Mentor Verification IP Altera Edition AMBA AXI4-Stream User Guide

Software Version 10.4c

January 2016

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 4

Chapter 5

Chapter 6

Chapter 9

List of Examples

List of Figures

List of Tables

Preface

About This User Guide

This user guide describes the application interface (API) of the Mentor® Verification IP (VIP) Altera® Edition (AE) and how it conforms to the *AMBA® 4 AXI4-Stream Protocol Specification*, Version 1.0, Issue A (ARM IHI 0051A).

AMBA AXI4-Stream Protocol Specification

The Mentor VIP AE conforms to the *AMBA 4 AXI4-Stream Protocol Specification*, Version 1.0, Issue A (ARM IHI 0051A). This user guide refers to this specification as the "AMBA AXI4-Stream Protocol Specification."

Mentor VIP AE License Requirements

Note

A license is required to access the Mentor Graphics VIP AE bus functional models and inline monitor.

- To access the Mentor Graphics VIP AE and upgrade to the Quartus II Subscription Edition software, Version 15.1, from a previous version, you must regenerate your license file.
- To access the Mentor VIP AE with the Quartus II Web Edition software, you must upgrade to Version 15.1 and purchase a Mentor VIP AE seat license by contacting your Altera sales representative.

You can generate and manage license files for Altera software and IP products by visiting the Self-Service Licensing Center of the Altera website.

Supported Simulators

Mentor VIP AE supports the following simulators:

- Mentor Graphics Questa SIM and ModelSim 10.4d
- Synopsys® VCS® and VCS-MX 2015.09 on Linux
- Cadence® Incisive® Enterprise Simulator (IES) 15.10.010 on Linux

Simulator GCC Requirements

Mentor VIP requires that the installation directory of the simulator includes the GCC libraries shown in Table 1. If the installation of the GCC libraries was an optional part of the simulator's installation and the Mentor VIP does not find these libraries, an error message similar to the following appears:

```
ModelSim / Questa SIM
# ** Error: (vsim-8388) Could not find the MVC shared library : GCC not
found in installation directory (/home/user/altera2/14.0/modelsim_ase) for
platform "linux".  Please install GCC version "gcc-4.7.4-linux"
```

Table 1. Simulator GCC Requirements

Simulator	Version	GCC Version(s)	Search Path
Mentor Questa SIM			
	10.4d	4.7.4 (Linux 32 bit)	<install dir>/gcc-4.7.4-linux
		4.7.4 (Linux 64 bit)	<install dir>/gcc-4.7.4-linux_x86_64
		4.2.1 (Windows 32 bit)	<install dir>/gcc-4.2.1-mingw32vc9
Mentor ModelSim			
	10.4d	4.7.4 (Linux 32 bit)	<install dir>/gcc-4.7.4-linux
		4.7.4 (Linux 64 bit)	<install dir>/gcc-4.7.4-linux_x86_64
		4.2.1 (Windows 32 bit)	<install dir>/gcc-4.2.1-mingw32vc9
Synopsys VCS/VCS-MX			
	2014.03-SP1	4.7.2 (Linux 32 bit)	$VCS_HOME/gnu/linux/4.7.2_32-shared
	or 2014.12		$VCS_HOME/gnu/4.7.2_32-shared
		4.7.2 (Linux 64 bit)	$VCS_HOME/gnu/linux/4.7.2_64-shared
			$VCS_HOME/gnu/4.7.2_64-shared
Note: If you set the environment variable VG_GNU_PACKAGE, then it is used instead of the VCS_HOME environment variable.			
Cadence Incisive			
	13.20.002 or 14.10.014	4.4 (Linux 32/64 bit)	<install dir>/tools/cdsgcc/gcc/4.4
Note: Use the *cds_tools.sh* executable to find the Incisive installation. Ensure $PATH includes the installation path and *<install dir>/tools/cdsgcc/gcc/4.4/install/bin*. Also, ensure the LD_LIBRARY_PATH includes *<install dir>/tools/cdsgcc/gcc/4.4/install/lib.*			

Chapter 1
Mentor VIP Altera Edition

The Mentor VIP AE provides bus functional models (BFMs) to simulate the behavior and to facilitate the verification of the IP. The Mentor VIP AE includes the following interfaces:

- AXI3 with master, slave, and inline monitor BFMs
- AXI4 with master, slave, and inline monitor BFMs
- AXI4-Lite with master, slave, and inline monitor BFMs
- AXI4-Stream with master, slave, and inline monitor BFMs

This user guide covers the AXI4-Stream BFMs only. Refer to the *Mentor Verification IP Altera Edition AXI3/AXI4 User Guide* for details of the AXI3 and AXI4 BFMs, and the *Mentor Verification IP Altera Edition AXI4-Lite User Guide* for details of the AXI4-Lite BFMs.

Advantages of Using BFMs and Monitors

Using the Mentor VIP AE has the following advantages:

- Accelerates the verification process by providing key verification test bench components
- Provides BFM components that implement the *AMBA 4 AXI4-Stream Protocol Specification*, which serves as a reference for the protocol
- Provides a full suite of configurable assertion checking within each BFM

Implementation of BFMs

The Mentor VIP AE BFMs, master, slave, and inline monitor components are implemented in SystemVerilog. Also included are wrapper components so that you can use the BFMs in VHDL verification environments with simulators that support mixed-language simulation.

The Mentor VIP AE provides a set of APIs for each BFM that you can use to construct, instantiate, control, and query signals in all BFM components. Your test programs must use only these public access methods and events to communicate with each BFM. To ensure support in current and future releases, your test programs must use the standard set of APIs to interface with the BFMs. Nonstandard APIs and user-generated interfaces may not be supported in future releases.

The test program drives the stimulus to the DUT and determines whether the behavior of the DUT is correct by analyzing the responses. The BFMs translate the test program stimuli (transactions), creating the signaling for the *AMBA 4 AXI4-Stream Protocol Specification*. The BFMs also check for protocol compliance by firing an assertion when a protocol error is observed.

What Is a Transaction?

A transaction for Mentor VIP AE represents an instance of information that is transferred between a master and a slave peripheral, and that it adheres to the protocol used to transfer the information. For example, a master transaction can communicate a data stream packet consisting of a number of transfers to a slave DUT. A subsequent data stream packet requires a new and unique transaction.

Each transaction has a dynamic Transaction Record that exists for the life of the transaction. The life of a transaction record starts when it is created, and ends when the transaction completes. The transaction record is automatically discarded when the transaction ends.

When created, a transaction contains *transaction fields* that you set to define two transaction aspects:

- *Protocol fields* are transferred over the protocol signals
- *Operation fields* determine how the information is transferred, and when the transaction is complete

For example, a master transaction record holds a byte definition in the *byte_type* protocol field, the value of this field is transferred over the TKEEP and TSTRB protocol signals. A master transaction also has a *transaction_done* operation field that indicates when the transaction is complete; this operation field is not transferred over the protocol signals. These two types of transaction fields, *protocol* and *operation*, establish a dynamic record during the life of the transaction.

In addition to transaction fields, you specify arguments to tasks, functions, and procedures that permit you to create, set, and get the dynamic transaction record during the lifetime of a transaction. Each BFM has an API that controls how you access the transaction record. How you access the record also depends on the source code language, whether it is VHDL or SystemVerilog. Methods for accessing transactions based on the language you use are explained in detail in the relevant chapters of this user guide.

AXI4-Stream Transactions

A complete transaction communicates information between a master and a slave. Transaction fields, described in the previous section, What Is a Transaction?, determine what is transferred and how information is transferred. During the lifetime of a transaction, the roles of the master and slave ensure that a transaction completes successfully, and that transferred information adheres to the protocol specification. Information flows from the master to the slave during a transaction, with the master initiating the transaction.

The AXI4-Stream protocol has a single channel to transfer protocol information. It has a pair of handshake signals, TVALID and TREADY, that indicate valid information on the channel, and the acceptance of the information from the channel.

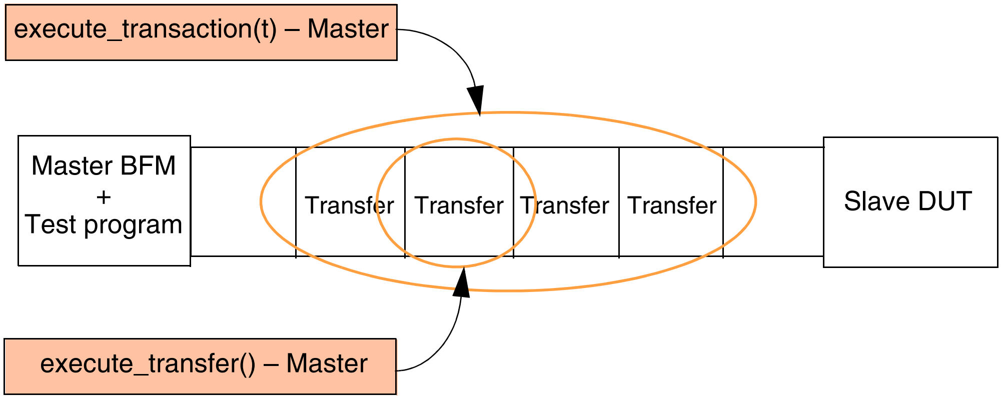
Master BFM and Slave BFM Roles

Note

The following description of a master transaction references SystemVerilog BFM API tasks. There are equivalent VHDL BFM API procedures that perform the same functionality.

For a master transaction, the master calls the *create_master_transaction()* task to define the information to be transferred, and then calls the *execute_transaction()* task to initiate the communication of information, as shown in Figure 1-1.

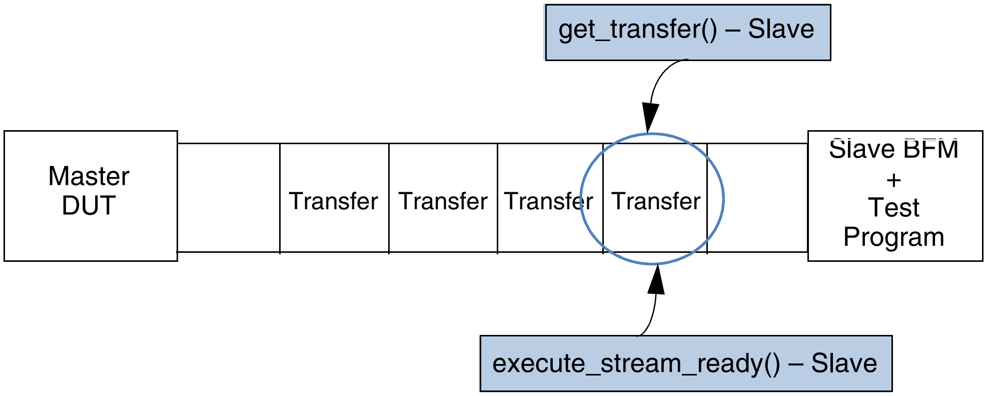
Figure 1-1. Master BFM Test Program Role



The *execute_transaction()* task results in the master calling the *execute_transfer()* task a multiple number of times, equal to the number of transfers in the transaction.

The slave also creates a transaction by calling the *create_slave_transaction()* task to accept the transfer of information from the master. The transfer is received by the slave calling the *get_transfer()* task, as shown in Figure 1-2.

Figure 1-2. Slave BFM Test Program Role



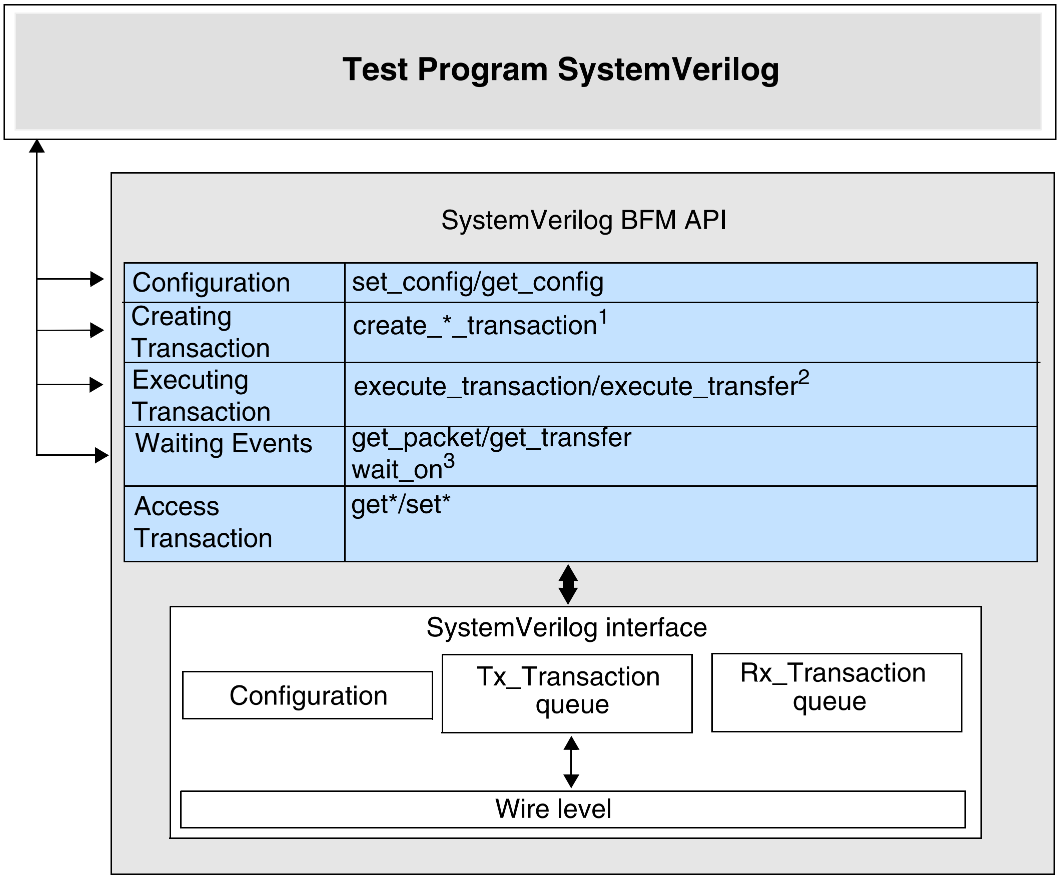
The slave can cause back-pressure to the master using the *execute_stream_ready()* task to set the TREADY protocol signal to “0” to inhibit subsequent “transfers” from the master.

Chapter 2
SystemVerilog API Overview

This section provides the functional description of the SystemVerilog (SV) API for all the BFM (master, slave, and monitor) components. For each BFM, you can configure the protocol transaction fields that are executed on the protocol signals, as well as control the operational transaction fields that set delay and timeout values.

In addition, each BFM API has tasks that wait for certain events to occur on the system clock and reset signals, and tasks to get and set information about a particular transaction.

Figure 2-1. SystemVerilog BFM Internal Structure



Configuration	set_config/get_config
Creating Transaction	create_*_transaction[1]
Executing Transaction	execute_transaction/execute_transfer[2]
Waiting Events	get_packet/get_transfer wait_on[3]
Access Transaction	get*/set*

Notes: 1. Refer to create_*_transaction()
2. Refer to execute_transaction()
3. Refer to set*()

Configuration

Configuration sets timeout delays, error reporting, and other attributes of the BFM.

Each BFM has a *set_config()* function that sets the configuration of the BFM. Refer to the individual BFM APIs for details.

Each BFM also has a *get_config()* function that returns the configuration of the BFM. Refer to the individual BFM APIs for details.

set_config()

Example 2-1 shows how to set the burst timeout factor to 1000 for a transaction in the master BFM test program.

Example 2-1. BFM Test Program Set Configuration

```
// Setting the burst timeout factor to 1000
master_bfm.set_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

Example 2-2 shows how to get the signal hold time in the master BFM test program.

Example 2-2. BFM Test Program Get Configuration

```
 // Getting hold time value
hold_time = master_bfm.get_config(AXI4STREAM_CONFIG_HOLD_TIME);
```

Creating Transactions

To transfer information between a master BFM and slave DUT over the protocol signals, you must create a transaction in the master test program. Similarly, to transfer information between a master DUT and a slave BFM, you must create a transaction in the slave test program. To monitor the transfer of information using a monitor BFM, you must create a transaction in the monitor test program.

When you create a transaction, a Transaction Record is created and exists for the life of the transaction. This transaction record can be accessed by the BFM test programs during the life of the transaction as it transfers information between the master and slave.

Transaction Record

The transaction record contains two types of transaction fields, *protocol* and *operational*, that either transfer information over the protocol signals, or define how and when a transfer occurs, respectively.

Protocol fields contain transaction information that is transferred over the protocol signals. For example, the *id* field is transferred over the TID protocol signals during a transaction to identify a data stream.

Operational fields define how and when the transaction is transferred. Their content is not transferred over protocol signals. For example, the *operation_mode* field controls the blocking/nonblocking operation of a transaction, but this information is not transferred over the protocol signals.

Transaction Definition

The transaction record exists as a SystemVerilog class definition in each BFM. Example 2-3 shows the definition of the *axi4stream_transaction* class members that form the transaction record.

Example 2-3. Transaction Record Definition

```
// Global Transaction Class
class axi4stream_transaction;
    // Protocol
    byte unsigned data[];
    axi4stream_byte_type_e byte_type[];
    bit [((`MAX_AXI4_ID_WIDTH) - 1):0]  id;
    bit [((`MAX_AXI4_DEST_WIDTH) - 1):0]  dest;
    bit [((`MAX_AXI4_USER_WIDTH) - 1):0] user_data [];
    int valid_delay[];
    int ready_delay[];

    // Housekeeping
    axi4stream_operation_mode_e
      operation_mode   = AXI4STREAM_TRANSACTION_BLOCKING;
    bit transfer_done[];
    bit transaction_done;

...

endclass
```

Note

The *axi4stream_transaction* class code above is shown for information only. Access to each transaction record during its life is performed by various *set*()* and *get*()* tasks described later in this chapter.

The contents of the transaction record is detailed in Table 2-1.

Table 2-1. Transaction Record Fields

Transaction Field	Description
Protocol Transaction Fields	
data	An unsized array of bytes to hold the data of an AXI4-Stream packet. The field content is transferred over the TDATA protocol signals during a transaction.
byte_type	An unsized array to hold the enumerated type of each data byte within an AXI4-Stream packet. The field content is transferred over the TSTRB and TKEEP protocol signals during a transaction. The following are types of byte: AXI4STREAM_DATA_BYTE AXI4STREAM_NULL_BYTE AXI4STREAM_POS_BYTE AXI4STREAM_ILLEGAL_BYTE
id	A bit vector (of length equal to the TID protocol signal bus width) to hold the data stream identifier of the data packet. The field content is transferred over the TID protocol signals during a transaction.
dest	A bit vector (of length equal to the TDEST protocol signal bus width) to hold the routing information for the data stream packet. The field content is transferred over the TDEST protocol signals during a transaction.
user_data	An unsized bit vector (of length equal to the TUSER protocol signal bus width) to hold the user-defined sideband information. The field content is transferred over the TUSER protocol signals during a transaction.
Operational Transaction Fields	
valid_delay	An unsized array of integers to hold the delay value of the TVALID protocol signal (measured in ACLK cycles) for each transfer within a packet. The field content is not transferred over the protocol signals during a transaction.
ready_delay	An unsized array of integers to hold the delay value of the TREADY protocol signal (measured in ACLK cycles) for each transfer within a packet. The field content is not transferred over the protocol signals during a transaction.

Table 2-1. Transaction Record Fields (cont.)

Transaction Field	Description
operation_mode	An enumeration to hold the operation mode of the transaction. The following are two types of operation mode: AXI4STREAM_TRANSACTION_NON_BLOCKING AXI4STREAM_TRANSACTION_BLOCKING The field content is not transferred over the AXI4-Stream protocol signals during a transaction.
transfer_done	An unsized bit array to hold the *done* flag for each transfer within a packet. The field content is not transferred over the protocol signals during a transaction.
transaction_done	A bit to hold the *done* flag for a complete transaction. The field content is not transferred over the protocol signals during a transaction.

The SystemVerilog Master BFM API allows you to create a master transaction by providing only an optional *burst_length* argument to indicate the number of transfers within a packet. All other protocol transaction fields automatically default to legal protocol values to create a master transaction record. Refer to *create_master_transaction()* for default protocol transaction field values.

The SystemVerilog Slave BFM API allows you to create a slave transaction with no arguments. All protocol transaction fields automatically default to legal protocol values to create a slave transaction record. Refer to *create_slave_transaction()* for default protocol transaction field values.

The SystemVerilog Monitor BFM API allows you to create a monitor transaction with no arguments. All protocol transaction fields automatically default to legal protocol values to create a complete monitor transaction record. Refer to *create_monitor_transaction()* for default protocol transaction field values.

Note

If you change the default value of a protocol transaction field, it is valid for all future transactions until you set a new value.

create_*_transaction()

The *create_master_transaction()*, *create_slave_transaction()* and *create_monitor_transaction()* BFM API functions create a master, a slave, and a monitor transaction, respectively.

Example 2-4 shows a master BFM test program creating a master transaction with a packet length of 10 transfers.

Example 2-4. Master BFM Test Program Transaction Creation

```
// Define a variable trans of type axi4stream_transaction to hold
// master transaction record
axi4stream_transaction trans;

...

// Create master transaction with 10 transfers
trans = bfm.create_master_transaction(10);
```

Example 2-5 shows a slave BFM test program creating a slave transaction.

Example 2-5. Slave BFM Test Program Transaction Creation

```
// Define a variable trans of type axi4stream_transaction to hold
// slave transaction record
axi4stream_transaction trans;

...

// Create a slave transaction
trans = bfm.create_slave_transaction();
```

Executing Transactions

Executing a transaction in a master/slave BFM test program initiates the transaction onto the protocol signals. Each master/slave BFM API has execution tasks that push transactions into the BFM internal transaction queues. Figure 2-1 on page 19 illustrates the internal BFM structure.

execute_transaction()

If the DUT is a slave, then the *execute_transaction()* task is called in the master BFM test program. Example 2-6 shows a master test program executing a master transaction.

Example 2-6. Master Test Program Transaction Execution

```
// Define a variable trans of type axi4stream_transaction to hold the
// master transaction record.
axi4stream_transaction trans;

...

// Create a master transaction with 10 transfers.
trans = bfm.create_master_transaction(10);

...

// By default the execution of a transaction will block.
bfm.execute_transaction(trans);
```

Waiting Events

Each BFM API has tasks that block the test program code execution until an event has occurred.

The *wait_on()* task blocks the test program execution until an ACLK or ARESETn signal event has occurred before proceeding.

The *get_packet(), get_transfer()* tasks block the test program code execution until a complete stream packet, or transfer, has occurred.

wait_on()

Example 2-7 shows a BFM test program waiting for the positive edge of the ARESETn signal.

Example 2-7. Test Program Wait for Event

```
// Block test program execution until the positive edge of the
// ARESETn signal.
bfm.wait_on(AXI4STREAM_RESET_POSEDGE);
```

get_packet(), get_transfer()

Example 2-8 shows a slave BFM test program using the *get_transfer()* task to block until it has received a data stream transfer.

Example 2-8. Slave Test Program get_transfer() Task

```
// Create a slave transaction.
trans = bfm.create_slave_transaction();

...

// Wait for a data stream transfer to occur.
bfm.get_transfer(trans, 0, last);
```

Access Transaction Record

Each BFM API has tasks that can access a complete or partially complete Transaction Record. The _set*()_ and _get*()_ tasks are used in a test program to set and get information from the transaction record.

Note

The _set*()_ and _get*()_ tasks are not explicitly detailed within each BFM API chapter. The simple rule for the task name is *set_* or *get_* followed by the name of the transaction field to be accessed. Refer to "Transaction Record Fields" on page 22 for transaction field name details.

set*()

Example 2-9 shows the master test program calling the *set_byte_type()* task to set the first data *byte_type* in the transaction.

Example 2-9. Master Test Program set_byte_type() Task

```
trans.set_byte_type(AXI4STREAM_DATA_BYTE, 0);
```

get*()

Example 2-10 shows the slave test program calling the *get_byte_type()* task to get the first data *byte_type* in the transaction.

Example 2-10. Slave Test Program get_byte_type() Task

```
// Define a variable of type axi4stream_byte_type_e to hold the byte
// type of the data stream byte.
axi4stream_byte_type_e slave_byte_type;

...

// Create a slave transaction.
trans = bfm.create_slave_transaction();

...

// Wait for a data stream transfer to occur.
bfm.get_transfer(trans, 0, last);

...

// Get the byte_type for the first data byte of the data stream transfer
slave_byte_type = trans.get_byte_type(0);
```

Operational Transaction Fields

Operational transaction fields control the way in which a transaction is executed onto the protocol signals. These fields also indicate when an individual data transfer or transaction is complete.

Operation Mode

By default, each transaction performs a blocking operation, which prevents a following transaction from starting until the current active transaction completes.

You can configure this behavior to be nonblocking by setting the *operation_mode* transaction field to the enumerate type value AXI4STREAM_TRANSACTION_NON_BLOCKING instead of the default AXI4STREAM_TRANSACTION_BLOCKING.

Example 2-11 shows a master BFM test program creating a transaction by calling the *create_master_transaction()* task. Before executing the transaction, the *operation_mode* task is changed to nonblocking.

Example 2-11. Master Test Program operation_mode() Task

```
// Define a variable trans of type axi4stream_transaction to hold the
// master transaction record.
axi4stream_transaction trans;

// Create a master transaction to create a transaction record
trans = bfm.create_master_transaction(1);

// Change the operation_mode to be nonblocking in the transaction record
trans.operation_mode(AXI4STREAM_TRANSACTION_NON_BLOCKING);
```

Handshake Delay

You can configure the TVALID and TREADY handshake signals to insert a delay before their assertion.

TVALID Signal Delay Transaction Field

The Transaction Record contains a *valid_delay* transaction field to configure the delay of the TVALID signal. The setting of the *valid_delay* transaction field is performed in the master BFM test program by calling the *set_valid_delay()* task.

TREADY Signal Delay Transaction Field

The Transaction Record contains a *ready_delay* transaction field to configure the delay of the TREADY signal. The setting of the *ready_delay* transaction field is performed in the slave BFM test program by calling the local *ready_delay()* task.

Example 2-12 shows the slave BFM test program implementing a *ready_delay()* task that inserts a specified delay before the assertion of the TREADY signal.

Example 2-12. Slave Test Program ready_delay() Task

```
// Task : ready_delay
// This is used to set ready delay to extend the transfer
task ready_delay();
   // Making TREADY '0'. This will consume one cycle.
   bfm.execute_stream_ready(0);
   // Two clock cycle wait. In total 3 clock wait.
   repeat(2) bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE);
   // Making TREADY '1'.
   bfm.execute_stream_ready(1);
endtask
```

Transfer Done

A *transfer_done* transaction field is set to 1 to indicate when each protocol “transfer” completes.

Transaction Done

A *transaction_done* transaction field is set to 1 to indicate when each protocol “transaction” completes.

In a slave BFM test program, you call the *get_transfer()* task to investigate whether a transaction is complete. If complete, the task returns the *last* argument of the task set to 1, and the transaction record will have the *transaction_done* field set to 1.

Chapter 3
SystemVerilog Master BFM

This section provides information about the SystemVerilog master BFM. It has an API that contains tasks and functions to configure the BFM and to access the dynamic Transaction Record during the life of the transaction.

Master BFM Protocol Support

The master BFM supports the full AMBA AXI4-Stream protocol.

Master Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI4-Stream Protocol Specification chapter, which you can use to reference details of the following master BFM API timing and events.

The AMBA AXI4-Stream Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the master BFM does not contain any timescale, timeunit, or timeprecision declarations. The signal setup and hold times are specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the test bench and design IP as a result of these directives, declarations, options, or initialization files:

- ` timescale directives in design elements
- Timeprecision declarations in design elements
- Compiler command-line options
- Simulation command-line options
- Local, or site-wide, simulator initialization files

If there is no timescale directive, the default time unit and time precision are tool specific. The recommended practice is to use timeunit and timeprecision declarations. For details, refer to Section 3.14, "System Time Units and Precision," of the *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language,* IEEE Std 1800™-2012 , February 21, 2013. This user guide refers to this document as the *IEEE Standard for SystemVerilog*.

Master BFM Configuration

A master BFM supports the full range of signals defined for the AMBA AXI4-Stream Protocol Specification. It has parameters that you use to configure the widths of the data and ID signals, and transaction fields to configure timeout factors, setup and hold times, and so on.

You can change the data and ID signals widths from their default settings by assigning them new values, usually in the top-level module of the test bench. These new values are then passed into the master BFM using a parameter port list of the master BFM module. Example 3-1 shows the master BFM with the data and ID signal widths defined in *module top()* and passed in to the master BFM *mgc_axi4stream_master* parameter port list.

Example 3-1. Master BFM Configuration

```
module top ();

   parameter AXI4STREAM_ID_WIDTH = 18;
   parameter AXI4STREAM_USER_WIDTH = 4;
   parameter AXI4STREAM_DEST_WIDTH = 4;
   parameter AXI4STREAM_DATA_WIDTH = 32;

   mgc_axi4stream_master #(AXI4STREAM_ID_WIDTH, AXI4STREAM_USER_WIDTH,
AXI4STREAM_DEST_WIDTH, AXI4STREAM_DATA_WIDTH) bfm_master(....);
```

Note

In the above code extract, the *mgc_axi4stream_master* is the master BFM interface.

Table 3-1 lists the parameter names for the data and ID signals, and their default values.

Table 3-1. Master BFM Signal Width Parameters

Signal Width Parameter	Description
AXI4_ID_WIDTH	ID signal width in bits. This applies to the TID signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_USER_WIDTH	User data signal width in bits. This applies to the TUSER signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 8.
AXI4_DEST_WIDTH	Destination routing signal width in bits. This applies to the TDEST signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_DATA_WIDTH	Data signal width in bits. This applies to the TDATA signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 1024.

A master BFM has configuration fields that you set by calling the *set_config()* function to configure timeout factors, setup and hold times, and so on. You get the value of a configuration field using the *get_config()* function. Table 3-2 describes the full list of configuration fields.

Table 3-2. Master BFM Configuration

Configuration Field	Description
Timing Variables	
AXI4STREAM_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay permitted between the individual transfer transactions in clock cycles. Default: 10000.
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ ASSERTION_TO_TREADY	The maximum delay permitted between the assertion of TVALID to the assertion of TREADY. Default: 10000.
Master Attributes	
AXI4STREAM_LAST_DURING_IDLE	Controls the value of *T*LAST during idle. 0 = TLAST driven to 0 during idle (default) 1 = TLAST driven to 1 during idle
Error Detection	
AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
AXI4STREAM_CONFIG_ENABLE_ASSERTION	Individual enable/disable of an assertion check in the BFM. Refer to the Master Assertions chapter for details 0 = disabled 1 = enabled (default)

[1] Refer to Master Timing and Events for details of simulator time-steps.

Master Assertions

The master BFM performs protocol error checking using built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

By default, all built-in assertions are enabled in the master BFM. To globally disable them in the master BFM, use the *set_config()* command as shown in Example 3-2.

Example 3-2. Master BFM Disable All Assertions

```
set_config(AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS,0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. Example 3-3 shows how to disable assertion checking for the TLAST signal changing between the TVALID and TREADY handshake signals.

