires two separate make files for synthesis and simulation which differ by just this one unit. In both cases the Serdes is called pcs_pipe_top.

The verilog path to the test-case module is

```
<path_to_serdes>.U1_BFM.U4_SG_VLOG.U_TEST_CASE
```

4.1.2 Type-0 Configuration Space Read Sequence

```
cfgrd0(addr, pload, be_first, cpl_wait, cpl_sta);
```

Description

Generate a type 0 configuration space read cycle. The twelve bit register address is passed in the first parameter. Addresses larger than twelve bits will be truncated. The expected value of the returned data double-word is passed in the second parameter.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[11:0]	in	Configuration space register address
be_first	vector [3:0]	in	Specifies the byte enable configuration
cpl_sta	vector[2:0]	in	Specifies the expected completion response form the DUT
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT
pload	vector[31:0]	in	Expected value of data double-word returned by the DUT

Example

```
cfgrd0(c_csreg_vend_id, 'h47110815, c_be_all, c_wait_cpl, c_cpl_sta_ur);
cfgrd0(12'h078, 'h12345678, c_be_all, c_wait_cpl, c_cpl_sta_ur);
```

4.1.3 Type-0 Configuration Space Write Sequence

```
cfgwr0(addr, pload, be_first, cpl_wait, cpl_sta);
```

Description

Generate a type 0 configuration space write cycle. The twelve bit register address is passed in the first parameter. Addresses larger than twelve bits will be truncated.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[11:0]	in	Configuration space register address
be_first	vector [3:0]	in	Specifies the byte enable configuration
cpl_sta	vector[2:0]	in	Specifies the expected completion response form the DUT
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT
pload	vector[31:0]	in	Data double-word written to the DUT

Example

```
cfgwr0(c_csreg_bar0, 'hffffffff, c_be_all, c_cpl_wait, c_cpl_sta_sc);
```

4.1.4 Type-1 Configuration Space Read Sequence

```
cfgrd1(addr, pload, be_first, cpl_wait, cpl_sta);
```

Description

Generate a type-1 configuration space read cycle. The twelve bit register address is passed in the first parameter. Addresses larger than twelve bits will be truncated. The expected value of the returned data double-word is passed in the second parameter.

Note that it is a protocol error if a PCI Express endpoint is the target of a type-1 configraiton space access. Type-1 configuration space accesses should only be directed at a PCI or PCI Express bridge. An end-point which is the target of a type-1 configuration space access should always return a status of 'unsupported request'

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[11:0]	in	Configuration space register address
be_first	vector [3:0]	in	Specifies the byte enable configuration
cpl_sta	vector[2:0]	in	Specifies the expected completion response form the DUT
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT
pload	vector[31:0]	in	Expected value of data double-word returned by the DUT

Example

```
cfgrd1(c_csreg_vend_id, 'h47110815, c_be_all, c_wait_cpl, c_cpl_sta_ur);
cfgrd1(12'h078, 'h12345678, c_be_all, c_wait_cpl, c_cpl_sta_ur);
```

4.1.5 Type-1 Configuration Space Write Sequence

```
cfgwr1(addr, pload, be_first, cpl_wait, cpl_sta);
```

Description

Generate a type-1 configuration space write cycle. The twelve bit register address is passed in the first parameter. Addresses larger than twelve bits will be truncated.

Note that it is a protocol error if a PCI Express endpoint is the target of a type-1 configigraiton space access. Type-1 configuration space accesses should only be directed at a PCI or PCI Express bridge. An end-point which is the target of a type-1 configuration space access should always return a status of 'unsupported request'

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[11:0]	in	Configuration space register address
be_first	vector [3:0]	in	Specifies the byte enable configuration
cpl_sta	vector[2:0]	in	Specifies the expected completion response form the DUT
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT
pload	vector[31:0]	in	Data double-word written to the DUT

Example

```
cfgwr1(c_csreg_bar0, 'hffffffff, c_be_all, c_cpl_wait, c_cpl_sta_sc);
```

4.1.6 Completion without Payload

```
cpl(req_id, tag);
```

Description

Generate a completion sequence without payload. This method is intended for testing the DUT response to an unexpected completion. Completion packets in response to a non-posted request from the DUT (e.g. DMA memory read) are automatically generated by the BFM.

Parameter Description

Parameter	Type	Dir.	Description
req_id	vector[15:0]	in	Value to use as requestor ID. Bits [15:8] are the bus number, bits [7:3] the device number and bits [2:0] the function number.
tag	vector [7:0]	in	Specifies the tag to use in the completion packet

Example

```
cpl(16'h0004, 0);
```

4.1.7 Completion with Payload

```
cpld(pload, length, req_id, tag);
```

Description

Generate a completion sequence with payload. This method is intended for testing the DUT response to an unexpected completion. Completion packets in response to a non-posted request from the DUT (e.g. DMA memory read) are automatically generated by the BFM.

Parameter Description

Parameter	Type	Dir.	Description
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord
pload	vector[32767:0]	in	Data payload which is written to the DUT
req_id	vector[15:0]	in	Value to use as requestor ID. Bits [15:8] are the bus number, bits [7:3] the device number and bits [2:0] the function number.
tag	vector [7:0]	in	Specifies the tag to use in the completion packet

Example

```
cpld({32'h0, 32'h01020304, 32'h02030405}, 3, 16'h0004, 0);
```

4.1.8 I/O Space Read Sequence

```
iord(addr, pload, be_first, cpl_wait, cpl_sta);
```

Description

Generate a PCIexpress I/O space read cycle. The thirty-two bit register address is passed in the first parameter, the expected value of the returned data double-word is passed in the second parameter.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[31:0]	in	I/O space register address
be_first	vector [3:0]	in	Specifies the byte enable configuration
cpl_sta	vector[2:0]	in	Specifies the expected completion response form the DUT
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT
pload	vector[31:0]	in	Expected value of data double-word returned by the DUT

Example

```
iord('habcdba94, 'h47110815, c_be_all, c_wait_cpl, c_cpl_sta_sc);
iord('hfedcba94, 'h47110815, c_be_all, c_wait_cpl, c_cpl_sta_ur);
```

4.1.9 I/O Space Write Sequence

```
iowr(addr, pload, be_first, cpl_wait, cpl_sta);
```

Description

Generate a PCIexpress I/O space write cycle. The thirty-two bit register address is passed in the first parameter, the data double-word is passed in the second parameter.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[31:0]	in	I/O space register address
be_first	vector [3:0]	in	Specifies the byte enable configuration
cpl_sta	vector[2:0]	in	Specifies the expected completion response form the DUT
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT
pload	vector[31:0]	in	Data double-word written to the DUT

Example

```
iowr('hfedcba94, 'h87654321, c_be_all, c_wait_cpl, c_cpl_sta_sc);
```

4.1.10 Memory Space Read Sequence

```
memrd(addr, pload, length, be_first, be_last, cpl_wait, cpl_sta);
```

Description

Generate a PCIexpress memory space locked read cycle. The sixty-four bit start address of the memory region is passed in the first parameter, the expected values of the returned data are passed in the second parameter. The payload comparison data is left extended to 32768 bits representing the maximum permissible PCIexpress payload size of 1024 DWords.

