Avalon Verification IP Suite

User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

UG-01073-3.2

Document last updated for Altera Complete Design Suite version: 13.0
Document publication date: May 2013

Feedback Subscribe

Contents

Chapter 2. Avalon-MM Slave BFM

Chapter 3. Avalon-MM Monitor

Section IV. Avalon-ST BFMs

Chapter 1. Avalon-ST Source BFM

Chapter 2. Avalon-ST Sink BFM

Chapter 3. Avalon-ST Monitor

Chapter 2. Nios II Custom Instruction Slave BFM

Section VII. Tutorials

Chapter 1. Qsys Tutorial

Chapter 2. Using the VHDL BFMs

Additional Information

Section I. Introduction to Avalon Verification IP Suite

The Avalon® Verification IP Suite provides bus functional models (BFMs) to simulate the behavior of and to facilitate the verification of IP that includes the following interfaces and components:

- Avalon Memory-Mapped (Avalon-MM) master and slave interfaces
- Avalon Streaming (Avalon-ST) source and sink interfaces
- Conduit interfaces and Avalon Tri-State conduit (Avalon-TC) interfaces
- Clock source and reset source
- Interrupt source and sink
- Custom instruction master and slave
- External memory

This suite also provides the following monitors to verify the respective Avalon protocols:

- Avalon-MM monitor
- Avalon-ST monitor

Advantages of Using BFMs and Monitors

Using the Altera-provided BFMs and monitors has the following advantages:

- It accelerates the verification process by providing key components of the verification testbench.
- It provides Avalon BFM components that implement the standard Avalon-MM and Avalon-ST protocols, serving as a reference for those protocols.
- For SystemVerilog users, it provides a platform that you can use to implement constraint-driven randomized tests, including traffic scenario drivers, scoreboard and coverage facilities, and assertion checkers.

BFM Implementation

The Avalon Verification IP Suite BFMs (excluding the Clock Source and Reset Source BFMs that are written in VHDL) are implemented in SystemVerilog. The BFM components use primarily Verilog HDL with a few basic SystemVerilog constructs that are supported by ModelSim®-Altera Edition (AE).

The Quartus II software version 13.0 and higher extends VHDL BFM support in Qsys. The VHDL BFMs wrap the SystemVerilog implementation and additional logic to support VDHL.

Refer to Table 1–1 for a summary of BFM language support.

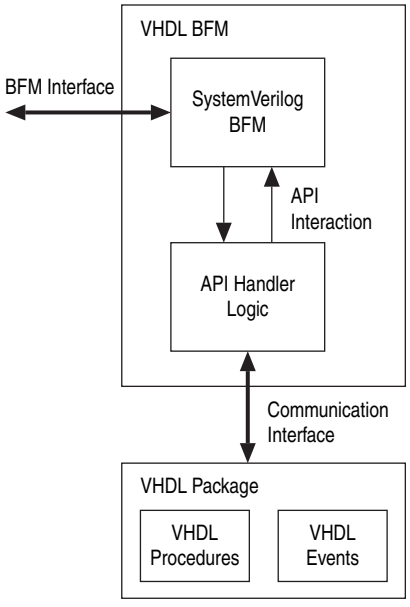
Table 1–1. BFM Language Support

BFM	Verilog HDL Support	VHDL Support
Clock Source and Reset Source	✓	✓
Avalon Interrupt Source and Sink	✓	Version 13.0 and higher
Avalon-MM Master, Slave, and Monitor	✓	Version 13.0 and higher
Avalon-ST Source, Sink, and Monitor	✓	Version 13.0 and higher
Conduit and Tri-State Conduit	✓	—
External Memory	✓	Version 13.0 and higher
Nios II Custom Instruction Master and Slave	✓	Version 13.0 and higher

The VHDL BFM has four parts (see Figure 1–1):

- *SystemVerilog BFM*—Contains the BFM implementation and behavioral model, and the SystemVerilog API. The SystemVerilog code is IEEE encrypted for use in single-language simulators.
- *VHDL package*—Provides the VHDL API used to control the BFM and interface with your test program. The package contains VHDL procedures and events.
- *API handler logic*—SystemVerilog logic block that translates your test program's VHDL API calls to SystemVerilog API calls. The SystemVerilog code is IEEE encrypted for use in single-language simulators.
- *API communication interface*—Bridges the VHDL API to the API handler logic.

Figure 1–1. VHDL Component BFM



For more information on using the VHDL BFMs, refer to Chapter 2, Using the VHDL BFMs.

The monitor components use the SystemVerilog Assertion (SVA) language and are supported only by simulators that support SVA, including: Modelsim-Altera Starter Edition (ASE), Synopsys VCS, and Mentor Graphics® Questa.

Application Programming Interface

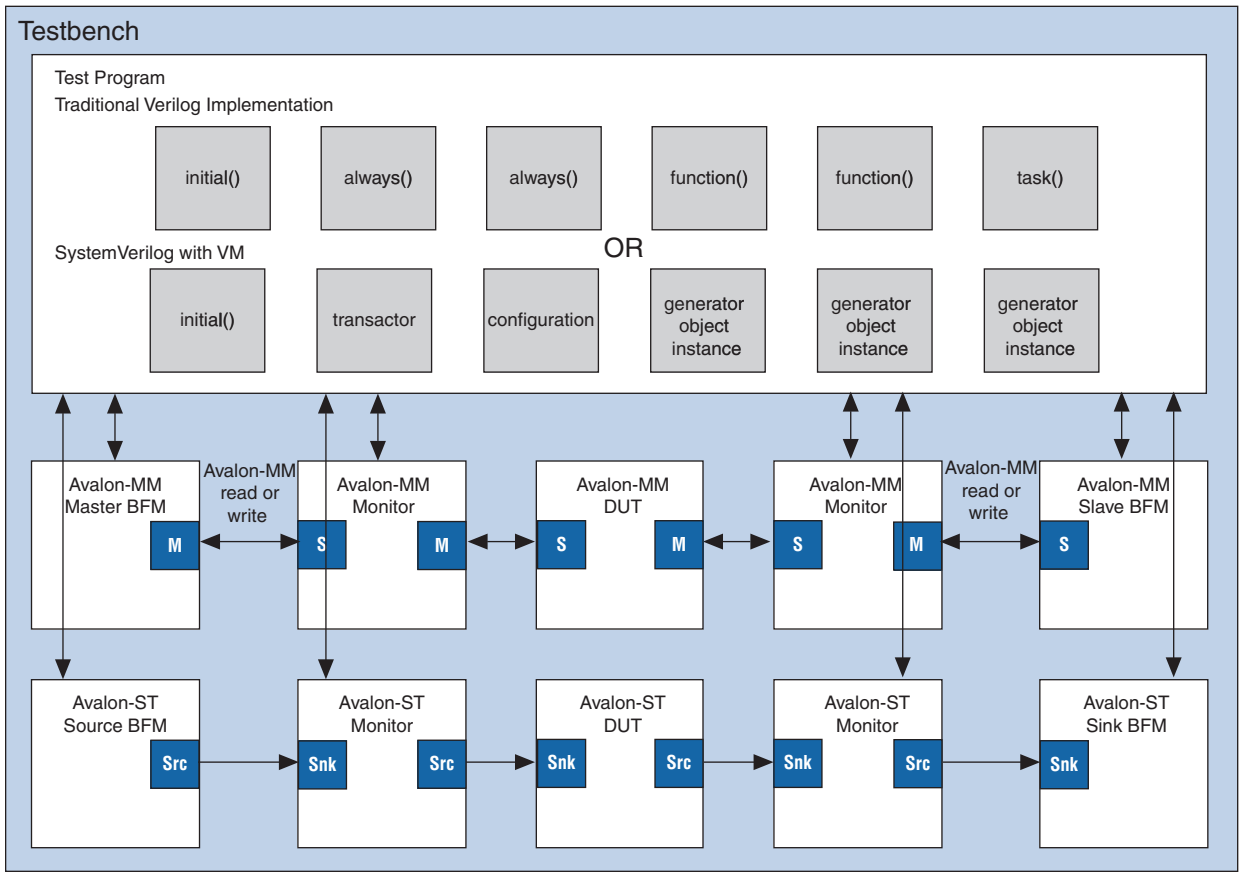
Altera provides you with a set of application programming interface (API) for each Avalon Verification IP Suite BFM that you can use to construct, instantiate, control, and query signals in all BFM components. Your test programs must use only these public access methods and events to communicate with each BFM.

While you can use methods other than the API, Altera does not guarantee continued support or backwards compatibility of custom methods.

Application Example of BFMs

Figure 1–1 shows the top-level blocks in a typical testbench to verify components with Avalon-MM and Avalon-ST interfaces.

Figure 1–1. Avalon Verification IP Suite Testbench



As Figure 1–1 illustrates, it is possible to write a testbench using a traditional Verilog HDL implementation or using SystemVerilog with VMM. For illustration purposes, Figure 1–1 shows an Avalon-MM design under test (DUT) that includes both Avalon-MM master and slave interfaces, and an Avalon-ST DUT that includes both source and sink interfaces, although typical components might include a single Avalon interface.

When verifying a component with Avalon-MM or Avalon-ST interfaces, a monitor is inserted between the master or source BFM and the slave or sink interface of the DUT. A second monitor can be interposed between the slave or sink BFM and the master or source interface of the DUT. The monitors do not have to be placed between a BFM component and another component. They can be inserted anywhere in the system to provide protocol assertion checking and functional coverage reporting.

The test program drives the stimulus to the DUTs and determines whether the DUTs' behavior is correct, by analyzing the responses. The BFMs translate the test program stimuli, creating the signalling for the Avalon-MM and Avalon-ST protocols. The monitors verify Avalon protocol compliance and provide test coverage reports.