Example 3-3. Master BFM Individual Assertion Enable/Disable

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector =
bfm.get_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION);

// Assign the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion bit to 0
config_assert_bitvector[AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION,
config_assert_bitvector);
```

Note
Do not confuse the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector with the AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion, follow the code sequence in Example 3-3 and assign the assertion enable within the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI4-Stream assertions, refer to “Assertions” on page 203.

SystemVerilog Master API

This section describes the SystemVerilog master BFM API.

Each task and function available within the master BFM API is detailed with the exception of the *set*()* and *get*()* tasks that operate on the Transaction Record. The simple rule for the task name is *set_* or *get_* followed by the name of the transaction field to be accessed. Refer to "Transaction Record" on page 21 for details of transaction field names.

Note

The master BFM API is the *axi4stream/bfm//mgc_axi4stream_master.sv* file packaged within the Mentor Verification IP Altera Edition.

set_config()

This function sets the configuration of the master BFM.

Prototype

```
function void set_config
(
   input axi4stream_config_e config_name,
   input axi4stream_max_bits_t config_val
);
```

Arguments

config_name	Configuration name: AXI4STREAM_CONFIG_SETUP_TIME AXI4STREAM_CONFIG_HOLD_TIME AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR AXI4STREAM_CONFIG_LAST_DURING_IDLE AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS AXI4STREAM_CONFIG_ENABLE_ASSERTION
config_val	See “Master BFM Configuration” on page 30 for more details.

Returns None

Example

```
set_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

This function gets the configuration of the master BFM.

Prototype

```
function axi4stream_max_bits_t get_config
(
   input axi4stream_config_e config_name,
);
```

Arguments config_name Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

Returns config_val See “Master BFM Configuration” on page 30 for more details.

Example

```
get_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR);
```

create_master_transaction()

This nonblocking function creates a master transaction with an optional *burst_length* argument. All other transaction fields default to legal protocol values, unless previously assigned a value. It returns with the *axi4stream_transaction* record.

Prototype

```
function automatic axi4stream_transaction
create_master_transaction
(
   input int burst_length = 1 // optional
);
```

Arguments	burst_length	(Optional) Number of transfers within a packet. Default: 1.
Protocol Transaction Fields	data	Data array in bytes.

	byte_type	Byte type array:
		AXI4STREAM_DATA_BYTE; (default)
		AXI4STREAM_NULL_BYTE;
		AXI4STREAM_POS_BYTE;
		AXI4STREAM_ILLEGAL_BYTE;

	id	Data stream identifier.
	dest	Destination routing information.
	user_data	User data array.
Operational Transaction Fields	operation_ mode	Operation mode: AXI4STREAM_TRANSACTION_NON_BLOCKING; AXI4STREAM_TRANSACTION_BLOCKING; (default)
	valid_delay	TVALID delay measured in ACLK cycles for this transaction. (default = 0).
	ready_delay	TREADY delay measured in ACLK cycles for this transaction. (default = 0).
	transfer_done	Transfer *done* flag array for this transaction
	transaction_ done	Transaction *done* flag for this transaction
Returns	trans	The *axi4stream_transaction* record.

Example

```
// Create a master transaction with a data burst length of 3.
trans = bfm.create_write_transaction(3);
trans.set_data[0] = 'hACE0ACE1;
trans.set_data[1] = 'hACE2ACE3;
trans.set_data[2] = 'hACE4ACE5;
```

execute_transaction()

This task executes a master transaction previously created by the *create_master_transaction()* function. The transaction may be blocking (default) or nonblocking, as defined by the transaction record *operation_mode* field.

It calls the *execute_transfer()* task for each transfer within a packet, with the number of transfers defined by the transaction *burst_length* field.

Prototype

```
task automatic execute_transaction
(
   axi4stream_transaction trans
)
```

Arguments	trans	The *axi4stream_transaction* record.
Returns	None	

Example

```
// Declare a local variable trans to hold the transaction record.
axi4stream_transaction trans;

// Create a master transaction with a transfer count of 3 and assign
// it to the local trans variable.
trans = bfm.create_master_transaction(3);

....

// Execute the trans transaction.
bfm.execute_transaction(trans);
```

execute_transfer()

This task executes a master transfer previously created by the *create_master_transaction()* function. This task may be blocking (default) or nonblocking, as defined by the transaction *operation_mode* field.

It sets the TVALID protocol signal at the appropriate time defined by the transaction *valid_delay* field, and sets the *transfer_done* array *index* element field to 1 when the transfer is complete.

If this is the last transfer of the transaction, then it sets the *transaction_done* field to 1 and returns the *last* argument set to 1 to indicate the whole transaction is complete.

Prototype

```
task automatic execute_transfer
(
   axi4stream_transaction trans,
   int index = 0, // Optional
   output bit last
);
```

Arguments

Arguments	trans	The *axi4stream_transaction* record.
	index	(Optional) Transfer number.
Returns	last	

Example

```
// Declare a local variable to hold the transaction record.
axi4stream_transaction trans;

// Create a master transaction with a transfer count of 3 and assign
// it to the local trans variable.
trans = bfm.create_master_transaction(3);

....

// Execute the first transfer of the trans transaction.
bfm.execute_transfer(trans, 0, last);

// Execute the second transfer of the trans transaction0.
bfm.execute_transfer(trans, 1, last);
```

get_stream_ready()

This blocking task returns the value of the TREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype	`task automatic get_stream_ready`
	`(`
	`   output bit ready`
	`);`	

Arguments	ready	The value of the TREADY signal.
Returns	ready	

Example

```
// Get the value of the TREADY signal
bfm.get_stream_ready(ready);
```

wait_on()

This blocking task waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
task automatic wait_on
(
   axi4stream_wait_e phase,
   input int count = 1 //Optional
);
```

Arguments

phase	Wait for: AXI4STREAM_CLOCK_POSEDGE AXI4STREAM_CLOCK_NEGEDGE AXI4STREAM_CLOCK_ANYEDGE AXI4STREAM_CLOCK_0_TO_1 AXI4STREAM_CLOCK_1_TO_0 AXI4STREAM_RESET_POSEDGE AXI4STREAM_RESET_NEGEDGE AXI4STREAM_RESET_ANYEDGE AXI4STREAM_RESET_0_TO_1 AXI4STREAM_RESET_1_TO_0

Example

```
bfm.wait_on(AXI4STREAM_RESET_POSEDGE);
bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE, 10);
```

Chapter 4
SystemVerilog Slave BFM

This section provides information about the SystemVerilog slave BFM. It has an API that contains tasks and functions to configure the BFM and to access the dynamic Transaction Record during the lifetime of a transaction.

Slave BFM Protocol Support

The slave BFM supports the full AMBA AXI4-Stream protocol.

Slave Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI4-Stream Protocol Specification chapter, which you can reference for details of the following slave BFM API timing and events.

The AMBA AXI4-Stream Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the slave BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the test bench and design IP as a result of these directives, declarations, options, or initialization files:

- ` timescale directives in design elements
- Timeprecision declarations in design elements
- Compiler command-line options
- Simulation command-line options
- Local or site-wide simulator initialization files

If there is no timescale directive, the default time unit and time precision are tool specific. The recommended practice is to use timeunit and timeprecision declarations. Refer to the *IEEE Standard for SystemVerilog*, Section 3.14, for details.

Slave BFM Configuration

The slave BFM supports the full range of signals defined for the AMBA AXI4-Stream Protocol Specification. It has parameters that you use to configure the widths of the data and ID signals, and transaction fields to configure timeout factors, setup and hold times, and so on.

You can change the data and ID signal widths from their default settings by assigning them new values, usually in the top-level module of the test bench. These new values are then passed into the slave BFM using a parameter port list of the slave BFM module. Example 4-1 shows the slave BFM with the data and ID signal widths defined in *module top()* and passed in to the slave BFM *mgc_axi4stream_slave* parameter port list.

Example 4-1. Slave BFM Configuration

```
module top ();

   parameter AXI4STREAM_ID_WIDTH = 18;
   parameter AXI4STREAM_USER_WIDTH = 4;
   parameter AXI4STREAM_DEST_WIDTH = 4;
   parameter AXI4STREAM_DATA_WIDTH = 32;

   mgc_axi4stream_slave #(AXI4STREAM_ID_WIDTH, AXI4STREAM_USER_WIDTH,
AXI4STREAM_DEST_WIDTH, AXI4STREAM_DATA_WIDTH) bfm_slave(....);
```

Note

In the Example 4-1 code extract, the *mgc_axi4stream_slave* is the slave BFM interface.

Table 4-1 lists the parameter names for the data and ID signals and their default values.

Table 4-1. Slave BFM Signal Width Parameters

Signal Width Parameter	Description
AXI4_ID_WIDTH	ID signal width in bits. This applies to the TID signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_USER_WIDTH	User data signal width in bits. This applies to the TUSER signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 8.
AXI4_DEST_WIDTH	Destination routing signal width in bits. This applies to the TDEST signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_DATA_WIDTH	Data signal width in bits. This applies to the TDATA signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 1024.

A slave BFM has configuration fields that you can set using the *set_config()* function to configure timeout factors, setup and hold times, and so on. You can also get the value of a configuration field with the *get_config()* function. Table 4-2 lists the configuration fields.

Table 4-2. Slave BFM Configuration

Configuration Field	Description
Timing Variables	
AXI4STREAM_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay permitted between the individual transfer transactions in clock cycles. Default: 10000.
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY	The maximum delay permitted between the assertion of TVALID to the assertion of TREADY. Default: 10000.
Master Attributes	
AXI4STREAM_LAST_DURING_IDLE	Controls the value of TLAST during idle. 0 = TLAST driven to 0 during idle (default) 1 = TLAST driven to 1 during idle
Error Detection	
AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
AXI4STREAM_CONFIG_ENABLE_ASSERTION	Individual enable/disable of an assertion check in the BFM. Refer to Slave Assertions for details 0 = disabled 1 = enabled (default)

[1] Refer to Slave Timing and Events for details of simulator time-steps.

Slave Assertions

The slave BFM performs protocol error checking using built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

By default, all built-in assertions are enabled in the slave BFM. To globally disable them in the slave BFM, use the *set_config()* command as shown in Example 4-2.

Example 4-2. Slave BFM Disable All Assertions

```
set_config(AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS,0)
```

Alternatively, individual built-in assertions may be disabled by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. Example 4-3 shows how to disable assertion checking for the TLAST signal changing between the TVALID and TREADY handshake signals.

Example 4-3. Slave BFM Individual Assertion Enable/Disable

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector =
bfm.get_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION);

// Assign the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion bit to 0
config_assert_bitvector[AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION,
config_assert_bitvector);
```

Note
Do not confuse the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector with the AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion, follow the code sequence in Example 4-3 and assign the assertion within the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI4-Stream assertions, refer to “Assertions” on page 203.

SystemVerilog Slave API

This section describes the SystemVerilog slave BFM API

Each task and function available within the slave BFM API is detailed with the exception of the *set*()* and *get*()* tasks that operate on the Transaction Record. The simple rule for the task name is *set_* or *get_* followed by the name of the transaction field to be accessed. Refer to "Transaction Record" on page 21 for details of transaction field names.

Note

The slave BFM API is the *axi4stream/bfm//mgc_axi4stream_slave.sv* file packaged within the Mentor Verification IP Altera Edition.

set_config()

This function sets the configuration of the slave BFM.

Prototype

```
function void set_config
(
   input axi4stream_config_e config_name,
   input axi4stream_max_bits_t config_val
);
```

Arguments

config_name Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

config_val See "Slave BFM Configuration" on page 42 for more details.

Returns None

Example

```
set_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

This function gets the configuration of the slave BFM.

Prototype

```
function axi4stream_max_bits_t get_config
(
   input axi4stream_config_e config_name,
);
```

Arguments config_name — Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

Returns config_val — See “Slave BFM Configuration” on page 42 for more details.

Example

```
get_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR);
```

create_slave_transaction()

This nonblocking function creates a slave transaction. All transaction fields default to legal protocol values unless previously assigned a value. It returns with the *axi4stream_transaction* record.

Prototype	`function automatic axi4stream_transaction create_slave_transaction();`	
Protocol Transaction Fields	burst_length	(Optional) Number of transfers within a packet. Default: 1.
	data	Data array in bytes.

	byte_type	Byte type:
		AXI4STREAM_DATA_BYTE; (default)
		AXI4STREAM_NULL_BYTE;
		AXI4STREAM_POS_BYTE;
		AXI4STREAM_ILLEGAL_BYTE;

	id	Data stream identifier.
	dest	Destination routing information.
	user_data	User data array.
Operational Transaction Fields	operation_ mode	Operation mode: AXI4STREAM_TRANSACTION_NON_BLOCKING; AXI4STREAM_TRANSACTION_BLOCKING; (default)
	valid_delay	TVALID delay measured in ACLK cycles for this transaction. (default = 0).
	ready_delay	TREADY delay measured in ACLK cycles for this transaction. (default = 0).
	transfer_done	Transfer *done* flag array for this transaction.
	transaction_ done	Transaction *done* flag for this transaction.
Returns	trans	The *axi4stream_transaction* record.

Example

```
// Create a slave transaction.
trans = bfm.create_slave_transaction();
```

get_transfer()

This blocking task gets a slave transfer previously created by the *create_slave_transaction()* function, and identified by the optional *index* argument.

It sets the TREADY protocol signal at the appropriate time defined by the transaction *ready_delay* field, and sets the *transfer_done* array *index* element field to 1 when the transfer is complete.

If this is the last transfer of the transaction, then it sets the *transaction_done* field to 1 and returns the *last* argument set to 1 to indicate the whole transaction is complete.

Prototype

```
task automatic get_transfer
(
   axi4stream_transaction trans,
   int index = 0, // Optional
   output bit last
);
```

Arguments

trans	The *axi4stream_transaction* record.
index	(Optional) Transfer number.
last	Flag to indicate the last transfer in the packet.

Returns last

Example

```
// Declare a local variable to hold the transaction record.
axi4stream_transaction trans;

// Create a slave transaction and assign it to the local
// trans variable.
trans = bfm.create_slave_transaction();

....

// Get the first transfer of the trans transaction.
bfm.get_transfer(trans, 0, last);

// Get the second transfer of the trans transaction.
bfm.get_transfer(trans, 1, last);
```

execute_stream_ready()

This task executes a slave ready by placing the state of the *ready* input argument onto the TREADY signal. This task may be blocking (default) or nonblocking, as defined by the optional *non_blocking_mode* input argument.

Prototype

```
task automatic execute_stream_ready
(
   input bit ready,
   input bit non_blocking_mode = 0 // Optional
);
```

Arguments

ready	The value to be placed onto the TREADY signal.
non_blocking_mode	(Optional) Controls the blocking or nonblocking mode of the task. 0 = blocking (default) 1 = nonblocking

Returns None

Example

```
// Assign TREADY = '0'. This will consume one cycle.
bfm.execute_stream_ready(0);

// Two clock cycle wait.
repeat(2) bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE);

// Assign TREADY = '1'.
bfm.execute_stream_ready(1);
```

wait_on()

This blocking task waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
task automatic wait_on
(
   axi4stream_wait_e phase,
   input int count = 1 //Optional
);
```

Arguments		
	phase	Wait for: AXI4STREAM_CLOCK_POSEDGE AXI4STREAM_CLOCK_NEGEDGE AXI4STREAM_CLOCK_ANYEDGE AXI4STREAM_CLOCK_0_TO_1 AXI4STREAM_CLOCK_1_TO_0 AXI4STREAM_RESET_POSEDGE AXI4STREAM_RESET_NEGEDGE AXI4STREAM_RESET_ANYEDGE AXI4STREAM_RESET_0_TO_1 AXI4STREAM_RESET_1_TO_0

Example

```
bfm.wait_on(AXI4STREAM_RESET_POSEDGE);
bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE, 10);
```

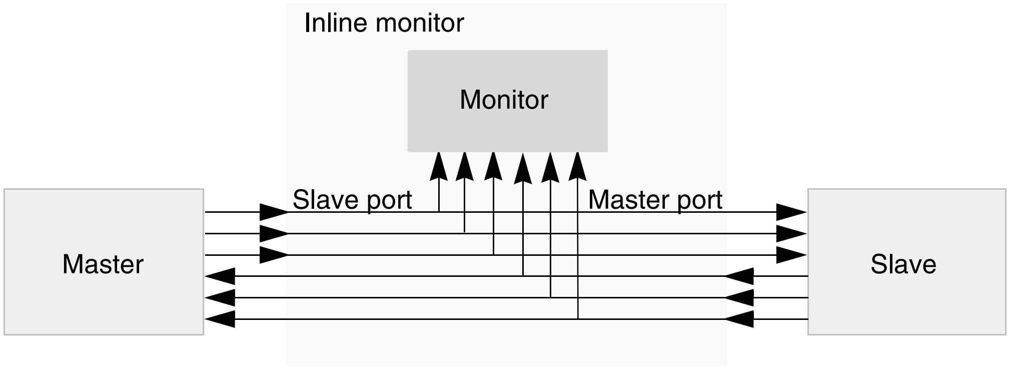
Chapter 5
SystemVerilog Monitor BFM

This section provides information about the SystemVerilog monitor BFM. It has an API that contains tasks and functions to configure the BFM and to access the dynamic Transaction Record during the life of the transaction.

Inline Monitor Connection

The connection of a monitor BFM to a test environment differs from that of a master and slave BFM. It is wrapped within an inline monitor interface and connected inline between a master and slave, as shown in Figure 5-1. It has separate master and slave ports, and monitors protocol traffic between a master and slave. The monitor has access to all the facilities provided by the monitor BFM.

Figure 5-1. Inline Monitor Connection Diagram



Monitor BFM Protocol Support

The monitor BFM supports the full AMBA AXI4-Stream Protocol.

Monitor Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI4-Stream Protocol Specification chapter, which you can reference for details of the following monitor BFM API timing and events.

The AMBA AXI4-Stream Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the monitor BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the test bench and design IP as a result of these directives, declarations, options, or initialization files:

- ` timescale directives in design elements
- Timeprecision declarations in design elements
- Compiler command-line options
- Simulation command-line options
- Local, or site-wide, simulator initialization files

If there is no timescale directive, the default time unit and time precision are tool specific. The recommended practice is to use timeunit and timeprecision declarations. Refer to the *IEEE Standard for SystemVerilog*, Section 3.14, for details.

Monitor BFM Configuration

The monitor BFM supports the full range of signals defined for the AMBA AXI4-Stream Protocol Specification. It has parameters that you use to configure the widths of the data and ID signals, and transaction fields to configure timeout factors, setup and hold times, and so on.

You can change the data and ID signals widths from their default settings by assigning them with new values, usually in the top-level module of the test bench. These new values are then passed into the monitor BFM using a parameter port list of the monitor BFM module. Example 5-1 shows the monitor BFM with the data and ID signal widths defined in *module top()* and passed in to the monitor BFM *mgc_axi4stream_monitor* parameter port list.

Example 5-1. Monitor BFM Configuration

```
module top ();

   parameter AXI4STREAM_ID_WIDTH = 18;
   parameter AXI4STREAM_USER_WIDTH = 4;
   parameter AXI4STREAM_DEST_WIDTH = 4;
   parameter AXI4STREAM_DATA_WIDTH = 32;

   mgc_axi4stream_monitor #(AXI4STREAM_ID_WIDTH, AXI4STREAM_USER_WIDTH,
AXI4STREAM_DEST_WIDTH, AXI4STREAM_DATA_WIDTH) bfm_monitor(....);
```

Note

In the above code extract, the *mgc_axi4stream_monitor* is the monitor BFM interface.

Table 5-1 describes the parameter names for the data and ID signals and their default values.

Table 5-1. Monitor BFM Signal Width Parameters

Signal Width Parameter	Description
AXI4_ID_WIDTH	ID signal width in bits. This applies to the TID signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_USER_WIDTH	User data signal width in bits. This applies to the TUSER signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 8.
AXI4_DEST_WIDTH	Destination routing signal width in bits. This applies to the TDEST signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_DATA_WIDTH	Data signal width in bits. This applies to the TDATA signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 1024.

A monitor BFM has configuration fields that set with the *set_config()* function to configure timeout factors, setup and hold times, and so on. You get the value of a configuration field with the *get_config()* function. Table 5-2 describes the configuration fields.

Table 5-2. Monitor BFM Configuration

Configuration Field	Description
Timing Variables	
AXI4STREAM_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay permitted between the individual transfer transactions in clock cycles. Default: 10000.
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY	The maximum delay permitted between the assertion of TVALID to the assertion of TREADY. Default: 10000.

Table 5-2. Monitor BFM Configuration (cont.)

Configuration Field	Description
Master Attributes	
AXI4STREAM_LAST_DURING_IDLE	Controls the value of TLAST during idle. 0 = TLAST driven to 0 during idle (default) 1 = TLAST driven to 1 during idle
Error Detection	
AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
AXI4STREAM_CONFIG_ENABLE_ASSERTION	Individual enable/disable of an assertion check in the BFM. Refer to Monitor Assertions for details 0 = disabled 1 = enabled (default)

[1] Refer to Monitor Timing and Events for details of simulator time-steps.

Monitor Assertions

The monitor BFM performs protocol error checking using built-in assertions.

Note

The built-in BFM assertions are independent of programming language and simulator.

By default, all built-in assertions are enabled in the monitor BFM. To globally disable them in the monitor BFM, use the *set_config()* command as shown in Example 5-2.

Example 5-2. Monitor BFM Disable All Assertions

```
set_config(AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS,0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. Example 5-3 shows how to disable assertion checking for the TLAST signal changing between the TVALID and TREADY handshake signals.

Example 5-3. Monitor BFM Individual Assertion Enable/Disable

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector =
bfm.get_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION);

// Assign the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion bit to 0
config_assert_bitvector[AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION,
config_assert_bitvector);
```

Note

Do not confuse the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector with the AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion, follow the code sequence in Example 5-3 and assign the assertion within the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI4-Stream assertions, refer to “Assertions” on page 203.

SystemVerilog Monitor API

This section describes the SystemVerilog monitor BFM API.

Each task and function available within the monitor BFM API is detailed with the exception of the *set*()* and *get*()* tasks that operate on the Transaction Record. The simple rule for the task name is *set_* or *get_* followed by the name of the transaction field to be accessed. Refer to “Transaction Record” on page 21 for details of transaction field names

Note

The monitor BFM API is the *axi4stream/bfm//mgc_axi4stream_monitor.sv* file packaged within the Mentor Verification IP Altera Edition.

set_config()

This function sets the configuration of the monitor BFM.

Prototype

```
function void set_config
(
   input axi4stream_config_e config_name,
   input axi4stream_max_bits_t config_val
);
```

Arguments

config_name Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_
TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

config_val See “Monitor BFM Configuration” on page 54 for more details.

Returns None

Example

```
set_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

This function gets the configuration of the monitor BFM.

Prototype

```
function axi4stream_max_bits_t get_config
(
   input axi4stream_config_e config_name,
);
```

Arguments config_name Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

Returns config_val See “Monitor BFM Configuration” on page 54 for more details.

Example

```
get_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR);
```

create_monitor_transaction()

This nonblocking function creates a monitor transaction. All transaction fields default to legal protocol values, unless previously assigned a value. It returns with the *axi4stream_transaction* record.

Prototype

```
function automatic axi4stream_transaction
create_monitor_transaction();
```

Protocol Transaction Fields	burst_length	(Optional) Number of transfers within a packet. Default: 1.
	data	Data array in bytes.

	byte_type	Byte type:
		AXI4STREAM_DATA_BYTE; (default)
		AXI4STREAM_NULL_BYTE;
		AXI4STREAM_POS_BYTE;
		AXI4STREAM_ILLEGAL_BYTE;

	id	Data stream identifier.
	dest	Destination routing information.
	user_data	User data array.
Operational Transaction Fields	operation_ mode	Operation mode: AXI4STREAM_TRANSACTION_NON_BLOCKING; AXI4STREAM_TRANSACTION_BLOCKING; (default)
	valid_delay	TVALID delay measured in ACLK cycles for this transaction. (default = 0).
	ready_delay	TREADY delay measured in ACLK cycles for this transaction. (default = 0).
	transfer_done	Transfer *done* flag array for this transaction
	transaction_ done	Transaction *done* flag for this transaction
Returns	trans	The *axi4stream_transaction* record.

Example

```
// Create a monitor transaction
trans = bfm.create_monitor_transaction();
```

get_packet()

This blocking task gets a monitor packet previously created by the *create_monitor_transaction()* function.

It calls the *get_transfer()* task for each transfer of the packet with the number of transfers defined by the transaction record *burst_length* field.

Prototype

```
task automatic get_packet
(
   axi4stream_transaction trans
);
```

Arguments trans The *axi4stream_transaction* record.

Returns None

Example

```
// Declare a local variable to hold the transaction record.
axi4stream_transaction trans;

// Create a monitor transaction and assign it to the local
// trans variable.
trans = bfm.create_monitor_transaction();

....