If the upper thirty address bits are omitted or are all zero, the BFM will automatically generate a three double-word transaction header. Otherwise, a four double-word transaction header will be generated.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[63:0]	in	Memory space start address
be_first	vector[3:0]	in	Specifies the byte enable configuration for the fist DWord
be_last	vector[3:0]	in	Specifies the byte enable configuration for the last DWord
cpl_sta	vector[2:0]	in	Specifies the expected completion response from the DUT
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord
pload	vector[32767:0]	in	Compare data against which the received data is optionally compared.

Example

```
   // Expect 2 Dwords, the first being 0x01234567
memrd('h84200000, {32'h01234567, 32'h89abcdef}, 2,
      c_be_all, c_be_all, c_cpl_wait, c_cpl_sta_sc);

   // expect 16 DWords, all having the value zero
memrd('h02468aec, 0, 16,
      c_be_all, c_be_all, c_cpl_wait, c_cpl_sta_sc);
```

4.1.11 Locked Memory Read Sequence

```
memrd_lk(addr, pload, length, be_first, be_last, cpl_wait, cpl_sta);
```

Description

Generate a PCIexpress memory space read cycle. The sixty-four bit start address of the memory region is passed in the first parameter, the expected values of the returned data are passed in the second parameter. The payload comparison data is left extended to 32768 bits representing the maximum permissible PCIexpress payload size of 1024 DWords.

If the upper thirty address bits are omitted or are all zero, the BFM will automatically generate a three double-word transaction header. Otherwise, a four double-word transaction header will be generated.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[63:0]	in	Memory space start address
be_first	vector[3:0]	in	Specifies the byte enable configuration for the fist DWord
be_last	vector[3:0]	in	Specifies the byte enable configuration for the last DWord
cpl_sta	vector[2:0]	in	Specifies the expected completion response from the DUT
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord
pload	vector[32767:0]	in	Compare data against which the received data is optionally compared.

Example

```
   // Expect 2 Dwords, the first being 0x01234567
memrd_lk('h84200000, {32'h01234567, 32'h89abcdef}, 2,
         c_be_all, c_be_all, c_cpl_wait, c_cpl_sta_sc);

   // expect 16 DWords, all having the value zero
memrd_lk('h02468aec, 0, 16,
         c_be_all, c_be_all, c_cpl_wait, c_cpl_sta_sc);
```

4.1.12 Memory Space Write Sequence

```
memwr(addr, pload, length, be_first, be_last);
```

Description

Generate a PCIexpress memory space write cycle. The sixty-four bit start address of the memory region is passed in the first parameter, the expected values of the returned data are passed in the second parameter. The payload data is left extended to 32768 bits representing the maximum permissible PCIexpress payload size of 1024 DWords.

If the upper thirty address bits are omitted or are all zero, the BFM will automatically generate a three double-word transaction header. Otherwise, a four double-word transaction header will be generated.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[63:0]	in	Memory space start address
be_first	vector[3:0]	in	Specifies the byte enable configuration for the fist DWord
be_last	vector[3:0]	in	Specifies the byte enable configuration for the last DWord
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord
pload	vector[32767:0]	in	Data payload which is written to the DUT

Example

```
memwr('h84200000, {32'h01234567, 32'h89abcdef}, 2,
      c_be_all, c_be_all);
memwr('h02468aec, 0, 16, c_be_all, c_be_all);
```

4.1.13 Message Request without Payload

```
msg(msg_code, routing, hdr_dw2, hdr_dw3);
```

Description

Generates a PCI Express message transaction. The transaction is of type Msg, message without payload. For normal use the predefined messages described in sections 4.1.14 to 4.1.17 should be used. The msg() method described here can be used to simulate the insertion of invalid messages.

Parameter Description

Parameter	Type	Dir.	Description
hdr_dw2	vector[31:0]	in	Specifies the second DWord in the PCI Express message header. For details please consult the PCI Express literature
hdr_dw3	vector[31:0]	in	Specifies the third DWord in the PCI Express message header. For details please consult the PCI Express literature
msg_code	vector[7:0]	in	Specifies the eight bit PCI Express message code.
routing	vector[2:0]	in	Specifies the three bit routing code. It is recommended to use the pre-defined constants c_route_broadcast_from_rc, c_route_by_address, c_route_by_id, c_route_forward_ro_rc, c_route_gathered_to_rc, c_route_local_rcv_term

Example

```
msg(c_msg_pme_to, rc_route_broadcast_from_rc, 0, 0);
```

4.1.14 Power Management Active State NAK Message

```
msg_pm_active_state_nak();
```

Description

Generates a PCI Express PM_Active_State_Nak message. This message is issued by a root complex if an end-points request to enter the L1 power down state is rejected

Parameter Description

This BFM method does not have parameters

Example

```
msg_pm_active_state_nak;
```

4.1.15 Power Management PME Turn Off Message

```
msg_pme_turn_off();
```

Description

Generates a PCI Express Power Management Event turn-off message. This message is issued by a root complex when a system is being powered down to disable power management event messages from downstream components

Parameter Description

This BFM method does not have parameters

Example

```
msg_pme_turn_off;
```

4.1.16 Unlock Message

```
msg_unlock();
```

Description

Generates a PCI Express unlock message. This message is issued by a root complex to terminate a locked transaction sequence with a legacy end-point.

Parameter Description

This BFM method does not have parameters

Example

```
msg_unlock;
```

4.1.17 Vendor Defined Message without Payload

```
msg_vendor_defined(vend_id, vend_dw, routing, type_id);
```

Description

Generates a vendor defined PCI Express message without payload. The message type can be specified as type-0 or type-1. A type 0 vendor-defined message must be handled as ERR_CORR or ERR_NONFATAL if it is not supported by the receiver. An unsupported type1 message is silently dropped.

Parameter Description

Parameter	Type	Dir.	Description
routing	vector[2:0]	in	Specifies the three bit routing code. It is recommended to use the pre-defined constants c_route_broadcast_from_rc, c_route_by_address, c_route_by_id, c_route_forward_ro_rc, c_route_gathered_to_rc, c_route_local_rcv_term
type_id	scalar	in	If this paramter is set to 1, a type-1 message will be generated. Otherwise, a type-0 message is generated.
vend_dw	vector[31:0]	in	Specifies a user-specific fourth DWord for the message header
vend_id	vector[15:0]	in	Specifies the message vendor ID.

Example

```
msg_vendor_defined(16'h8086, 32'b0, rc_route_broadcast_from_rc, 1'b0);
```

4.1.18 Message Request with Payload

```
msgd(msg_code, pload, length, routing, hdr_dw2, hdr_dw3);
```

Description

Generates a PCI Express message transaction. The transaction is of type MsgD, message with payload. For normal use the predefined messages described in sections 4.1.19 to 4.1.20 should be used. The msgD() method described here can be used to simulate the insertion of invalid messages.