In This User Guide

The *Avalon Verification IP Suite User Guide* provides a reference document for each of the BFMs and Avalon Monitors. It includes the following sections:

- Section II, Clock, Reset, and Interrupt BFMs
  This section contains chapters that describe the parameters and API of the Clock Source, Reset Source, Interrupt Source, and the Interrupt Sink BFMs.
- Section III, Avalon-MM BFMs
  This section contains chapters that describe the parameters, functional description, and the API of the Avalon-MM Master and Slave BFMs. This section also includes a tutorial on using the Avalon-MM BFMs.
- Section IV, Avalon-ST BFMs
  This section contains chapters that describe the parameters, functional description, and the API of the Avalon-ST Source and Sink BFMs. This section also includes a tutorial on using the Avalon-ST BFMs.
- Section V, Conduit and External Memory BFMs
  This section contains chapters that describe the blocks, parameters, and API of the conduit, tri-state conduit, and the external memory BFMs.
- Section VI, Nios II Custom Instruction BFMs
  This section contains chapters that describe the blocks, parameters, and API of the Nios II custom instruction master and slave BFMs.
- Section VII, Tutorials
  This section contains chapters that provide a tutorial on how to use the BFMs to verify IP interfaces and components in Qsys and describe how to use the VHDL BFMs.

Section II. Clock, Reset, and Interrupt BFMs

This section provides information about Clock Source, Reset Source, Avalon Interrupt Source, and Avalon Interrupt Sink BFMs. This section includes the following chapters:

- Chapter 1, Clock Source BFM
- Chapter 2, Reset Source BFM
- Chapter 3, Avalon Interrupt Source and Interrupt Sink BFMs

1. Clock Source BFM

The Avalon Verification IP Suite includes a Clock Source BFM that you can use to generate a clock signal for your testbench.

The Clock Source BFM is only supported in Qsys.

Parameters

Table 1–1 lists the parameter settings for the clock signal.

Table 1–1. Clock Source BFM Parameter Settings

Option	Default Value	Legal Values	Description
Clock rate	10	—	Specifies the clock rate in MHz.

Application Program Interface

This section describes the API for the Clock Source BFM.

clock_start()

Prototype: `clock_start()`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Turns on the clock.
Language support: Verilog HDL

clock_stop()

Prototype: `clock_stop()`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Turns off the clock.
Language support: Verilog HDL

get_run_state()

Prototype: `get_run_state()`

Arguments: Verilog HDL: None

VHDL: N.A.

Returns: `bit`

Description: Returns the state of the clock source; 1=running, 0=stop.

Language support: Verilog HDL

get_version()

Prototype: `string get_version()`

Arguments: Verilog HDL: None

VHDL: N.A.

Returns: `string`

Description: Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".

Language support: Verilog HDL

2. Reset Source BFM

The Avalon Verification IP Suite includes a Reset Source BFM that you can use to generate a reset signal in your testbench.

☞ The Reset Source BFM is only supported in Qsys.

Parameters

Table 2–1 lists the parameter settings for the reset signal.

Table 2–1. Reset Source BFM Parameter Settings

Option	Default Value	Legal Values	Description
Assert reset high	**On**	**On/Off**	Specifies the polarity of the reset signal. Turn on this option to set the reset signal active high.
Cycles of initial reset	**0**	—	Specifies the number of cycles that the reset signal is asserted at the initial stage of the simulation.

Application Program Interface

This section describes the API for the Reset Source BFM.

reset_assert

Prototype: `reset_assert`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`.
Description: Asserts the reset signal.
Language support: Verilog HDL

reset_deassert

Prototype: `reset_deassert`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`.
Description: Deasserts the reset signal.
Language support: Verilog HDL

get_version()

Prototype: `string get_version()`

Arguments: Verilog HDL: None

VHDL: N.A.

Returns: `String`.

Description: Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".

Language support: Verilog HDL

3. Avalon Interrupt Source and Interrupt Sink BFMs

The Avalon Verification IP Suite includes Avalon Interrupt Source and Avalon Interrupt Sink BFMs for you to generate interrupt signals in your testbench.

The Avalon Interrupt Source and Sink BFMs are only supported in Qsys.

Parameters

Table 3–1 lists the parameter settings for the interrupt signals.

Table 3–1. Avalon Interrupt Source and Avalon Interrupt Sink BFMs Parameter Settings

Option	Default Value	Legal Values	Description
Interrupt Source			
Assert IRQ high	**On**	**On/Off**	Specifies the polarity of the interrupt source signal. Turn on this option to change the name of the interrupt source signal port from `irq` to `irq_n`.
IRQ width	**1**	**1–32**	Specifies the width of the interrupt source signal.
Asynchronous IRQ	**Off**	**On/Off**	Specifies whether the interrupt signal is asserted or deasserted immediately after an API call or one clock cycle after an API call. Turn on this option to allow changes to the interrupt signal immediately after an API call or turn off this option to allow changes to the interrupt signal on the next clock edge.
VHDL BFM ID	**0**	**0–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.
Interrupt Sink			
Assert IRQ high	**On**	**On/Off**	Specifies the polarity of the interrupt sink signal. Turn on this option to change the name of the interrupt sink signal port from `irq` to `irq_n`.
IRQ width	**1**	**1–32**	Specifies the width of the interrupt sink signal.

Application Program Interface

This section describes the API for the Avalon Interrupt Source and Avalon Interrupt Sink BFMs.

clear_irq()

Prototype:	`int clear_irq()`
Arguments:	Verilog HDL: `int interrupt_bit`
	VHDL: `int interrupt_bit`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Asserts the interrupt signal and sets the interrupt signal to 0, regardless of the value you set for **Assert IRQ high** in the parameter editor.
Language support:	Verilog HDL, VHDL

get_irq()

Prototype:	`get_irq()`
Arguments:	Verilog HDL: None
	VHDL: `bfm_id`, `req_if`
Returns:	`logic[AV_IRQ_W-1:0]`
Description:	Returns the current value of the register holding the latched interrupt signal.
Language support:	Verilog HDL, VHDL

get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`String`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".
Language support:	Verilog HDL

set_irq()

Prototype:	`set_irq()`
Arguments:	Verilog HDL: `int interrupt_bit`
	VHDL: `int interrupt_bit`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Asserts the interrupt signal and sets the interrupt signal to 1, regardless of the value you set for **Assert IRQ high** in the parameter editor.
Language support:	Verilog HDL, VHDL

Section III. Avalon-MM BFMs

This section provides information about Avalon-MM BFMs. This section includes the following chapters:

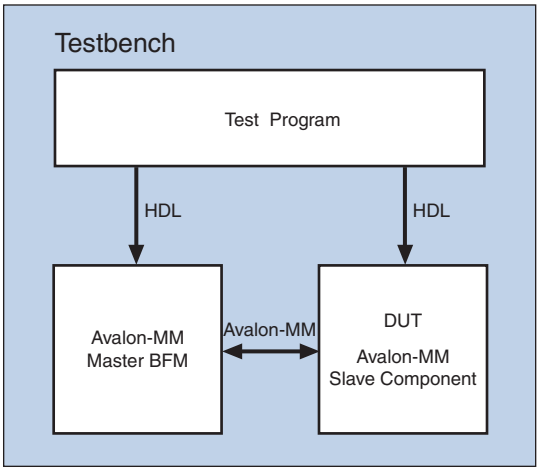
- Chapter 1, Avalon-MM Master BFM
- Chapter 2, Avalon-MM Slave BFM
- Chapter 3, Avalon-MM Monitor

1. Avalon-MM Master BFM

The Avalon-MM Master BFM implements the Avalon-MM interface protocol, including: read, write, burst read, and burst write. Figure 1–1 shows the top-level modules for a typical testbench that uses the Avalon-MM BFM to verify an Avalon-MM slave component. In addition to the Altera-provided Avalon-MM Master BFM component, the typical testbench includes a test program and the DUT that includes an Avalon-MM slave interface. The Altera-provided Avalon-MM BFM highlights any misinterpretation of the protocol implemented by the DUT that might be missed in a testbench designed by a single engineer.

The BFMs allow illegal transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitor components verify protocol compliance.

Figure 1–1. Top-Level Module to Verify an Avalon-MM Slave Device



For more information about the Avalon-MM specification supported in Qsys, refer to the *Avalon Interface Specifications (version 2.1)*.

Functional Description

This section provides a functional description of the Avalon-MM Master BFM. It includes the following topics:

Timing

The timing diagram in Figure 1–2 illustrates the sequence of events for an Avalon-MM Master BFM driving interleaved writes and reads when the `readdatavalid` signal is present. This diagram serves as a reference for the following discussion of API and events.