// Get the packet of the trans transaction.
bfm.get_packet(trans);
```

get_transfer()

This blocking task gets a monitor transfer previously created by the *create_monitor_transaction()* function and identified by the optional *index* argument.

It sets the *transfer_done* array *index* element field to 1 when the transfer completes.

If this is the last transfer of the transaction, then it sets the *transaction_done* field to 1 and returns the *last* argument set to 1 to indicate the whole transaction is complete.

Prototype

```
task automatic get_transfer
(
   axi4stream_transaction trans,
   int index = 0, // Optional
   output bit last
);
```

Arguments	trans	The *axi4stream_transaction* record.
	index	(Optional) Transfer number.
	last	Flag to indicate the last transfer in the packet.
Returns	last	

Example

```
// Declare a local variable to hold the transaction record.
axi4stream_transaction trans;

// Create a monitor transaction and assign it to the local
// trans variable.
trans = bfm.create_monitor_transaction();

....

// Get the first transfer of the trans transaction.
bfm.get_transfer(trans, 0, last);

// Get the second transfer of the trans transaction.
bfm.get_transfer(trans, 1, last);
```

get_stream_ready()

This blocking task gets the state of the TREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
task automatic get_stream_ready
(
   output bit ready
);
```

Arguments	ready	The value on the TREADY signal.
Returns	ready	

Example

```
// Get the value of the TREADY signal
bfm.get_stream_ready(ready);
```

wait_on()

This blocking task waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
task automatic wait_on
(
   axi4stream_wait_e phase,
   input int count = 1 //Optional
);
```

Arguments

phase Wait for:

AXI4STREAM_CLOCK_POSEDGE
AXI4STREAM_CLOCK_NEGEDGE
AXI4STREAM_CLOCK_ANYEDGE
AXI4STREAM_CLOCK_0_TO_1
AXI4STREAM_CLOCK_1_TO_0
AXI4STREAM_RESET_POSEDGE
AXI4STREAM_RESET_NEGEDGE
AXI4STREAM_RESET_ANYEDGE
AXI4STREAM_RESET_0_TO_1
AXI4STREAM_RESET_1_TO_0

Example

```
bfm.wait_on(AXI4STREAM_RESET_POSEDGE);
bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE, 10);
```

Chapter 6
SystemVerilog Tutorials

This chapter discusses how to use the Mentor VIP AE master and slave BFMs to verify slave and master DUT components, respectively.

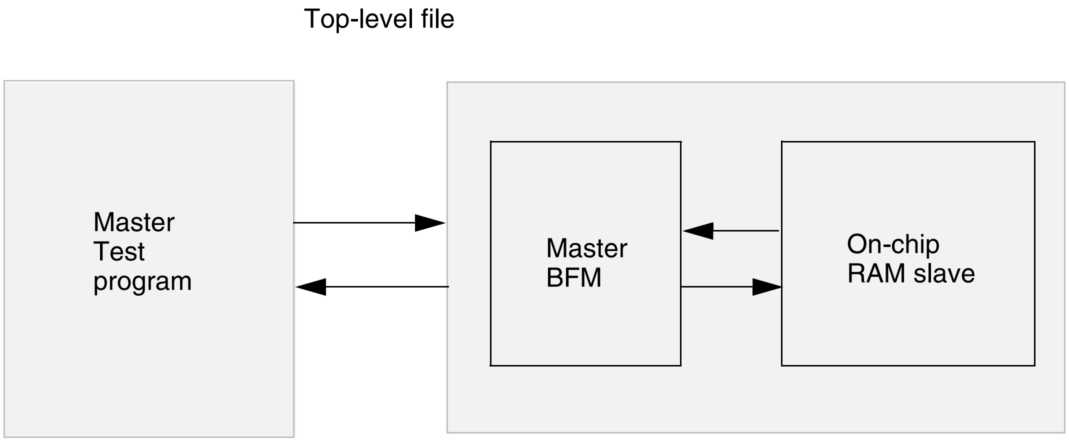
In the Verifying a Slave DUT tutorial, the slave is verified using a master BFM and test program. In the Verifying a Master DUT tutorial, the master issues "transfers" that are verified using a slave BFM and test program.

Following this top-level discussion of how you verify a master and a slave component using the Mentor VIP AE is a brief example of how to run Qsys, the powerful system integration tool in Quartus® II software. This procedure shows you how to use Qsys to create a top-level DUT environment. For more details about this example, refer to "Getting Started with Qsys and the BFMs" on page 187.

Verifying a Slave DUT

A slave DUT component is connected to a master BFM at the signal level. A master test program written at the transaction level generates stimulus using the master BFM to verify the slave DUT. Figure 6-1 illustrates a typical top-level test bench environment.

Figure 6-1. Slave DUT Top-Level Test Bench Environment



A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

Master BFM Test Program

A master BFM test program is capable of creating a wide range of stimulus scenarios to verify a slave DUT. For a complete code listing of this master test program, refer to "SystemVerilog Master Test Program" on page 207.

The master test program contains an Initial Block that creates and executes master transactions over the protocol signals. The following sections describe the main procedures and variables:

Initial Block

Within an *initial* block, the master test program defines a transaction variable *trans* of type *axi4stream_transaction* to hold a record of a transaction during its life, as shown in Example 6-1. The initial wait for the ARESETn signal to be deactivated, followed by a positive ACLK edge, satisfies the protocol requirement detailed in Section 2.7.2 of the AMBA AXI4-Stream Protocol Specification.

Example 6-1. Definition and Initialization

```
initial
begin
    axi4stream_transaction trans;
    static int byte_count = AXI4_DATA_WIDTH/8;
    int transfer_count;
    bit last;
    /******************
    ** Initialisation **
    ******************/
    bfm.wait_on(AXI4STREAM_RESET_POSEDGE);
    bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE);
```

An outer *for* loop increments the *transfer_count* on each iteration of the loop, as shown in Example 6-2. Calling the *create_master_transaction()* function creates a master transaction, passing in the optional *transfer_count* as an argument to the function. The created master transaction is then assigned to the transaction variable *trans*. The TID and TDEST signal values are then assigned for the data stream. Each iteration of the outer loop creates a master transaction with the *transfer_count* per transaction passed as an argument.

An inner *for* loop calls the *trans.set_data()* task to load a byte into the *data* transaction field, and calls the *trans.set_byte_type()* task to load the *byte_type* transaction field for the byte.

Calling the *execute_transaction()* task executes the *trans* transaction onto the protocol signals.

Example 6-2. Master Transaction Creation and Execution

```
/************************
** Traffic generation: **
************************/
// 10 x packet with
// Number of transfer = i % 10. Values : 1, 2 .. 10
// id = i % 15. Values 0, 1, 2 .. 14
// dest = i %20. Values 0, 1, 2 .. 19
for(int i = 0; i < 10; ++i)
begin
   transfer_count = (i % 10) + 1;
   trans = bfm.create_master_transaction(transfer_count);
   trans.set_id = (i % 15);
   trans.set_dest = (i % 20);
   for(int j = 0; j < (transfer_count * byte_count); ++j)
   begin
      trans.set_data(i + j, j);
      if(((i + j)% 5) == 0)
      begin
         trans.set_byte_type(AXI4STREAM_NULL_BYTE, j);
      end
      else if(((i + j)% 5) == 1)
      begin
         trans.set_byte_type(AXI4STREAM_POS_BYTE, j);
      end
      else
      begin
         trans.set_byte_type(AXI4STREAM_DATA_BYTE, j);
      end
   end
   bfm.execute_transaction(trans);
end
```

The master test program repeats the creation of master transactions similar to that shown in Example 6-2, but instead calls the *execute_transfer()* task per iteration of the inner *for* loop, as shown in Example 6-3.

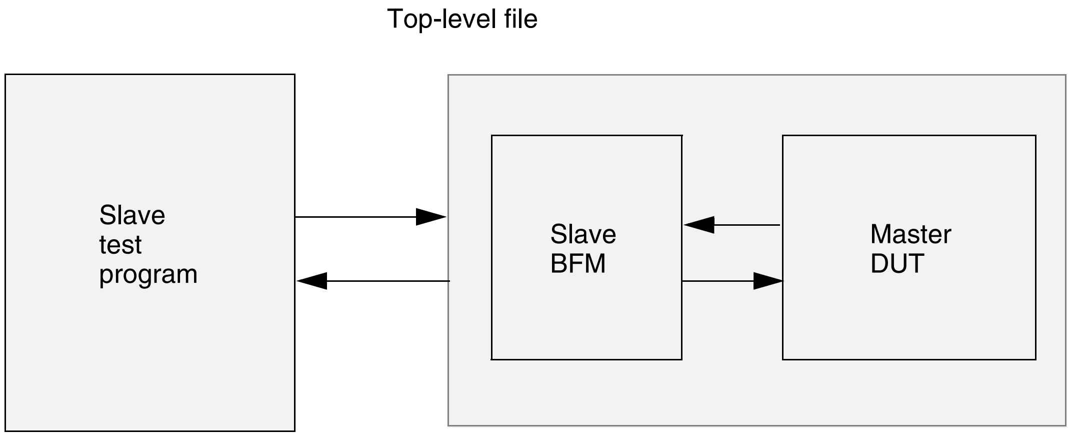
Example 6-3. Master Transfer Execution

```
// 10 x packet at transfer level with
// Number of transfer = i % 10. Values : 1, 2 .. 10
// id = i % 15. Values 0, 1, 2 .. 14
// dest = i %20. Values 0, 1, 2 .. 19
for(int i = 0; i < 10; ++i)
begin
   transfer_count = (i % 10) + 1;
   trans = bfm.create_master_transaction(transfer_count);
   trans.set_id = (i % 15);
   trans.set_dest = (i % 20);
   for(int j = 0; j < transfer_count; j= j + byte_count)
   begin
      for(int k = j; k < byte_count; ++k)
      begin
         trans.set_data(k, k);
         if(((i + j)% 5) == 0)
         begin
            trans.set_byte_type(AXI4STREAM_NULL_BYTE, k);
         end
         else if(((i + j)% 5) == 1)
         begin
            trans.set_byte_type(AXI4STREAM_POS_BYTE, k);
         end
         else
         begin
            trans.set_byte_type(AXI4STREAM_DATA_BYTE, k);
         end
      end
      bfm.execute_transfer(trans, j / byte_count, last);
   end
end
```

Verifying a Master DUT

A master DUT component is connected to a slave BFM at the signal level. A slave test program written at the transaction level generates stimulus using the slave BFM to verify the master DUT. Figure 6-2 illustrates a typical top-level test bench environment.

Figure 6-2. Master DUT Top-Level Test Bench Environment



A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

Slave BFM Test Program

The slave test program contains a Basic Slave Test Program API Definition that implements a simplified interface for you to start verifying a master DUT with minimal effort. The API allows the slave BFM to control back-pressure to the master DUT by configuring the delay for the assertion of the TREADY signal. No other slave test program editing is required in this case.

The Advanced Slave Test Program API Definition allows the slave BFM to receive protocol transfers and insert a delay for the assertion of the TREADY signal. No further analysis of the protocol transfer content is performed. If further analysis is required then the slave test program will require editing to add this feature.

For a complete code listing of the slave test program, refer to "SystemVerilog Slave Test Program" on page 209.

Basic Slave Test Program API Definition

The Basic Slave Test Program API contains the following:

- Configuration variable *m_insert_wait* to insert a delay in the assertion of the TREADY protocol signal.

- Task *ready_delay()* to configure the delay of the TREADY signal.

m_insert_wait

The *m_insert_wait* configuration variable controls the insertion of a delay for the TREADY signal defined by the *ready_delay()* task. To insert a delay set *m_insert_wait* to 1 (default); otherwise, set to 0, as shown in Example 6-4.

Example 6-4. m_insert_wait

```
// This member controls the wait insertion in axi4 stream transfers
// coming from master.
// Assigning m_insert_wait to 0 turns off the wait insertion.
bit m_insert_wait = 1;
```

ready_delay()

The *ready_delay* task inserts a delay for the TREADY signal. The delay value extends the length of a protocol transfer by a defined number of ACLK cycles. The starting point of the delay is determined by the completion of a previous transfer, or from the first positive ACLK edge after reset at the start of simulation.

The *ready_delay()* task initially sets TREADY to 0 by calling the *execute_stream_ready()* task, as shown in Example 6-5. The delay is inserted by calling the *wait_on()* task within a *repeat()* statement. You can edit the number of repetitions to change the delay. After the delay, the *execute_stream_ready()* task is called again to set the TREADY signal to 1.

Example 6-5. ready_delay()

```
// Task : ready_delay
// This is used to set ready delay to extend the transfer
task ready_delay();
   // Making TREADY '0'. This will consume one cycle.
   bfm.execute_stream_ready(0);
   // Two clock cycle wait. In total 3 clock wait.
   repeat(2) bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE);
   // Making TREADY '1'.
   bfm.execute_stream_ready(1);
endtask
```

Note

In addition to the above tasks and variables, you can configure other aspects of the slave BFM by using these functions: *“set_config()”* on page 46 and *“get_config()”* on page 47.

Advanced Slave Test Program API Definition

Note

You are not required to edit the following Advanced Slave Test Program API unless further analysis of the protocol transfer is required.

The remaining section of this tutorial presents a walk-through of the Advanced Slave Test Program API within the slave BFM test program. It consists of a single *initial block()* that receives protocol transfers, inserting a delay in the assertion of the TREADY signal, as detailed in the *Basic Slave Test Program API Definition*.

initial block()

Within an *initial* block, the slave test program defines a transaction variable *trans* of type *axi4stream_transaction* to hold the Transaction Record of the transaction, as shown in Example 6-6. The initial wait for the ARESETn signal to be deactivated, followed by a positive ACLK edge, satisfies the protocol requirement detailed in Section 2.7.2 of the AXI4-Stream Protocol Specification.

Example 6-6. Initialization

```
initial
  begin
    int i;
    bit last;
    axi4stream_transaction trans;
    /*******************
    ** Initialisation **
    *******************/
    bfm.wait_on(AXI4STREAM_RESET_POSEDGE);
    bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE);
```

To receive protocol transfers, you must create a slave transaction. Within a *forever* loop, the *create_slave_transaction()* function is used to create a slave transaction and assigned to the transaction variable *trans*, as shown in Example 6-7.

An inner *while* loop iterates until the *last* transfer has been received. On each iteration, a delay is inserted before the TREADY signal is set to 1 by calling the *ready_delay()* task if *m_insert_wait* is set to 1. After any TREADY delay, the blocking *get_transfer()* task is called and waits for a transfer to be received.

If further analysis of the received transfer is required, you need to edit the Advanced Slave API to achieve this. You can obtain details of the Transaction Record for the received transfer by using the *get*()* tasks within the SystemVerilog Slave BFM.

Example 6-7. Transfer Receiving

```
   // Packet receiving
   forever
   begin
      trans = bfm.create_slave_transaction();
      i = 0;
      last = 0;
      while(!last)
      begin
      if(m_insert_wait)
         begin
            ready_delay();
         end
         bfm.get_transfer(trans, i, last);
         ++i;
      end
   end
end
```

Chapter 7
VHDL API Overview

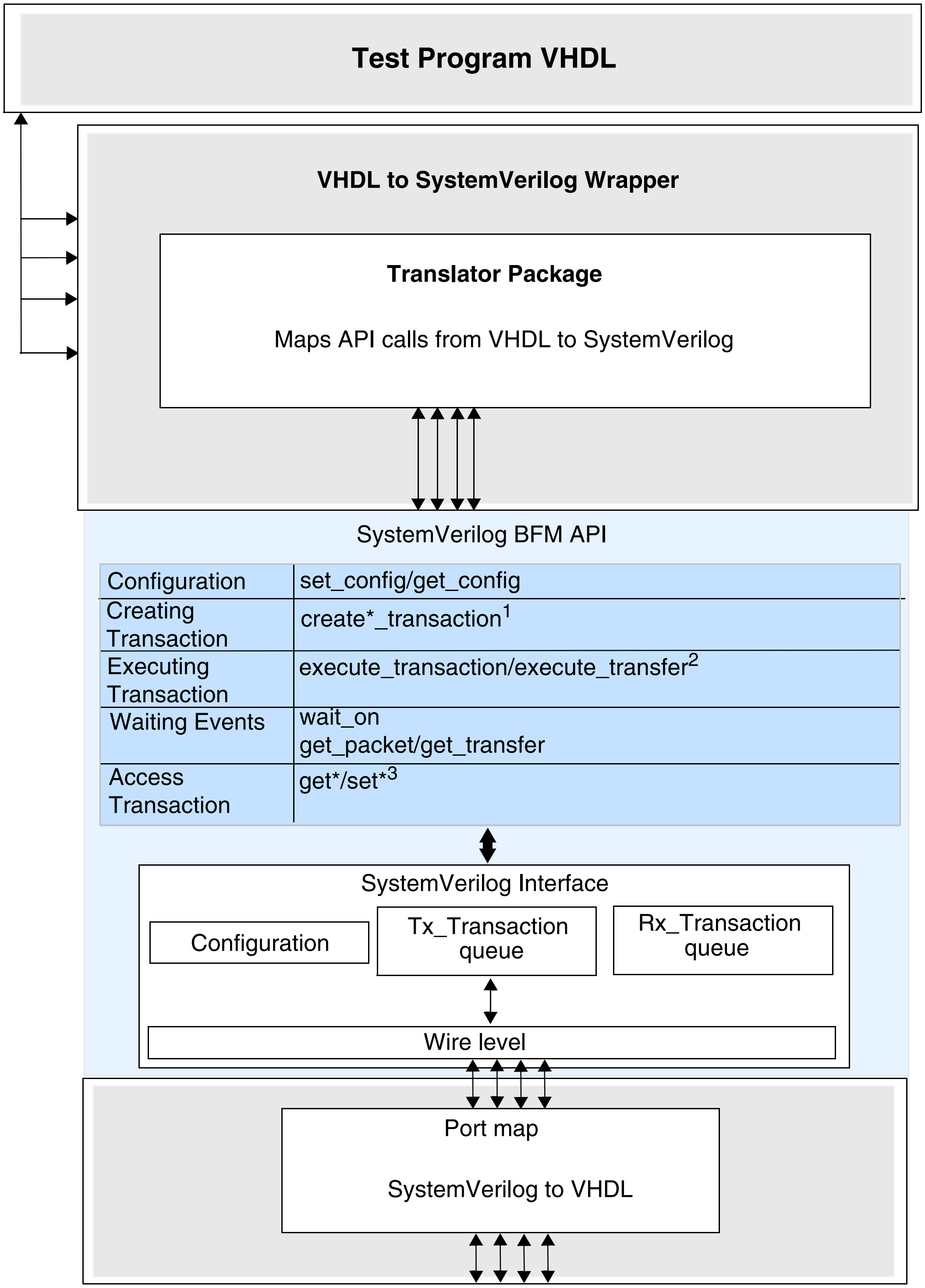
This section describes the VHDL API procedures for the BFM (master, slave, and monitor) components. For each BFM, you can configure protocol transaction fields that execute on the protocol signals and control the operational transaction fields that permit delays between the handshake signals.

In addition, each BFM API has procedures that wait for certain events to occur on the system clock and reset signals, and procedures to “get" and “set" information about a particular transaction.

Note

The VHDL API is built on the SystemVerilog API. An internal VHDL to SystemVerilog (SV) wrapper casts the VHDL BFM API procedure calls to the SystemVerilog BFM API tasks and functions.

Figure 7-1. VHDL BFM Internal Structure



Configuration	set_config/get_config
Creating Transaction	create*_transaction[1]
Executing Transaction	execute_transaction/execute_transfer[2]
Waiting Events	wait_on get_packet/get_transfer
Access Transaction	get*/set*[3]

Notes: 1. Refer to the create*_transaction()
2. Refer to the execute_transaction()
3. Refer to the get*()

Configuration

Configuration sets timeout delays, error reporting, and other attributes of the BFM.

Each BFM has a *set_config()* procedure that sets the configuration of the BFM. Refer to the individual BFM API for details. Each BFM has a *get_config()* procedure that sets the configuration of the BFM. Refer to the individual BFM API for details.

set_config()

Example 7-1 shows how to set the burst timeout factor to 1000 for a transaction in the master BFM test program.

Example 7-1. BFM Test Program Set Configuration

```
-- Setting the burst timeout factor to 1000
set_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi4stream_tr_if_0(bfm_index))
```

In the above example, the *bfm_index* specifies the actual master BFM.

get_config()

Example 7-2 shows how to get the protocol signal hold time in the master BFM test program.

Example 7-2. BFM Test Program Get Configuration

```
-- Getting the burst timeout factor
get_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, config_value,
      bfm_index, axi4stream_tr_if_0(bfm_index))
```

In the above example, the *bfm_index* specifies the actual master BFM.

Creating Transactions

To transfer information between a master BFM and slave DUT over the protocol signals, you must create a transaction in the master test program. Similarly, to transfer information between a master DUT and a slave BFM, you must create a transaction in the slave test program. To monitor the transfer of information using a monitor BFM, you must create a transaction in the monitor test program.

When you create a Transaction Record, it exists for the life of the transaction. The BFM test programs can access this transaction record during the life of the transaction as it transfers information between the master and slave.

Transaction Record

The transaction record contains two types of transaction fields, *protocol* and *operational*, that either transfer information over the protocol signals, or define how and when a transfer occurs, respectively.

Protocol fields contain transaction information that is transferred over the protocol signals. For example, the *id* field is transferred over the TID protocol signals during a transaction to identify a data stream.

Operational fields define how and when the transaction is transferred. Their content is not transferred over protocol signals. For example, the *operation_mode* field controls the blocking/nonblocking operation of a transaction, but this information is not transferred over the protocol signals.

Transaction Definition

The transaction record exists as a SystemVerilog class definition in each BFM. Example 7-3 shows the definition of the *axi4stream_transaction* class members that form the transaction record.

Example 7-3. Transaction Record Definition

```
// Global Transaction Class
class axi4stream_transaction;
    // Protocol
    byte unsigned data[];
    axi4stream_byte_type_e byte_type[];
    bit [((`MAX_AXI4_ID_WIDTH) - 1):0]  id;
    bit [((`MAX_AXI4_DEST_WIDTH) - 1):0]  dest;
    bit [((`MAX_AXI4_USER_WIDTH) - 1):0] user_data [];
    int valid_delay[];
    int ready_delay[];

    // Housekeeping
    axi4stream_operation_mode_e
      operation_mode = AXI4STREAM_TRANSACTION_BLOCKING;
    bit transfer_done[];
    bit transaction_done;

...

endclass
```

Note

The *axi4stream_transaction* class code above is shown for information only. Access to each transaction record during its life is performed by various VHDL *set*()* and *get*()* procedures described later in this chapter.

The contents of the transaction record is detailed in Table 7-1.

Table 7-1. Transaction Record Fields

Transaction Field	Description
Protocol Transaction Fields	
data	An unsized array of bytes to hold the data of an AXI4-Stream packet. The field content is transferred over the TDATA protocol signals during a transaction.
byte_type	An unsized array to hold the enumerated type of each byte within an AXI4-Stream packet. The field content is transferred over the TSTRB and TKEEP protocol signals during a transaction. The types of byte are as follows: AXI4STREAM_DATA_BYTE AXI4STREAM_NULL_BYTE AXI4STREAM_POS_BYTE AXI4STREAM_ILLEGAL_BYTE
id	A bit vector (of length equal to the TID protocol signal bus width) to hold the data stream identifier of the data packet. The field content is transferred over the TID protocol signals during a transaction.
dest	A bit vector (of length equal to the TDEST protocol signal bus width) to hold the routing information for the data stream packet. The field content is transferred over the TDEST protocol signals during a transaction.
user_data	An unsized bit vector (of length equal to the TUSER protocol signal bus width) to hold the user-defined sideband information. The field content is transferred over the TUSER protocol signals during a transaction.
Operational Transaction Fields	
valid_delay	An unsized array of integers to hold the delay value of the TVALID protocol signal (measured in ACLK cycles) for each transfer within a packet. The field content is not transferred over the protocol signals during a transaction.
ready_delay	An unsized array of integers to hold the delay value of the TREADY protocol signal (measured in ACLK cycles) for each transfer within a packet. The field content is not transferred over the protocol signals during a transaction.

Table 7-1. Transaction Record Fields (cont.)

Transaction Field	Description
operation_mode	A enumeration to hold the operation mode of the transaction. There are two types of operation mode: AXI4STREAM_TRANSACTION_NON_BLOCKING AXI4STREAM_TRANSACTION_BLOCKING The field content is not transferred over the AXI4-Stream protocol signals.
transfer_done	An unsized bit array to hold the *done* flag for each transfer within a packet. The field content is not transferred over the protocol signals during a transaction.
transaction_done	A bit to hold the *done* flag for a complete transaction. The field content is not transferred over the protocol signals during a transaction.

The VHDL Master BFM API allows you to create a master transaction by providing only an optional *burst_length* argument to indicate the number of transfers within a packet. All other protocol transaction fields automatically default to legal protocol values to create a master transaction record. Refer to *create_master_transaction()* for default protocol transaction field values.

The VHDL Slave BFM API allows you to create a slave transaction with no arguments. All protocol transaction fields automatically default to legal protocol values to create a slave transaction record. Refer to *create_slave_transaction()* for default protocol transaction field values.

The VHDL Monitor BFM API allows you to create a monitor transaction with no arguments. All protocol transaction fields automatically default to legal protocol values to create a complete monitor transaction record. Refer to *create_monitor_transaction()* for default protocol transaction field values.

Note

If you change the default value of a protocol transaction field, it is valid for all future transactions until you set a new value.

create*_transaction()

There *create_master_transaction()*, *create_slave_transaction()* and *create_monitor_transaction()* BFM API procedures create master, slave, and monitor transactions, respectively.

Example 7-4 shows a master BFM test program creating a master transaction with a packet length of 10 transfers.

Example 7-4. Master BFM Test Program Transaction Creation

```
-- Define a local variable trans to hold the transaction record.
variable trans: integer;

-- Create a master transaction of 10 transfers.
create_master_transaction(10, trans, bfm_index,
                              axi4stream_tr_if_0(bfm_index));
```

Example 7-5 shows a slave BFM test program creating a slave transaction.

Example 7-5. Slave BFM Test Program Transaction Creation

```
-- Define a local variable trans to hold the transaction record.
variable trans: integer;

-- Create a slave transaction.
create_slave_transaction(trans, bfm_index,axi4stream_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the actual BFM.

Executing Transactions

Executing a transaction in a master/slave BFM test program initiates the transaction onto the protocol signals. Each master/slave BFM API has execution procedures that push transactions into the BFM internal transaction queues. Figure 7-1 on page 74 illustrates the internal BFM structure.

execute_transaction()

If the DUT is a slave, then the *execute_transaction()* procedure is called in the master BFM test program. Example 7-6 shows a master test program executing a master transaction.

Example 7-6. Master Test Program Transaction Execution

```
-- Define a local variable trans to hold the transaction record.
variable trans: integer;

...

-- Create a master transaction with 10 transfers.
create_master_transaction(10, trans, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

...

-- By default the execution of a transaction will block.
execute_transaction(trans, bfm_index, axi4stream_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the actual slave BFM.

Waiting Events

Each BFM API has procedures that block the test program code execution until an event has occurred.

The *wait_on()* procedure blocks the test program until an ACLK or ARESETn signal event has occurred before proceeding.

The *get_packet(), get_transfer()* procedures block the test program code execution until a complete stream packet or transfer has occurred.

wait_on()

Example 7-7 shows a BFM test program waiting for the positive edge of the ARESETn signal.

Example 7-7. Test Program Wait for Event

```
-- Block test program execution until the positive edge of the
-- ARESETn signal.
wait_on(AXI4STREAM_RESET_POSEDGE, bfm_index,
            axi4stream_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the actual master BFM.

get_packet(), get_transfer()

Example 7-8 shows a slave BFM test program using the *get_packet()* procedure to block until it has received a data stream transfer.