Parameter Description

Parameter	Type	Dir.	Description
hdr_dw2	vector[31:0]	in	Specifies the second DWord in the PCI Express message header. For details please consult the PCI Express literature
hdr_dw3	vector[31:0]	in	Specifies the third DWord in the PCI Express message header. For details please consult the PCI Express literature
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord
msg_code	vector[7:0]	in	Specifies the eight bit PCI Express message code.
pload	vector[32767:0]	in	Data payload which is written to the DUT
routing	vector[2:0]	in	Specifies the three bit routing code. It is recommended to use the pre-defined constants c_route_broadcast_from_rc, c_route_by_address, c_route_by_id, c_route_forward_ro_rc, c_route_gathered_to_rc, c_route_local_rcv_term

Example

```
msgd(c_msg_set_slot_pwr_limit, 'h0201, 1, rc_route_local_rcv_term, 0, 0);
```

4.1.19 Set-Slot-Power-Limit (SSPL) Message

```
msgd_set_slot_power_limit(value, scale);
```

Description

A common end-point design error is incorrect handling of the set-slot-power limit message. Most endpoints ignore this message (unsupported request) but the posted header and payload credits must be returned. An end-point must not generate a completion packet in response to this message.

Parameter Description

Parameter	Type	Dir.	Description
value	vector[7:0]	in	Specifies the power-limit value
scale	vector[1:0]	in	Specifies the scaling factor to use with the value field

Example

```
msgd_set_slot_power_limit(0, 0);
```

4.1.20 Vendor Defined Message with Payload

```
msgd_vendor_defined(vend_id, pload, length, vend_dw, routing, type_id);
```

Description

Generates a vendor defined PCI Express message with payload. The message type can be specified as type-0 or type-1. A type 0 vendor-defined message must be handled as ERR_CORR or ERR_NONFATAL if it is not supported by the receiver. An unsupported type1 message is silently dropped.

Parameter Description

Parameter	Type	Dir.	Description
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord
pload	vector[32767:0]	in	Data payload which is written to the DUT
routing	vector[2:0]	in	Specifies the three bit routing code. It is recommended to use the pre-defined constants c_route_broadcast_from_rc, c_route_by_address, c_route_by_id, c_route_forward_ro_rc, c_route_gathered_to_rc, c_route_local_rcv_term
type_id	scalar	in	If this paramter is set to 1, a type-1 message will be generated. Otherwise, a type-0 message is generated.
vend_dw	vector[31:0]	in	Specifies a user-specific fourth DWord for the message header
vend_id	vector[15:0]	in	Specifies the message vendor ID.

Example

```
msgd_vendor_defined(16'h8086, {32'h0, 32'h01020304}, 2, 32'b0,
                       rc_route_broadcast_from_rc, 1'b1);
```

4.1.21 Reading from BFM System Memory

```
sys_memrd(addr, pload, length, be_first, be_last);
```

Description

Generate a read sequence to the BFM system memory. BFM system memory emulates the memory region external to the DUT. It is addressed by the DUT when the DUT performs DMA accesses. The test-case can also read and write to BFM memory allowing communication between the test-case and the DUT.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[63:0]	in	Memory space start address
be_first	vector[3:0]	in	Specifies the byte enable configuration for the fist DWord
be_last	vector[3:0]	in	Specifies the byte enable configuration for the last DWord
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord
pload	vector[32767:0]	in	Compare data against which the received data is optionally compared.

Example

```
   // Expect 2 Dwords, the first being 0x01234567
sys_memrd('h84200000, {32'h01234567, 32'h89abcdef}, 2,
          c_be_all, c_be_all);

   // expect 16 DWords, all having the value zero
sys_memrd('h02468aec, 0, 16, c_be_all, c_be_all);
```

4.1.22 Writing to BFM System Memory

```
sys_memwr(addr, pload, length, be_first, be_last);
```

Description

Generate a read sequence to the BFM system memory. BFM system memory emulates the memory region external to the DUT. It is addressed by the DUT when the DUT performs DMA accesses. The test-case can also read and write to BFM memory allowing communication between the test-case and the DUT.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[63:0]	in	Memory space start address
be_first	vector[3:0]	in	Specifies the byte enable configuration for the fist DWord
be_last	vector[3:0]	in	Specifies the byte enable configuration for the last DWord
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord
pload	vector[32767:0]	in	Data payload which is written to the DUT

Example

```
sys_memwr('h84200000, {32'h01234567, 32'h89abcdef}, 2,
          c_be_all, c_be_all);
sys_memwr('h02468aec, 0, 16, c_be_all, c_be_all);
```

4.1.23 Polling Read to Configuration Space

```
cfg_poll(addr, be_first, cpl_wait);
```

Description

Generate a type 0 configuration space read cycle. The twelve bit register address is passed in the first parameter. Addresses larger than twelve bits will be automatically truncated.

This task is useful for reading a configuration space location when the state of one or more bits is unknown, or when the test-program is simply waiting for the contents of a configuration space location to change. This task is typically called together with the get_cpl_buffer() function (see section 4.1.33 on page 72)

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[11:0]	in	Configuration space register address
be_first	vector [3:0]	in	Specifies the byte enable configuration
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT

Example

```
reg [31:0] s_inword;

cfg_poll(c_csreg_status, c_be_all, c_wait_cpl);
s_inword = get_cpl_buffer(0);
if (s_inword & c_bsel_intx_pending)
   $display(“Interrupt pending”);
```

4.1.24 Polling Read to I/O Space

```
io_poll(addr, be_first, cpl_wait);
```

Description

Generate a PCIexpress I/O space read cycle. The thirty-two bit register address is passed in the first parameter.

This task is useful for reading an I/O space location when the state of some bits is unknown, or when the test-program is waiting for the contents of an I/O space location to change. This task is typically called together with the get_cpl_buffer() function (see section 4.1.33 on page 72)

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[31:0]	in	I/O space register address
be_first	vector [3:0]	in	Specifies the byte enable configuration
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT

Example

```
io_poll('habcdba94, c_be_all, c_wait_cpl);
```

4.1.25 Polling Read from Memory Space

```
mem_poll(addr, length, be_first, be_last, cpl_wait);
```

Description

Generate a PCIexpress memory space read cycle. The sixty-four bit start address of the memory region is passed in the first parameter.