Figure 1–2. Avalon-MM Master Driving Interleaved Write and Read Transactions

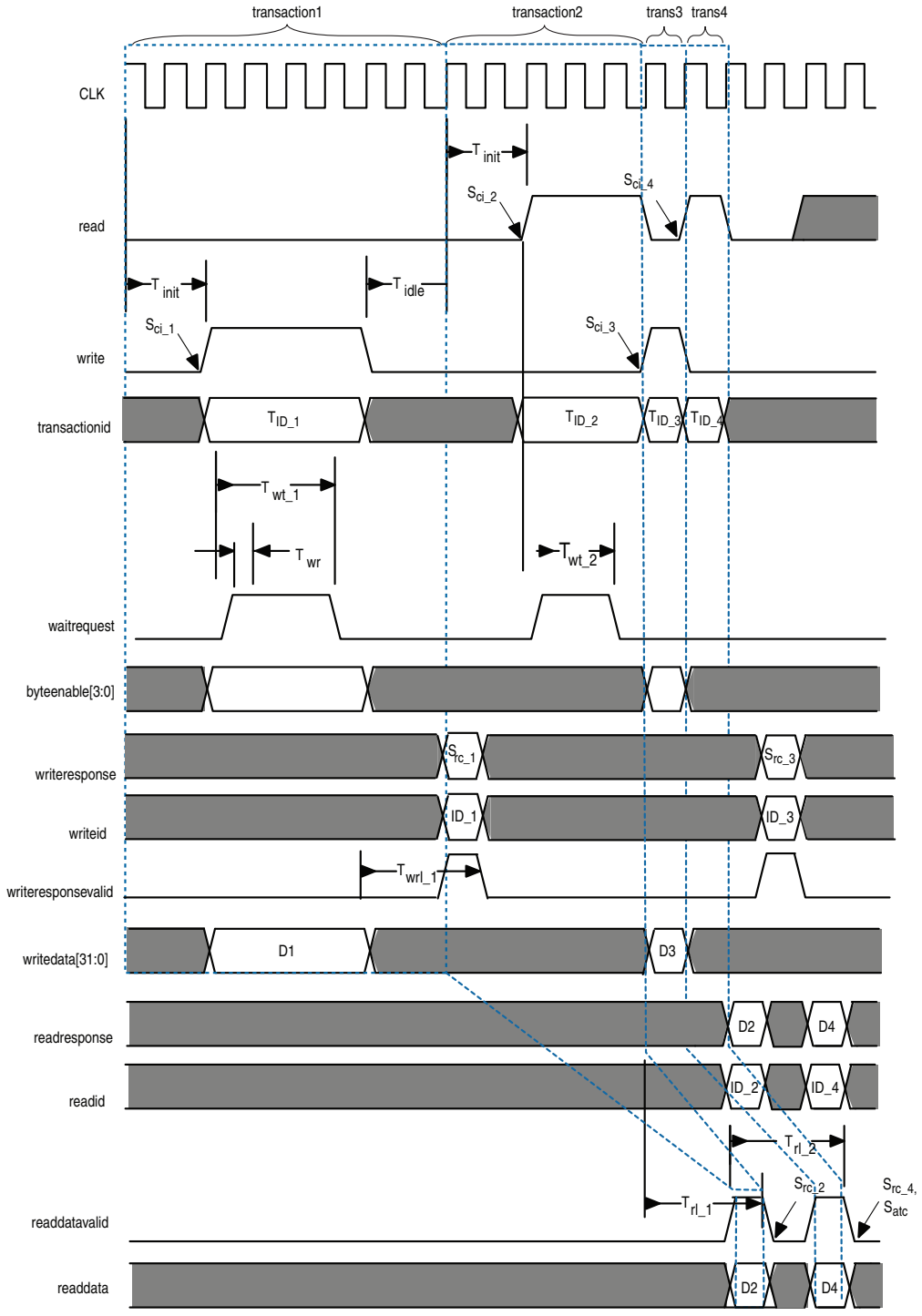


Table 1–1 lists the annotations used in Figure 1–2.

Table 1–1. Key to the Figure 1–2 Annotations

Symbol	Description
T_{init}	The initial command latency, which is two cycles for transactions 1 and 2. This time is set by the API command `set_command_init_latency`.
T_{wt_1}	The response wait time, which is three cycles. This time is determined by the number of cycles that the `waitrequest` signal is asserted by the slave.The program gets this value using the `get_response_wait_time` command.
T_{wr}	`waitrequest` is always sampled #1 after the falling edge of `clk`.
T_{idle}	The idle time after each transaction. This time is set by the command `set_command_idle`.
T_{rl_1}	The response latency for the first read, which is three cycles. This is the time between when the read command is accepted, and the read response is provided by the slave. The program gets this time using the `get_response_latency` command. Note if the Avalon-MM slave component has defined a fixed read latency by defining the `readLatency` interface property, the `readdatavalid` signal is not used. For more information refer to the *Avalon Interface Specifications*.
T_{rl_2}	The response latency for the second read, which is three cycles. The program gets this time using the `get_response_latency` command.
T_{wrl_1}	The write response latency for the first write, which is three cycles. This is the time between when the write command is accepted, and the write response is provided by the slave. The program gets this time using the `get_response_latency` command.
S_{ci_1}–S_{ci_4}	Signals when write or read commands are presented on the interface. The event name is `signal_command_issued`.
S_{rc_1},S_{rc_3}	Signals write responses. The event name is `signal_response_complete`.
S_{rc_2},S_{rc_4}	Signals read responses. The event name is `signal_response_complete`.
S_{atc}	Signals the end of the test. The event name is `signal_all_transactions_complete`
T_{ID_1}–T_{ID_4}	Reference number to identify each read or write transaction.
ID_1, ID_3	Reference number to identify each write transaction.
ID_2, ID_4	Reference number to identify each read transaction.

The timing diagram in Figure 1–3 shows the sequence of events for an Avalon-MM Master BFM driving a write followed by a read when the `readdatavalid` signal is not present.

Figure 1–3. Avalon-MM Master Driving Write and Read Transactions with No readdatavalid Signal

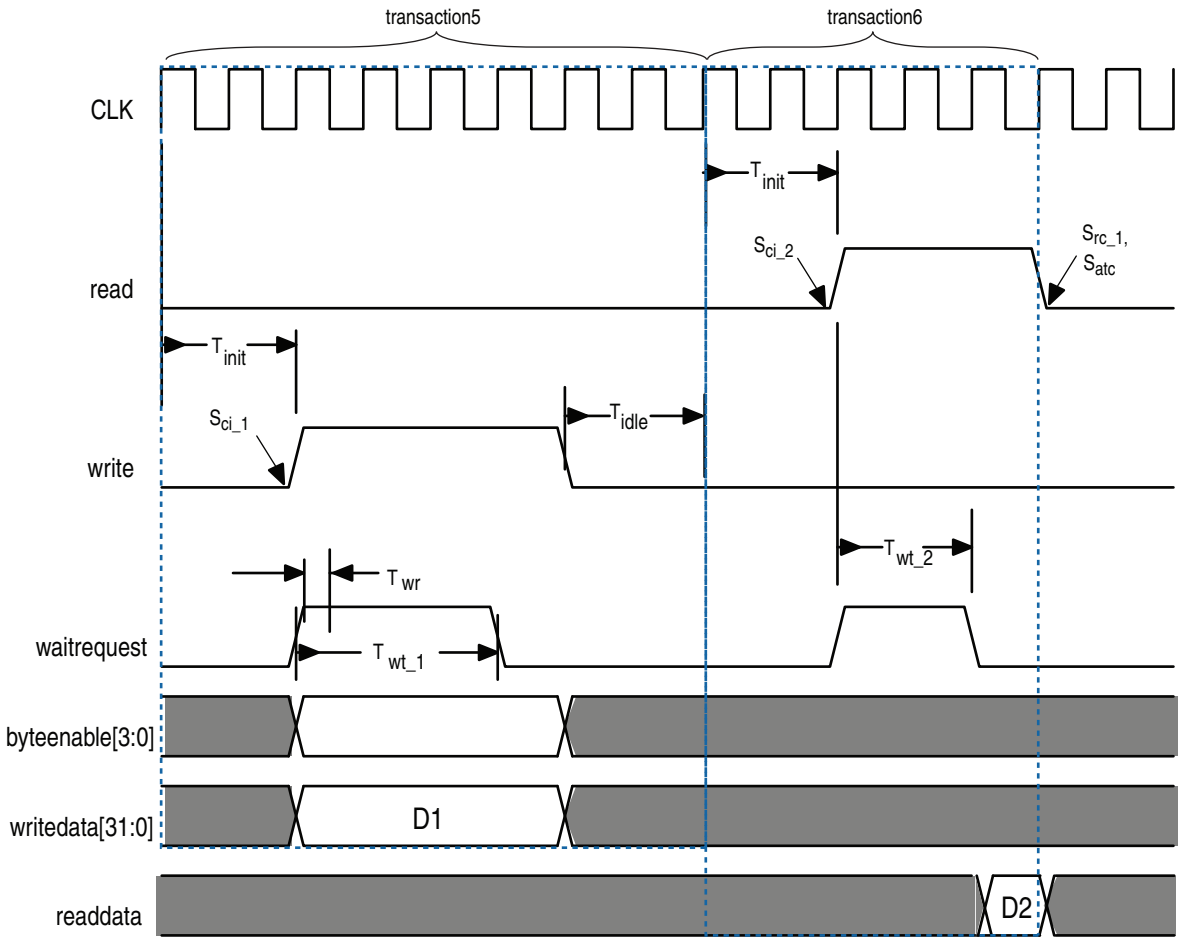


Table 1–2 lists the annotations used in Figure 1–3.

Table 1–2. Key to the Figure 1–3 Annotations (Part 1 of 2)

Symbol	Description
T_{init}	The initial command latency, which is two cycles for transactions 1 and 2. This time is set by the API command `set_command_init_latency`.
T_{wt_1}	The response wait time, which is three cycles. This time is determined by the number of cycles that the `waitrequest` signal is asserted by the slave.The program gets this value using the `get_response_wait_time` command.
T_{wt_2}	The response wait time for the first read, which is two cycles. This time is determined by the number of cycles that the `waitrequest` signal is asserted by the slave.The program gets this value using the `get_response_wait_time` command.

Table 1–2. Key to the Figure 1–3 Annotations (Part 2 of 2)

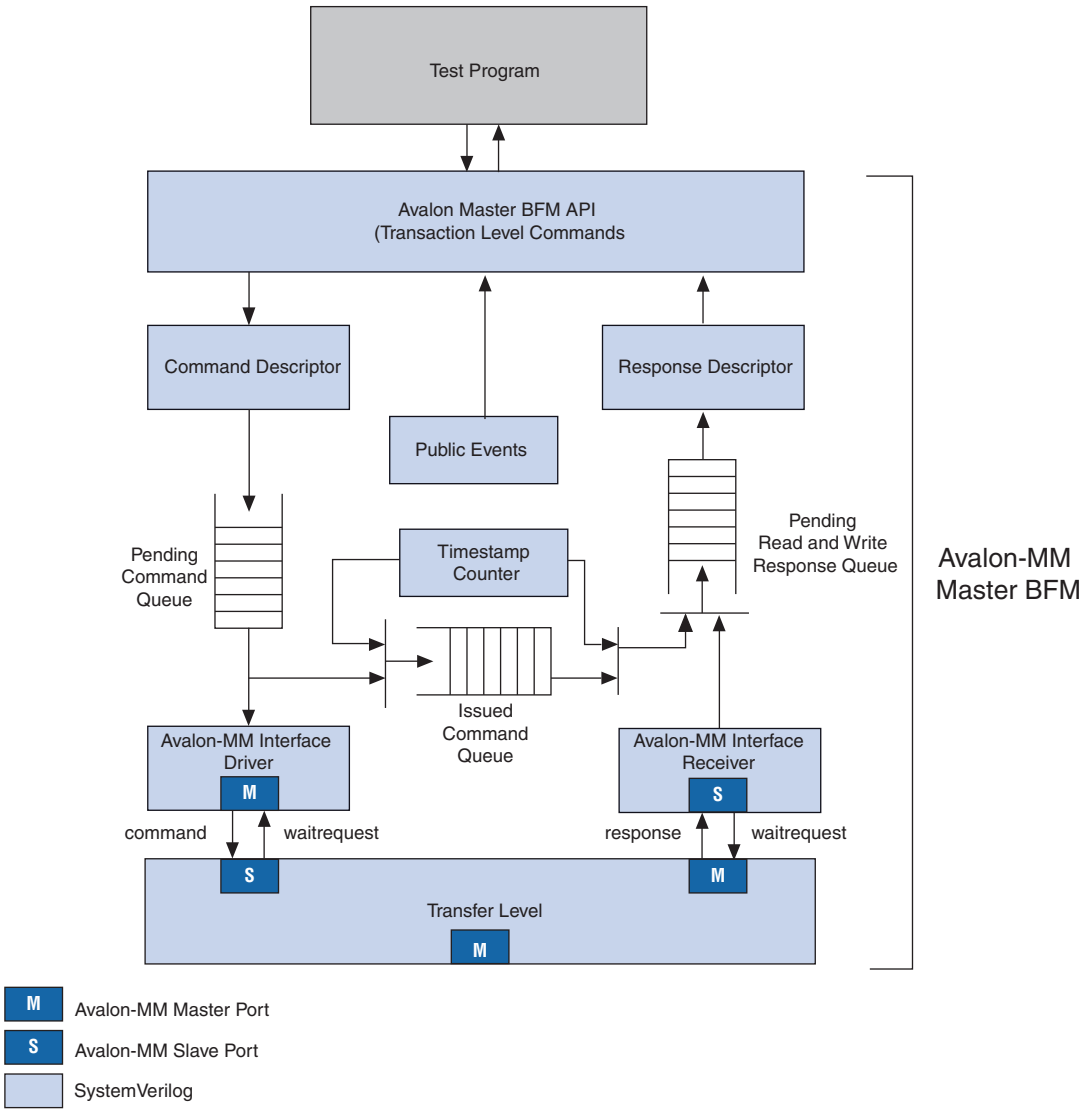
Symbol	Description
T_{wr}	`waitrequest` is always sampled #1 after the falling edge of `clk`.
T_{idle}	The idle time after a transaction. This time is set by the command `set_command_idle`.
S_{ci_1}–S_{ci_2}	Signals when write and read commands are presented on the interface. The event name is `signal_command_issued`.
S_{rc_1}	Signals the first read response. The event name is `signal_response_complete`.
S_{atc}	Signals the end of the test. The event name is `signal_all_transactions_complete`.