Example 7-8. Slave Test Program get_packet() Procedure

```
-- Define a local variable trans to hold the transaction record
variable trans: integer;

...

-- Create a slave transaction
create_slave_transaction(trans, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

...

--Wait for the first data stream transfer to occur.
get_transfer(trans, 0, last, bfm_index, axi4stream_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the actual slave BFM.

Access Transaction Record

Each BFM API has procedures that can access a complete or partially complete Transaction Record. The *set*()* and *get*()* procedures are used in a test program to set and get information from the transaction record.

set*()

Example 7-9 shows the master test program calling the *set_byte_type()* procedure to set the first *byte_type* in the transaction.

Example 7-9. Master Test Program set_byte_type() Procedure

```
-- Define a local variable trans to hold the transaction record.
variable trans: integer;

-- Create a master transaction with 10 transfers.
create_master_transaction(10, trans, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

-- Set the first byte_type in the transfer.
set_byte_type(AXI4STREAM_DATA_BYTE, 0, trans, bfm_index,
                  axi4stream_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the actual master BFM.

get*()

Example 7-10 shows the slave test program calling the *get_byte_type()* procedure to get the first data *byte_type* of a transaction.

Example 7-10. Slave Test Program get_byte_type() Procedure

```
-- Define a local variable trans to hold the transaction record.
variable trans: integer;

-- Define a local variable to hold the transaction byte type
variable byte_type: integer;

-- Create a slave transaction
create_slave_transaction(trans, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

-- Get the first byte_type of a transaction.
get_byte_type(byte_type, 0, trans, bfm_index,
                  axi4stream_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the actual slave BFM.

Operational Transaction Fields

Operational transaction fields control the way in which a transaction is executed onto the protocol signals. These fields also indicate when an individual data transfer or transaction is complete.

Operation Mode

By default, each transaction performs a blocking operation that prevents a following transaction from starting until the current active transaction completes.

You can configure this behavior to be nonblocking by setting the *operation_mode* transaction field to the enumerate type value AXI4STREAM_TRANSACTION_NON_BLOCKING instead of the default AXI4STREAM_TRANSACTION_BLOCKING.

Example 7-11shows a master BFM test program creating a transaction by calling the *create_master_transaction()* procedure. Before executing the transaction, the *operation_mode* is changed to nonblocking.

Example 7-11. Master Test Program set_operation_mode() Procedure

```
-- Define a local variable trans to hold the transaction record.
variable trans: integer;

-- Create a master transaction with 10 transfers.
create_master_transaction(10, trans, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

// Change the operation_mode to be nonblocking in the transaction record
set_operation_mode(AXI4STREAM_TRANSACTION_NON_BLOCKING, trans, bfm_index,
                  axi4stream_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the actual master BFM.

Handshake Delay

You can configure the TVALID and TREADY handshake signals to insert a delay before their assertion.

TVALID Signal Delay Transaction Field

The Transaction Record contains a *valid_delay* transaction field to configure the delay of the TVALID signal. The setting of the *valid_delay* transaction field is performed in the master BFM test program by calling the *set_valid_delay()* procedure.

TREADY Signal Delay Transaction Field

The Transaction Record contains a *ready_delay* transaction field to configure the delay of the TREADY signal. The setting of the *ready_delay* transaction field is performed in the slave BFM test program by calling the local *ready_delay()* procedure.

Example 7-12 shows the slave BFM test program implementing a *ready_delay()* procedure that inserts a specified delay before the assertion of the TREADY signal.

Example 7-12. Slave Test Program ready_delay() Procedure

```
-- Procedure : ready_delay
-- This is used to set ready delay to extend the transfer
procedure ready_delay(signal tr_if : inout axi4stream_vhd_if_struct_t) is
begin
   --  Making TREADY '0'. This will consume one cycle.
   execute_stream_ready(0, index, tr_if);
   -- Two clock cycle wait. In total 3 clock wait.
   for i in 0 to 1 loop
      wait_on(AXI4STREAM_CLOCK_POSEDGE, index, tr_if);
   end loop;
   -- Making TREADY '1'.
   execute_stream_ready(1, index, tr_if);
end ready_delay;
```

Transfer Done

A *transfer_done* transaction field is set to 1 to indicate when each protocol transfer completes.

Transaction Done

A *transaction_done* transaction field is set to 1 to indicate when each protocol transaction completes.

In a slave BFM, you call the *get_packet()* BFM procedure to investigate whether a transaction is complete. If complete, the procedure returns the *last* argument set to 1, and the transaction record has the *transaction_done* field set to 1.

Chapter 8
VHDL Master BFM

This section provides information about the VHDL master BFM. It has an API that contains procedures to configure the BFM and to access the dynamic Transaction Record during the life of the transaction.

Overloaded Procedure Common Arguments

The BFMs use VHDL procedure overloading, which results in the prototype having a number of definitions for each procedure. Their arguments are unique to each procedure and concern the protocol or operational transaction fields for a transaction. These procedures have several common arguments that may be optional and include the arguments described below:

- *transaction_id* is an index number that identifies a specific transaction. Each new transaction automatically increments the index number until reaching 255, the maximum value, and then the index number automatically wraps to zero. The *transaction_id* uniquely identifies each transaction when there are a number of concurrently active transactions.
- *bfm_id* is a unique identification number for each master, slave, and monitor BFM within a multiple BFM test bench.
- *tr_if* is a signal definition that passes the content of a transaction between the VHDL and SystemVerilog environments.

Master BFM Protocol Support

The AXI4-Stream master BFM supports the full AMBA AXI4-Stream Protocol Specification.

Master Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI4-Stream Protocol Specification chapter, which you can reference for details of the following master BFM API timing and events.

The AMBA AXI4-Stream Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the master BFM does not contain any timescale, timeunit, or timeprecision declarations. The signal setup and hold times are specified in units of simulator time-steps.

Master BFM Configuration

A master BFM supports the full range of signals defined for the AMBA AXI4-Stream Protocol Specification. It has parameters that you use to configure the widths of the data and ID signals, and transaction fields to configure timeout factors, setup and hold times, and so on.

You can change the data and ID signal widths from their default setting by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the master BFM using a parameter port list of the master BFM module.

Table 8-1 lists the parameter names for the data and ID signals and their default values.

Table 8-1. Master BFM Signal Width Parameters

Signal Width Parameter	Description
AXI4_ID_WIDTH	ID signal width in bits. This applies to the TID signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_USER_WIDTH	User data signal width in bits. This applies to the TUSER signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 8.
AXI4_DEST_WIDTH	Destination routing signal width in bits. This applies to the TDEST signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_DATA_WIDTH	Data signal width in bits. This applies to the TDATA signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 1024.

A master BFM has configuration fields that you set by calling the *set_config()* procedure to configure timeout factors, setup and hold times, etc. You get the value of a configuration field by calling the *get_config()* procedure. Table 8-2 describes the full list of configuration fields.

Table 8-2. Master BFM Configuration

Configuration Field	Description
Timing Variables	
AXI4STREAM_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.

Table 8-2. Master BFM Configuration (cont.)

Configuration Field	Description
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay permitted between individual transfers in clock cycles. Default: 10000.
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ ASSERTION_TO_TREADY	The maximum delay permitted between the assertion of TVALID to the assertion of TREADY. Default: 10000.
Master Attributes	
AXI4STREAM_LAST_DURING_IDLE	Controls the value of TLAST during idle. 0 = TLAST driven to 0 during idle (default) 1 = TLAST driven to 1 during idle
Error Detection	
AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
AXI4STREAM_CONFIG_ENABLE_ASSERTION	Individual enable/disable of an assertion check in the BFM. Refer to the Master Assertions chapter for details. 0 = disabled 1 = enabled (default)

[1] Refer to Master Timing and Events for details of simulator time-steps.

Master Assertions

The master BFM performs protocol error checking via built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

By default, all built-in assertions are enabled in the master BFM. To globally disable them in the master BFM, use the *set_config()* command as shown in Example 8-1.

Example 8-1. Master BFM Disable All Assertions

```
set_config(AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS, 0, bfm_index,
axi4stream_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. Example 8-2 shows how to disable assertion checking for the TLAST signal changing between the TVALID and TREADY handshake signals.

Example 8-2. Master BFM Individual Assertion Enable/Disable

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector :
std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4stream_tr_if_0(bfm_index));

-- Assign the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion bit to 0
config_assert_bitvector(AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY) := ‘0’;

-- Set the new value of the assertion bit vector
set_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4stream_tr_if_0(bfm_index));
```

Note

Do not confuse the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector with the AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion, follow the code sequence in Example 8-2 and assign the assertion enable within the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to “Assertions” on page 203.

VHDL Master BFM API

This section describes the VHDL master BFM API.

Each procedure available within the master BFM API is detailed in the following chapter. The *set*()* and *get*()* procedures that operate on the Transaction Record fields have a simple rule for the procedure name: *set_* or *get_* followed by the name of the transaction field to be accessed. Refer to "Transaction Record" on page 21 for details of transaction field names.

Note

The master BFM API package is the *axi4stream/bfm/mgc_axi4stream_bfm_pkg.vhd* file packaged within the Mentor Verification IP Altera Edition.

set_config()

This nonblocking procedure sets the configuration of the master BFM.

Prototype

```
procedure set_config
(
   config_name   : in std_logic_vector(7 downto 0);
   config_val    : in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
   downto 0)|integer;
   bfm_id        : in integer;
   signal tr_if  : inout axi4stream_vhd_if_struct_t
 );
```

Arguments

config_name — Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

config_val — Refer to "Master BFM Configuration" on page 86 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
set_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi4stream_tr_if_0(bfm_index));
```

get_config()

This nonblocking procedure gets the configuration of the master BFM.

Prototype

```
procedure get_config
(
   config_name   : in std_logic_vector(7 downto 0);
   config_val   : out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
   downto 0)|integer;
   bfm_id        : in integer;
   signal tr_if  : inout axi4stream_vhd_if_struct_t
);
```

Arguments

config_name — Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

config_val — Refer to "Master BFM Configuration" on page 86 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns

config_val

Example

```
get_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, config_value,
     bfm_index, axi4stream_tr_if_0(bfm_index));
```

create_master_transaction()

This nonblocking procedure creates a master transaction with an optional *burst_length* argument. All other transaction fields default to legal protocol values, unless previously assigned a value. This procedure creates and returns the *transaction_id* argument.

Prototype

```
procedure create_master_transaction
(
   burst_length    : in integer; --optional
   transaction_id  : out integer;
   bfm_id          : in integer;
   signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments

burst_length	(Optional) Number of transfers within a packet. Default: 1.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Protocol Transaction Fields

data	Data array in bytes

byte_type	Byte type array:
	AXI4STREAM_DATA_BYTE; (default)
	AXI4STREAM_NULL_BYTE;
	AXI4STREAM_POS_BYTE;
	AXI4STREAM_ILLEGAL_BYTE;

id	Data stream identifier.
dest	Destination routing information.
user_data	User data array.

Operational Transaction Fields

operation_mode	Operation mode: AXI4STREAM_TRANSACTION_NON_BLOCKING; AXI4STREAM_TRANSACTION_BLOCKING; (default)
valid_delay	TVALID delay measured in ACLK cycles for this transaction (default = 0).
ready_delay	TREADY delay measured in ACLK cycles for this transaction (default = 0).
transfer_done	Transfer *done* flag array for this transaction
transaction_done	Transaction *done* flag for this transaction

Returns

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85.

Example

```
-- Create a master transaction containing 3 transfers.
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
      axi4stream_tr_if_0(bfm_index);
```

set_data()

This nonblocking procedure sets a *data* field array element for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *data* byte is identified by the optional *index* argument. If no *index* is supplied, then the first *data* byte is accessed in the array.

Prototype

```
set_data
(
     data: in integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

data	Data byte.
index	(Optional) Array element index number for *data*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers.
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Set the data field to 2 for the first byte
-- of the tr_id transaction.
set_data(2, 0, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Set the data field to 3 for the second byte
-- of the tr_id transaction.
set_data(3, 1, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_data()

This nonblocking procedure gets a *data* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *data* byte is identified by the optional *index* argument. If no *index* is supplied, then the first *data* byte is accessed in the array.

Prototype

```
get_data
(
    data: out integer;
    index : in integer; --optional
    transaction_id  : in integer;
    bfm_id : in integer;
    signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

data	Data byte.
index	(Optional) Array element index number for *data*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns

data

Example

Note

You would not normally use this procedure within a Master Test Program.

set_byte_type()

This nonblocking procedure sets a *byte_type* field array element for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *byte_type* is identified by the optional *index* argument. If no *index* is supplied, then the first *byte_type* is accessed in the array.

Prototype

```
set_byte_type
(
     byte_type: in integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

byte_type — Byte type array:

AXI4STREAM_DATA_BYTE; (default)
AXI4STREAM_NULL_BYTE;
AXI4STREAM_POS_BYTE;
AXI4STREAM_ILLEGAL_BYTE;

index — (Optional) Array element index number for *byte_type*.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns

None

Example

```
-- Create a master transaction containing 3 transfers.
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Set the byte_type field to data for the first byte
-- of the tr_id transaction.
set_byte_type(AXI4STREAM_DATA_BYTE, 0, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Set the byte_type field to null for the second byte
-- of the tr_id transaction.
set_byte_type(AXI4STREAM_NULL_BYTE, 1, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));
```

get_byte_type()

This nonblocking procedure gets a *byte_type* field array element for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *byte_type* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *byte_type* is accessed in the array.

Prototype

```
get_byte_type
(
     byte_type: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

byte_type — Byte type array:

AXI4STREAM_DATA_BYTE; (default)
AXI4STREAM_NULL_BYTE;
AXI4STREAM_POS_BYTE;
AXI4STREAM_ILLEGAL_BYTE;

index — (Optional) Array element index number for *byte_type*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns byte_type

Example

Note

You would not normally use this procedure within a Master Test Program.

set_id()

This nonblocking procedure sets the data stream identifier *id* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

Prototype

```
set_id
(
     id: in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0 )
     | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

id	Data stream identifier value placed on the *TID* signals.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers.
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Set the id field to 2 for the tr_id transaction.
set_id(2, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_id()

This nonblocking procedure gets the data stream identifier *id* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

Prototype

```
get_id
(
     id: out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0
     ) | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

id	Data stream identifier value placed on the *TID* signals.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns id

Example

Note

You would not normally use this procedure within a Master Test Program.

set_dest()

This nonblocking procedure sets the routing information *dest* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

Prototype

```
set_dest
(
     dest: in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0
     ) | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

dest	Data stream routing information value placed on the *TDEST* signals.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers.
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Set the dest field to 2 for the tr_id transaction.
set_dest(2, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_dest()

This nonblocking procedure gets the routing information *id* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

Prototype

```
get_dest
(
     dest: out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto
     0 ) | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

dest — Data stream routing information value placed on the *TDEST* signals.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns dest

Example

Note

You would not normally use this procedure within a Master Test Program.

set_user_data()

This nonblocking procedure sets the *user_data* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *user_data* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *user_data* is accessed in the array.

Prototype

```
set_user_data
(
    user_data: in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
    downto 0 ) | integer;
    index : in integer; --optional
    transaction_id  : in integer;
    bfm_id : in integer;
    signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

user_data
User data array values placed on the *TUSER* signals.

index
(Optional) Array element index number for *user_data*.

transaction_id
Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id
BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if
Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns

None

Example

```
-- Create a master transaction containing 3 transfers.
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Set the user_data field to 2 for the first transfer
-- of the tr_id transaction.
set_user_data(2, 0, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Set the user_data field to 3 for the second transfer
-- of the tr_id transaction.
set_user_data(3, 1, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_user_data()

This nonblocking procedure gets a *user_data* field array element for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *user_data* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *user_data* is accessed in the array.

Prototype

```
get_user_data
(
     user_data: out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
     downto 0 ) | integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

user_data — User data array values placed on the *TUSER* signals.

index — (Optional) Array element index number for *user_data*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns

user_data

Example

Note

You would not normally use this procedure within a Master Test Program.

set_valid_delay()

This nonblocking procedure sets the *valid_delay* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *valid_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *valid_delay* is accessed in the array.

Prototype

```
set_valid_delay
(
    valid_delay: in integer;
    index : in integer; --optional
    transaction_id  : in integer;
    bfm_id : in integer;
    signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

valid_delay	Valid delay array to store TVALID delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for *valid_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Set the TVALID delay to 3 ACLK cycles for the first transfer
-- of the tr_id transaction.
set_valid_delay(3, 0, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Set the TVALID delay to 2 ACLK cycles for the second transfer
-- of the tr_id transaction.
set_valid_delay(2, 1, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_valid_delay()

This nonblocking procedure gets the *valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *valid_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *valid_delay* is accessed in the array.

Prototype

```
get_valid_delay
(
     valid_delay: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

valid_delay	Valid delay array to store TVALID delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for *valid_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns valid_delay

Example

Note

You would not normally use this procedure within a Master Test Program.

set_ready_delay()

This nonblocking procedure sets the *ready_delay* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *ready_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *ready_delay* is accessed in the array.

Prototype

```
set_ready_delay
(
     ready_delay: in integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

ready_delay	Ready delay array to hold TREADY delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for *ready_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

Note

You would not normally use this procedure within a Master Test Program.

get_ready_delay()

This nonblocking procedure gets the *ready_delay* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *ready_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then first *ready_delay* is accessed in the array.

Prototype

```
get_ready_delay
(
     ready_delay: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

ready_delay	Read data channel array to hold TREADY delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for *ready_delay*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns ready_delay

Example

```
-- Create a master transaction containing 3 transfers.
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Get the TREADY delay for the first transfer of the tr_id transaction.
get_ready_delay(ready_delay, 0, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Get the TREADY delay for the second transfer of the tr_id transaction.
get_ready_delay(ready_delay, 1, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));
```

set_operation_mode()

This nonblocking procedure sets the *operation_mode* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

Prototype

```
set_operation_mode
(
     operation_mode: in integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

operation_mode — Operation mode:

AXI4STREAM_TRANSACTION_NON_BLOCKING;
AXI4STREAM_TRANSACTION_BLOCKING; (default)

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers.
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Set the operation mode field to nonblocking for the tr_id transaction.
set_operation_mode(AXI4STREAM_TRANSACTION_NON_BLOCKING, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));
```

get_operation_mode()

This nonblocking procedure gets the *operation_mode* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

Prototype

```
get_operation_mode
(
     operation_mode: out integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

operation_mode	Operation mode: AXI4STREAM_TRANSACTION_NON_BLOCKING; AXI4STREAM_TRANSACTION_BLOCKING; (default)
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns operation_mode

Example

Note

You would not normally use this procedure within a Master Test Program.

set_transfer_done()

This nonblocking procedure sets a *transfer_done* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *transfer_done* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *transfer_done* is accessed in the array.

Prototype

```
set_transfer_done
(
    transfer_done : in integer;
    index : in integer; --optional
    transaction_id  : in integer;
    bfm_id : in integer;
    signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transfer_done	Transfer *done* array for this transaction.
index	(Optional) Array element index number for *transfer_done*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

....

-- Set the transfer_done flag for the first transfer
-- of the tr_id transaction.
set_transfer_done(1, 0, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));

....

-- Set the transfer_done flag for the second transfer
-- of the tr_id transaction.
set_transfer_done(1, 1, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_transfer_done()

This nonblocking procedure gets a *transfer_done* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

The *transfer_done* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *transfer_done* is accessed in the array.

Prototype

```
get_transfer_done
(
    transfer_done : out integer;
    index : in integer; --optional
    transaction_id  : in integer;
    bfm_id : in integer;
    signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transfer_done	Transfer *done* array for this transaction.
index	(Optional) Array element index number for *transfer_done*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns transfer_done

Example

Note

You would not normally use this procedure within a Master Test Program.

set_transaction_done()

This nonblocking procedure sets the *transaction_done* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

Prototype

```
set_transaction_done
(
     transaction_done : in integer;
     transaction_id   : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

....

-- Set the transaction_done flag of the tr_id transaction.
set_transaction_done(1, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_transaction_done()

This nonblocking procedure gets the *transaction_done* field for a master transaction that is uniquely identified by the *transaction_id* field previously created by the *create_master_transaction()* procedure.

Prototype

```
get_transaction_done
(
     transaction_done : out integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns transaction_done

Example

Note

You would not normally use this procedure within a Master Test Program.

execute_transaction()

This procedure executes a master transaction that is uniquely identified by the *transaction_id* argument previously created by the *create_master_transaction()* procedure. A transaction can be blocking (default) or nonblocking, based on the setting of the transaction *operation_mode* field.

It calls the *execute_transfer()* procedure for each transfer within a packet, with the number of transfers defined by the transaction *burst_length* field.

Prototype

```
procedure execute_transaction
(
   transaction_id : in integer;
   bfm_id : in integer;
   signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Set the ID to 1 for this transaction
set_id(1, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Execute the tr_id transaction.
execute_transaction(tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

execute_transfer()

This procedure executes a master transfer that is uniquely identified by the *transaction_id* argument previously created by the *create_master_transaction()* procedure. This transfer may be blocking (default) or nonblocking, as defined by the transaction *operation_mode* field.

It sets the TVALID protocol signal at the appropriate time defined by the transaction *valid_delay* field and sets the *transfer_done* array *index* element field to 1 when the transfer is complete.

If this is the last transfer of the transaction, then it sets the *transaction_done* field to 1 and returns the *last* argument set to 1 to indicate the whole transaction is complete.

Prototype

```
procedure execute_transfer
(
     transaction_id  : in integer;
     index : in integer; --optional
     bfm_id          : in integer;
     signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
index	(Optional) Data phase (beat) number.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

....

-- Execute the first transfer of the packet for the
-- tr_id transaction.
execute_transfer(tr_id, 0, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Execute the second transfer of the packet for the
-- tr_id transaction.
execute_transfer(tr_id, 1, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_stream_ready()

This procedure gets the value of the TREADY signal by returning it via the *ready* argument. It will block for one ACLK period.

Prototype

```
procedure get_stream_ready
(
   ready : out integer;
   bfm_id          : in integer;
   signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments

ready	The value of the TREADY signal
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns ready

Example

```
-- Get the state of the TREADY signal.
get_stream_ready(ready, bfm_index,  axi4stream_tr_if_0(bfm_index));
```

print()

This nonblocking procedure prints a transaction record that is uniquely identified by the *transaction_id* argument previously created by the *create_master_transaction()* procedure.

Prototype

```
procedure print
(
   transaction_id  : in integer;
   print_delays : in integer;
   bfm_id          : in integer;
   signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

print_delays — (Optional) Print delay values flag:

0 = do not print the delay values (default).
1 = print the delay values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi4stream_tr_if_0(bfm_index));
```

destruct_transaction()

This blocking procedure removes a transaction record for cleanup purposes and memory management that is uniquely identified by the *transaction_id* argument previously created by the *create_master_transaction()* procedure.

Prototype

```
procedure destruct_transaction
(
   transaction_id   : in integer;
   bfm_id           : in integer;
   signal tr_if     : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a master transaction containing 3 transfers
-- Creation returns tr_id to identify the transaction.
create_master_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

wait_on()

This blocking procedure waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
procedure wait_on
(
   phase            : in integer;
   count: in integer; -optional
   bfm_id           : in integer;
   signal tr_if     : inout axi4stream_vhd_if_struct_t
);
```

Arguments

phase — Wait for:

AXI4STREAM_CLOCK_POSEDGE
AXI4STREAM_CLOCK_NEGEDGE
AXI4STREAM_CLOCK_ANYEDGE
AXI4STREAM_CLOCK_0_TO_1
AXI4STREAM_CLOCK_1_TO_0
AXI4STREAM_RESET_POSEDGE
AXI4STREAM_RESET_NEGEDGE
AXI4STREAM_RESET_ANYEDGE
AXI4STREAM_RESET_0_TO_1
AXI4STREAM_RESET_1_TO_0

count — (Optional) Wait for a number of events to occur set by *count*.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns None

Example

```
wait_on(AXI4STREAM_RESET_POSEDGE, bfm_index,
axi4stream_tr_if_0(bfm_index));

wait_on(AXI4STREAM_CLOCK_POSEDGE, 10, bfm_index,
axi4stream_tr_if_0(bfm_index));
```

Chapter 9
VHDL Slave BFM

This section provides information about the VHDL slave BFM. It has an API that contains procedures to configure the BFM and to access the dynamic Transaction Record during the life of the transaction.

Slave BFM Protocol Support

The AXI4-Stream slave BFM supports the full AMBA AXI4-Stream protocol.

Slave Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI4-Stream Protocol Specification chapter, which you can reference for details of the following slave BFM API timing and events.

The AMBA AXI4-Stream Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the slave BFM does not contain any timescale, timeunit, or timeprecision declarations. The signal setup and hold times are specified in units of simulator time-steps.

Slave BFM Configuration

A slave BFM supports the full range of signals defined for the AMBA AXI4-Stream Protocol Specification. It has parameters that you use to configure the widths of the data and ID signals, and transaction fields to configure timeout factors, setup and hold times, and so on.

You can change the data and ID signals widths from their default setting by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the slave BFM using a parameter port list of the slave BFM module.

Table 9-1 lists the parameter names for the data and ID signals and their default values.

Table 9-1. Slave BFM Signal Width Parameters

Signal Width Parameter	Description
AXI4_ID_WIDTH	ID signal width in bits. This applies to the TID signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.

Table 9-1. Slave BFM Signal Width Parameters (cont.)

Signal Width Parameter	Description
AXI4_USER_WIDTH	User data signal width in bits. This applies to the TUSER signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 8.
AXI4_DEST_WIDTH	Destination routing signal width in bits. This applies to the TDEST signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_DATA_WIDTH	Data signal width in bits. This applies to the TDATA signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 1024.

A slave BFM has configuration fields that you set by calling the *set_config()* procedure to configure timeout factors, setup and hold times, and so on. You get the value of a configuration field by calling the *get_config()* procedure. Table 9-2 describes the full list of configuration fields.

Table 9-2. Slave BFM Configuration

Configuration Field	Description
Timing Variables	
AXI4STREAM_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay between individual transfers in clock cycles. Default: 10000.
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY	The maximum delay permitted between the assertion of TVALID to the assertion of TREADY. Default: 10000.
Master Attributes	
AXI4STREAM_LAST_DURING_IDLE	Controls the value of TLAST during idle. 0 = TLAST driven to 0 during idle (default) 1 = TLAST driven to 1 during idle

Table 9-2. Slave BFM Configuration (cont.)

Configuration Field	Description
Error Detection	
AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
AXI4STREAM_CONFIG_ENABLE_ASSERTION	Individual enable/disable of an assertion check in the BFM. Refer to Slave Assertions chapter for details. 0 = disabled 1 = enabled (default)

[1] Refer to Slave Timing and Events for details of simulator time-steps.

Slave Assertions

The slave BFM performs protocol error checking via built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

By default, all built-in assertions are enabled in the slave BFM. To globally disable them in the slave BFM, use the *set_config()* command as shown in Example 9-1.

Example 9-1. Slave BFM Disable All Assertions

```
set_config(AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS, 0, bfm_index,
axi4stream_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. Example 9-2 shows how to disable assertion checking for the TLAST signal changing between the TVALID and TREADY handshake signals.