This task is useful for reading a memory space location when the state of some bits is unknown, or when the test-program is waiting for the contents of a memory space location to change. This task is typically called together with the get_cpl_buffer() function (see section 4.1.33 on page 72)

If the upper thirty address bits are omitted or are all zero, the BFM will automatically generate a three double-word transaction header. Otherwise, a four double-word transaction header will be generated.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[63:0]	in	Memory space start address
be_first	vector[3:0]	in	Specifies the byte enable configuration for the fist DWord
be_last	vector[3:0]	in	Specifies the byte enable configuration for the last DWord
cpl_wait	bit	in	Specifies whether the test program waits until a completion has been returned by the DUT
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord

Example

```
integer   s_memword;
   . . . . . .

   // Read 16 DWords from the DUT, copy the 8th word to the local
   // integer s_memword
   // Check that the lower four bits of the 8th memword are set to 0xa
   // c_cpl_wait is set, so the call waits until the completion packet
   // has been returned
mem_poll('h02468aec, 16, c_be_all, c_be_all, c_cpl_wait);
s_memword = get_cpl_buffer(7);
if (s_memword[3:0] != 4'ha)
   $display(“Error: Lower four bits not as expected”);
```

4.1.26 Polling Read from BFM System Memory

```
sys_mem_poll(addr, length, be_first, be_last);
```

Description

Generate a read sequence to the BFM system memory. The payload returned by the system memory unit is copied to a local BFM buffer where individual Dwords or bits can be accessed for further processing. This task is typically called together with the get_mem_buffer() BFM API call as described in section 4.1.34 on page 73.

BFM system memory emulates the memory region external to the DUT. It is addressed by the DUT when the DUT performs DMA accesses. The test-case can also read and write to BFM memory allowing communication between the test-case and the DUT.

Parameter Description

Parameter	Type	Dir.	Description
addr	vector[63:0]	in	Memory space start address
be_first	vector[3:0]	in	Specifies the byte enable configuration for the fist DWord
be_last	vector[3:0]	in	Specifies the byte enable configuration for the last DWord
length	integer	in	Specifies the number of valid DWords contained in the pload parameter. The first Dword is specified in the leftmost position. i.e. the first valid payload bit is given by (32 * length) - 1. This bit represents bit 31 of the first valid payload DWord

Example

```
   // Read 16 Dwords from BFM system memory
sys_mem_poll('h84200000, 16, c_be_all, c_be_all);
```

4.1.27 Assigning additional Completion Credits to the BFM

```
svc_ca_cplx(hdr, data)
```

Description

In the default configuration after reset, the BFM advertises the minimum number of completion credits allowed by the PCI Express specification. These are one completion header and eight completion payload (data) credits. The eight completion credits indicate the capacity to accept one completion with payload having a size of 128 bytes. The user can assign additional completion credits to the BFM using the svc_ca_cplx() command.

Note that only a switch is permitted to advertise non-infinite completion credits. An endpoint or root-complex must always advertise infinite credits. A consequence of this rule is that a root-complex or end-point can only issue a request when local resources are avilable to accept any resulting completions. Wih regard to completion credits, the BFM is playing the role of a possible switch between root complex and end-point.

The command can be issued any time (or multiple times) during a test-case. The user must ensure that in total no more than 127 header credits or 2047 payload credits are issued. This is a requirement of the PCI Express specification. Issuing larger values can lead to packet corruption.

If either field is set to zero, no additional credits will be advertised for the credit class associated with that field.

Parameter Description

Parameter	Type	Dir.	Description
hdr	integer	in	Number of additional completion header credits
data	integer	in	Number of additional completion data (payload) credits

Example

```
svc_ca_cplx(3, 15);
```

4.1.28 Assigning additional Non-Posted Credits to the BFM

```
svc_ca_np(hdr, data)
```

Description

The default setting for the BFM is to advertise one non-posted header credit and one non-posted payload credits which is the equivalent of four double-words or sixteen bytes. This setting restricts the maximum rate at which the DUT can generate non-posted requests (I/O or Configuration Space writes). By increasing the number of non-posted credits advertised by the BFM, the DUT can achieve a higher emission rate for non-posted requests.

Note that a PCIexpress endpoint id not permitted to generate configuration space or I/O spaces requests.

If either field is set to zero, no additional credits will be advertised for the credit class associated with that field.

Parameter Description

Parameter	Type	Dir.	Description
hdr	integer	in	Number of additional non-posted header credits
data	integer	in	Number of additional non-posted data (payload) credits

Example

```
svc_ca_np(3, 15);
```

4.1.29 Assigning additional Posted Credits to the BFM

```
svc_ca_p(hdr, data)
```

Description

The default setting for the BFM is to advertise one posted header credit and eight posted payload credits which is the equivalent of 32 double-words or one-hundred-and-twenty-eight bytes. This setting restricts the maximum rate at which the DUT can generate posted requests (memory writes including MSIs or messages). By increasing the number of posted credits advertised by the BFM, the DUT can achieve a higher emission rate for posted requests.

If either field is set to zero, no additional credits will be advertised for the credit class associated with that field.

Parameter Description

Parameter	Type	Dir.	Description
hdr	integer	in	Number of additional posted header credits
data	integer	in	Number of additional posted data (payload) credits

Example

```
svc_ca_p(7, 56);
```

4.1.30 Assigning a Unique ID (Bus No. / Function No. / Device No.) to the DUT

```
set_dut_id(bus_no, dev_no, func_no);
```

Description

The set_dut_id() task assigns a unique ID to the DUT. The unique ID consists of a bus number, device number and function number and is transferred to the DUT with the next type-0 configuration space write cfgwr0() call. The bus number is an eight bit field, the device number a five bit field and the function number a three bit field. Larger parameter values are automatically truncated without warning. During truncation, the upper bits are lost and the lower bits retained.

The DUT will use the unique ID as the completer ID in all successive completion packets. Typically this is one of the first procedure calls in a test-case and should be issued before sending any PCI Express request to the DUT.

Note that the function number should always be set to zero for a non multi-function device.

Parameter Description

Parameter	Type	Mode	Dir.	Description
bus_no	natural	variable	in	Specifies the PCI Express bus number assigned to the DUT. The bus number can be in the range 0 to 255. Larger values are automatically truncated without warning
dev_no	natural	variable	in	Specifies the PCI Express device number assigned to the DUT. The device number can be in the range 0 to 31. Larger values are automatically truncated without warning.
func_no	natural	variable	in	Specifies the PCI Express function number assigned to the DUT. The function number can be in the range 0 to 7. Larger values are automatically truncated without warning.

Example

Assign a completer ID to the DUT

```
set_dut_id(2, 5, 0);
```

4.1.31 Querying the Credits Advertised by the DUT

```
show_credits()
```

This task can be used to display the status of the credits displayed by the DUT. This task is useful for verifying that all incoming requests, including unsupported requests, have been correctly handled and all credits consumed by the BFM have been re-advertised. If this task is called at the end of simulation, the values displayed should match the values configured for the DUT in the IPexpress mask.

Parameter Description

This task does not take any parameters.

Example

```
show_credits;
```

4.1.32 Waiting for all pending Requests to complete

```
wait_all_cplx_pending()
```

Waits until all outstanding non-posted requests (configuration request, I/O request, memory read request) to the DUT have completed. This task is typically used to synchronise the test-case program flow to the DUT hardware activity.