Block Diagram

Figure 1–4 shows a block diagram of the Avalon-MM Master BFM. As this figure illustrates, the BFM includes the following major blocks:

- *Avalon-MM Master API*—Provides methods to create Avalon-MM transactions and query the state of all queues.
- *Command Descriptor*—Accumulates the fields of an Avalon-MM command transaction using the `set_command` API calls and inserts completed commands onto the pending command queue.
- *Avalon-MM Interface Driver*—Issues transfers to the system interconnect fabric and holds each transfer until `waitrequest` is deasserted. For burst transfers, there is a separate transfer for each word of the burst. The system interconnect fabric can assert `waitrequest` for each word of the burst, as necessary.
- *Timestamp Counter*—Records a timestamp with commands for use in timing calculations. The driver and monitor both use the timestamp counter for timing calculations.
- *Avalon-MM Interface Monitor*—Monitors the system interconnect fabric and records responses for read transfers in the response queue.
- *Response Descriptor*—Collects information about completed transactions using the `get_response_<rolename>` API calls. The testbench uses this information for further analysis.
- *Public Events*—Provides status response that arrives together with the data. The public event signals indicate the status of the Master's request such as successful completion, timeout, or error.

Figure 1–4. Block Diagram of the Avalon-MM Master BFM



Parameters

The Avalon-MM BFM supports the full range of signals defined for the Avalon-MM master interface. You can customize the Avalon-MM master interface using the parameters described in Table 1–3.

Table 1–3. Parameters for the Avalon-MM Master BFM (Part 1 of 2)

Parameter	Default Value	Legal Values	Description
Port Widths			
Address width	**32**	—	Address width in bits.
Symbol width	**8**	—	Data symbol width in bits. The symbol width should be 8 for byte-oriented interfaces.
Read Response width	**8**	—	Read response signal width in bits.
Write Response width	**8**	—	Write response signal width in bits.
Parameters			
Number of symbols	**4**	—	Number of symbols per word.
Burstcount width	**3**	—	The width of the burst count in bits.
Port Enables			
Use the read signal	**On**	**On/Off**	When **On**, the interface includes a `read` pin.
Use the write signal	**On**	**On/Off**	When **On**, the interface includes a `write` pin.
Use the address signal	**On**	**On/Off**	When **On**, the interface includes `address` pins.
Use the byteenable signal	**On**	**On/Off**	When **On**, the interface includes `byteenable` pins.
Use the burstcount signal	**On**	**On/Off**	When **On**, the interface includes `burstcount` pins.
Use the readdata signal	**On**	**On/Off**	When **On**, the interface includes a `readdata` pin.
Use the readdatavalid signal	**On**	**On/Off**	When **On**, the interface includes a `readdatavalid` pin.
Use the writedata signal	**On**	**On/Off**	When **On**, the interface includes a `writedata` pin.
Use the begintransfer signal	**Off**	**On/Off**	When **On**, the interface includes `writedata` pins
Use the beginbursttransfer signal	**Off**	**On/Off**	When **On**, the interface includes a `beginbursttransfer` pins.
Use the arbiterlock signal	**Off**	**On/Off**	When **On**, the interface includes an `arbiterlock` pin.
Use the lock signal	**Off**	**On/Off**	When **On**, the interface includes a `lock` pin.
Use the debugaccess signal	**Off**	**On/Off**	When **On**, the interface includes a `debugaccess` pin.
Use the waitrequest signal	**On**	**On/Off**	When **On**, the interface includes a `waitrequest` pin.
Use the transactionid signal	**Off**	**On/Off**	When **On**, the interface includes a `transactionid` pin.
Use the write response signals	**Off**	**On/Off**	When **On**, the interface includes a `writeresponse` pin.
Use the read response signals	**Off**	**On/Off**	When **On**, the interface includes a `readresponse` pin.
Use the clken signals	**Off**	**On/Off**	When **On**, the interface includes a `clken` pin.
Port Polarity			
Assert reset high	**On**	**On/Off**	When **On**, `reset` is asserted high.
Assert waitrequest high	**On**	**On/Off**	When **On**, `waitrequest` is asserted high.

Table 1–3. Parameters for the Avalon-MM Master BFM (Part 2 of 2)

Parameter	Default Value	Legal Values	Description
Assert read high	**On**	**On/Off**	When **On**, `read` is asserted high.
Assert write high	**On**	**On/Off**	When **On**, `write` is asserted high.
Assert byteenable high	**On**	**On/Off**	When **On**, `byteenable` is asserted high.
Assert readdatavalid high	**On**	**On/Off**	When **On**, `readdatavalid` is asserted high.
Assert arbiterlock high	**On**	**On/Off**	When **On**, `arbiterlock` is asserted high.
Assert lock high	**On**	**On/Off**	When **On**, `lock` is asserted high.
Burst Attributes			
Linewrap burst	**On**	**On/Off**	When **On**, the address for bursts wraps instead of an incrementing. With a wrapping burst, when the address reaches a burst boundary, it wraps back to the previous burst boundary such that only the low order bits need to be used for addressing.
Burst on burst boundaries only	**On**	**On/Off**	When **On**, memory bursts are aligned to the address size.
Miscellaneous			
Maximum pending reads	**1**	—	The maximum number of pending reads that can be queued by the slave.
Fixed read latency (cycles)	**1**	—	Sets the read latency for fixed-latency slaves. Not used on interfaces that include the `readdatavalid` signal.
VHDL BFM ID	**0**	**0–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.
Timing			
Fixed read wait time (cycles)	**1**	—	For master interfaces that do not use the `waitrequest` signal, the read wait time indicates the number of cycles before the master responds to a read. The timing is as if the master asserted `waitrequest` for this number of cycles.
Fixed write wait time (cycles)	**0**	—	For master interfaces that do not use the `waitrequest` signal, the write wait time indicates the number of cycles before the master accepts a write.
Registered waitrequest	**Off**	**On/Off**	Specifies whether to turn on the register stage.
Registered Incoming Signals	**Off**	**On/Off**	Specifies whether to register incoming signals.
Interface Address Type			
Set master interface address type to symbols or words	**WORDS**	**WORDS/ SYMBOLS**	Sets slave interface address type to symbols or words.

Application Program Interface

This section describes the API for the Avalon-MM Master BFM.

all_transactions_complete()

Prototype:	`bit all_transactions_complete()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit`.
Description:	Queries the BFM component to determine whether all issued commands have been completed. A return value of 1 means that there are no more transactions in the transaction queue or in progress.
Language support:	Verilog HDL, VHDL

event_all_transactions_complete()

Prototype:	`event_all_transactions_complete()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that all commands have completed.
Language support:	VHDL

event_command_issued()

Prototype:	`event_command_issued()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench a command was driven to the bus.
Language support:	VHDL

event_max_command_queue_size()

Prototype:	`event_max_command_queue_size()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the command queue size reached its maximum limit.
Language support:	VHDL

event_min_command_queue_size()

Prototype: `event_min_command_queue_size()`
Arguments: Verilog HDL: N.A.
VHDL: `bfm_id`, `req_if`
Returns: `void`
Description: Notifies the testbench that the command queue size reached its minimum limit.
Language support: VHDL

event_read_response_complete()

Prototype: `event_read_response_complete()`
Arguments: Verilog HDL: N.A.
VHDL: `bfm_id`, `req_if`
Returns: `void`
Description: Notifies the testbench that a read response was received.
Language support: VHDL

event_response_complete()

Prototype: `event_response_complete()`
Arguments: Verilog HDL: N.A.
VHDL: `bfm_id`, `req_if`
Returns: `void`
Description: Notifies the testbench that a read/write response was received.
Language support: VHDL

event_write_response_complete()

Prototype: `event_write_response_complete()`
Arguments: Verilog HDL: N.A.
VHDL: `bfm_id`, `req_if`
Returns: `void`
Description: Notifies the testbench that a write response was received.
Language support: VHDL

get_command_issued_queue_size()

Prototype:	`int get_command_issued_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the issued command queue to determine the number of commands that have been driven to the system interconnect fabric, but not completed.
Language support:	Verilog HDL, VHDL

get_command_pending_queue_size()

Prototype:	`int get_command_pending_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the command queue to determine number of pending commands waiting to be driven out as Avalon requests.
Language support:	Verilog HDL, VHDL

get_read_response_queue_size()

Prototype:	`int get_read_response_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the read response queue to determine number of response descriptors currently stored in the BFM. This is the number of responses the test program can immediately remove from the response queue for further processing.
Language support:	Verilog HDL, VHDL

get_response_address()

Prototype:	`bit [AV_ADDRESS_W-1:0] get_response_address()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit`
Description:	Returns the transaction address in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_byte_enable()

Prototype: `bit [AV_NUMSYMBOLS-1:0] get_response_byte_enable(int index)`

Arguments: Verilog HDL: `index`

VHDL: `index`, `bfm_id`, `req_if`

Returns: `bit`

Description: Returns the value of the byte enables in the response descriptor that has been removed from the response queue. Each cycle of a burst response is addressed individually by the specified index.