Example 9-2. Slave BFM Individual Assertion Enable/Disable

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector :
std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4stream_tr_if_0(bfm_index));

-- Assign the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion bit to 0
config_assert_bitvector(AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY) := ‘0’;

-- Set the new value of the assertion bit vector
set_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4stream_tr_if_0(bfm_index));
```

Note

Do not confuse the *AXI4STREAM_CONFIG_ENABLE_ASSERTION* bit vector with the *AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS* global enable/disable.

To re-enable the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion, follow the code sequence in Example 9-2 and assign the assertion within the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to “Assertions” on page 203.

VHDL Slave BFM API

This section describes the VHDL Slave BFM API.

Each procedure available within the slave BFM API is detailed in the following chapter. The *set*()* and *get*()* procedures that operate on the Transaction Record fields have a simple rule for the procedure name: *set_* or *get_* followed by the name of the transaction field to be accessed. Refer to “Transaction Record” on page 21 for details of transaction field names.

Note

The slave BFM API package is the *axi4stream/bfm/mgc_axi4stream_bfm_pkg.vhd* file packaged within the Mentor Verification IP Altera Edition.

set_config()

This nonblocking procedure sets the configuration of the slave BFM.

Prototype

```
procedure set_config
(
   config_name    : in std_logic_vector(7 downto 0);
   config_val     : in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
   downto 0)|integer;
   bfm_id         : in integer;
   signal tr_if   : inout axi4stream_vhd_if_struct_t
 );
```

Arguments

config_name — Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

config_val — Refer to “Slave BFM Configuration” on page 121 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns None

Example

```
set_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi4stream_tr_if_0(bfm_index));
```

get_config()

This nonblocking procedure gets the configuration of the slave BFM.

Prototype

```
procedure get_config
(
   config_name   : in std_logic_vector(7 downto 0);
   config_val    : out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
   downto 0)|integer;
   bfm_id        : in integer;
   signal tr_if  : inout axi4stream_vhd_if_struct_t
);
```

Arguments

config_name — Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

config_val — Refer to “Slave BFM Configuration” on page 121 for description and valid values.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns — config_val

Example

```
get_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, config_value,
     bfm_index, axi4stream_tr_if_0(bfm_index));
```

create_slave_transaction()

This nonblocking procedure creates a slave transaction. All transaction fields default to legal protocol values, unless previously assigned a value. This procedure creates and returns the *transaction_id* argument.

Prototype

```
procedure create_slave_transaction
(
   transaction_id  : out integer;
   bfm_id          : in integer;
   signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments	transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85.
	bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85.
	tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85.
Protocol Transaction Fields	data	Data array in bytes

	byte_type	Byte type array:
		AXI4STREAM_DATA_BYTE; (default)
		AXI4STREAM_NULL_BYTE;
		AXI4STREAM_POS_BYTE;
		AXI4STREAM_ILLEGAL_BYTE;

	id	Data stream identifier.
	dest	Destination routing information.
	user_data	User data array.
Operational Transaction Fields	operation_ mode	Operation mode: AXI4STREAM_TRANSACTION_NON_BLOCKING; AXI4STREAM_TRANSACTION_BLOCKING; (default)
	valid_delay	TVALID delay measured in ACLK cycles for this transaction (default = 0).
	ready_delay	TREADY delay measured in ACLK cycles for this transaction (default = 0).
	transfer_done	Transfer *done* flag array for this transaction
	transaction_ done	Transaction *done* flag for this transaction
Returns	transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85.

Example

```
-- Create a slave transaction
-- Returns the transaction ID (tr_id) for this created transaction.
create_slave_transaction(tr_id, bfm_index,
                              axi4stream_tr_if_3(bfm_index));
```

set_data()

This nonblocking procedure sets a *data* field array element for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *data* byte is identified by the optional *index* argument. If no *index* is supplied, then the first *data* byte is accessed in the array.

Prototype

```
set_data
(
     data: in integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

data	Data byte.
index	(Optional) Array element index number for *data*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

Note

You would not normally use this procedure within a Slave Test Program.

get_data()

This nonblocking procedure gets a *data* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *data* byte is identified by the optional *index* argument. If no *index* is supplied, then the first *data* byte is accessed in the array.

Prototype

```
get_data
(
     data: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

data — Data byte.

index — (Optional) Array element index number for *data*.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns data

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Get the data field for the first byte of the tr_id transaction.
get_data(data, 0, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Get the data field for the second byte of the tr_id transaction.
get_data(data, 1, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

set_byte_type()

This nonblocking procedure sets a *byte_type* field array element for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *byte_type* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *byte_type* is accessed in the array.

Prototype

```
set_byte_type
(
     byte_type: in integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

byte_type — Byte type array:

AXI4STREAM_DATA_BYTE; (default)
AXI4STREAM_NULL_BYTE;
AXI4STREAM_POS_BYTE;
AXI4STREAM_ILLEGAL_BYTE;

index — (Optional) Array element index number for *byte_type*.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

Note

You would not normally use this procedure within a Slave Test Program.

get_byte_type()

This nonblocking procedure gets a *byte_type* field array element for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *byte_type* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *byte_type* is accessed in the array.

Prototype

```
get_byte_type
(
     byte_type: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

byte_type — Byte type array:

AXI4STREAM_DATA_BYTE; (default)
AXI4STREAM_NULL_BYTE;
AXI4STREAM_POS_BYTE;
AXI4STREAM_ILLEGAL_BYTE;

index — (Optional) Array element index number for *byte_type*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns byte_type

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

-- Get the byte_type field for the first byte of the tr_id transaction.
get_byte_type(byte_type, 0, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));

-- Get the byte_type field for the second byte of the tr_id transaction.
get_byte_type(byte_type, 1, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));
```

set_id()

This nonblocking procedure sets the data stream identifier *id* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
set_id
(
     id: in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0 )
     | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

id

Data stream identifier value placed on the TID signals.

transaction_id

Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id

BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if

Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns

None

Example

Note

You would not normally use this procedure within a Slave Test Program.

get_id()

This nonblocking procedure gets the data stream identifier *id* field for a slave transaction that is uniquely identified by the *transaction_id* field and previously created by the *create_slave_transaction()* procedure.

Prototype

```
get_id
(
     id: out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0
     ) | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

id	Data stream identifier value placed on the TID signals.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns id

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

....

-- Get the id field of the tr_id transaction.
get_id(id, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

set_dest()

This nonblocking procedure sets the routing information *dest* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
set_dest
(
     dest: in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0
     ) | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

dest	Data stream routing information value placed on the TDEST signals.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

Note

You would not normally use this procedure within a Slave Test Program.

get_dest()

This nonblocking procedure gets the routing information *id* field for a slave transaction that is uniquely identified by the *transaction_id* field and previously created by the *create_slave_transaction()* procedure.

Prototype

```
get_dest
(
     dest: out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto
     0 ) | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

dest	Data stream routing information value placed on the TDEST signals.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns dest

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

....

-- Get the dest field of the tr_id transaction.
get_dest(dest, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

set_user_data()

This nonblocking procedure sets a *user_data* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *user_data* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *user_data* is accessed in the array.

Prototype

```
set_user_data
(
     user_data: in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
     downto 0 ) | integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

user_data — User data array values placed on the TUSER signals.

index — (Optional) Array element index number for *user_data*.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

Note

You would not normally use this procedure within a Slave Test Program.

get_user_data()

This nonblocking procedure gets a *user_data* field array element for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *user_data* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *user_data* is accessed in the array.

Prototype

```
get_user_data
(
     user_data: out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
     downto 0 ) | integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

user_data	User data array values placed on the TUSER signals.
index	(Optional) Array element index number for *user_data*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns user_data

Example

```
-- Create a slave transaction containing 3 transfers.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(3, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Get the user_data field for the first transfer
-- of the tr_id transaction.
get_user_data(user_data, 0, tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Get the user_data field for the second transfer
-- of the tr_id transaction.
get_user_data(user_data, 1, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));
```

set_valid_delay()

This nonblocking procedure sets the *valid_delay* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *valid_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *valid_delay* is accessed in the array.

Prototype

```
set_valid_delay
(
     valid_delay: in integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

valid_delay	Valid delay array to store TVALID delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for *valid_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

Note

You would not normally use this procedure within a Slave Test Program.

get_valid_delay()

This nonblocking procedure gets the *valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *valid_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *valid_delay* is accessed in the array.

Prototype

```
get_valid_delay
(
     valid_delay: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

valid_delay	Valid delay array to store TVALID delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for *valid_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns valid_delay

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
axi4stream_tr_if_0(bfm_index));

-- Get the TVALID delay for the first transfer of the tr_id transaction.
get_valid_delay(valid_delay, 0, tr_id, bfm_index,
                     axi4stream_tr_if_0(bfm_index));

-- Get the TVALID delay or the second transfer of the tr_id transaction.
get_valid_delay(valid_delay, 1, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));
```

set_ready_delay()

This nonblocking procedure sets the *ready_delay* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *ready_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *ready_delay* is accessed in the array.

Prototype

```
set_ready_delay
(
     ready_delay: in integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

ready_delay	Ready delay array to hold TREADY delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for *ready_delay*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns None

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

-- Set the TREADY delay to 3 ACLK cycles for the first transfer
-- of the tr_id transaction.
set_ready_delay(3, 0, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Set the TREADY delay to 2 ACLK cycles for the first transfer
-- of the tr_id transaction.
set_ready_delay(2, 1, tr_id, bfm_index, axistream_tr_if_0(bfm_index));
```

get_ready_delay()

This nonblocking procedure gets the *ready_delay* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *ready_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *ready_delay* is accessed in the array.

Prototype

```
get_ready_delay
(
     ready_delay: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

ready_delay	Read data channel array to hold RREADY delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for *ready_delay*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns ready_delay

Example

Note

You would not normally use this procedure within a Slave Test Program.

set_operation_mode()

This nonblocking procedure sets the *operation_mode* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
set_operation_mode
(
    operation_mode: in integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

operation_mode	Operation mode:
	AXI4STREAM_TRANSACTION_NON_BLOCKING;
	AXI4STREAM_TRANSACTION_BLOCKING (default);

transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns None

Example

Note

You would not normally use this procedure within a Slave Test Program.

get_operation_mode()

This nonblocking procedure gets the *operation_mode* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
get_operation_mode
(
   operation_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

operation_mode — Operation mode:

AXI4STREAM_TRANSACTION_NON_BLOCKING;
AXI4STREAM_TRANSACTION_BLOCKING; (default)

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns

operation_mode

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
                    axi4stream_tr_if_0(bfm_index));

....

-- Get the operation mode field of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index,
                axi4stream_tr_if_0(bfm_index));
```

set_transfer_done()

This nonblocking procedure sets a *transfer_done* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *transfer_done* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *transfer_done* is accessed in the array.

Prototype

```
set_transfer_done
(
     transfer_done : in integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transfer_done — Transfer *done* array for this transaction.

index — (Optional) Array element index number for *transfer_done*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns None

Example

Note

You would not normally use this procedure within a Slave Test Program.

get_transfer_done()

This nonblocking procedure gets a *transfer_done* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

The *transfer_done* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *transfer_done* is accessed in the array.

Prototype

```
get_transfer_done
(
     transfer_done : out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transfer_done	Transfer *done* array for this transaction.
index	(Optional) Array element index number for *transfer_done*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns transfer_done

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

....

-- Get the trans_done flag for the first transfer
-- of the tr_id transaction.
get_transfer_done(trans_done, 0, tr_id, bfm_index,
                     axi4stream_tr_if_0(bfm_index));

....

-- Get the trans_done flag for the second transfer
-- of the tr_id transaction.
get_transfer_done(trans_done, 1, tr_id, bfm_index,
                     axi4stream_tr_if_0(bfm_index));
```

set_transaction_done()

This nonblocking procedure sets the *transaction_done* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
set_transaction_done
(
     transaction_done : in integer;
     transaction_id   : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

Note

You would not normally use this procedure within a Slave Test Program.

get_transaction_done()

This nonblocking procedure gets the *transaction_done* field for a slave transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
get_transaction_done
(
     transaction_done : out integer;
     transaction_id   : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns transaction_done

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

....

-- Get the transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index,
                        axi4stream_tr_if_0(bfm_index));
```

get_packet()

This blocking procedure gets a slave packet that is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure.

It calls the *get_transfer()* procedure for each transfer of the packet, with the number of transfers defined by the transaction record *burst_length* field.

Prototype

```
procedure get_packet
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

index — (Optional) Data phase (beat) number.

last — Last data phase (beat) of the burst:
0 = data burst not complete
1 = data burst complete

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns last

Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

....

-- Get the packet of the tr_id transaction.
get_packet(tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_transfer()

This blocking procedure gets a slave transfer that is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure.

The transfer number within a packet is identified by the optional *index* argument. If no transfer *index* is supplied, then the first transfer within a packet is accessed.

It sets the *transfer_done* array *index* element to 1 when the transfer is completed. If this is the last transfer of the transaction, it sets the *transaction_done* field to 1 and returns the *last* argument set to 1 to indicate the whole transaction is complete.

Prototype

```
procedure get_transfer
(
   transaction_id   : in integer;
   index : in integer; --optional
   last : out integer;
   bfm_id           : in integer;
   signal tr_if     : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

index — (Optional) Data phase (beat) number.

last — Last data phase (beat) of the burst:

0 = data burst not complete
1 = data burst complete

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns last

Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

....

-- Get the first transfer of the tr_id transaction.
get_transfer(tr_id, 0, last, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Get the second transfer of the tr_id transaction.
get_transfer(tr_id, 1, last, bfm_index, axi4stream_tr_if_0(bfm_index));
```

execute_stream_ready()

This procedure executes a stream ready by placing the *ready* argument value onto the TREADY signal.

Prototype

```
procedure execute_stream_ready
(
   ready : in integer;
   non_blocking_mode : in integer; -- Optional
   bfm_id            : in integer;
   signal tr_if      : inout axi4stream_vhd_if_struct_t
);
```

Arguments

ready	The value of the TREADY signal
non_blocking_mode	(Optional) Controls the blocking or nonblocking mode of the procedure. 0 = blocking (default) 1 = nonblocking
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Assign TREADY = '0'. This will consume one cycle.
execute_stream_ready(0, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Two clock cycle wait.
for i in 0 to 1 loop
   wait_on(AXI4STREAM_CLOCK_POSEDGE, bfm_index,
            axi4stream_tr_if_0(bfm_index));
end loop;

-- Assign TREADY = '1'.
execute_stream_ready(1, bfm_index, axi4stream_tr_if_0(bfm_index));
```

print()

This nonblocking procedure prints a transaction record that is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure.

Prototype

```
procedure print
(
   transaction_id  : in integer;
   print_delays : in integer;
   bfm_id          : in integer;
   signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

print_delays — Print delay values flag:
0 = do not print the delay values (default).
1 = print the delay values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi4stream_tr_if_0(bfm_index));
```

destruct_transaction()

This blocking procedure removes a transaction record, for cleanup purposes and memory management, uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure.

Prototype

```
procedure destruct_transaction
(
   transaction_id   : in integer;
   bfm_id           : in integer;
   signal tr_if     : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index,
                              axi4stream_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

wait_on()

This blocking procedure waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
procedure wait_on
(
   phase            : in integer;
   count: in integer; -optional
   bfm_id           : in integer;
   signal tr_if     : inout axi4stream_vhd_if_struct_t
);
```

Arguments

phase

Wait for:

AXI4STREAM_CLOCK_POSEDGE
AXI4STREAM_CLOCK_NEGEDGE
AXI4STREAM_CLOCK_ANYEDGE
AXI4STREAM_CLOCK_0_TO_1
AXI4STREAM_CLOCK_1_TO_0
AXI4STREAM_RESET_POSEDGE
AXI4STREAM_RESET_NEGEDGE
AXI4STREAM_RESET_ANYEDGE
AXI4STREAM_RESET_0_TO_1
AXI4STREAM_RESET_1_TO_0

count

(Optional) Wait for a number of events to occur set by *count*.

bfm_id

BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if

Transaction signal interface. Refer to
for more details.

Returns

None

Example

```
wait_on(AXI4STREAM_RESET_POSEDGE, bfm_index,
            axi4stream_tr_if_0(bfm_index));

wait_on(AXI4STREAM_CLOCK_POSEDGE, 10, bfm_index,
            axi4stream_tr_if_0(bfm_index));
```

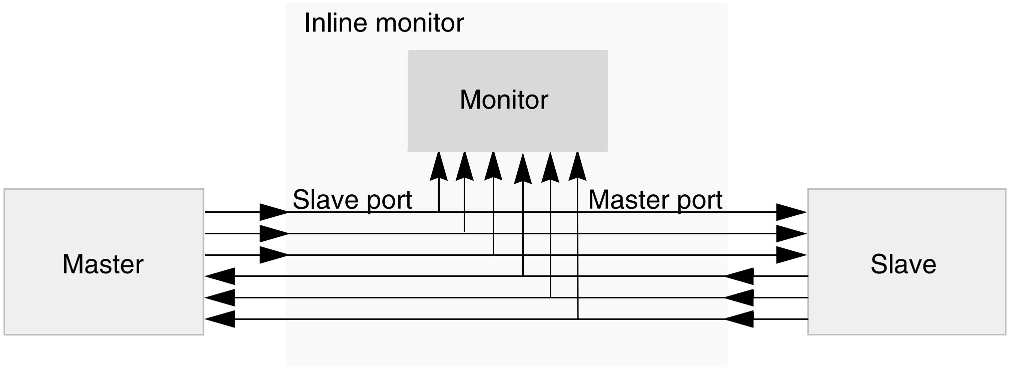
Chapter 10
VHDL Monitor BFM

This section provides information about the VHDL monitor BFM. It has an API that contains procedures to configure the BFM and to access the dynamic Transaction Record during the life of the transaction.

Inline Monitor Connection

The connection of a monitor BFM to a test environment differs from that of a master and slave BFM. It is wrapped within an inline monitor interface and connected inline between a master and slave, as shown in Figure 10-1. It has separate master and slave ports, and monitors protocol traffic between a master and slave. The monitor has access to all the facilities provided by the monitor BFM.

Figure 10-1. Inline Monitor Connection Diagram



Monitor BFM Protocol Support

The monitor BFM supports the full AMBA AXI4-Stream protocol.

Monitor Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI4-Stream Protocol Specification chapter, which you can reference for details of the following monitor BFM API timing and events.

The AMBA AXI4-Stream Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the monitor BFM does not contain any timescale, timeunit, or timeprecision declarations, with the signal setup and hold times specified in units of simulator time-steps.

Monitor BFM Configuration

The monitor BFM supports the full range of signals defined for the AMBA AXI4-Stream Protocol Specification. It has parameters that you use to configure the widths of the data and ID signals, and transaction fields to configure timeout factors, setup and hold times, and so on.

You can change the data and ID signals widths from their default settings by assigning them new values, usually in the top-level module of the test bench. These new values are then passed into the monitor BFM using a parameter port list of the monitor BFM module

Table 10-1 lists the parameter names for the data and ID signals and their default values.

Table 10-1. Monitor BFM Signal Width Parameters

Signal Width Parameter	Description
AXI4_ID_WIDTH	ID signal width in bits. This applies to the TID signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_USER_WIDTH	User data signal width in bits. This applies to the TUSER signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 8.
AXI4_DEST_WIDTH	Destination routing signal width in bits. This applies to the TDEST signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 18.
AXI4_DATA_WIDTH	Data signal width in bits. This applies to the TDATA signal. Refer to the AMBA AXI4-Stream Protocol Specification for more details. Default: 1024.

A monitor BFM has configuration fields that you set with the *set_config()* procedure to configure timeout factors, setup and hold times, and so on. You get the value of a configuration field with the *get_config()* procedure. Table 10-2 describes the full list of configuration fields.

Table 10-2. Monitor BFM Configuration

Configuration Field	Description
Timing Variables	
AXI4STREAM_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.

Table 10-2. Monitor BFM Configuration (cont.)

Configuration Field	Description
AXI4STREAM_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY	The maximum delay permitted between the assertion of TVALID to the assertion of TREADY. Default: 10000.
Master Attributes	
AXI4STREAM_LAST_DURING_IDLE	Controls the value of TLAST during idle. 0 = TLAST driven to 0 during idle (default) 1 = TLAST driven to 1 during idle
Error Detection	
AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
AXI4STREAM_CONFIG_ENABLE_ASSERTION	Individual enable/disable of an assertion check in the BFM. Refer to Assertions chapter for details 0 = disabled 1 = enabled (default)

[1] Refer to Monitor Timing and Events for details of simulator time-steps.

Monitor Assertions

The monitor BFM performs protocol error checking using built-in assertions.

Note

The built-in BFM assertions are independent of programming language and simulator.

By default, all built-in assertions are enabled in the monitor BFM. To globally disable them in the monitor BFM, use the *set_config()* command as shown in Example 10-1.

Example 10-1. Monitor BFM Disable All Assertions

```
set_config(AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS,0,bfm_index,
axi4stream_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. Example 10-2 shows how to disable assertion checking for the TLAST signal changing between the TVALID and TREADY handshake signals.

Example 10-2. Monitor BFM Individual Assertion Enable/Disable

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector :
std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4stream_tr_if_0(bfm_index));

-- Assign the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion bit to 0
config_assert_bitvector(AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY) := ‘0’;

-- Set the new value of the assertion bit vector
set_config(AXI4STREAM_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4stream_tr_if_0(bfm_index));
```

Note

Do not confuse the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector with the AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4STREAM_TLAST_CHANGED_BEFORE_TREADY assertion, follow the code sequence in Example 10-2 and assign the assertion enable within the AXI4STREAM_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to “Assertions” on page 203.

VHDL Monitor BFM API

This section describes the VHDL monitor API.

Each procedure available within the monitor BFM API is detailed in the following chapter. The *set*()* and *get*()* procedures that operate on the Transaction Record fields have a simple rule for the procedure name: *set_* or *get_* followed by the name of the transaction field to be accessed. Refer to “Transaction Record” on page 21 for details of transaction field names.

Note

The monitor BFM API package is the *axi4stream/bfm/mgc_axi4stream_bfm_pkg.vhd* file packaged within the Mentor Verification IP Altera Edition.

set_config()

This nonblocking procedure sets the configuration of the monitor BFM.

Prototype

```
procedure set_config
(
   config_name   : in std_logic_vector(7 downto 0);
   config_val : in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
   downto 0)|integer;
   bfm_id        : in integer;
   signal tr_if  : inout axi4stream_vhd_if_struct_t
);
```

Arguments

config_name — Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_
  ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

config_val — Refer to "Overloaded Procedure Common Arguments" on page 85 for description and valid values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
set_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi4stream_tr_if_0(bfm_index));
```

get_config()

This nonblocking procedure gets the configuration of the monitor BFM.

Prototype

```
procedure get_config
(
   config_name   : in std_logic_vector(7 downto 0);
   config_val    : out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
   downto 0)|integer;
   bfm_id        : in integer;
   signal tr_if  : inout axi4stream_vhd_if_struct_t
);
```

Arguments

config_name — Configuration name:

AXI4STREAM_CONFIG_SETUP_TIME
AXI4STREAM_CONFIG_HOLD_TIME
AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR
AXI4STREAM_CONFIG_LAST_DURING_IDLE
AXI4STREAM_CONFIG_MAX_LATENCY_TVALID_ASSERTION_TO_TREADY

AXI4STREAM_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4STREAM_CONFIG_ENABLE_ASSERTION

config_val — Refer to "Monitor BFM Configuration" on page 156 for description and valid values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns config_val

Example

```
get_config(AXI4STREAM_CONFIG_BURST_TIMEOUT_FACTOR, config_value,
     bfm_index, axi4stream_tr_if_0(bfm_index));
```

create_monitor_transaction()

This nonblocking procedure creates a monitor transaction. All transaction fields default to legal protocol values, unless previously assigned a value. This procedure creates and returns the *transaction_id* argument.

Prototype

```
procedure create_monitor_transaction
(
   transaction_id   : out integer;
   bfm_id           : in integer;
   signal tr_if     : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Protocol Transaction Fields

data — Data array in bytes

byte_type — Byte type array:

AXI4STREAM_DATA_BYTE; (default)
AXI4STREAM_NULL_BYTE;
AXI4STREAM_POS_BYTE;
AXI4STREAM_ILLEGAL_BYTE;

id — Data stream identifier.

dest — Destination routing information.

user_data — User data array.

Operational Transaction Fields

operation_mode — Operation mode:

AXI4STREAM_TRANSACTION_NON_BLOCKING;
AXI4STREAM_TRANSACTION_BLOCKING; (default)

valid_delay — TVALID delay measured in ACLK cycles for this transaction. (default = 0).

ready_delay — TREADY delay measured in ACLK cycles for this transaction. (default = 0).

transfer_done — Transfer *done* flag array for this transaction

transaction_done — Transaction *done* flag for this transaction

Returns

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85.

Example

```
-- Create a monitor transaction
-- Returns the transaction ID (tr_id) for this created transaction.
create_monitor_transaction(tr_id, bfm_index,
axi4stream_tr_if_3(bfm_index));
```

get_data()

This nonblocking procedure gets a *data* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

The *data* byte is identified by the optional *index* argument. If no *index* is supplied, then the first *data* byte is accessed in the array.

Prototype

```
get_data
(
     data: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

data	Data byte.
index	(Optional) Array element index number for *data*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns data

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

-- Get the data field for the first byte of the tr_id transaction.
get_data(data, 0, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Get the data field for the second byte of the tr_id transaction.
get_data(data, 1, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_byte_type()

This nonblocking procedure gets a *byte_type* field array element for a monitor transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

The *byte_type* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *byte_type* is accessed in the array.

Prototype

```
get_byte_type
(
     byte_type: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

byte_type — Byte type array:

AXI4STREAM_DATA_BYTE; (default)
AXI4STREAM_NULL_BYTE;
AXI4STREAM_POS_BYTE;
AXI4STREAM_ILLEGAL_BYTE;

index — (Optional) Array element index number for *byte_type*.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns

byte_type

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

-- Get the byte_type field for the first byte of the tr_id transaction.
get_byte_type(byte_type, 0, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));

-- Get the byte_type field for the second byte of the tr_id transaction.
get_byte_type(byte_type, 1, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));
```

get_id()

This nonblocking procedure gets the data stream identifier *id* field for a monitor transaction that is uniquely identified by the *transaction_id* field and previously created by the *create_monitor_transaction()* procedure.

Prototype

```
get_id
(
     id: in std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto 0 )
     | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

id	Data stream identifier value placed on the TID signals.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns id

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

....

-- Get the id field of the tr_id transaction.
get_id(id, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_dest()

This nonblocking procedure gets the routing information *id* field for a monitor transaction that is uniquely identified by the *transaction_id* field and previously created by the *create_monitor_transaction()* procedure.

Prototype

```
get_dest
(
     dest: out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1 downto
     0 ) | integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

dest	Data stream routing information value placed on the TDEST signals.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns dest

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

....

-- Get the dest field of the tr_id transaction.
get_dest(dest, tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_user_data()

This nonblocking procedure gets a *user_data* field array element for a monitor transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

The *user_data* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *user_data* is accessed in the array.

Prototype

```
get_user_data
(
     user_data: out std_logic_vector(AXI4STREAM_MAX_BIT_SIZE-1
     downto 0 ) | integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

user_data	User data array values placed on the TUSER signals.
index	(Optional) Array element index number for *user_data*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns user_data

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(3, tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

-- Get the user_data field for the first transfer
-- of the tr_id transaction.
get_user_data(user_data, 0, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));

-- Get the user_data field for the second transfer
-- of the tr_id transaction.
get_user_data(user_data, 1, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));
```

get_valid_delay()

This nonblocking procedure gets the *valid_delay* field for a monitor transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

The *valid_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *valid_delay* is accessed in the array.

Prototype

```
get_valid_delay
(
    valid_delay: out integer;
    index : in integer; --optional
    transaction_id  : in integer;
    bfm_id : in integer;
    signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

valid_delay	Valid delay array to store TVALID delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for *valid_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns valid_delay

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

-- Get the TVALID delay for the first transfer of the tr_id transaction.
get_valid_delay(valid_delay, 0, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));

-- Get the TVALID delay or the second transfer of the tr_id transaction.
get_valid_delay(valid_delay, 1, tr_id, bfm_index,
                     axi4stream_tr_if_0(bfm_index));
```

get_ready_delay()

This nonblocking procedure gets the *ready_delay* field for a monitor transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

The *ready_delay* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *ready_delay* is accessed in the array.

Prototype

```
get_ready_delay
(
     ready_delay: out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

ready_delay — Read data channel array to hold RREADY delays measured in ACLK cycles for this transaction. Default: 0.

index — (Optional) Array element index number for *ready_delay*.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns ready_delay

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(3, tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

-- Get the TREADY delay for the first transfer of the tr_id transaction.
get_ready_delay(ready_delay, 0, tr_id, bfm_index,
                     axi4stream_tr_if_0(bfm_index));

-- Get the TREADY delay or the second transfer of the tr_id transaction.
get_ready_delay(ready_delay, 1, tr_id, bfm_index,
                  axi4stream_tr_if_0(bfm_index));
```

get_operation_mode()

This nonblocking procedure gets the *operation_mode* field for a monitor transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
get_operation_mode
(
     operation_mode: out integer;
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

operation_mode — Operation mode:

AXI4STREAM_TRANSACTION_NON_BLOCKING;
AXI4STREAM_TRANSACTION_BLOCKING; (default)

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns

operation_mode

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(3, tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

....

-- Get the operation mode field of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index,
                          axi4stream_tr_if_0(bfm_index));
```

get_transfer_done()

This nonblocking procedure gets a *transfer_done* field for a monitor transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

The *transfer_done* array element is identified by the optional *index* argument. If no *index* is supplied, then the first *transfer_done* is accessed in the array.