Parameter Description

This task does not take any parameters.

Example

```
wait_all_cplx_pending;
```

4.1.33 Accessing the Contents of the last Completion Packet

```
get_cpl_buffer(index)
```

When the DUT is the target of a PCIexpress read request (Configuration space, I/O space or memory space), the returned data is stored in a 1024 double-word sized buffer in the BFM. This buffer can be accessed by the test case by calling the get_cpl_buffer() function. The first location in the buffer is accessed with the index 0, the last with the index 1023.

Parameter Description

Parameter	Type	Dir.	Description
index	integer	in	Index into Completion buffer
return	vector[31:0]	out	Data value at the specified location

Example

```
reg [31:0] s_inword;
   .  .  .  .

   // Request 16 words. Pass the third (offset = 2) to a register
mem_poll('h02468aec, 0, 16, c_be_all, c_be_all, c_cpl_wait);
s_inword = get_cpl_buffer(2);
```

4.1.34 Accessing the Contents of the BFM Memory Read Buffer

```
get_mem_buffer(index)
```

When the test issues a read request to the BFM system memory region (sys_memrd() or poll_sys_mem()), the returned data is stored in a 1024 double-word sized buffer in the BFM. This buffer can be accessed by the test case by calling the get_mem_buffer() function. The first location in the buffer is accessed with the index 0, the last with the index 1023.

Parameter Description

Parameter	Type	Dir.	Description
index	integer	in	Index into BFM Memory buffer
return	vector[31:0]	out	Data value at the specified location

Example

```
get_mem_buffer(2);
```

4.1.35 Idle Task

```
idle(count);
```

The idle() task can be used to insert a defined number of wait states (clock cycles) in the test program flow.

Parameter Description

Parameter	Type	Dir.	Description
count	integer	in	Number of BFM cycles to wait. One cycle is 8 ns.

Example

```
idle(127);
```

4.1.36 Waiting for all pending Requests to complete

```
wait_all_cplx_pending()
```

Waits until all outstanding non-posted requests (configuration request, I/O request, memory read request) to the DUT have completed. This task is typically used to synchronise the test-case program flow to the DUT hardware activity.

Parameter Description

This task does not take any parameters.

Example

```
wait_all_cplx_pending;
```

4.1.37 Predefined Constants

The following text block lists the pre-defined constants available in Verilog mode.

```
   // Predefined parameter Settings for Verilog BFM API
localparam  [3:0]  c_be_all      = 4'hf;
localparam  [3:0]  c_be_none     = 4'h0;
localparam         c_no_wait_cpl = 1'b0;
localparam         c_wait_cpl    = 1'b1;

   // PCI Config. Space: Status / Command Register Bits
localparam  [31:0] c_bsel_bus_mst_en          = 32'h00000004;
localparam  [31:0] c_bsel_cap_list_present    = 32'h00100000;
localparam  [31:0] c_bsel_intx_disable        = 32'h00000400;
localparam  [31:0] c_bsel_intx_pending        = 32'h00080000;
localparam  [31:0] c_bsel_io_space_en         = 32'h00000001;
localparam  [31:0] c_bsel_mem_space_en        = 32'h00000002;

   // PCI Config. Space: MSI Capability Structure.
   //                     Enable / Config settings
localparam  [31:0] c_bsel_msi_en              = 32'h00010000;
localparam  [31:0] c_bsel_msi_mm_1            = 32'h00000000;
localparam  [31:0] c_bsel_msi_mm_2            = 32'h00100000;
localparam  [31:0] c_bsel_msi_mm_4            = 32'h00200000;
localparam  [31:0] c_bsel_msi_mm_8            = 32'h00300000;

   // PCI Express Completion Status Codes
localparam  [2:0]  c_cpl_sta_ca               = 3'b100;
localparam  [2:0]  c_cpl_sta_crs              = 3'b010;
localparam  [2:0]  c_cpl_sta_sc               = 3'b000;
localparam  [2:0]  c_cpl_sta_ur               = 3'b001;

   // PCI Express Configuration Space Register Addresses
localparam  [11:0] c_csreg_vend_id            = 12'h000;
localparam  [11:0] c_csreg_dev_id             = 12'h002;
localparam  [11:0] c_csreg_command            = 12'h004;
localparam  [11:0] c_csreg_status             = 12'h006;
localparam  [11:0] c_csreg_rev_id             = 12'h008;
localparam  [11:0] c_csreg_class_code_high    = 12'h00a;
localparam  [11:0] c_csreg_bar0               = 12'h010;
localparam  [11:0] c_csreg_bar1               = 12'h014;
localparam  [11:0] c_csreg_bar2               = 12'h018;
localparam  [11:0] c_csreg_bar3               = 12'h01c;
localparam  [11:0] c_csreg_bar4               = 12'h020;
localparam  [11:0] c_csreg_bar5               = 12'h024;
localparam  [11:0] c_csreg_cis_ptr            = 12'h028;
localparam  [11:0] c_csreg_subs_vend_id       = 12'h02c;
localparam  [11:0] c_csreg_subs_dev_id        = 12'h02e;
localparam  [11:0] c_csreg_expansion_rom      = 12'h030;
localparam  [11:0] c_csreg_cap_ptr            = 12'h034;
   // The following will vary for different PCIe Cores
localparam  [11:0] c_csreg_pmcsr              = 12'h054;
localparam  [11:0] c_csreg_msi_cap            = 12'h070;
localparam  [11:0] c_csreg_msi_control        = 12'h072;
localparam  [11:0] c_csreg_msi_addr_low       = 12'h074;
localparam  [11:0] c_csreg_msi_addr_high      = 12'h078;
localparam  [11:0] c_csreg_msi_msg_data       = 12'h07c;
localparam  [11:0] c_csreg_pcie_cap           = 12'h090;
localparam  [11:0] c_csreg_pcie_dev_ctrl      = 12'h098;
localparam  [11:0] c_csreg_pcie_dev_stat      = 12'h09a;
localparam  [11:0] c_csreg_pcie_link_ctrl     = 12'h0a0;
localparam  [11:0] c_csreg_pcie_link_stat    = 12'h0a2;

   // PCI Express permissible values for Max. Payload Size
localparam  [2:0] c_max_payload_128 = 3'b000;
localparam  [2:0] c_max_payload_256 = 3'b001;
localparam  [2:0] c_max_payload_512 = 3'b010;
localparam  [2:0] c_max_payload_1024 = 3'b011;
localparam  [2:0] c_max_payload_2048 = 3'b100;
localparam  [2:0] c_max_payload_4096 = 3'b101;

   // PCI Express permissible values for Max. Read-Request Size
localparam  [2:0] c_max_read_req_128 = 3'b000;
localparam  [2:0] c_max_read_req_256 = 3'b001;
localparam  [2:0] c_max_read_req_512 = 3'b010;
localparam  [2:0] c_max_read_req_1024 = 3'b011;
localparam  [2:0] c_max_read_req_2048 = 3'b100;
localparam  [2:0] c_max_read_req_4096 = 3'b101;

   // PCI Express Pre-defined Message Codes
localparam  [7:0] c_msg_err_corr              = 8'h30;
localparam  [7:0] c_msg_err_fatal             = 8'h33;
localparam  [7:0] c_msg_err_non_fatal         = 8'h31;
localparam  [7:0] c_msg_inta_assert           = 8'h20;
localparam  [7:0] c_msg_inta_deassert         = 8'h24;
localparam  [7:0] c_msg_intb_assert           = 8'h21;
localparam  [7:0] c_msg_intb_deassert         = 8'h25;
localparam  [7:0] c_msg_intc_assert           = 8'h22;
localparam  [7:0] c_msg_intc_deassert         = 8'h26;
localparam  [7:0] c_msg_intd_assert           = 8'h23;
localparam  [7:0] c_msg_intd_deassert         = 8'h27;
localparam  [7:0] c_msg_pm_as_nak             = 8'h14;
localparam  [7:0] c_msg_pm_pme                = 8'h18;
localparam  [7:0] c_msg_pm_pme_to             = 8'h19;
localparam  [7:0] c_msg_pm_pme_to_ack         = 8'h1b;
localparam  [7:0] c_msg_set_slot_pwr_limit    = 8'h50;
localparam  [7:0] c_msg_unlock                = 8'h00;
localparam  [7:0] c_msg_vendor_defined_0      = 8'h7e;
localparam  [7:0] c_msg_vendor_defined_1      = 8'h7f;

   // Routing method specifiers
localparam [2:0] c_route_broadcast_from_rc  = 3'h3;
localparam [2:0] c_route_by_address         = 3'h1;
localparam [2:0] c_route_by_id              = 3'h2;
localparam [2:0] c_route_forward_to_rc      = 3'h0;
localparam [2:0] c_route_gathered_to_rc     = 3'h5;
localparam [2:0] c_route_local_rcv_term     = 3'h4;

   // PCI Express Commands
localparam  [7:0] c_tlp_mrd32    = 8'h00;
localparam  [7:0] c_tlp_mrd64    = 8'h20;
localparam  [7:0] c_tlp_mrdlk32  = 8'h01;
localparam  [7:0] c_tlp_mrdlk64  = 8'h21;
localparam  [7:0] c_tlp_mwr32    = 8'h40;
localparam  [7:0] c_tlp_mwr64    = 8'h60;
localparam  [7:0] c_tlp_iord     = 8'h02;
localparam  [7:0] c_tlp_iowr     = 8'h42;
localparam  [7:0] c_tlp_cfgrd0   = 8'h04;
localparam  [7:0] c_tlp_cfgwr0   = 8'h44;
localparam  [7:0] c_tlp_cfgrd1   = 8'h05;
localparam  [7:0] c_tlp_cfgwr1   = 8'h45;
localparam  [7:0] c_tlp_msg      = 8'h30;
localparam  [7:0] c_tlp_msgd     = 8'h70;
localparam  [7:0] c_tlp_cpl      = 8'h0a;
localparam  [7:0] c_tlp_cpld     = 8'h4a;
localparam  [7:0] c_tlp_cpllk    = 8'h0b;
localparam  [7:0] c_tlp_cpldlk   = 8'h4b;
```