Language support: Verilog HDL, VHDL

get_response_burst_size()

Prototype: `bit [AV_BURSTCOUNT_W-1:0]get_response_burst_size()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `bit`

Description: Returns the size of the response transaction burst count in the response descriptor that has been removed from the response queue.

Language support: Verilog HDL, VHDL

get_response_data()

Prototype: `bit [AV_DATA_W-1:0] get_response_data(int index)`

Arguments: Verilog HDL: `index`

VHDL: `index`, `bfm_id`, `req_if`

Returns: `bit`

Description: Returns the transaction read data in the response descriptor that has been removed from the response queue. Each cycle in a burst response is addressed individually by the specified index. In the case of read responses, the data is the data captured on the `avm_readdata` interface pin. In the case of write responses, the data on the driven `avm_writedata` pin is captured and reflected here.

Language support: Verilog HDL, VHDL

get_response_latency()

Prototype: `int get_response_latency(int index)`

Arguments: Verilog HDL: `index`

VHDL: `index`, `bfm_id`, `req_if`

Returns: `bit`

Description: Returns the transaction read latency in the response descriptor that has been removed from the response queue. Each cycle in a burst read has its own latency entry.

Language support: Verilog HDL, VHDL

get_response_queue_size()

Prototype:	`int get_response_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the response queue to determine number of response descriptors currently stored in the BFM. This is the number of responses the test program can immediately remove from the response queue for further processing.
Language support:	Verilog HDL, VHDL

get_response_read_id()

Prototype:	`[AV_TRANSACTIONID_W-1:0] get_response_read_id()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`AvalonTransactionId_t`
Description:	Returns the read id of the transaction in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_read_response()

Prototype:	`bit[2**(AV_BURSTCOUNT_W-1) - 1:0] [AV_READRESPONSE_W-1:0] get_response_read_response(int index)`
Arguments:	Verilog HDL: `int index` VHDL: `int index`, `bfm_id`, `req_if`
Returns:	`AvalonReadResponse_t`
Description:	Returns the transaction read status in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_request()

Prototype:	`enum int[REQ_READ = 0, REQ_WRITE = 1, RED_IDLE = 2] get_response_request()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`Request_t`
Description:	Returns the transaction command type in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_wait_time()

Prototype:	`int get_response_wait_time(int index)`
Arguments:	Verilog HDL: `index` VHDL: `index`, `bfm_id`, `req_if`
Returns:	`int`
Description:	Returns the wait latency for transaction in the response descriptor that has been removed from the response queue. Each cycle in a burst has its own wait latency entry.
Language support:	Verilog HDL, VHDL

get_response_write_id()

Prototype:	`bit [AV_TRANSACTIONID_W-1:0] get_response_write_id()`
Arguments:	Verilog HDL: None VHDL: `index`, `bfm_id`, `req_if`
Returns:	`AvalonTransactionId_t`
Description:	Returns the write id of the transaction in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_write_response()

Prototype:	`bit [2**(AV_BURSTCOUNT_W-1)-1:0] [AV_WRITERESPONSE_W-1:0] get_response_write_response(int index)`
Arguments:	Verilog HDL: `int index` VHDL: `int index`, `bfm_id`, `req_if`
Returns:	`AvalonWriteResponse_t`
Description:	Returns the transaction write status in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_write_response_queue_size()

Prototype:	`int get_write_response_queue_size()`
Arguments:	Verilog HDL: None VHDL: `index`, `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the write response queue to determine number of response descriptors currently stored in the BFM. This is the number of responses the test program can immediately pop off the response queue for further processing.
Language support:	Verilog HDL, VHDL

get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`String`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".
Language support:	Verilog HDL

init()

Prototype:	`init`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Initializes the Avalon-MM master interface.
Language support:	Verilog HDL, VHDL

pop_response()

Prototype:	`void pop_response()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Removes the oldest response descriptor from the response queue, such that transaction information is available using the `get_response_`<*rolename*> commands.
Language support:	Verilog HDL, VHDL

push_command()

Prototype:	`void push_command()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Inserts the fully populated transaction descriptor onto the pending transaction command queue.
Language support:	Verilog HDL, VHDL

set_clken()

Prototype:	`void set_clken(bit state)`
Arguments:	Verilog HDL: `bit state`
	VHDL: `bit state`, `bfm_id`, `req_ig`
Returns:	`void`
Description:	Sets the assertion and deassertion of the clock enable signal.
Language support:	Verilog HDL, VHDL

set_command_address()

Prototype:	`void set_command_address(bit[AV_ADDRESS_W-1:0]addr)`
Arguments:	Verilog HDL: `addr`
	VHDL: `addr`, `bfm_id`, `req_ig`
Returns:	`void`
Description:	Sets the transaction address in the command descriptor.
Language support:	Verilog HDL, VHDL

set_command_arbiterlock()

Prototype:	`void set_command_arbiterlock (bit state)`
Arguments:	Verilog HDL: `bit state`
	VHDL: `bit state`, `bfm_id`, `req_ig`
Returns:	`void`
Description:	Controls the assertion or deassertion of the arbiterlock interface signal. The arbiterlock control is on the transaction boundaries and is not used when the Avalon-MM Master BFM is operating in burst mode.
Language support:	Verilog HDL, VHDL

set_command_byte_enable()

Prototype:	`void set_command_byte_enable(bit[AV_NUMSYMBOLS-1:0] byte_enable, int index)`
Arguments:	Verilog HDL: `byte_enable`, `index`
	VHDL: `byte_enable`, `index`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the transaction byte enable field for the cycle of the burst command descriptor indicated by index. This field applies to both read and write operations.
Language support:	Verilog HDL, VHDL

set_command_burst_count()

Prototype:	`void set_command_burst_count(bit[AV_BURSTCOUNT_W-1:0] burst_count)`
Arguments:	Verilog HDL: `burst_count` VHDL: `burst_count`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the value driven on the Avalon interface `burstcount` pin. Generates a warning message if the specified `burst_count` is out of range. Not available if the `USE_BURSTCOUNT` parameter is false.
Language support:	Verilog HDL, VHDL

set_command_burst_size()

Prototype:	`void set_command_burst_size (bit[AV_BURSTCOUNT_W-1:0] burst_size)`
Arguments:	Verilog HDL: `burst_size` VHDL: `burst_size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the transaction burst count in the command descriptor to determine the number of words driven on the write burst command. The value might be different from the value specified in `set_command_burst_count` to generate illegal traffic for testing. Generates a warning if the value is different.
Language support:	Verilog HDL, VHDL

set_command_data()

Prototype:	`void set_command_data(bit[AV_DATA_W-1:0] data, int index)`
Arguments:	Verilog HDL: `data`, `index` VHDL: `data`, `index`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the transaction write data in the command descriptor. For burst transactions, the command descriptor holds an array of data, with each element individually set by this method.
Language support:	Verilog HDL, VHDL

set_command_debugaccess()

Prototype:	`void set_command_debugaccess`
Arguments:	Verilog HDL: `bit state` VHDL: `bit state`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Controls the assertion or deassertion of the debugaccess interface signal. The debugaccess control is on transaction boundaries.
Language support:	Verilog HDL, VHDL

set_command_idle()

Prototype: `void set_command_idle(int idle, int index)`

Arguments: Verilog HDL: `int idle`, `int index`

VHDL: `int idle`, `int index`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets idle cycles at the end of each transaction cycle. In the case of read commands, idle cycles are inserted at the end of the command cycle. In the case of burst write commands, idle cycles are inserted at the end of each write data cycle within the burst.

Language support: Verilog HDL, VHDL

set_command_init_latency()

Prototype: `void set_command_init_latency(int cycles)`

Arguments: Verilog HDL: `cycles`

VHDL: `cycles`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the number of cycles to postpone the start of a command.

Language support: Verilog HDL, VHDL

set_command_lock()

Prototype: `void set_command_lock (bit state)`

Arguments: Verilog HDL: `bit state`

VHDL: `bit state`, `bfm_id`, `req_if`

Returns: `void`

Description: Controls the assertion or deassertion of the lock interface signal. The lock control is on the transaction boundaries and is not used when the Avalon-MM Master BFM is operating in burst mode.

Language support: Verilog HDL, VHDL

set_command_request()

Prototype: `void set_command_request(Request_t request)`

Arguments: Verilog HDL: `Request_t request`

VHDL: `Request_t request`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the transaction type to read or write in the command descriptor. The enumeration type defines `REQ_READ` = 0 and `REQ_WRITE` = 1.