Prototype

```
get_transfer_done
(
     transfer_done : out integer;
     index : in integer; --optional
     transaction_id  : in integer;
     bfm_id : in integer;
     signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transfer_done	Transfer *done* array for this transaction.
index	(Optional) Array element index number for *transfer_done*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                axi4stream_tr_if_0(bfm_index));

....

-- Get the transfer_done flag for the first transfer
-- of the tr_id transaction.
get_transfer_done(transfer_done, 0, tr_id, bfm_index,
                     axi4stream_tr_if_0(bfm_index));

....

-- Get the transfer_done flag for the second transfer
-- of the tr_id transaction.
get_transfer_done(transfer_done, 1, tr_id, bfm_index,
                        axi4stream_tr_if_0(bfm_index));
```

get_transaction_done()

This nonblocking procedure gets the *transaction_done* field for a monitor transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
get_transaction_done
(
    transaction_done : out integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    signal tr_if : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns transaction_done

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

....

-- Get the transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index,
                        axi4stream_tr_if_0(bfm_index));
```

get_packet()

This blocking procedure gets a monitor packet that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

It calls the *get_transfer()* procedure for each transfer of the packet with the number of transfers defined by the transaction record *burst_length* field.

Prototype

```
procedure get_packet
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
index	(Optional) Data phase (beat) number.
last	Last data phase (beat) of the burst: 0 = data burst not complete 1 = data burst complete
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns last

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                  axi4stream_tr_if_0(bfm_index));

....

-- Get the packet of the tr_id transaction.
get_packet(tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

get_transfer()

This blocking procedure gets a monitor transfer that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

The transfer number within a packet is identified by the optional *index* argument. If no transfer *index* is supplied, then the first transfer within a packet is accessed.

It sets the *transfer_done* array *index* element to 1 when the transfer completes. If this is the last transfer of the transaction, it sets the *transaction_done* field to 1 and returns the *last* argument set to 1 to indicate the whole transaction is complete.

Prototype

```
procedure get_transfer
(
   transaction_id   : in integer;
   index : in integer; --optional
   last : out integer;
   bfm_id           : in integer;
   signal tr_if     : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

index (Optional) Data phase (beat) number.

last Last data phase (beat) of the burst:
0 = data burst not complete
1 = data burst complete

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 85 for more details.

Returns last

Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

....

-- Get the first transfer of the tr_id transaction.
get_transfer(tr_id, 0, last, bfm_index, axi4stream_tr_if_0(bfm_index));

-- Get the second transfer of the tr_id transaction.
get_transfer(tr_id, 1, last, bfm_index, axi4stream_tr_if_0(bfm_index));
```

print()

This nonblocking procedure prints a transaction record that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

Prototype

```
procedure print
(
   transaction_id  : in integer;
   print_delays : in integer;
   bfm_id          : in integer;
   signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

print_delays — Print delay values flag:
   0 = do not print the delay values (default).
   1 = print the delay values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                  axi4stream_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi4stream_tr_if_0(bfm_index));
```

destruct_transaction()

This blocking procedure removes a transaction record, for cleanup purposes and memory management, that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

Prototype

```
procedure destruct_transaction
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   signal tr_if    : inout axi4stream_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns None

Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index,
                                 axi4stream_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi4stream_tr_if_0(bfm_index));
```

wait_on()

This blocking procedure waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
procedure wait_on
(
   phase            : in integer;
   count: in integer; -optional
   bfm_id           : in integer;
   signal tr_if     : inout axi4stream_vhd_if_struct_t
);
```

Arguments

phase

Wait for:

AXI4STREAM_CLOCK_POSEDGE
AXI4STREAM_CLOCK_NEGEDGE
AXI4STREAM_CLOCK_ANYEDGE
AXI4STREAM_CLOCK_0_TO_1
AXI4STREAM_CLOCK_1_TO_0
AXI4STREAM_RESET_POSEDGE
AXI4STREAM_RESET_NEGEDGE
AXI4STREAM_RESET_ANYEDGE
AXI4STREAM_RESET_0_TO_1
AXI4STREAM_RESET_1_TO_0

count

(Optional) Wait for a number of events to occur set by *count*.

bfm_id

BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

tr_if

Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 85 for more details.

Returns

None

Example

```
wait_on(AXI4STREAM_RESET_POSEDGE, bfm_index,
            axi4stream_tr_if_0(bfm_index));

wait_on(AXI4STREAM_CLOCK_POSEDGE, 10, bfm_index,
            axi4stream_tr_if_0(bfm_index));
```

Chapter 11
VHDL Tutorials

This chapter discusses how to use the Mentor VIP AE master and slave BFMs to verify slave and master components, respectively.

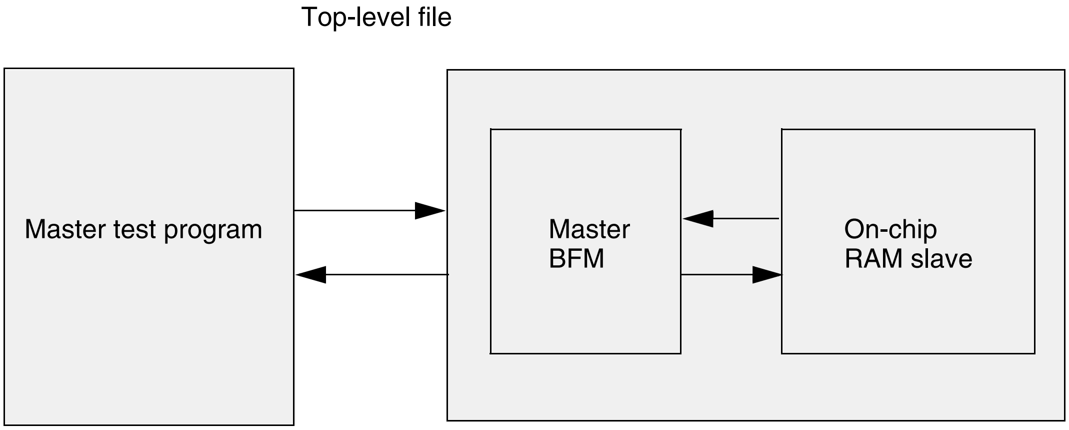
In the Verifying a Slave DUT tutorial, the slave is verified using a master BFM and test program. In the Verifying a Master DUT tutorial, the master issued transfers are verified using a slave BFM and test program.

Following this top-level discussion of how you verify a master and a slave component using the Mentor VIP AE is a brief example of how to run Qsys, the powerful system integration tool in the Quartus II software. This procedure shows you how to use Qsys to create a top-level DUT environment. For more details about this example, refer to "Getting Started with Qsys and the BFMs" on page 187.

Verifying a Slave DUT

A slave DUT component is connected to a master BFM at the signal-level. A master test program, written at the transaction-level, generates stimulus via the master BFM to verify the slave DUT. Figure 11-1 illustrates a typical top-level test bench environment.

Figure 11-1. Slave DUT Top-Level Test Bench Environment



A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

Master BFM Test Program

A master BFM test program is capable of creating a wide range of stimulus scenarios to verify a slave DUT. For a complete code listing of this master test program, refer to "VHDL Master BFM Code Example" on page 211.

The master test program contains a Traffic Generation process that creates and executes master transactions over the protocol signals. The following section describes the main process and variables.

Traffic Generation

The traffic generation process creates and executes master transactions, as shown in Example 11-1. The process defines a number of local variables to hold the transaction record, the byte count within a transfer, the transfer count, and inner and outer loop counters. Execution then waits for the ARESETn signal to be deasserted, followed by a positive ACLK edge. This satisfies the protocol requirements detailed in Section 2.7.2 of the AMBA 4 AXI4-Stream Protocol Specification.

Example 11-1. Definition and Initialization

```
process
   variable trans: integer;
   variable byte_count : integer := AXI4_DATA_WIDTH/8;
   variable transfer_count : integer;
   variable k    : integer;
   variable m    : integer;
begin
   wait_on(AXI4STREAM_RESET_POSEDGE, index, axi4stream_tr_if_0(index));
   wait_on(AXI4STREAM_CLOCK_POSEDGE, index, axi4stream_tr_if_0(index));
```

An outer *for* loop increments the *transfer_count* on each iteration of the loop, as shown in Example 11-2. Calling the *create_master_transaction()* procedure creates a master transaction, passing in the optional *transfer_count* as an argument to the procedure. The TID and TDEST signal values are then assigned for the data stream. Each iteration of the outer loop creates a master transaction with the *transfer_count* per transaction passed as an argument.

An inner *for* loop calls the *set_data()* procedure to load a byte into the *data* transaction field, and calls the *set_byte_type()* procedure to load the *byte_type* transaction field for the byte.

Calling the *execute_transaction()* procedure executes the *trans* transaction onto the protocol signals.

Example 11-2. Master Transaction Creation and Execution

```
--************************
-- Traffic generation: **
--************************
-- 10 x packet with
-- Number of transfer = i % 10. Values : 1, 2 .. 10
-- id = i % 15. Values 0, 1, 2 .. 14
-- dest = i %20. Values 0, 1, 2 .. 19
for i in  0 to 9 loop
   transfer_count := (i mod 10) + 1;
   create_master_transaction(transfer_count, trans, index,
                                 axi4stream_tr_if_0(index));
   set_id(i mod 15, trans, index, axi4stream_tr_if_0(index));
   set_dest(i mod 20, trans, index, axi4stream_tr_if_0(index));
   for j in  0 to ((transfer_count * byte_count) - 1) loop
      set_data(i + j, j, trans, index, axi4stream_tr_if_0(index));
      if(((i + j) mod 5) = 0) then
         set_byte_type(AXI4STREAM_NULL_BYTE, j, trans, index,
                           axi4stream_tr_if_0(index));
      elsif(((i + j) mod 5) = 1) then
         set_byte_type(AXI4STREAM_POS_BYTE, j, trans, index,
                           axi4stream_tr_if_0(index));
      else
         set_byte_type(AXI4STREAM_DATA_BYTE, j, trans, index,
                           axi4stream_tr_if_0(index));
      end if;
   end loop;
   execute_transaction(trans, index, axi4stream_tr_if_0(index));
end loop;
```

The master test program repeats the creation of master transactions similar to that shown in Example 11-2, but instead calls the *execute_transfer()* task per iteration of the inner *for* loop, as shown in Example 11-3.

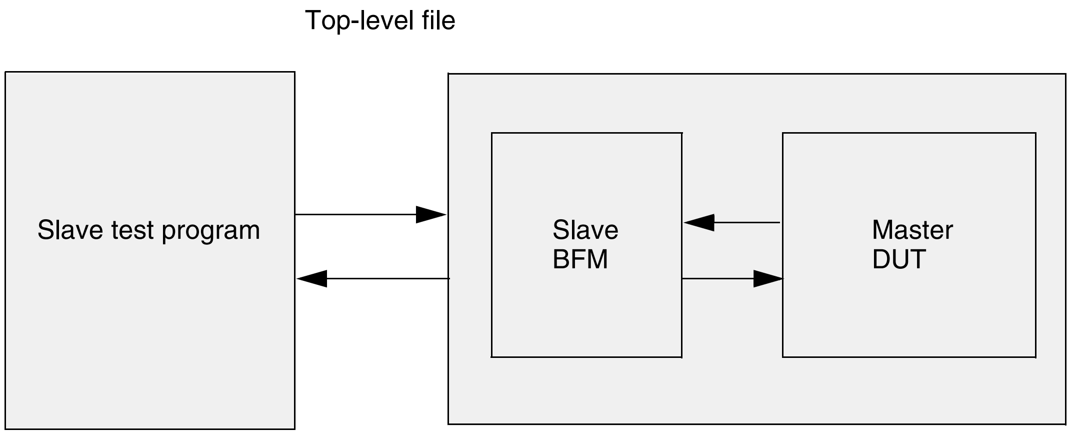
Example 11-3. Master Transfer Execution

```
-- 10 x packet at transfer level with
-- Number of transfer = i % 10. Values : 1, 2 .. 10
-- id = i % 15. Values 0, 1, 2 .. 14
-- dest = i %20. Values 0, 1, 2 .. 19
for i in  0 to 9 loop
   transfer_count := (i mod 10) + 1;
   create_master_transaction(transfer_count, trans, index,
                                 axi4stream_tr_if_0(index));
   set_id(i mod 15, trans, index, axi4stream_tr_if_0(index));
   set_dest(i mod 20, trans, index, axi4stream_tr_if_0(index));
   m := 0;
   while(m < transfer_count) loop
      k := m;
      while(k < transfer_count) loop
         set_data(k, k, trans, index, axi4stream_tr_if_0(index));
         if(((i + m) mod 5) = 0) then
            set_byte_type(AXI4STREAM_NULL_BYTE, m, trans, index,
                              axi4stream_tr_if_0(index));
         elsif(((i + m) mod 5) = 1) then
            set_byte_type(AXI4STREAM_POS_BYTE, m, trans, index,
                              axi4stream_tr_if_0(index));
         else
            set_byte_type(AXI4STREAM_DATA_BYTE, m, trans, index,
                              axi4stream_tr_if_0(index));
         end if;
         k := k + 1;
      end loop;
      execute_transfer(trans, m / byte_count, index,
                           axi4stream_tr_if_0(index));
      m := m + byte_count;
   end loop;
end loop;
```

Verifying a Master DUT

A master DUT component is connected to a slave BFM at the signal-level. A slave test program, written at the transaction-level, generates stimulus using the slave BFM to verify the master DUT. Figure 11-2 illustrates a typical top-level test bench environment.

Figure 11-2. Master DUT Top-Level Test Bench Environment



A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

Slave BFM Test Program

The slave test program contains a Basic Slave Test Program API Definition that implements a simplified interface for you to start verifying a master DUT with minimal effort. The API allows the slave BFM to control back-pressure to the master DUT by configuring the delay for the assertion of the TREADY signal. No other slave test program editing is required in this case.

The Advanced Slave Test Program API Definition allows the slave BFM to receive protocol transfers and insert a delay for the assertion of the TREADY signal. No further analysis of the protocol transfer content is performed. If analysis is required, you need to edit the slave test program to add this feature.

For a complete code listing of the slave test program, refer to “VHDL Slave BFM Code Example” on page 214.

Basic Slave Test Program API Definition

The Basic Slave Test Program API contains the following:

- Configuration variable *m_insert_wait* to insert a delay in the assertion of the TREADY protocol signal
- Procedure *ready_delay()* to configure the delay of the TREADY signal

m_insert_wait

The *m_insert_wait* configuration signal controls the insertion of a delay for the TREADY protocol signal defined by the *ready_delay()* procedure. To insert a delay, set *m_insert_wait* to 1 (default); otherwise, set to 0 as shown in Example 11-4.

Example 11-4. m_insert_wait

```
-- This signal controls the wait insertion in axi4 stream transfers
-- coming from master.
-- Making ~m_insert_wait~ to '0' truns off the wait insertion.
signal m_insert_wait : std_logic := '1';
```

ready_delay()

The *ready_delay* procedure inserts a delay for the TRREADY signal. The delay value extends the length of a protocol transfer by a defined number of ACLK cycles. The starting point of the delay is determined by the completion of a previous transfer, or from the first positive ACLK edge after reset at the start of simulation.

The *ready_delay()* task initially sets TREADY to 0 by calling the *execute_stream_ready()* procedure, as shown in Example 11-5. The delay is inserted by calling the *wait_on()* procedure within a *for* loop statement. You can edit the number of repetitions to change the delay. After the delay, the *execute_stream_ready()* procedure is called again to set the TREADY signal to 1.

Example 11-5. ready_delay

```
procedure ready_delay(signal tr_if : inout axi4stream_vhd_if_struct_t);

--////////////////////////////////////////////////
-- Code user could edit according to requirements
--////////////////////////////////////////////////

-- Procedure : ready_delay
-- This is used to set ready delay to extend the transfer
procedure ready_delay(signal tr_if : inout axi4stream_vhd_if_struct_t) is
begin
   --  Making TREADY '0'. This will consume one cycle.
   execute_stream_ready(0, index, tr_if);
   -- Two clock cycle wait. In total 3 clock wait.
   for i in 0 to 1 loop
      wait_on(AXI4STREAM_CLOCK_POSEDGE, index, tr_if);
   end loop;
   -- Making TREADY '1'.
   execute_stream_ready(1, index, tr_if);
end ready_delay;
```

Note

In addition to the above procedures and variables, you can configure other aspects of the slave BFM by using these procedures: *set_config()* and *get_config()*.

Advanced Slave Test Program API Definition

The remaining section of this tutorial presents a walk-through of the Advanced Slave Test Program API within the slave BFM test program. It consists of a single *initial block()* process that receives protocol transfers, inserting a delay in the assertion of the TREADY signal as detailed in the *Basic Slave Test Program API Definition*.

initial block()

Within a process, the slave test program defines a local variable *trans* to hold the Transaction Record of the transaction, as shown in Example 11-6. The initial wait for the ARESETn signal to be deactivated, followed by a positive ACLK edge, satisfies the protocol requirement detailed in Section 2.7.2 of the AMBA 4 AXI4-Stream Protocol Specification.

Example 11-6. Initialization

```
--/////////////////////////////////////////////////////////////////////
-- Code user do not need to edit
--/////////////////////////////////////////////////////////////////////
process
   variable trans: integer;
   variable i : integer;
   variable last : integer;
begin
   --*******************
   --** Initialisation **
   --********************
   wait_on(AXI4STREAM_RESET_POSEDGE, index, axi4stream_tr_if_0(index));
   wait_on(AXI4STREAM_CLOCK_POSEDGE, index, axi4stream_tr_if_0(index));
```

To receive protocol transfers, you must create a slave transaction. Within a *loop,* the *create_slave_transaction()* procedure is called to create a slave transaction, returning the *transaction_id* field of the transaction via the *trans* variable, as shown in Example 11-7.

An inner *while* loop iterates until the *last* transfer has been received. On each iteration, a delay is inserted before the TREADY signal is set to 1 by calling the *ready_delay()* procedure if *m_insert_wait* is set to 1. After any TREADY delay, the blocking *get_transfer()* procedure is called and waits for a transfer to be received.

If further analysis of the received transfer is required, then you need to edit the Advanced Slave API to achieve this. You can obtain details of the Transaction Record for the received transfer using the *get*()* procedures within the VHDL Slave BFM.

Example 11-7. Transfer Receiving

```
   loop
      create_slave_transaction(trans, index, axi4stream_tr_if_0(index));
      i := 0;
      last := 0;
      while(last = 0) loop
         if(m_insert_wait = '1') then
            -- READY is through path 0
            ready_delay(axi4stream_tr_if_0(index));
         end if;
         get_transfer(trans, i, last, index, axi4stream_tr_if_0(index));
         i := i + 1;
      end loop;
   end loop;