5 Compiling the Bus Functional Model

The Bus Functional Model (BFM) sources consist of two files one verilog and one VHDL. These should be compiled to a different library than is used for the user design. The BFM requires the Lattice PMI and ECP3 libraries but the user design can be based on any technology library such as ECP2/M, ECP3 or ECP5.

Even if the user design also uses ECP3 it is still required that the BFM be compiled to a different work library. BFM core modules may otherwise clash with PCI Express modules used by the user leading to incorrect simulation results.

Typically the BFM is compiles as follows:

```
% vlib pcie_bfm_lib pcie_bfm_lib
% vlog -v2k5 -work pcie_bfm_lib +incdir+$PATH_TO_BFM_SOURCE/bfm_lspcie_rc/src \
-l ovi_ecp3 -l pmi_work $PATH_TO_BFM_SOURCE/bfm_lspcie_rc/src/bfm_lspcie_rc.v
% vcom -2008 -quiet -work pcie_bfm_lib \
 $PATH_TO_BFM_SOURCE/bfm_lspcie_rc/src/bfm_lspcie_rc.vhd
```

This can be done in the active-hdl console or in a Windows command shell having the appropriate Aldec Path and variables set. For Aldec Riviera the commands are typically entered in a Linux shell.

6 Building a Test-Bench

When building a testbench for an FPGA based on the Lattice PCI Epxress core, the only external circuitry required in all cases is a reset generator. Depending on the other external interfaces implemented in the FPGA solution, other interfaces or BFMs may of course also be required.

To incorporate the PCI Express BFM described in this document, the following additional steps must be taken:

- Replace the SERDES in the PCI Express top unit generated by IPexpress with the BFM. This step does not require any physical changes to the design. For a verilog design, it can be enforced by linking to the BFM directory before linking to any other design or Lattice simulation libraries. For a VHDL design, it can be enforced by compiling the pcs_pipe_top wrapper unit provided with the BFM into the design work directory. This wrapper unit automatically references the BFM library.
- Ensure that each member of a hdinp/hdinn pair in an active lane is set to a different level
- Ensure that each member of a hdinp/hdinn pair in an inactive lane is set to the same level. If lanes 1 to 3 are not used in a design (typical for x1 configuration) these lanes can be left unassigned. The default setting in the BFM is for these lane inputs to be initialised to logic zero.

The BFM is precompiled into the simulation library 'pcie_bfm_lib'. The user is free to map this library into the FPGA workspace either verbatim or using an alias.

6.1 Simulating a Verilog Design using the PCI Express BFM

6.1.1 Using the Aldec active-hdl simulator

- Link to the BFM library using the console command

```
amap pcie_bfm_lib <path_to_bfm_root>/bfm_lspcie_rc/bfm_lspcie_rc.LIB
```

Alternatively, the library manager in the pull-down menus can be used.

- When compiling the PCI Express wrapper generated by IPexpress, specify the BFM Library first. For example assuming the PCI Express core was generated using 'pcie_x1_unit' as the user name:

```
alog -l pcie_bfm_lib -l ecp2m <some_file_path>/pcie_x1_unit.v
```

Alternatively, the bfm library can be specified first in the list of search libraries using the menu Design → Settings and then adding files to the 'Verilog libraries' box

6.1.2 Using the Aldec riviera simulator

- Link to the BFM library

```
vlib pcie_bfm_lib <path_to_bfm_root>/bfm_lspcie_rc
```

- When compiling the PCI Express wrapper generated by IPexpress, specify the BFM Library first. For example assuming the PCI Express core was generated using 'pcie_x1_unit' as the user name:

```
vlog -l pcie_bfm_lib -l ecp2m -work prj_work <some_file_path>/pcie_x1_unit.v
```

6.2 Compiling a Verilog Test-Case

All BFM test-cases must be compiled into the BFM simulation library. The defualt naem for this library is *pcie_bfm_lib.* In principle, the user is however free to chose any name for the BFM library and simply mapping the chosen name to the library file (bfm_lspcie_rc.LIB) in the BFM installation.