Language support: Verilog HDL, VHDL

set_command_timeout()

Prototype:	`void set_command_timeout(int cycles)`
Arguments:	Verilog HDL: `int cycles`
	VHDL: `int cycles`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the number of elapsed cycles between waiting for a `waitrequest` and when time out is asserted. Disables time-out by setting the value to 0.
Language support:	Verilog HDL, VHDL

set_command_transaction_id()

Prototype:	`void set_command_transaction_id(bit[AV_TRANSACTIONID_W-1:0] id)`
Arguments:	`AvalonTransactionId_t id.`
	Verilog HDL: `AvalonTransactionId_t id`
	VHDL: `AvalonTransactionId_t id`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the transaction id number in the command descriptor.
Language support:	Verilog HDL, VHDL

set_command_write_response_request()

Prototype:	`void set_command_write_response_request (logic request)`
Arguments:	Verilog HDL: `logic request`
	VHDL: `logic request`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the flag that enables or disables the write response requests in the command descriptor.
Language support:	Verilog HDL, VHDL

set_max_command_queue_size()

Prototype:	`void set_max_command_queue_size(int size)`
Arguments:	Verilog HDL: `int size`
	VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the pending command queue size maximum threshold.
Language support:	Verilog HDL, VHDL

set_min_command_queue_size()

Prototype:	`void set_min_command_queue_size(int size)`
Arguments:	Verilog HDL: `int size`
	VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the pending command queue size minimum threshold.
Language support:	Verilog HDL, VHDL

set_response_timeout()

Prototype:	`void set_response_timeout(int cycles)`
Arguments:	Verilog HDL: `int cycles`
	VHDL: `int cycles`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the number of cycles that may elapse before response time out. Disable time-out by setting the value to 0.
Language support:	Verilog HDL, VHDL

signal_all_transactions_complete

Prototype:	`signal_all_transactions_complete`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Signals that all queued transactions have completed.
Language support:	Verilog HDL

signal_command_issued

Prototype:	`signal_command_issued`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Signals that the currently pending command has been driven to the interface.
Language support:	Verilog HDL

signal_fatal_error

Prototype:	`signal_fatal_error`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a fatal error has occured in this module.
Language support:	Verilog HDL

signal_max_command_queue_size

Prototype:	`signal_max_command_queue_size`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that the maximum pending transaction queue size threshold has been exceeded.
Language support:	Verilog HDL

signal_min_command_queue_size

Prototype:	`signal_min_command_queue_size`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that the pending transaction queue size is below the minimum threshold.
Language support:	Verilog HDL

signal_read_response_complete

Prototype:	`signal_read_response_complete`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that the read response has been received and inserted into the response queue.
Language support:	Verilog HDL

signal_response_complete

Prototype:	`signal_response_complete`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Triggers when either `signal_read_response_complete` or `signal_write_response_complete` is triggered.
Language support:	Verilog HDL

signal_write_response_complete

Prototype:	`signal_write_response_complete`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that the write response has been received and inserted into the response queue.
Language support:	Verilog HDL

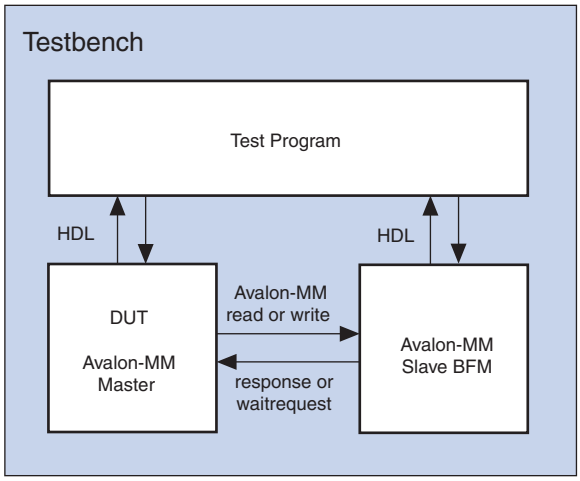
2. Avalon-MM Slave BFM

The Avalon-MM Slave BFM implements the slave side of the Avalon-MM interface protocol. This is a standard memory-mapped protocol including reads and writes typical of simple peripherals and the reads, writes, burst reads, and burst writes for typical memory devices. This BFM also includes a procedural interface to monitor incoming commands, pass these to the test program, accept response transactions from the test program, and drive responses.

Figure 2–1 shows the top-level modules for a testbench that uses the Avalon-MM Slave BFM to verify an Avalon-MM Master device. In addition to the Altera-provided Avalon-MM Slave BFM, the example testbench shown in Figure 2–1 includes a test program and the DUT. The test program, written in HDL, programs the Avalon-MM master to issue Avalon-MM transactions, programs the Avalon-MM Slave BFM to respond, and analyzes the results.

The BFMs allow illegal response transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitor components verify protocol compliance.

Figure 2–1. Top-Level Module to Verify an Avalon-MM Master



For more information about the Avalon-MM specification supported in Qsys, refer to the *Avalon Interface Specifications (version 2.1)*.

Functional Description

This section provides a functional description of the Avalon-MM Master BFM. It includes the following topics:

- "Timing" on page 2–2
- "Block Diagram" on page 2–5

Timing

The timing diagram in Figure 2–2 illustrates the sequence of events for an Avalon-MM Slave BFM responding to interleaved writes and reads when the `readdatavalid` signal is present.

Figure 2–2. Avalon-MM Slave Responding to Interleaved Write and Read Transactions

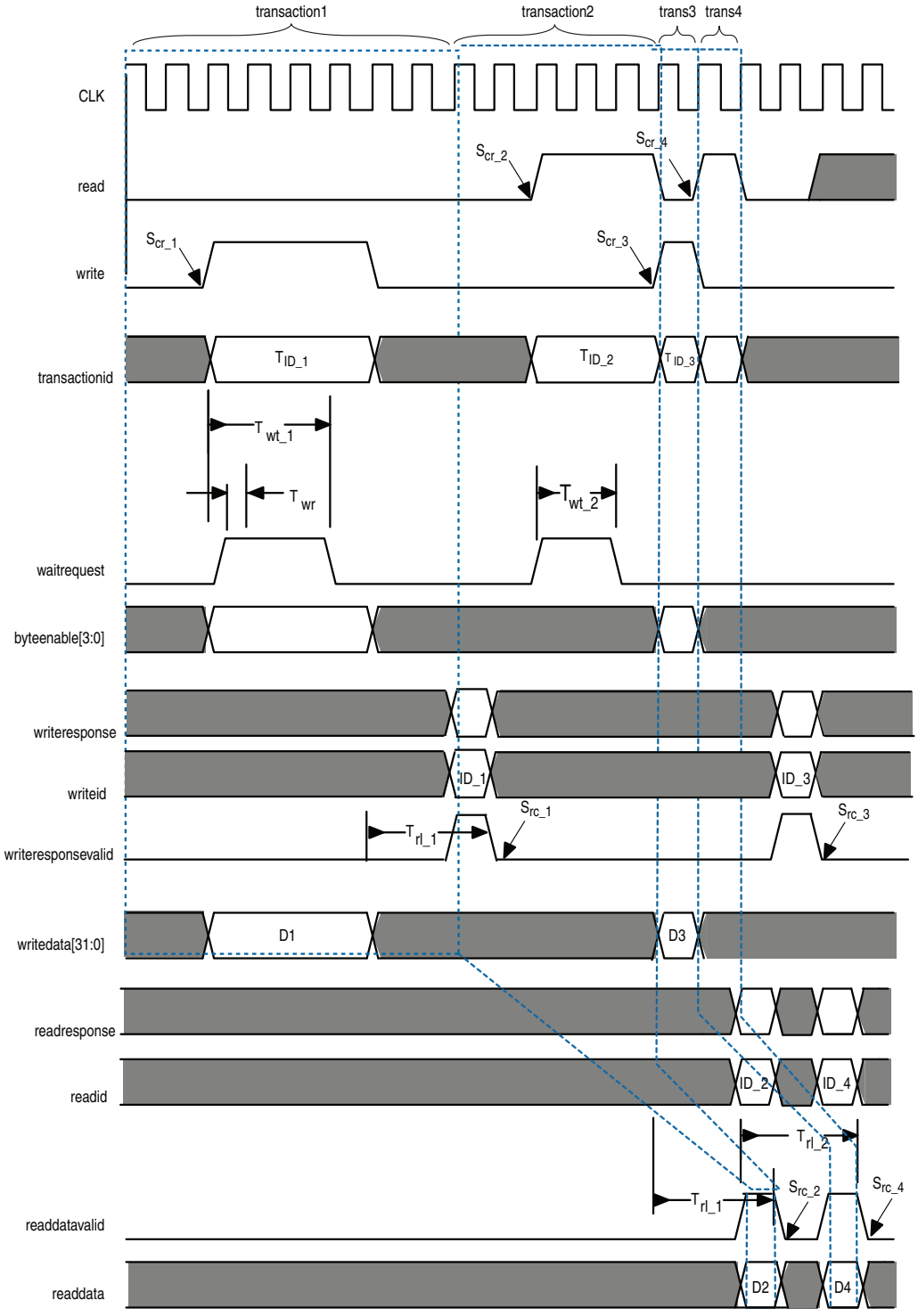


Table 2–1 lists the annotations used in Figure 2–2.

Table 2–1. Key to Annotations in Figure 2–2

Symbol	Description
T_{wt_1}	The response wait time, which is three cycles. The slave sets this value using the `set_interface_wait_time` command.
T_{wr}	`waitrequest` is sampled #1 after the falling edge of `clk`.
T_{wt_2}	The response wait time for the first read, which is two cycles. The slave sets this value using the `set_interface_wait_time` command.
S_{cr_1}–S_{cr_2}	Signals when read commands were received. The event name is `signal_command_received`.
T_{rl_1},T_{rl_2}	The response latency for the reads, which is three cycles. The slave sets this time using the `set_response_latency` command.
T_{wrl_1}	The write response latency for the first write, which is three cycles. This is the time between when the write command is accepted, and the write response is provided by the slave. T
S_{rc_1},S_{rc_3}	Signals write responses. The event name is `signal_response_issued`.
S_{rc_2},S_{rc_4}	Signals read responses. The event name is `signal_response_issued`.
T_{ID_1}–T_{ID_4}	Reference number to identify each read or write transaction.
ID_1, ID_3	Reference number to identify write transactions.
ID_2, ID_4	Reference number to identify read transactions.

The timing diagram in Figure 2–3 illustrates the sequence of events for an Avalon-MM Slave BFM receiving a write followed by a read when the `readdatavalid` signal is not present.