   wait;
end process;
```

Chapter 12
Getting Started with Qsys and the BFMs

Note

A license is required to access the Mentor Graphics VIP AE bus functional models and inline monitor. See Mentor VIP AE License Requirements for details.

This example shows you how to use the Qsys tool in Quartus II software to create a top-level design environment. You will be using the *ex1_back_to_back_sv*, a SystemVerilog example from the *$QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/axi4stream/qsys-examples* directory in the Altera Complete Design Suite (ACDS) installation.

Do the following tasks to set up the design environment:

1. Create a work directory.
2. Copy the example to the work directory.
3. Invoke Qsys from the Quartus II software *Tools* menu.
4. Generate a top-level netlist.
5. Run the simulation by referencing the *README* text file and command scripts for your simulation environment.

Setting Up a Simulation from a UNIX Platform

The following steps outline how to set up the simulation environment from a UNIX platform.

1. Create a work directory into which you copy the example directory *qsys-examples*, which contains the directory *ex1_back_to_back_sv* from the *Installation*.

   a. Using the *mkdir* command, create the work directory into which you will copy the *qsys-examples* directory.

   ```
   mkdir axi4stream-qsys-example
   ```

   b. Using the *cp* command, copy the *qsys-examples* directory from the *Installation* directory into your work directory.

   ```
   cp -r $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/axi4stream/\
      qsys-examples/* axi4stream-qsys-example/
   ```

2. Using the *cd* command, change the directory path to your local path where the example resides.

   ```
   cd axi4stream-qsys-example/ex1_back_to_back_sv
   ```

3. Open the Qsys tool. Refer to the Running the Qsys Tool section for details.

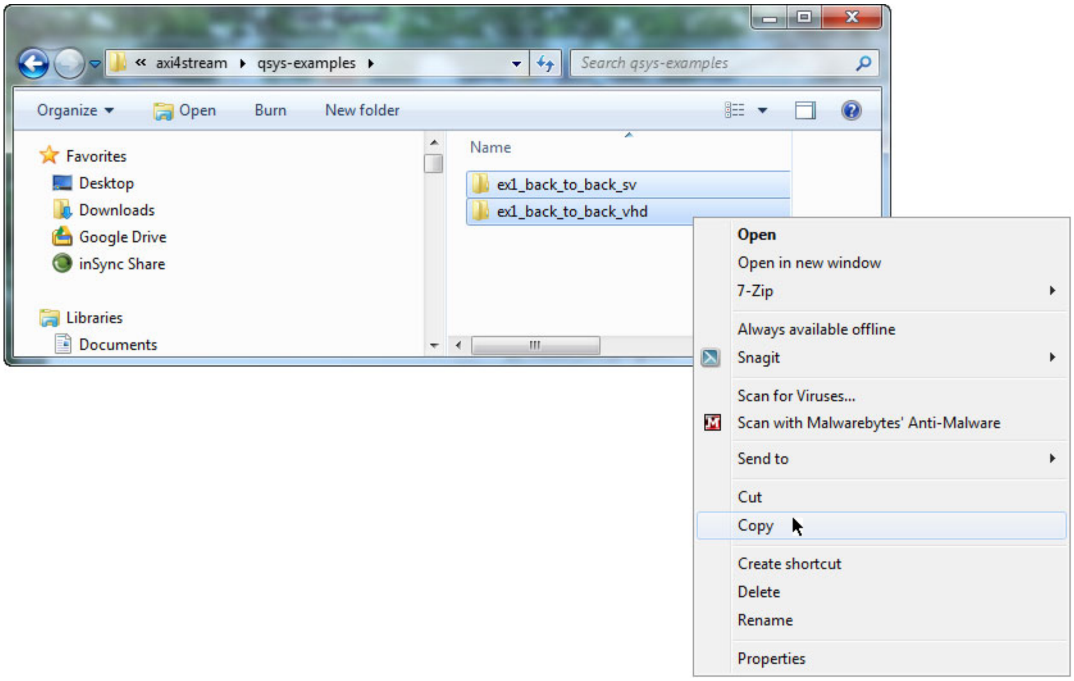
Setting Up Simulation from the Windows GUI

The following steps outline how to set up the simulation environment from a Windows GUI. This example uses the Windows7 platform.

1. Create a work folder into which you copy the contents of the *qsys-examples* folder, which includes the *ex1_back_to_back_sv* folder from the *Installation*.

    a. Using the GUI, select a location for your work folder, then click the *New folder* option on the window's menu bar to create and name a work folder. For this example name the work folder *axi4stream-qsys-examples.* Refer to figures 12-1 and 12-3 below.

Figure 12-1. Copy *qsys-examples* from the Installation Folder



    b. Copy the contents of the *qsys-examples* folder from the Installation folder to your work folder.

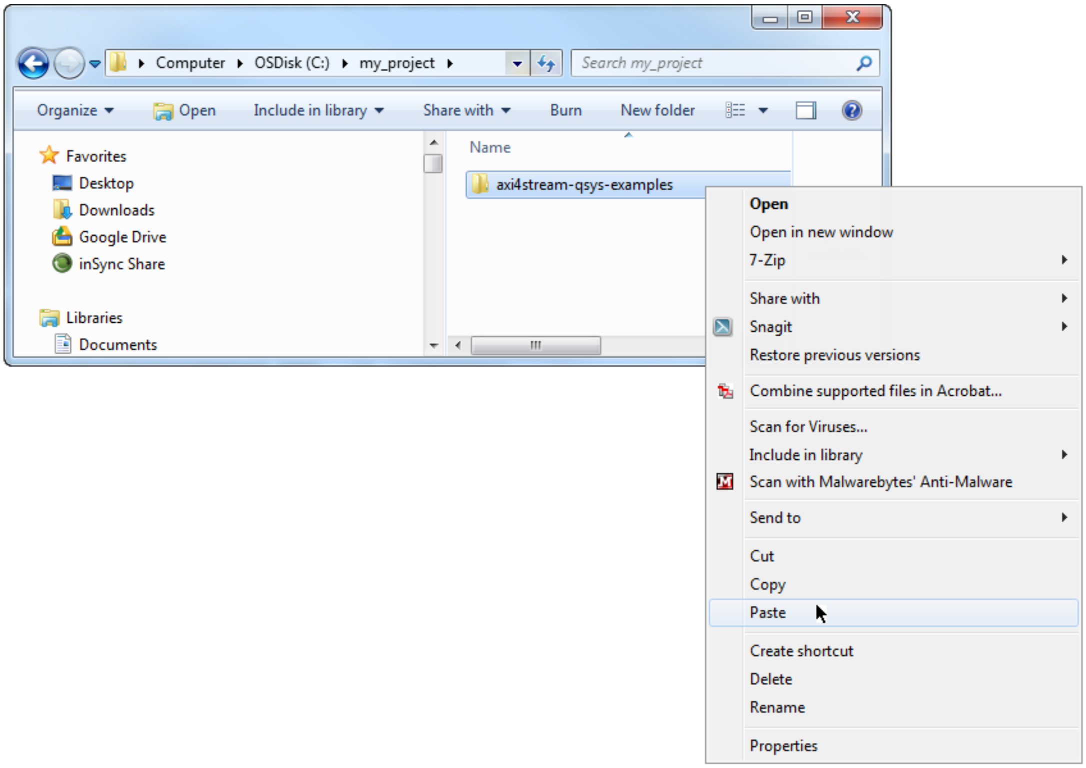
    Open the Installation and work folders. In the Installation folder, double-click the *qsys-examples* folder to select and open it. When the folder opens, type `CRTL/A` to select the contents of the directory, then right-click to display the drop-down menu and select *Copy* from the drop-down menu.

    Go to the open work folder. Double-click on the folder.

    When the folder opens, right-click inside the work folder and select *Paste* from the drop-down menu to copy the contents of the *qsys-examples* folder to the new *axi4stream-qsys-examples* work folder.

Paste the *qsys-examples* from the *Installation* folder in to the *axi4stream-qsys-examples* work folder (refer to Figure 12-2).

Figure 12-2. Paste qsys-examples from Installation to Work Folder



Note

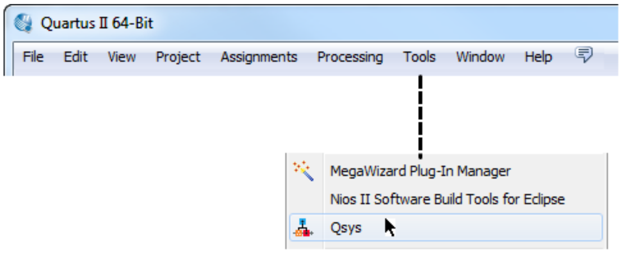
Alternatively, open both folders, the *Installation* folder containing the *qsys-examples* folder and the new *axi4stream-qsys-examples* work folder. Use the Windows *select*, *drag*, and *drop* functions to select the *qsys-examples* folder in the *Installation* folder, and then drag the contents to and drop it in the new *axi4stream-qsys-examples* work folder.

2. After creating the new *axi4stream-qsys-examples* work folder and copying the contents of the *qsys-examples* to it, open the Qsys tool. Refer to Running the Qsys Tool section for details.

Running the Qsys Tool

1. Open Qsys in the Quartus II software menu.
   To do this, start the Quartus II software. When the Quartus II GUI appears, select *Tools>Qsys* (refer to Figure 12-3).

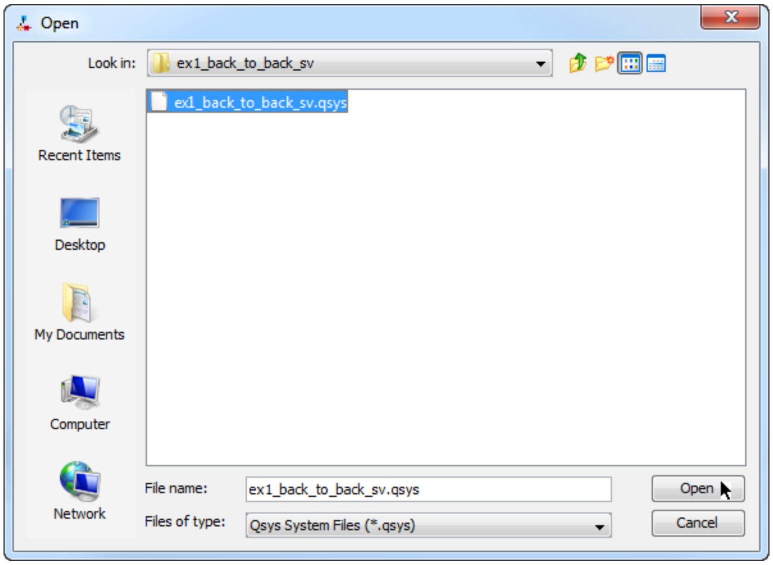
Figure 12-3. Select Qsys from the Quartus II Software Top-Level Menu



2. From the Qsys open window, use the *File>Open* command to open and select the file *ex1_back_to_back_sv.qsys*. This Qsys file is in the directory *axi4stream-qsys-examples\ex1_back_to_back_sv* (refer to Figure 12-3).

   Select and Open the *ex1_back_to_back_sv.qsys* example.

Figure 12-4. Open the *ex1_back_to_back_sv.qsys* Example



Look in: ex1_back_to_back_sv

ex1_back_to_back_sv.qsys

File name: ex1_back_to_back_sv.qsys

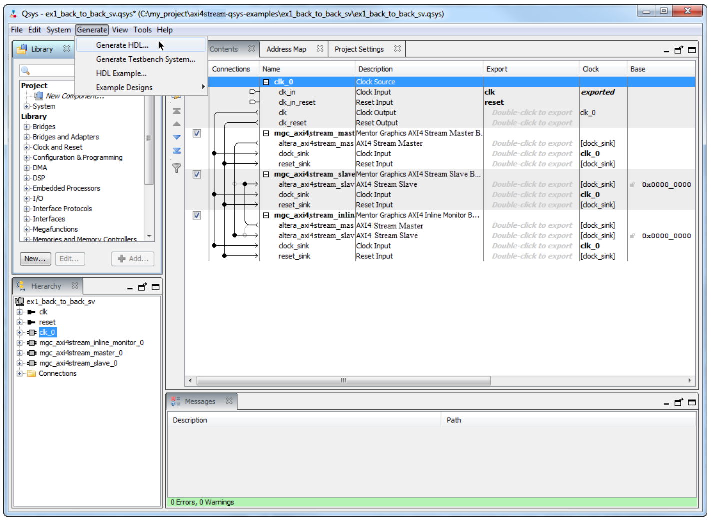
Files of type: Qsys System Files (*.qsys)

Note

If you open the Qsys tool in a subsequent session, a Qsys dialog asks you if you want to open this file.

3. Qsys displays the connectivity of the selected example as shown in Figure 12-5.

Figure 12-5. Quartus II Software Displays the Connectivity of the Example



Connections	Name	Description	Export	Clock	Base
	clk_0	Clock Source			
	clk_in	Clock Input	**clk**	***exported***	
	clk_in_reset	Reset Input	**reset**		
	clk	Clock Output	*Double-click to export*	clk_0	
	clk_reset	Reset Output	*Double-click to export*		
☑	**mgc_axi4stream_mast**	Mentor Graphics AXI4 Stream Master B.			
	altera_axi4stream_mas	AXI4 Stream Master	*Double-click to export*	[clock_sink]	
	clock_sink	Clock Input	*Double-click to export*	**clk_0**	
	reset_sink	Reset Input	*Double-click to export*	[clock_sink]	
☑	**mgc_axi4stream_slave**	Mentor Graphics AXI4 Stream Slave B...			
	altera_axi4stream_slav	AXI4 Stream Slave	*Double-click to export*	[clock_sink]	0x0000_0000
	clock_sink	Clock Input	*Double-click to export*	**clk_0**	
	reset_sink	Reset Input	*Double-click to export*	[clock_sink]	
☑	**mgc_axi4stream_inlin**	Mentor Graphics AXI4 Inline Monitor B...			
	altera_axi4stream_mas	AXI4 Stream Master	*Double-click to export*	[clock_sink]	
	altera_axi4stream_slav	AXI4 Stream Slave	*Double-click to export*	[clock_sink]	0x0000_0000
	clock_sink	Clock Input	*Double-click to export*	**clk_0**	
	reset_sink	Reset Input	*Double-click to export*	[clock_sink]	

Description	Path

0 Errors, 0 Warnings

Note

If you are using VHDL, you must select each BFM and verify that the index number specified for the BFM is correct. An information dialog displays the properties of the BFM when you select it. Ensure the specified BFM *index* is correct in this dialog. If you do not know the correct index number, check the VHDL code for the BFM.

4. Click the *Generate* drop-down menu on the Qsys toolbar, and select *Generate HDL* to open the Generation options window, as shown in Figure 12-5.

5. Specify the Generation window options shown in the following:

    a. Synthesis section

        i. Set the *Create HDL design files for synthesis* to *None* to inhibit the generation of synthesis files.

        ii. Uncheck the *Create block symbol file (.bsf)* check box.

    b. Simulation section

        i. Set the *Create simulation model* to *Verilog*.

    c. Change the path of the example. In the *Path* field of the Output Directory section, ensure the path correctly specifies the subdirectory *ex1_back_to_back_sv*, which is the subdirectory containing the example that you just copied into a temporary directory.

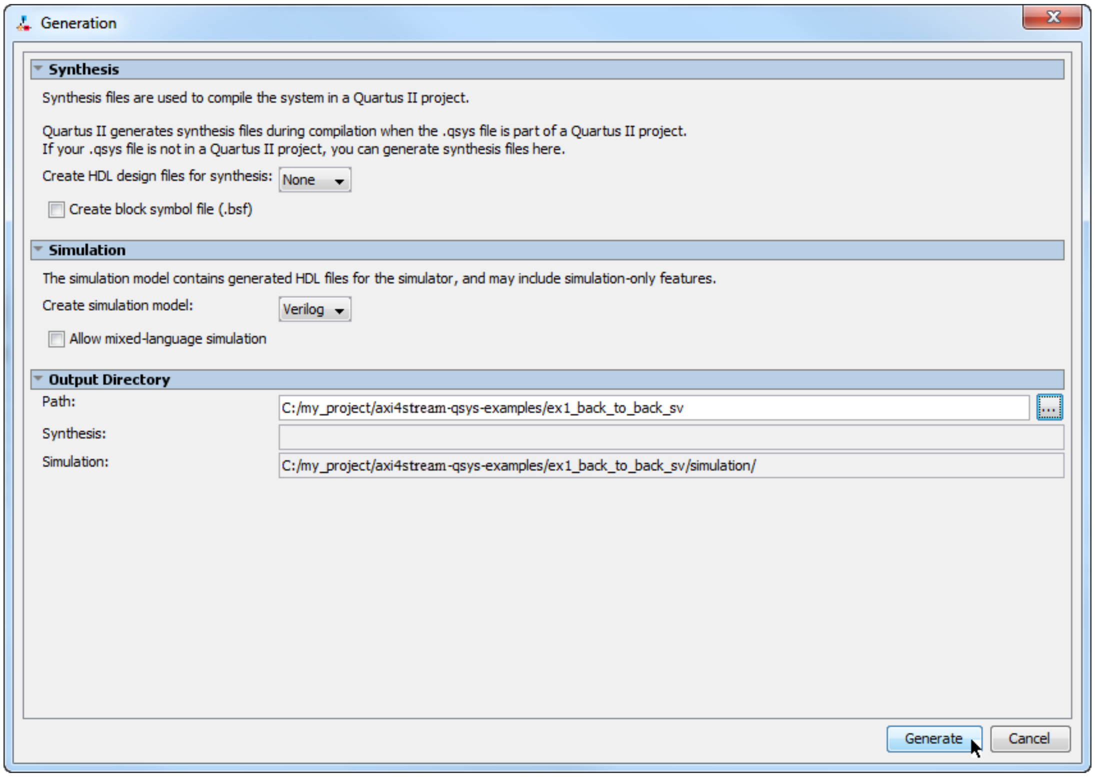
Note

If the subdirectory name of the example is duplicated in the *Path* field, you must remove one of the duplicated subdirectory names. To reset the path, double-click the square browse button to the right of the *Path* field and locate the correct path of the example.

The path name of the example specified in the *Path* field of the Output Directory section **must be correct before** generating the HDL for the example.

6. Click the *Generate* button on the bottom right side of the window, as shown in Figure 12-6.

Figure 12-6. Qsys Generation Window Options



Synthesis

Synthesis files are used to compile the system in a Quartus II project.

Quartus II generates synthesis files during compilation when the .qsys file is part of a Quartus II project.
If your .qsys file is not in a Quartus II project, you can generate synthesis files here.

Create HDL design files for synthesis: None

□ Create block symbol file (.bsf)

Simulation

The simulation model contains generated HDL files for the simulator, and may include simulation-only features.

Create simulation model: Verilog

□ Allow mixed-language simulation

Output Directory

Path: C:/my_project/axi4stream-qsys-examples/ex1_back_to_back_sv

Synthesis:

Simulation: C:/my_project/axi4stream-qsys-examples/ex1_back_to_back_sv/simulation/

7. Refer to the section Running a Simulation for steps to start the simulation.

Running a Simulation

The choice of simulator determines the process that you follow to run a simulation. The process for each simulator is detailed in the following sections:

- ModelSim Simulation
- Questa Simulation
- Cadence IES Simulation
- Synopsys VCS Simulation

For each simulator, a *README* text file and a command script file is provided in the installed Mentor VIP AE directory location *axi4stream/qsys-examples/ex1_back_to_back_sv*. Table 12-1 details the *README* text file instructions to load a model into the simulator, and the script command file to start the simulation.

Table 12-1. SystemVerilog README Files and Script Names for all Simulators

	Questa Simulation	ModelSim Simulation	IES Simulation	VCS Simulation
README	README-Questa.txt	README-ModelSim.txt	README-IUS.txt	README-VCS.txt
Script File	example.do	example.do	example-ius.sh	example-vcs.sh

Note

The VHDL example *axi4stream/qsys-examples/ex1_back_to_back_vhd* has equivalent *README* text files and command script files. The process to follow for VHDL simulation is similar to that for SystemVerilog simulation.

ModelSim Simulation

You can run a ModelSim simulation from a GUI interface or a command line. Before starting a simulation, you must do the following:

- Check that the *$QUARTUS_ROOTDIR* environment variable points to the Quartus II software directory in the Quartus II software installation. The example command script *example.do* requires this variable to locate the installed Mentor VIP AE BFMs during simulation.
- Ensure that the environment variable *MvcHome* points to the location of the installed Mentor VIP AE BFM. You can set the location of *MvcHome* using one of the following options:
  - To set the *MvcHome* variable in the *modelsim.ini* file, refer to the section "Editing the modelsim.ini File."
  - To specify the *-mvchome* option on the command line, refer to the section "Starting a Simulation from a UNIX Command Line."

The following sections outline how to run a ModelSim simulation from either a GUI or a command line.

Starting a Simulation From the ModelSim GUI

To start a simulation with the ModelSim simulator GUI:

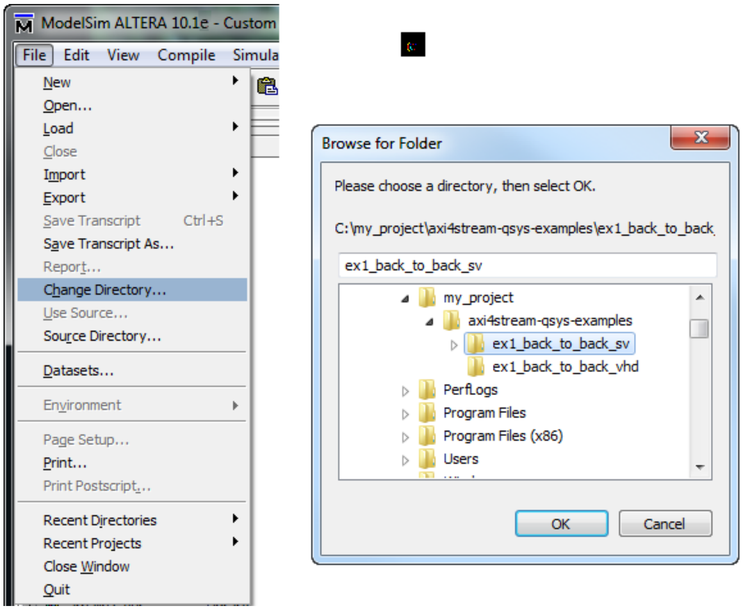
1. Start the ModelSim GUI.

```
vsim -mvchome $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common
```

2. Change directory to the work directory that contains the example to be simulated with method (a) or (b) below.

    a. From the *File* menu, click the *Change Directory* option. When the *Browse for Folder* dialog appears, select the work directory that contains the example.

Figure 12-7. Select the Work Directory



Please choose a directory, then select OK.

C:\my_project\axi4stream-qsys-examples\ex1_back_to_back

ex1_back_to_back_sv

    b. In the ModelSim Transcript window, change to the work directory containing the example to simulate.

```
vsim> cd axi4stream-qsys-examples/ex1_back_to_back_sv
```

3. Run the *example.do* script within the Transcript window by typing the following command to compile and elaborate the test programs:

```
vsim> do example.do
```

Note

For details about the processing performed by the *example.do* script, refer to the section ModelSim Example Script Processing.

4. In the Transcript window, start the simulation and run to completion.

```
vsim> run -all
```

Starting a Simulation from a UNIX Command Line

To start a simulation with the ModelSim simulator from a UNIX command line:

1. Change the directory to the work directory containing the example to be simulated.

   ```
   cd axi4stream-qsys-examples/ex1_back_to_back_sv
   ```

2. In a shell, start the Modelsim simulator with the *example.do* script.

   ```
   vsim -mvchome $QUARTUS_ROOTDIR/../ip/altera/\
         mentor_vip_ae/common -gui -do example.do
   ```

Note

For details about the processing performed by the *example.do* script, refer to the section ModelSim Example Script Processing.

3. In the Transcript window, start the simulation and run to completion.

   ```
   vsim> run -all
   ```

ModelSim Example Script Processing

The *example.do* script described below is contained in the installed Mentor VIP AE directory location *axi4stream/qsys-examples/ex1_back_to_back_sv*.

The Mentor VIP AE BFMs for AXI4-Stream are compiled.

```
set TOP_LEVEL_NAME top
set QSYS_SIMDIR    simulation

source $QSYS_SIMDIR/mentor/msim_setup.tcl
if {![info exists env(MENTOR_VIP_AE)]}
{
  set env(MENTOR_VIP_AE) $env(QUARTUS_ROOTDIR)/../ip/altera/mentor_vip_ae
}

   ensure_lib libraries
   ensure_lib libraries/work
   vmap work  libraries/work

   vlog -work work -sv \
      $env(MENTOR_VIP_AE)/common/questa_mvc_svapi.svh \
      $env(MENTOR_VIP_AE)/axi4stream/bfm/mgc_common_axi.sv \
      $env(MENTOR_VIP_AE)/axi4stream/bfm/mgc_axi_monitor.sv \
      $env(MENTOR_VIP_AE)/axi4stream/bfm/mgc_axi_inline_monitor.sv \
      $env(MENTOR_VIP_AE)/axi4stream/bfm/mgc_axi_master.sv \
      $env(MENTOR_VIP_AE)/axi4stream/bfm/mgc_axi_slave.sv
```

The two *tcl* alias commands *dev_com* and *com* compile the required design files. These alias commands are defined in the *msim_setup.tcl* simulation script generated by Qsys, along with the simulation model files.

```
# Compile device library files
dev_com

# Compile Qsys-generated design files
com
```

The three example test programs are compiled:

```
# Compile example test program files
vlog  master_test_program.sv
vlog   slave_test_program.sv
vlog monitor_test_program.sv
```

The example top-level file is compiled:

```
# Compile top-level design file
vlog top.sv
```

Simulation starts with the *elab* alias defined in the *msim_setup.tcl* simulation script generated by Qsys:

```
# Simulate
elab
```

Editing the modelsim.ini File

The ModelSim simulator does not have a default installation directory path defined for the environment variable *MvcHome*; therefore, you must define a path for this variable.

Note

Setting *MvcHome* within the *modelsim.ini* file eliminates the need to specify the *-mvchome* option on the *vsim* command line.

To provide the installation directory path of the Mentor VIP AE for running a ModelSim simulation:

1. Edit the *modelsim.ini* file and find the section that starts with *[vsim]*.
2. Search for *MvcHome*. If it is not already defined in the *modelsim.ini* file, you must add it. You can add this variable at any location in the *[vsim]* section.

   If the *modelsim.ini* file is read-only, you must modify the permissions of the file to allow write access.
3. Add or change the *MvcHome* path to point to the location where the Mentor VIP AE is installed. Do not forget the *common* subdirectory.

   ```
   MvcHome = $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common
   ```

Note

Do not use the ModelSim *vmap* command to specify the installed location of the Mentor VIP AE because this places the definition of the environment variable *MvcHome* in the *[library]* section of *modelsim.ini*. For example, do not use the command `vmap MvcHome $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common`.

Questa Simulation

To run a Questa simulation, follow the process detailed in the ModelSim Simulation section.

Cadence IES Simulation

Before starting a Cadence IES simulation, you must do the following:

- Check that the *$QUARTUS_ROOTDIR* environment variable points to the Quartus II software directory in the Quartus II software installation. The example script *example-ius.sh* requires this variable to locate the Mentor VIP AE BFMs during simulation.
- Set the environment variable *CDS_ROOT* to the installation directory of the IES Verilog compiler *ncvlog*. The *cds_root* command returns the installation directory of the specified tool *ncvlog*.

```
setenv CDS_ROOT        `cds_root ncvlog`
```

Starting a Simulation from a UNIX Command Line

To start a simulation with the Cadence IES simulator from a UNIX command line:

1. Change the directory to the work directory containing the example to be simulated.

   ```
   cd axi4stream-qsys-examples/ex1_back_to_back_sv
   ```

2. Start the Cadence IES simulator with the *example-ius.sh* script.

   - For a 32-bit simulation, execute this command:

   ```
   sh example-ius.sh 32
   ```

   - For a 64-bit simulation execute the command:

   ```
   sh example-ius.sh 64
   ```

Note

For details about the process steps performed by the *example-ius.sh* script, see the section Cadence IES Example Script Processing.

Cadence IES Example Script Processing

The *example-ius.sh* script described below is contained in the installed Mentor VIP AE directory location *axi4stream/qsys-examples/ex1_back_to_back_sv.*

The Mentor VIP AE BFMs for AXI4-Stream are compiled. The *ncsim_setup.sh* simulation script is generated by Qsys, along with the simulation model files.

```
#!/bin/sh

# Usage: <command> [32|64]
# 32 bit mode is run unless 64 is passed in as the first argument.

MENTOR_VIP_AE=${MENTOR_VIP_AE:-$QUARTUS_ROOTDIR/../ip/ \
                              altera/mentor_vip_ae}

if [ "$1" == "64" ]
then
   export QUESTA_MVC_GCC_LIB=$MENTOR_VIP_AE/common/ \
         questa_mvc_core/linux_x86_64_gcc-4.4_ius
   export INCA_64BIT=1
else
   export QUESTA_MVC_GCC_LIB=$MENTOR_VIP_AE/common/ \
         questa_mvc_core/linux_gcc-4.4_ius
fi
export LD_LIBRARY_PATH=$QUESTA_MVC_GCC_LIB:$LD_LIBRARY_PATH

cd simulation/cadence
# Run once, just to execute the 'mkdir' for the libraries.
source ncsim_setup.sh SKIP_DEV_COM=1 SKIP_COM=1 SKIP_ELAB=1 SKIP_SIM=1

# Compile VIP
   ncvlog -sv \
      "$MENTOR_VIP_AE/common/questa_mvc_svapi.svh" \
      "$MENTOR_VIP_AE/axi4stream/bfm/mgc_common_axi4stream.sv" \
      "$MENTOR_VIP_AE/axi4stream/bfm/mgc_axi4stream_monitor.sv" \
      "$MENTOR_VIP_AE/axi4stream/bfm/mgc_axi4stream_inline_monitor.sv" \
      "$MENTOR_VIP_AE/axi4stream/bfm/mgc_axi4stream_master.sv" \
      "$MENTOR_VIP_AE/axi4stream/bfm/mgc_axi4stream_slave.sv"
```

The three example test programs are compiled:

```
# Compile the test program
ncvlog -sv ../../master_test_program.sv
ncvlog -sv ../../monitor_test_program.sv
ncvlog -sv ../../slave_test_program.sv
```

The example top-level file is compiled:

```
# Compile the top
ncvlog -sv ../../top.sv
```

Elaboration and simulation starts with the *ncsim_setup.sh* command. The Cadence IES simulator requires the SystemVerilog library path *-sv_lib* to be passed to the simulator.

```
# Elaborate and simulate
source ncsim_setup.sh \
   USER_DEFINED_ELAB_OPTIONS="\"-timescale 1ns/1ns\"" \
   USER_DEFINED_SIM_OPTIONS="\"-MESSAGES \
      -sv_lib
$QUESTA_MVC_GCC_LIB/libaxi4stream_IN_SystemVerilog_IUS_full\"" \
   TOP_LEVEL_NAME=top
```

Synopsys VCS Simulation

Before starting a Synopsys VCS simulation, you must do the following:

- Check that the *$QUARTUS_ROOTDIR* environment variable points to the Quartus II software directory in the Quartus II software installation. The example script *example-vcs.sh* requires this variable to locate the Mentor VIP AE BFMs during simulation.
- Set the environment variable *VCS_HOME* to the installation directory of the VCS Verilog compiler.

  ```
  setenv VCS_HOME <Installation-of-VCS>
  ```

Starting a Simulation from a UNIX Command Line

To start a simulation with the Synopsys VCS simulator from a UNIX command line:

1. Change the directory to the work directory containing the example to be simulated.

   ```
   cd axi4stream-qsys-examples/ex1_back_to_back_sv
   ```

2. Start the Synopsys VCS simulator with the *example-vcs.sh* script.
   - For a 32-bit simulation execute the command:

     ```
     sh example-vcs.sh 32
     ```

   - For a 64-bit simulation execute the command:

     ```
     sh example-vcs.sh 64
     ```

Note

For details about the process steps performed by the *example-vcs.sh* script, see the section Synopsys VCS Example Script Processing.

Synopsys VCS Example Script Processing

The *example-vcs.sh* script described below is contained in the installed Mentor VIP AE directory location *axi4stream/qsys-examples/ex1_back_to_back_sv*.

The Mentor VIP AE BFMs for AXI4-Stream are compiled. The *vcs_setup.sh* simulation script is generated by Qsys, along with the simulation model files.

```
#!/bin/sh

# Usage: <command> [32|64]
# 32 bit mode is run unless 64 is passed in as the first argument.

MENTOR_VIP_AE=${MENTOR_VIP_AE:-
$QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae}

if [ "$1" == "64" ]
then
   export RUN_64bit=-full64
   export VCS_TARGET_ARCH=`getsimarch 64`
   export LD_LIBRARY_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_64-shared/lib64
   export QUESTA_MVC_GCC_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_64-shared
   export QUESTA_MVC_GCC_LIB=${MENTOR_VIP_AE}/common/ \
                           questa_mvc_core/linux_x86_64_gcc-4.7.2_vcs
else
   export RUN_64bit=
   export LD_LIBRARY_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_32-shared/lib
   export QUESTA_MVC_GCC_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_32-shared
   export QUESTA_MVC_GCC_LIB=${MENTOR_VIP_AE}/common/ \
                           questa_mvc_core/linux_gcc-4.7.2_vcs
fi

cd simulation/synopsys/vcs
rm -rf csrc simv simv.daidir transcript ucli.key vc_hdrs.h

# VCS accepts the -LDFLAGS flag on the command line, but the shell quoting
# is too difficult. Just set the LDFLAGS ENV variable for the compiler to
# pick up. Alternatively, use the VCS command line option '-file' with the
# LDFLAGS set (this avoids shell quoting issues).
# vcs-switches.f:
# -LDFLAGS "-L ${QUESTA_MVC_GCC_LIB} -Wl,-rpath ${QUESTA_MVC_GCC_LIB}
# -laxi4stream_IN_SystemVerilog_VCS_full"
export LDFLAGS="-L ${QUESTA_MVC_GCC_LIB} -Wl, \
-rpath ${QUESTA_MVC_GCC_LIB} -laxi4stream_IN_SystemVerilog_VCS_full"

USER_DEFINED_ELAB_OPTIONS="\"\
   $RUN_64bit \
   +systemverilogext+.sv +vpi +acc +vcs+lic+wait \
   -cpp ${QUESTA_MVC_GCC_PATH}/xbin/g++ \
   \
   $MENTOR_VIP_AE/common/questa_mvc_svapi.svh \
   $MENTOR_VIP_AE/axi4stream/bfm/mgc_common_axi4stream.sv \
   $MENTOR_VIP_AE/axi4stream/bfm/mgc_axi4stream_monitor.sv \
   $MENTOR_VIP_AE/axi4stream/bfm/mgc_axi4stream_inline_monitor.sv \
   $MENTOR_VIP_AE/axi4stream/bfm/mgc_axi4stream_slave.sv \
   $MENTOR_VIP_AE/axi4stream/bfm/mgc_axi4stream_master.sv \
   \
```

The three example test programs and top-level file are compiled:

```
      ../../../master_test_program.sv \
      ../../../monitor_test_program.sv  \
      ../../../slave_test_program.sv \
      ../../../top.sv  \""
```

Elaboration and simulation starts with the *vcs_setup.sh* command.