As mentioned in chapter 4.1 on page 47, the user test-case must be called *pcie_vlog_test_case* and must include the file *bfm_lspcie_rc_tlm_lib.v* found under the *src* directory in the BFM installation.

An exmple test-case skeleton is shown below.

```
`timescale 1ns / 1 ps
module pcie_vlog_test_case;

   `include "bfm_lspcie_rc_tlm_lib.v"

   initial begin
       // User Test goes here
   end
endmodule
```

On the command line, a test-case can be compiled by a command such as

```
vlog -work pcie_bfm_lib +incdir+/<path_to_bfm>/src <user_test_case.v>
```

However, if the user test-case contains syntax errors, the only console message will be

```
ERROR VCP1230 "Error in encrypted code."
```

This is because the file *bfm_lspcie_rc_tlm_lib.v* is an encrypted verilog file. To get more meaningful information, the user should compile with the BFM_CHECK define set. This is deon as follows:

```
vlog -work pcie_bfm_lib +incdir+/<path_to_bfm>/src <user_test_case.v> \
                        +define+BFM_CHECK
```

Setting the BFM_CHECK define, bypasses the encrypted verilog sources during compilation. The calls to the BFM API are still checked. The user user will see more meaningful log messages such as

```
ERROR VCP2897 "Mismatched number of actual and formal arguments in task cfg_poll
enable." "test_basic_01.v" 53 65
```

Once the test-case has been corrected, the test-case must be recompiled without specifying the BFM_CHECK define.

6.3 Simulating a VHDL Design using the PCI Express BFM

As illustrated in Figure 2 on page 8, working with the BFM in a VHDL design environment typically involves the use of two or more VHDL project libraries. At a minimum, the BFM library and the project library are required. The BFM always runs in the context of the BFM library. This library must also be the target library when compiling a test-case. The BFM top-level appears as a plug-compatible instance of the Lattice pcs_pipe_top library cell.

There are two approaches to making the BFM visible in the design project. The first and less desirable is to modify the top-level wrapper generated by IPexpress so that it searches for the pcs_pipe_top in the BFM library. This method is less desirable because it involves separate instances of the IPexpress top-level wrapper for synthesis and simulation.

The preferred approach is to use the pcs_pipe_top redirection wrapper provided by the BFM installation. This method is outlined below. The redirection wrapper is the only BFM object which should be compiled into the project work library instead of into the BFM work library. The redirection wrapper appears itself as the pcs_pipe_top implementation but transparently instantiates the pcs_pipe_top provided by the BFM library. With this method the only difference between simulation and synthesis is in the make files. The simulation makefiles must contain the redirection wrapper whereas the synthesis make files do not.

For ECP5 the redirection method must use the BFM top level pcie_x1_top_phy instead of pcs_pipe_top.

6.3.1 Using the Aldec active-hdl Simulator

➢ Link to the BFM library using the console command

```
amap pcie_bfm_lib <path_to_bfm_root>/bfm_lspcie_rc/bfm_lspcie_rc.LIB
```

Alternatively, the library manager in the pull-down menus can be used.

➢ Before compiling the PCI Express wrapper generated by IPexpress, compile the BFM Serdes redirection wrapper first. For example assuming the PCI Express core was generated using 'pcie_x1_unit' as the user name:

```
acom -work proj_work <path_to_bfm_root>/bfm_lspcie_rc/src/pcs_pipe_top-e.vhd
acom -work proj_work \
             <path_to_bfm_root>/bfm_lspcie_rc/src/pcs_pipe_top-a_wrap.vhd
acom -dbg -work proj_work <some_file_path>/pcie_x1_unit.vhd
```

6.3.2 Using the Aldec Riviera Simulator

➢ Link to the BFM library

```
vlib pcie_bfm_lib <path_to_bfm_root>/bfm_lspcie_rc
```

➢ Before compiling the PCI Express wrapper generated by IPexpress, compile the BFM Serdes redirection wrapper first. For example assuming the PCI Express core was generated using 'pcie_x1_unit' as the user name:

```
vcom -work proj_work <path_to_bfm_root>/bfm_lspcie_rc/src/pcs_pipe_top-e.vhd
vcom -work proj_work \
             <path_to_bfm_root>/bfm_lspcie_rc/src/pcs_pipe_top-a_wrap.vhd
vcom -dbg -work proj_work <some_file_path>/pcie_x1_unit.vhd
```

7 Tutorial

7.1 Simulating the Basic Example from the Lattice PCI Express Development Kit

This section describes how to set up a simulation with the BFM described in this document and the Lattice PCI Express Development Kit for ECP3 as is available at the time of writing. The development kit can be downloaded from the Lattice web-site and is available for Windows and Linux.

Note the line numbers indicated in the following code snippets may change over time as the Lattice PCI Express Development kit is enhanced. They are only intended as orientation.

7.1.1 Activating the PCI Express Lanes in the Testbench

The testbench top level provided with the PCI Express Development kit for ECP3 is located in the file Testbench/ecp3/tb.v. As described in Chapter 6 above, the BFM searches for active PCI Express lanes by detecting opposite logic levels on the 'P' and 'N' lane input. Line 54 in the code section below indicates how to mark all four lanes as active.

```
50 top dut (
51    .rstn(rstn),
52    .FLIP_LANES(1'b0), .LED_INV(1'b0),
53    .refclkp(clk_100p), .refclkn(clk_100n),
54    .hdinp(4'hf), .hdinn(4'h0),
55    .hdoutp(tx_pcie_p), .hdoutn(tx_pcie_n),
56    .dip_switch(8'd0)
57 );
```