Figure 2–3. Avalon-MM Slave Receiving Write and Read Commands with No readdatavalid Signal

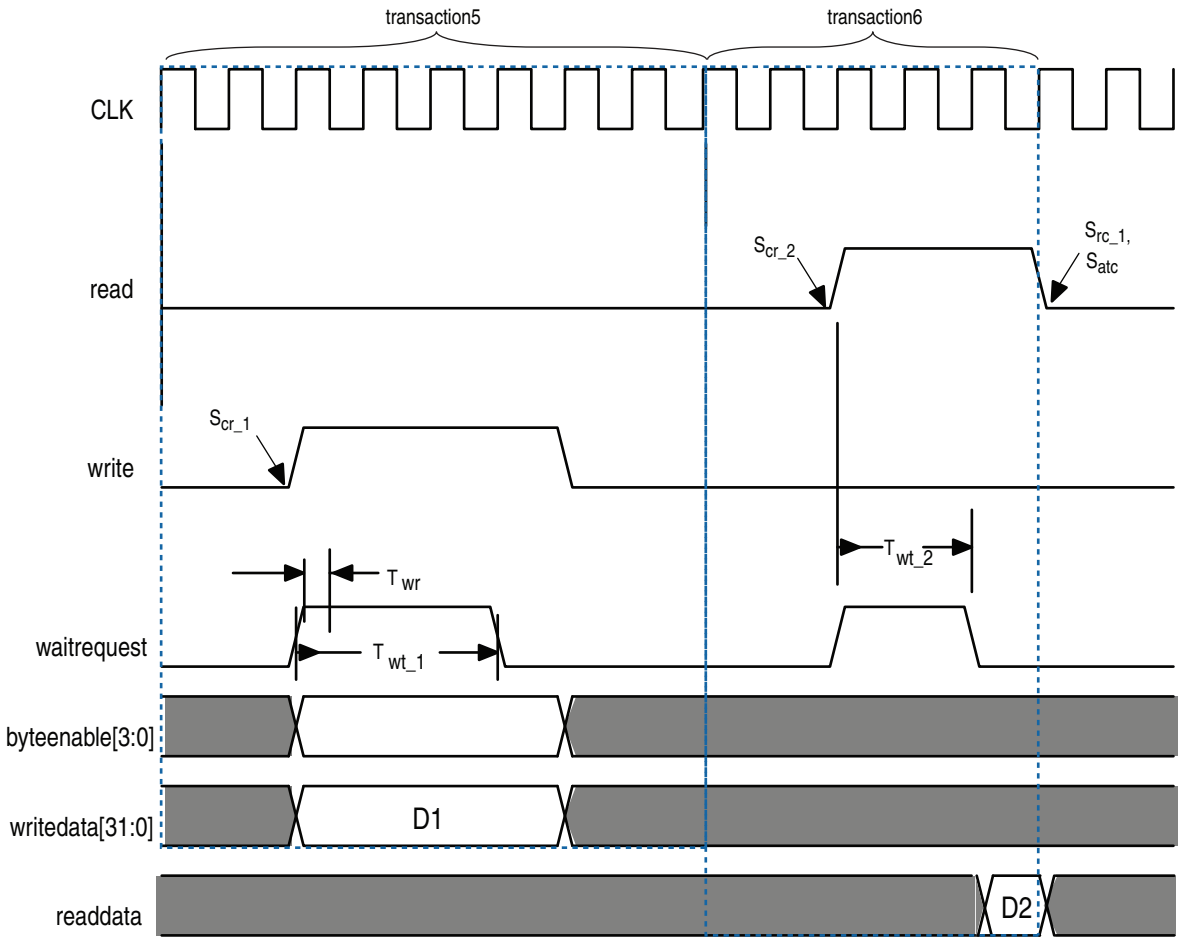


Table 2–2 lists the annotations used in Figure 2–3.

Table 2–2. Key to Annotations in Figure 2–3 (Part 1 of 2)

Symbol	Description
T_i	The initial command latency which is two cycles for transactions 1 and 2.
T_{wt_1}	The response wait time which is three cycles. The master gets this value using the `get_response_wait_time` command.
T_{wt_2}	The response wait time for the first read, which is two cycles. The slave sets this value using the `set_interface_wait_time` command.
T_{wr}	`waitrequest` is sampled #1 after the falling edge of `clk`.
T_{rl_1}	The response latency for the first read, which is zero cycles. The master gets this time using the `get_response_latency` command.
S_{cr_1}, S_{cr_2}	Signals write and read commands. The event name is `signal_command_issued`.

Table 2–2. Key to Annotations in Figure 2–3 (Part 2 of 2)

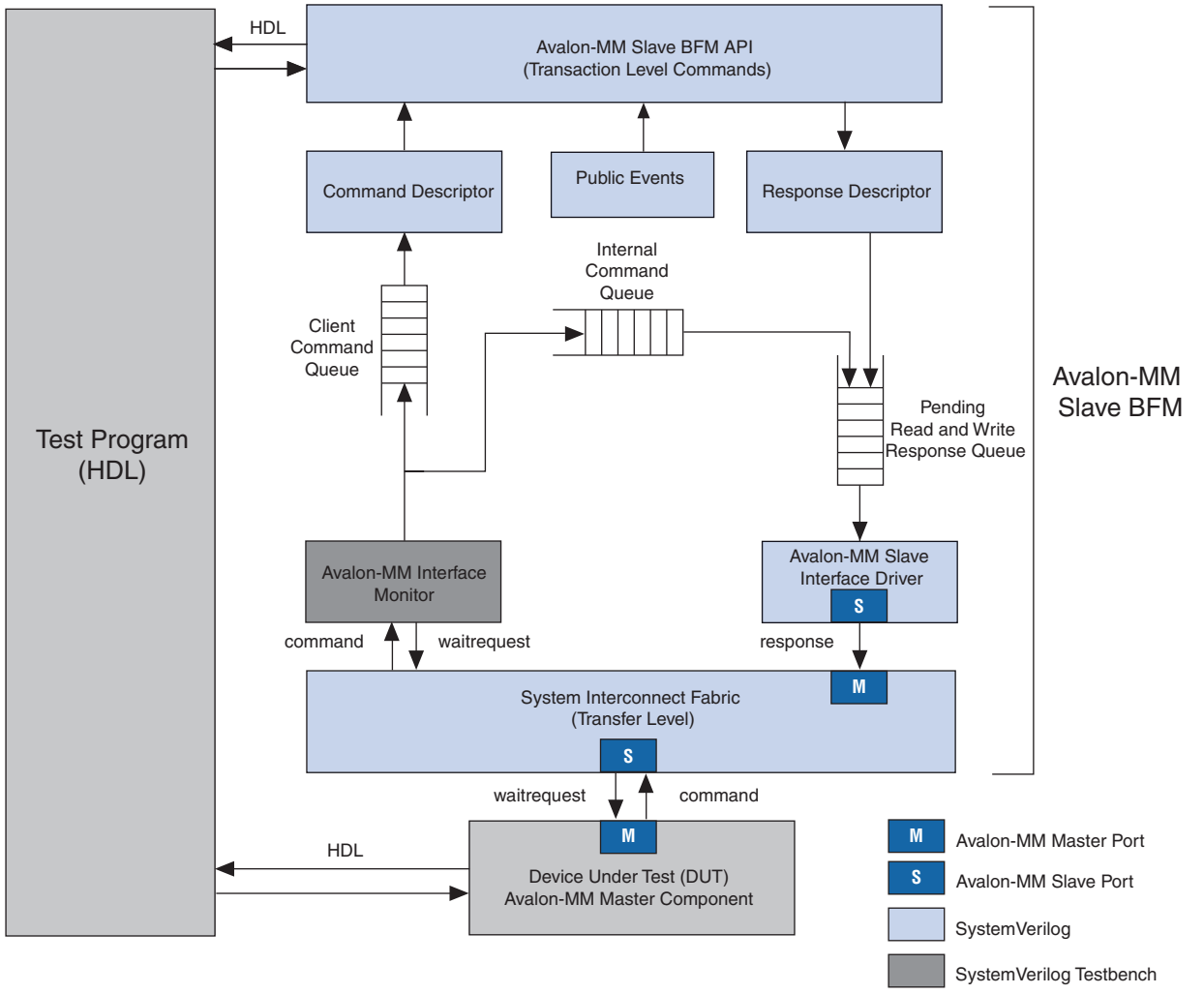
Symbol	Description
S_{rc_1}	Signals the first read response. The event name is `signal_response_complete`.
S_{atc}	Signals the end of the test. The event name is `signal_all_transactions_complete`

Block Diagram

Figure 2–4 shows a block diagram of the Avalon-MM Slave BFM. The BFM includes the following major blocks:

- *Avalon-MM Slave API*—Provides methods to get commands and create responses to commands from the Avalon-MM master (DUT).
- *Command Descriptor*—Accumulates the fields of a command sent by the Avalon-MM master and sends completed commands to the Avalon-MM Slave BFM when requested.
- *Avalon-MM Interface Monitor*—Monitors activity coming from the Avalon-MM Master (DUT) and stores commands in the Client Command Queue.
- *Response Generator and Data Cache*— In `memory_mode` the Slave BFM models a single port RAM. A write operation stores the data in an associative array and generates no response. A read operation fetches data from the array and drives it on the response side of the Avalon interface. This mode simplifies loopback testing.
- *Avalon-MM Slave Interface Driver*—Drives responses to the system interconnect fabric. For burst transfers, there is a separate transfer for each word of the burst. The client testbench can instruct the Slave BFM to assert `waitrequest` for each word of the burst to test the functionality of the Avalon-MM master.
- *Public Events*—Provides status response that arrives together with the data. The public event signals indicate the status of the Master's request such as successful completion, timeout, or error.

Figure 2–4. Avalon-MM Slave BFM Block Diagram



Parameters

The Avalon-MM Slave BFM supports the full range of signals defined for the Avalon-MM slave interface. You can customize the Avalon-MM slave interface using the parameters described in Table 2–3.