```
   source vcs_setup.sh \
      USER_DEFINED_ELAB_OPTIONS="$USER_DEFINED_ELAB_OPTIONS" \
      USER_DEFINED_SIM_OPTIONS="'-l transcript'" \
      TOP_LEVEL_NAME=top
```

Chapter 13
Assertions

The master, slave, and monitor BFMs all support error checking via the firing of one or more assertions when a property detailed in the AMBA AXI4-Stream Protocol Specification has been violated. Each assertion may be individually enabled/disabled using the *set_config()* function for a particular BFM. The Property Reference column of Table 13-1 references the section number in the AMBA AXI4-Stream Protocol Specification of the property the assertion covers.

Table 13-1. AXI4-Stream Assertions

Error Code	Error Name	Description	Property Ref
AXI4STREAM -60000	AXI4STREAM_TDATA_CHANGED_ BEFORE_TREADY_ ON_INVALID_LANE	On an invalid byte lane (TSTRB = 0) the value of TDATA has changed between TVALID asserted and TREADY asserted.	2.2.1
AXI4STREAM -60001	AXI4STREAM_TDATA_X_ ON_INVALID_LANE	On an invalid byte lane (TSTRB = 0), TDATA has an X value.	-
AXI4STREAM -60002	AXI4STREAM_TDATA_Z_ ON_INVALID_LANE	On an invalid byte lane (TSTRB = 0), TDATA has a Z value.	-
AXI4STREAM -60003	AXI4STREAM_TDATA_CHANGED_ BEFORE_TREADY_ ON_VALID_LANE	On a valid byte lane (TSTRB = 1) the value of TDATA has changed between TVALID asserted and TREADY asserted.	2.2.1
AXI4STREAM -60004	AXI4STREAM_TDATA_X_ ON_VALID_LANE	On a valid byte lane (TSTRB = 1), TDATA has an X value.	-
AXI4STREAM -60005	AXI4STREAM_TDATA_Z_ ON_VALID_LANE	On a valid byte lane (TSTRB = 1), TDATA has a Z value.	-
AXI4STREAM -60006	AXI4STREAM_TDEST_CHANGED_ BEFORE_TREADY	The value of TDEST has changed between TVALID asserted and TREADY asserted.	2.2.1
AXI4STREAM -60007	AXI4STREAM_TDEST_X	TDEST has an X value.	-
AXI4STREAM -60008	AXI4STREAM_TDEST_Z	TDEST has a Z value.	-
AXI4STREAM -60009	AXI4STREAM_TID_CHANGED_ BEFORE_TREADY	The value of TID has changed between TVALID asserted and TREADY asserted.	2.2.1
AXI4STREAM -60010	AXI4STREAM_TID_X	TID has an X value.	-
AXI4STREAM -60011	AXI4STREAM_TID_Z	TID has a Z value.	-

Table 13-1. AXI4-Stream Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4STREAM -60012	AXI4STREAM_TKEEP_CHANGED_ BEFORE_TREADY	The value of TKEEP has changed between TVALID asserted and TREADY asserted.	2.2.1
AXI4STREAM -60013	AXI4STREAM_TKEEP_X	TKEEP has an X value.	-
AXI4STREAM -60014	AXI4STREAM_TKEEP_Z	TKEEP has a Z value.	-
AXI4STREAM -60015	AXI4STREAM_TLAST_CHANGED_ BEFORE_TREADY	The value of TLAST has changed between TVALID asserted and TREADY asserted.	2.2.1
AXI4STREAM -60016	AXI4STREAM_TLAST_X	TLAST has an X value	-
AXI4STREAM -60017	AXI4STREAM_TLAST_Z	TLAST has a Z value	-
AXI4STREAM -60018	AXI4STREAM_TREADY_X	TREADY has an X value.	-
AXI4STREAM -60019	AXI4STREAM_TREADY_Z	TREADY has a Z value.	-
AXI4STREAM -60020	AXI4STREAM_TSTRB_CHANGED_ BEFORE_TREADY	The value of TSTRB has changed between TVALID asserted and TREADY asserted.	2.2.1
AXI4STREAM -60021	AXI4STREAM_TSTRB_X	TSTRB has an X value.	-
AXI4STREAM -60022	AXI4STREAM_TSTRB_Z	TSTRB has a Z value.	-
AXI4STREAM -60023	AXI4STREAM_TUSER_CHANGED_ BEFORE_TREADY	The value of TUSER has changed between TVALID asserted and TREADY asserted.	2.2.1
AXI4STREAM -60024	AXI4STREAM_TUSER_X	TUSER has an X value.	-
AXI4STREAM -60025	AXI4STREAM_TUSER_Z	TUSER has a Z value.	-
AXI4STREAM -60026	AXI4STREAM_TVALID_HIGH_ EXITING_RESET	TVALID should have been driven low when exiting reset.	2.7.2
AXI4STREAM -60027	AXI4STREAM_TVALID_HIGH_ ON_FIRST_CLOCK	A master interface must only begin driving TVALID at a rising edge of ACLK following a rising edge of ACLK at which TRESETn is deasserted.	2.7.2
AXI4STREAM -60028	AXI4STREAM_TVALID_CHANGED_ BEFORE_TREADY	The value of TVALID has changed between TVALID asserted and TREADY asserted.	2.2.1
AXI4STREAM -60029	AXI4STREAM_TVALID_X	TVALID has an X value.	-

Table 13-1. AXI4-Stream Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4STREAM -60030	AXI4STREAM_TVALID_Z	TVALID has a Z value.	-
AXI4STREAM -60030	AXI4STREAM_DATA_ WIDTH_VIOLATION	The data bus width of axi4 stream interface must be an integer number of bytes.	2.1
AXI4STREAM -60031	AXI4STREAM_TDEST_ MAX_WIDTH_VIOLATION	The recommended width of TDEST on AXI4-Stream interface must be less than 4-bits.	2.1
AXI4STREAM -60032	AXI4STREAM_TID_ MAX_WIDTH_VIOLATION	The recommended width of TID on AXI4-Stream interface must be less than 8-bits.	2.1
AXI4STREAM -60033	AXI4STREAM_TUSER_ MAX_WIDTH_VIOLATION	The recommended width of TUSER on AXI4-Stream interface must be an integer multiplication of data bus width in bytes.	2.1
AXI4STREAM -60034	AXI4STREAM_AUXM_TID_ TDEST_WIDTH	The value of AXI4STREAM_ID_WIDTH + AXI4STREAM_DEST_WIDTH must not exceed 24. See ARM AXI4STREAM Protocol Compliance checkers.	-
AXI4STREAM -60035	AXI4STREAM_TSTRB_ HIGH_WHEN_TKEEP_LOW	The combination of TSTRB HIGH and TKEEP LOW is a reserved value.	2.3.4
AXI4STREAM -60036	AXI4STREAM_TUSER_FIELD_ NONZERO_NULL_BYTE	If a null byte is inserted, then appropriate number of user bits must also be inserted, which must be fixed LOW. (STRM(2.8))	2.8
AXI4STREAM -60037	AXI4STREAM_TREADY_NOT_ ASSERTED_AFTER_TVALID	When TVALID is asserted, ARREADY should be asserted within *config_max_latency_TVALID_assertion_to_TREADY* clock periods	
AXI4STREAM -60038	AXI4STREAM_INTERNAL_ RESERVED	A value reserved for internal purposes of the BFM.	-

Appendix A SystemVerilog Master and Slave Test Programs

SystemVerilog Master Test Program

The example code in this section is a simplified AXI4-Stream master that illustrates how you can use the *mgc_axi4stream_master* BFM.

```
//
************************************************************************
****
//
// Copyright 2007-2013 Mentor Graphics Corporation
// All Rights Reserved.
//
// THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS
THE PROPERTY OF
// MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE
TERMS.
//
//
************************************************************************
****

/*
    This is a simple example of an axi4stream master to demonstrate the
mgc_axi4stream_master BFM usage.

   This master performs a directed test, initiating 10 sequential packets
at higher abstraction level
    followed by 10 transfer at phase level.

*/

import mgc_axi4stream_pkg::*;
module master_test_program #(int AXI4_ID_WIDTH = 18, int AXI4_USER_WIDTH =
8, int AXI4_DEST_WIDTH = 18, int AXI4_DATA_WIDTH = 1024)
(
    mgc_axi4stream_master bfm
);

initial
begin
    axi4stream_transaction trans;
    static int byte_count = AXI4_DATA_WIDTH/8;
    int transfer_count;
    bit last;
    /*******************
    ** Initialisation **
```

```
*******************/
bfm.wait_on(AXI4STREAM_RESET_POSEDGE);
bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE);

/************************
** Traffic generation: **
************************/
// 10 x packet with
// Number of transfer = i % 10. Values : 1, 2 .. 10
// id = i % 15. Values 0, 1, 2 .. 14
// dest = i %20. Values 0, 1, 2 .. 19
for(int i = 0; i < 10; ++i)
begin
  transfer_count = (i % 10) + 1;
  trans = bfm.create_master_transaction(transfer_count);
  trans.id = i % 15;
  trans.dest = i % 20;
  for(int j = 0; j < (transfer_count * byte_count); ++j)
  begin
    trans.set_data(i + j, j);
    if(((i + j)% 5) == 0)
    begin
      trans.set_byte_type(AXI4STREAM_NULL_BYTE, j);
    end
    else if(((i + j)% 5) == 1)
    begin
      trans.set_byte_type(AXI4STREAM_POS_BYTE, j);
    end
    else
    begin
      trans.set_byte_type(AXI4STREAM_DATA_BYTE, j);
    end
  end
  bfm.execute_transaction(trans);
end

// 10 x packet at transfer level with
// Number of transfer = i % 10. Values : 1, 2 .. 10
// id = i % 15. Values 0, 1, 2 .. 14
// dest = i %20. Values 0, 1, 2 .. 19
for(int i = 0; i < 10; ++i)
begin
  transfer_count = (i % 10) + 1;
  trans = bfm.create_master_transaction(transfer_count);
  trans.id = i % 15;
  trans.dest = i % 20;
  for(int j = 0; j < transfer_count; ++j)
  begin
    for(int k = 0; k < byte_count; ++k)
    begin
      trans.set_data(k+j, ((j*byte_count)+k));
      if(((i + j)% 5) == 0)
      begin
        trans.set_byte_type(AXI4STREAM_NULL_BYTE, ((j*byte_count)+k));
      end
      else if(((i + j)% 5) == 1)
      begin
        trans.set_byte_type(AXI4STREAM_POS_BYTE, ((j*byte_count)+k));
```

```
          end
          else
          begin
            trans.set_byte_type(AXI4STREAM_DATA_BYTE, ((j*byte_count)+k));
          end
        end
        bfm.execute_transfer(trans, j, last);
      end
    end

    #100
    $finish();
end
endmodule
```

SystemVerilog Slave Test Program

The example code in this section is a simplified AXI4-Stream slave that illustrates how you can use the *mgc_axi4stream_slave* BFM.

```
//
*************************************************************************
****
//
// Copyright 2007-2013 Mentor Graphics Corporation
// All Rights Reserved.
//
// THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS
THE PROPERTY OF
// MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE
TERMS.
//
//
*************************************************************************
****

/*
    This is a simple example of an AXI4STREAM Slave to demonstrate the
mgc_axi4stream_slave BFM usage.
*/


import mgc_axi4stream_pkg::*;

module slave_test_program #(int AXI4_ID_WIDTH = 18, int AXI4_USER_WIDTH =
8, int AXI4_DEST_WIDTH = 18, int AXI4_DATA_WIDTH = 1024)
(
    mgc_axi4stream_slave bfm
);

  /////////////////////////////////////////////////
  // Code user could edit according to requirements
  /////////////////////////////////////////////////
  // This member controls the wait insertion in axi4 stream transfers
coming from master.
  // Making ~m_insert_wait~ to 0 truns off the wait insertion.
  bit m_insert_wait = 1;

  // Task : ready_delay
  // This is used to set ready delay to extend the transfer
  task ready_delay();
    // Making TREADY '0'. This will consume one cycle.
    bfm.execute_stream_ready(0);
    // Two clock cycle wait. In total 3 clock wait.
    repeat(2) bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE);
    // Making TREADY '1'.
    bfm.execute_stream_ready(1);
  endtask

  ///////////////////////////////////////////////////////////////////////
  // Code user do not need to edit
  ///////////////////////////////////////////////////////////////////////
  initial
  begin
    int i;
    bit last;
    axi4stream_transaction trans;
    /*******************
    ** Initialisation **
    *******************/
    bfm.wait_on(AXI4STREAM_RESET_POSEDGE);
    bfm.wait_on(AXI4STREAM_CLOCK_POSEDGE);

    // Packet receiving
    forever
    begin
      trans = bfm.create_slave_transaction();
      i = 0;
      last = 0;
      while(!last)
      begin
        if(m_insert_wait)
        begin
          ready_delay();
        end
        bfm.get_transfer(trans, i, last);
        ++i;
      end
    end
  end

endmodule
```

Appendix B
VHDL Master and Slave Test Programs

This appendix contains two VHDL code examples: one for the master BFM, and the other for the slave BFM.

VHDL Master BFM Code Example

The example code in this section is a simplified AXI4-Stream slave that illustrates how you can use the *mgc_axi4stream_master* BFM.

```
--
*************************************************************************
****
--
-- Copyright 2007-2013 Mentor Graphics Corporation
-- All Rights Reserved.
--
-- THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS
THE PROPERTY OF
-- MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE
TERMS.
--
--
*************************************************************************
****

--    This is a simple example of an axi4stream master to demonstrate the
mgc_axi4stream_master BFM usage.
--
--    This master performs a directed test, initiating 10 sequential
packets at higher abstraction level
--    followed by 10 transfer at phase level.

library ieee ;
use ieee.std_logic_1164.all;

library work;
use work.all;
use work.mgc_axi4stream_bfm_pkg.all;
entity master_test_program is
 generic(
            AXI4_ID_WIDTH : integer := 18;
            AXI4_USER_WIDTH : integer := 8;
            AXI4_DEST_WIDTH : integer := 18;
            AXI4_DATA_WIDTH : integer := 1024;
            index : integer range 0 to 511 := 0
           );
```

```
end master_test_program;

architecture master_test_program_a of master_test_program is

begin
  process
    variable trans: integer;
    variable byte_count : integer := AXI4_DATA_WIDTH/8;
    variable transfer_count : integer;
    variable k    : integer;
    variable m    : integer;
  begin
    wait_on(AXI4STREAM_RESET_POSEDGE, index, axi4stream_tr_if_0(index));
    wait_on(AXI4STREAM_CLOCK_POSEDGE, index, axi4stream_tr_if_0(index));

    --************************
    -- Traffic generation: **
    --************************
    -- 10 x packet with
    -- Number of transfer = i % 10. Values : 1, 2 .. 10
    -- id = i % 15. Values 0, 1, 2 .. 14
    -- dest = i %20. Values 0, 1, 2 .. 19
    for i in  0 to 9 loop
      transfer_count := (i mod 10) + 1;
      create_master_transaction(transfer_count, trans, index,
axi4stream_tr_if_0(index));
      set_id(i mod 15, trans, index, axi4stream_tr_if_0(index));
      set_dest(i mod 20, trans, index, axi4stream_tr_if_0(index));
      for j in  0 to ((transfer_count * byte_count) - 1) loop
        set_data(i + j, j, trans, index, axi4stream_tr_if_0(index));
        if(((i + j) mod 5) = 0) then
          set_byte_type(AXI4STREAM_NULL_BYTE, j, trans, index,
axi4stream_tr_if_0(index));
        elsif(((i + j) mod 5) = 1) then
          set_byte_type(AXI4STREAM_POS_BYTE, j, trans, index,
axi4stream_tr_if_0(index));
        else
          set_byte_type(AXI4STREAM_DATA_BYTE, j, trans, index,
axi4stream_tr_if_0(index));
        end if;
      end loop;
      execute_transaction(trans, index, axi4stream_tr_if_0(index));
    end loop;

    -- 10 x packet at transfer level with
    -- Number of transfer = i % 10. Values : 1, 2 .. 10
    -- id = i % 15. Values 0, 1, 2 .. 14
    -- dest = i %20. Values 0, 1, 2 .. 19
    for i in  0 to 9 loop
      transfer_count := (i mod 10) + 1;
      create_master_transaction(transfer_count, trans, index,
axi4stream_tr_if_0(index));
      set_id(i mod 15, trans, index, axi4stream_tr_if_0(index));
      set_dest(i mod 20, trans, index, axi4stream_tr_if_0(index));
      m := 0;
      while(m < transfer_count) loop
        k := 0;
        while(k < byte_count) loop
```

```
          set_data(k, ((m*byte_count)+k), trans, index,
axi4stream_tr_if_0(index));
          if(((i + m) mod 5) = 0) then
           set_byte_type(AXI4STREAM_NULL_BYTE, ((m*byte_count)+k), trans,
index, axi4stream_tr_if_0(index));
          elsif(((i + m) mod 5) = 1) then
            set_byte_type(AXI4STREAM_POS_BYTE, ((m*byte_count)+k), trans,
index, axi4stream_tr_if_0(index));
          else
           set_byte_type(AXI4STREAM_DATA_BYTE, ((m*byte_count)+k), trans,
index, axi4stream_tr_if_0(index));
          end if;
          k := k + 1;
        end loop;
        execute_transfer(trans, m, index, axi4stream_tr_if_0(index));
        m := m + 1;
      end loop;
    end loop;

    wait;
  end process;
end master_test_program_a;
```

VHDL Slave BFM Code Example

The example code in this section is a simplified AXI4-Stream slave that illustrates how you can use the *mgc_axi4stream_master* BFM.

```
--
****************************************************************************
****
--
-- Copyright 2007-2013 Mentor Graphics Corporation
-- All Rights Reserved.
--
-- THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS
THE PROPERTY OF
-- MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE
TERMS.
--
--
****************************************************************************
****
--
--  This is a simple example of an AXI4STREAM Slave to demonstrate the
mgc_axi4stream_slave BFM usage.

library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.all;
use work.mgc_axi4stream_bfm_pkg.all;

entity slave_test_program is
   generic(
            AXI4_ID_WIDTH : integer := 18;
            AXI4_USER_WIDTH : integer := 8;
            AXI4_DEST_WIDTH : integer := 18;
            AXI4_DATA_WIDTH : integer := 1024;
            index : integer range 0 to 511 := 0
           );
 end slave_test_program;

architecture slave_test_program_a of slave_test_program is
  --This member controls the wait insertion in axi4 stream transfers
coming from master.
  -- Making ~m_insert_wait~ to '0' truns off the wait insertion.
  signal m_insert_wait : std_logic := '1';

  procedure ready_delay(signal tr_if : inout axi4stream_vhd_if_struct_t);

  --////////////////////////////////////////////////
  -- Code user could edit according to requirements
  --////////////////////////////////////////////////

  -- Procedure : ready_delay
  -- This is used to set ready delay to extend the transfer
  procedure ready_delay(signal tr_if : inout axi4stream_vhd_if_struct_t)
is
  begin
    --  Making TREADY '0'. This will consume one cycle.
    execute_stream_ready(0, index, tr_if);
    -- Two clock cycle wait. In total 3 clock wait.
    for i in 0 to 1 loop
      wait_on(AXI4STREAM_CLOCK_POSEDGE, index, tr_if);
    end loop;
     -- Making TREADY '1'.
    execute_stream_ready(1, index, tr_if);
  end ready_delay;

begin

  --///////////////////////////////////////////////////////////////////////
  -- Code user do not need to edit
  --///////////////////////////////////////////////////////////////////////
  process
    variable trans: integer;
    variable i : integer;
    variable last : integer;
  begin
     --*******************
    --** Initialisation **
    --********************
     wait_on(AXI4STREAM_RESET_POSEDGE, index, axi4stream_tr_if_0(index));
     wait_on(AXI4STREAM_CLOCK_POSEDGE, index, axi4stream_tr_if_0(index));

    ------------------------/
    -- Packet receiving:--
    ------------------------/
    loop
      create_slave_transaction(trans, index, axi4stream_tr_if_0(index));
      i := 0;
      last := 0;
      while(last = 0) loop
        if(m_insert_wait = '1') then
          -- READY is through path
          ready_delay(axi4stream_tr_if_0(index));
        end if;
        get_transfer(trans, i, last, index, axi4stream_tr_if_0(index));
        i := i + 1;
      end loop;
    end loop;

    wait;
  end process;
end slave_test_program_a;
```

Third-party Software for Mentor Verification IP Altera Edition

This section provides information on open source and third-party software that may be included in the Mentor Verification IP Altera Edition software product.

This software application may include GNU GCC 4.3.3 third-party software. GNU GCC v4.3.3 is distributed under the terms of the GNU General Public License version 3.0 and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at: <install_directory>/docs/legal/gnu_gpl_3.0.pdf. Portions of this software may be subject to the GNU Free Documentation License v1.2. You can view a copy of the GNU Free Documentation License v1.2 at: <install_directory>/docs/legal/gnu_free_doc_1.2.pdf. Portions of this software may be subject to the GNU Lesser General Public License v2.1. You can view a copy of the GNU Lesser General Public License v2.1 at: <install_directory>/docs/legal/ gnu_lgpl_2.1.pdf. Portions of this software may be subject to the GNU Library General Public License v2. You can view a copy of the GNU Library General Public License v2 at: <install_directory>/docs/legal/ gnu_library_gpl_2.0.pdf. Portions of this software may be subject to the W3C License. You can view a copy of the W3C License at: <install_directory>/docs/legal/w3c_2002.pdf. Portions of this software may be subject to the Boost License version 1.0. You can view a copy of the Boost License v1.0 at: <install_directory>/docs/legal/ boost_1.0.pdf. To obtain a copy of the GNU GCC v4.3.3 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed GNU GCC v4.3.3 and valid for as long as Mentor Graphics offers customer support for this Mentor Graphics product. GNU GCC v4.3.3 may be subject to the following copyrights:

© 1987 Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED ‘‘AS IS’’ AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

© 1983, 1990, 1991 Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. [rescinded 22 July 1999]

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1991, 2000, 2006 AT&T
The author of this software is David M. Gay.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

© 2004 World Wide Web Consortium

(Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. This work is distributed under the W3C(r) Software License [1] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

```
[1] http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231
```

Permission to copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications:

1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.

2. Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the W3C Software Short Notice should be included (hypertext is preferred, text is permitted) within the body of any redistributed or derivative code.

3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

© 1996-1999 Silicon Graphics Computer Systems, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1994, 2002 Hewlett-Packard Company

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1991-2005 Unicode, Inc.
All rights reserved.

Distributed under the Terms of Use in http://www.unicode.org/copyright.html

Permission is hereby granted, free of charge, to any person obtaining a copy of the Unicode data files and any associated documentation (the "Data Files") or Unicode software and any associated documentation (the "Software") to deal in the Data Files or Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Data Files or Software, and to permit persons to whom the Data Files or Software are furnished to do so, provided that (a) the above copyright notice(s) and this permission notice appear with all copies of the Data Files or Software, (b) both the above copyright notice(s) and this permission notice appear in associated documentation, and (c) there is clear notice in each modified Data File or in the Software as well as in the documentation associated with the Data File(s) or Software that the data or software has been modified.

THE DATA FILES AND SOFTWARE ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THE DATA FILES OR SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in these Data Files or Software without prior written authorization of the copyright holder.

© 2004 Ami Tavory and Vladimir Dreizin, IBM-HRL

Permission to use, copy, modify, sell, and distribute this software is hereby granted without fee, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation. None of the above authors, nor IBM Haifa Research Laboratories, make any representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1997 - 2002, Makoto Matsumoto and Takuji Nishimura
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1999 Citrus Project
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software application may include Binutils version 2.19.1 third-party software. Binutils v2.19.1 is distributed under the terms of the GNU General Public License version 3.0 and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_gpl_3.0.pdf. Portions of this software may be subject to the GNU Free Documentation License v1.2. You can view a copy of the GNU Free Documentation License v1.2 at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_free_doc_1.2.pdf. Portions of this software may be subject to the GNU Library General Public License v2. You can view a copy of the GNU Library General Public License v2 at: <your_Mentor_Graphics_documentation_directory>/legal/ gnu_library_gpl_2.0.pdf. Portions of this software may be subject to the GNU Lesser General Public License v3.0. You can view a copy of the GNU Lesser General Public License v3.0 at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_lgpl_3.0.pdf. To obtain a copy of the Binutils v2.19.1 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed Binutils v2.19.1 and valid for as long as Mentor Graphics offers customer support for this Mentor Graphics product. Binutils v2.19.1 may be subject to the following copyrights:

© 1983, 1990, 1991, 1993, 1998, 2001, 2002 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

This software application may include MinGW gcc version 4.2.1 third-party software. MinGW gcc version 4.2.1 is distributed under the terms of the GNU General Public License version 2.0 and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_gpl_2.0.pdf. Portions of this software may be subject to the GNU Free Documentation License v1.2. You can view a copy of the GNU Free Documentation License v1.2 at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_free_doc_1.2.pdf. Portions of this software may be subject to the GNU Lesser General Public License v2.1. You can view a copy of the GNU Lesser General Public License v2.1 at: <your_Mentor_Graphics_documentation_directory>/legal/ gnu_lgpl_2.1.pdf . To obtain a copy of the MinGW gcc version 4.2.1 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed MinGW gcc version 4.2.1. MinGW gcc version 4.2.1 may be subject to the following copyrights:

© 1994

Hewlett-Packard Company

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1996,1997, 1998

Silicon Graphics Computer Systems, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 2004 Ami Tavory and Vladimir Dreizin, IBM-HRL.

Permission to use, copy, modify, sell, and distribute this software is hereby granted without fee, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation. None of the above authors, nor IBM Haifa Research Laboratories, make any representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1982, 1986, 1992, 1993

The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

   This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1992, 1993, 1994 Henry Spencer. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.

2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.

3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.

4. This notice may not be removed or altered.

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. **ORDERS, FEES AND PAYMENT.**

1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement ("Order(s)"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.

1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.

1.3. All Products are delivered FCA factory (Incoterms 2000), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 5.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer requests any change or enhancement to Software, whether in the course of receiving support or consulting services, evaluating Software, performing beta testing or otherwise, any inventions, product

improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics' Embedded Software Channel ("ESC"), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are linked into a composite program as an integral part of Customer's compiled computer program, provided that Customer distributes these files only in conjunction with Customer's compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into Customer's products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

4. **BETA CODE.**

4.1. Portions or all of certain Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. **RESTRICTIONS ON USE.**

5.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use it except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any source code from Software. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Software or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Software, or disclose to any third party the results of, or information pertaining to, any benchmark.

5.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.

5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

5.4. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at http://supportnet.mentor.com/about/legal/.

7. **AUTOMATIC CHECK FOR UPDATES; PRIVACY.** Technological measures in Software may communicate with servers of Mentor Graphics or its contractors for the purpose of checking for and notifying the user of updates and to ensure that the Software in use is licensed in compliance with this Agreement. Mentor Graphics will not collect any personally identifiable data in this process and will not disclose any data collected to any third party without the prior written consent of Customer, except to Mentor Graphics' outside attorneys or as may be required by a court of competent jurisdiction.

8. **LIMITED WARRANTY.**

8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

9. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **HAZARDOUS APPLICATIONS.** CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING ITS PRODUCTS USED IN APPLICATIONS WHERE THE FAILURE OR INACCURACY OF ITS PRODUCTS MIGHT RESULT IN DEATH OR PERSONAL INJURY ("HAZARDOUS APPLICATIONS"). NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. **INDEMNIFICATION.** CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10. THE PROVISIONS OF THIS SECTION 11 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

12. **INFRINGEMENT.**

12.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

12.2. If a claim is made under Subsection 12.1 Mentor Graphics may, at its option and expense, (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

12.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

12.4. THIS SECTION 12 IS SUBJECT TO SECTION 9 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS FOR DEFENSE, SETTLEMENT AND DAMAGES, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

13. **TERMINATION AND EFFECT OF TERMINATION.** If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term.

13.1. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

13.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.

14. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products and information about the products to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

15. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

16. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

17. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FLEXlm or FLEXnet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 17 shall survive the termination of this Agreement.

18. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the United States. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, USA, if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. This section shall not

restrict Mentor Graphics' right to bring an action against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

19. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

20. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 100615, Part No. 246066