7.1.2 Removing any Transactors or Force Statements from the Testbench

Simulating with the BFM does not require any external PCI Express drivers or force statements. The only external function which must be addressed is the reset generator. Any lines such as indicated in the code block below must be removed or commented out.

```
30 //  // reads file tlps.txt for stimulus
31 //  iptx_tlpgen gen(.clk(clk_125), .rstn(rstn), .enable(gen_en),
32 //                  .req(tx_req), .rdy(tx_rdy),
33 //                  .stlp(tx_st), .etlp(tx_end), .nlfy(tx_nlfy),
34 //                  .dwen(tx_dwen), .dout(tx_dout));
35
36 //  pcie_drv drv (
37 //     .rstn(rstn),
38 //     .refclkp(clk_100p),
39 //     .refclkn(clk_100n),
40 //     .clk_125(clk_125),
41 //     .tx_req(tx_req), .tx_rdy(tx_rdy),
42 //     .tx_st(tx_st), .tx_end(tx_end), .tx_nlfy(tx_nlfy),
43 //     .tx_dwen(tx_dwen), .tx_data(tx_dout),
44 //     .rx_st(), .rx_end(), .rx_dwen(), .rx_data(),
45 //     .hdinp(tx_pcie_p), .hdinn(tx_pcie_n),
46 //     .hdoutp(rx_pcie_p), .hdoutn(rx_pcie_n)
47 //  );

87    // Force a positive receiver detection for both DUT and DRV
88    //force drv.pcie.u1_pcs_pipe.ffs_pcie_con_0 = 1'b1;
89    //force drv.pcie.u1_pcs_pipe.ffs_pcie_con_1 = 1'b1;
```

```
   //force drv.pcie.u1_pcs_pipe.ffs_pcie_con_2 = 1'b1;
   //force drv.pcie.u1_pcs_pipe.ffs_pcie_con_3 = 1'b1;
   //force dut.pcie.u1_pcs_pipe.ffs_pcie_con_0 = 1'b1;
   //force dut.pcie.u1_pcs_pipe.ffs_pcie_con_1 = 1'b1;
   //force dut.pcie.u1_pcs_pipe.ffs_pcie_con_2 = 1'b1;
   //force dut.pcie.u1_pcs_pipe.ffs_pcie_con_3 = 1'b1;


   // Force DUT to L0 state of LTSSM to save sim time
   //force drv.pcie.no_pcie_train=1'b1;
   //force dut.pcie.no_pcie_train=1'b1;
```

7.1.3 Modifying the default Compile Script

The Lattice PCI Express Development Kit includes a compile-and-go simulation script for active-hdl which is located at .../Simulation/ecp3/PCIeBasic.aldec/cmpile.do. A simulation can be run by simply reading this script into active-hdl using the menu selection Tools → Execute Macro.

The lines below indicate how to set a link to the precompiled BFM library extracted from the BFM *.zip file. Also, the modifications to the compilation steps for the logic generated by IPexpress and the modification to the elaboration line are indicated.

In summary, the following changes must be made to the compilations script:

- Add an amap instruction to reference the BFM library
- Add the BFM library to the compile step for pcie_top.v (-l pcie_bfm_lib). Note lower case 'l'
- Add the BFM library to the elaboration step (asim call). Here, use -L pcie_bfm_lib. Note, 'L' is upper case
- Remove or comment out any compilation of the Lattice Serdes cell
- Remove or comment out any compilation steps to an existing bus driver
- Remove, comment out or adjust any 'add wave' commands. The hierarchy paths are probably different

```
cd "F:\Lattice_DevKits.V1.0\DK-ECP3-PCIE-020\Hardware
    \PCIe_x4\ecp3-95_PCIeBasic_SBx4
    \Simulation\ecp3\PCIeBasic.aldec"

design create PCIeBasic .
design open PCIeBasic
cd ../..
set sim_working_folder .

amap pcie_bfm_lib
     “F:\.....\bfm_lspcie_rc\bfm_lspcie_rc.LIB” # Path will need adjusting!!

# Compile the PCIe core from IPExpress
alog  +define+SIMULATE=1
              ../../../ipExpressGenCore/ecp3/pciex4/pci_exp_params.v
              ../../../ipExpressGenCore/ecp3/pciex4/pci_exp_ddefines.v
              ../../../ipExpressGenCore/ecp3/pciex4/pcie_beh.v
              ../../../ipExpressGenCore/ecp3/pciex4/pcie.v
     # Comment out the original line
#alog +define+SIMULATE=1
   ../../../ipExpressGenCore/ecp3/pciex4/pci_exp_params.v
   ../../../ipExpressGenCore/ecp3/pciex4/pcie_eval/models/ecp3/pipe_top.v
   ../../../ipExpressGenCore/ecp3/pciex4/pcie_eval/models/ecp3/pcs_top.v
   ../../../ipExpressGenCore/ecp3/pciex4/pcie_eval/models/ecp3/ctc.v
```

```
     ../../../ipExpressGenCore/ecp3/pciex4/pcie_eval/models/ecp3/sync1s.v
     ../../../ipExpressGenCore/ecp3/pciex4/pcie_eval/models/ecp3/PCSD.v
     ../../../ipExpressGenCore/ecp3/pciex4/pcie_eval/models/ecp3/pcs_pipe_top.v

             . . . . . . .

alog "../../../Source/32kebr/wbs_32kebr.v"
alog "../../../Source/gpio/wbs_gpio.v"
alog "../../../Source/wb_arb/wb_arb.v"
alog "../../../Source/ip_tx_arbiter.v"
alog "../../../Source/ip_rx_crpr.v"
alog "../../../Source/ip_crpr_arb.v"
alog "../../../Source/UR_gen/UR_gen.v"
     # Reference the BFM Library when compiling the wrapper from IPexpress
alog -l pcie_bfm_lib "../../../Source/ecp3/pcie_top.v"
alog "../../../Source/ecp3/top_basic.v"

# Compile the Driver
#alog ../../../Testbench/ecp3/pcie_drv/pcie_drv.v
#alog ../../../Testbench/iptx_tlpgen_sim.v

# Compile the TB
alog ../../../Testbench/ecp3/tb.v

#asim PCIeBasic.tb -L pcsd_aldec_work -L pmi_work -L ovi_ecp2
  # Refer to the BFM Library first when elaborating
asim PCIeBasic.tb -L pcie_bfm_lib -L pmi_work -L ovi_ecp2
```

7.1.4 Adding Test Cases to the Design Workspace

Existing Verilog or VHDL test cases can be added to the design workspace using the Design → Add Files to Design menu items. Please note that all test-cases whether Verilog or VHDL must target the BFM library and not the design library for compilation (see also 2 on page 8).

In active-hdl this is done by setting the properties on test-case file. The properties are accessible by selecting the file with the right mouse button and then selecting 'pcie_bfm_lib' from the 'Compile into Library' pull down.

For Verilog test-cases, two additional steps must be performed

- Select Verilog 1364-2005 as language dialect
- Add the 'src' sub-directory under the BFM installation to the Verilog search path. This is set up using the active-hdl menu item Design → Settings → Compilation → Verilog and then adding the src directory to the 'Include Directories' list-box

For VHDL test-cases, the language dialect 1076-2002 must be selected.

7.1.5 Compiling a Test-Case

Once test-cases have been added to the design, a test-case can be compiled by selecting the test-case file with the right-hand mouse button and choosing compile. Note that selecting 'compile all' will leave only the last test-case in the list visible. Compilation results from intermediate test-cases will be overwritten.

If a design contains both Verilog and VHDL test-cases, the previously selected language must be cleared first before switching to the intermediate language. Otherwise, there will be contention between the two language interpreters as indicated in 1 on page 6. To 'erase' an existing language test-case, the corresponding file test_null.v or test_null.vhd provided in the BFM src directory should be compiled.