Table 2–3. Parameters for the Avalon-MM Slave BFM (Part 1 of 2)

Parameter	Default Value	Legal Values	Description
Port Widths			
Address width	**32**	—	Address width in bits.
Symbol width	**8**	—	Data symbol width in bits. Set `AV_SYMBOL_W` to 8 for byte-oriented interfaces.
Read Response width	**8**	—	Read status response width in bits.
Write Response width	**8**	—	Write status response width in bits.
Parameters			
Number of symbols	**4**	—	Number of symbols per word.
Burstcount width	**3**	—	The width of the burst count in bits.
Port Enables			
Use the read signal	**On**	**On/Off**	When **On**, the interface includes a `read` pin.
Use the write signal	**On**	**On/Off**	When **On**, the interface includes a `write` pin.
Use the address signal	**On**	**On/Off**	When **On**, the interface includes `address` pins.
Use the byte enable signal	**On**	**On/Off**	When **On**, the interface includes `byte_enable` pins.
Use the burstcount signal	**On**	**On/Off**	When **On**, the interface includes `burstcount` pins.
Use the readdata signal	**On**	**On/Off**	When **On**, the interface includes a `readdata` pin.
Use the readdatavalid signal	**On**	**On/Off**	When **On**, the interface includes a `readdatavalid` pin.
Use the writedata signal	**On**	**On/Off**	When **On**, the interface includes a `writedata` pin.
Use the begintransfer signal	**Off**	**On/Off**	When **On**, the interface includes `writedata` pins.
Use the beginbursttransfer signal	**Off**	**On/Off**	When **On**, the interface includes a `beginbursttransfer` pin.
Use the arbiterlock signal	**Off**	**On/Off**	When **On**, the interface includes an `arbiterlock` pin.
Use the lock signal	**Off**	**On/Off**	When **On**, the interface includes a `lock` pin.
Use the debugaccess signal	**Off**	**On/Off**	When **On**, the interface includes a `debugaccess` pin.
Use the waitrequest signal	**On**	**On/Off**	When **On**, the interface includes a `waitrequest` pin.
Use the transactionid signal	**Off**	**On/Off**	When **On**, the interface includes a `transactionid` pin.
Use the write response signals	**Off**	**On/Off**	When **On**, the interface includes a `writeresponse` pin.
Use the read response signals	**Off**	**On/Off**	When **On**, the interface includes a `readresponse` pin.
Use the clken signals	**Off**	**On/Off**	When **On**, the interface includes a `clken` pin.
Port Polarity			
Assert reset high	**On**	**On/Off**	When **On**, `reset` is asserted high.
Assert waitrequest high	**On**	**On/Off**	When **On**, `waitrequest` is asserted high.
Assert read high	**On**	**On/Off**	When **On**, `read` is asserted high.
Assert write high	**On**	**On/Off**	When **On**, `write` is asserted high.

Table 2–3. Parameters for the Avalon-MM Slave BFM (Part 2 of 2)

Parameter	Default Value	Legal Values	Description
Assert byteenable high	**On**	**On/Off**	When **On**, `byteenable` is asserted high.
Assert readdatavalid high	**On**	**On/Off**	When **On**, `readdatavalid` is asserted high.
Assert arbiterlock high	**On**	**On/Off**	When **On**, `arbiterlock` is asserted high.
Assert lock high	**On**	**On/Off**	When **On**, `lock` is asserted high.
Burst Attributes			
Linewrap burst	**On**	**On/Off**	When **On**, the address for bursts wraps instead of an incrementing. With a wrapping burst, when the address reaches a burst boundary, it wraps back to the previous burst boundary such that only the low order bits need to be used for addressing.
Burst on burst boundaries only	**On**	**On/Off**	When **On**, memory bursts are aligned to the address size.
Miscellaneous			
Maximum pending reads	**1**	—	The maximum number of pending reads which can be queued up by the slave.
VHDL BFM ID	**0**	**0–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.
Timing			
Fixed read latency (cycles)	**0**	—	Sets the read latency for fixed-latency slaves. Not used on interfaces that include the `readdatavalid` signal.
Fixed read wait time (cycles)	**1**	—	For slave interfaces that do not use the `waitrequest` signal, the read wait time indicates the number of cycles before the slave responds to a read. The timing is as if the slave asserted `waitrequest` for this number of cycles.
Fixed write wait time (cycles)	**0**	—	For slave interfaces that do not use the `waitrequest` signal, the write wait time indicates the number of cycles before the slave accepts a write.
Registered waitrequest	**On**	**On/Off**	Specifies whether to turn on the register stage.
Registered Incoming Signals	**On**	**On/Off**	Specifies whether to register incoming signals.
Interface Address Type			
Set slave interface address type to symbols or words	**WORDS**	**WORDS/ SYMBOLS**	Sets slave interface address type to symbols or words.

Application Program Interface

This section describes the API for the Avalon-MM Slave BFM.

event_error_exceed_max_pending_reads()

Prototype:	`event_error_exceed_max_pending_reads()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the BFM has more than the maximum pending reads in the pipelined read commands queue waiting to be processed.
Language support:	VHDL

event_command_received()

Prototype:	`event_command_received()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that a command was received.
Language support:	VHDL

event_response_issued()

Prototype:	`event_response_issued()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that a response was driven to the interface.
Language support:	VHDL

event_max_response_queue_size()

Prototype:	`event_max_response_queue_size()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the response queue size has reached the threshold limit.
Language support:	VHDL

event_min_response_queue_size()

Prototype: `event_min_response_queue_size()`
Arguments: Verilog HDL: N.A.
VHDL: `bfm_id`, `req_if`
Returns: `void`
Description: Notifies the testbench that the response queue size is below the minimum limit.
Language support: VHDL

get_clken()

Prototype: `logic get_clken()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `logic`
Description: Returns the clock enable signal status.
Language support: Verilog HDL, VHDL

get_command_address()

Prototype: `bit [AV_ADDRESS_W-1:0] get_command_address()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `bit [AV_ADDRESS_W-1:0]`
Description: Queries the received command descriptor for the transaction address.
Language support: Verilog HDL, VHDL

get_command_arbiterlock()

Prototype: `bit get_command_arbiterlock()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `bit`
Description: Queries the received command descriptor for the transaction arbiterlock.
Language support: Verilog HDL, VHDL

get_command_burst_count()

Prototype: `[AV_BURSTCOUNT_W-1:0] get_command_burst_count()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `[AV_BURSTCOUNT_W-1:0]`

Description: Queries the received command descriptor for the transaction burst count.

Language support: Verilog HDL, VHDL

get_command_burst_cycle()

Prototype: `int get_command_burst_cycle()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `Int`

Description: The slave BFM receives and processes write burst commands as a sequence of discrete commands. The number of commands corresponds to the burst count. A separate command descriptor is constructed for each write burst cycle, corresponding to a partially completed burst. This method returns a burst cycle field that tells the testbench which burst cycle was active when this descriptor was constructed. This facility enables the testbench to query partially completed write burst operations. In other words, the testbench can query the write data word on each burst cycle as it arrives and begin to process it immediately rather than waiting until the entire burst has been received, making it possible to perform pipelined write burst processing in the testbench.

Language support: Verilog HDL, VHDL

get_command_byte_enable()

Prototype: `bit [AV_NUMSYMBOLS-1:0] get_command_byte_enable (int index)`

Arguments: Verilog HDL: `index`

VHDL: `index`, `bfm_id`, `req_if`

Returns: `bit [AV_NUMSYMBOLS-1:0]`

Description: Queries the received command descriptor for the transaction byte enable. For burst commands with burst count greater than 1, the index selects the data cycle.

Language support: Verilog HDL, VHDL

get_command_data()

Prototype: `bit [AV_DATA_W-1:0] get_command_data(int index)`

Arguments: Verilog HDL: `index`

VHDL: `index`, `bfm_id`, `req_if`

Returns: `bit [AV_DATA_W-1:0]`

Description: Queries the received command descriptor for the transaction write data. For burst commands with burst count greater than 1, the index selects the write data cycle.

Language support: Verilog HDL, VHDL

get_command_debugaccess()

Prototype: `bit get_command_debugaccess()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `bit`

Description: Queries the received command descriptor for the transaction debugaccess.

Language support: Verilog HDL, VHDL

get_command_queue_size()

Prototype: `int get_command_queue_size()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `int`

Description: Queries the command queue to determine number of pending commands.

Language support: Verilog HDL, VHDL

get_command_lock()

Prototype: `bit get_command_lock()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `bit`

Description: Queries the received command descriptor for the transaction lock.

Language support: Verilog HDL, VHDL

get_command_request()

Prototype: `Request_t get_command_request()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `Request_t` (enumerated type)

Description: Gets the received command descriptor to determine command request type. A command type may be `REQ_READ` or `REQ_WRITE`. These type values are defined in the enumerated type called `Request_t`, which is imported with the package named `altera_avalon_mm_pkg`.

Language support: Verilog HDL, VHDL

get_command_transaction_id()

Prototype:	`AvalonTransactionId_t get_command_transaction_id()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`AvalonTransactionId_t`
Description:	Queries the received command descriptor for the transaction ID.
Language support:	Verilog HDL, VHDL

get_command_write_response_request()

Prototype:	`AvalonTransactionId_t get_command_write_response_request()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`AvalonTransactionId_t`
Description:	Queries the received command descriptor for the `write_response_request` field value. A value of 1 indicates that the master has requested for a write response.
Language support:	Verilog HDL, VHDL

get_pending_read_latency_cycle()

Prototype:	`int get_pending_read_latency_cycle()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the read command queue to determine the number of cycles needed for the Slave BFM to complete the current read response. This method notifies the master when the Slave BFM is ready to receive a command.
Language support:	Verilog HDL, VHDL

get_pending_write_latency_cycle()

Prototype:	`int get_pending_write_latency_cycle()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the write command queue to determine the number of cycles needed for the Slave BFM to complete the current write response.
Language support:	Verilog HDL, VHDL

get_response_queue_size()

Prototype:	`int get_response_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the response queue to determine number of response descriptors pending.
Language support:	Verilog HDL, VHDL

get_slave_bfm_status

Prototype:	`bit get_slave_bfm_status`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit`
Description:	Queries the Slave BFM component to determine when the read transaction in the Slave BFM has reached the maximum read transactions. A return value of 1 means that the Slave BFM can no longer accept a new read command.
Language support:	Verilog HDL, VHDL

get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`String`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".
Language support:	Verilog HDL

init()

Prototype:	`init()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Initializes the Avalon-MM slave interface.
Language support:	Verilog HDL, VHDL

pop_command()

Prototype:	`void pop_command()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Removes the command descriptor from the queue so that the testbench can query it using the `get_command` methods.
Language support:	Verilog HDL, VHDL

push_response()

Prototype:	`void push_response()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Inserts the fully populated response transaction descriptor onto the response queue. The BFM removes response descriptors off the queue as soon as they are available, reads them, and drives the Avalon-MM interface response plane.
Language support:	Verilog HDL, VHDL

set_command_transaction_mode()

Prototype:	`void set_command_transaction_mode (int mode);`
Arguments:	Verilog HDL: `mode` VHDL: `mode`, `bfm_id`, `req_if`
Returns:	`void`
Description:	By default, write burst commands are consolidated into a single command transaction containing the write data for all burst cycles in that command. This mode is set when the mode argument equals 0. When the mode argument is set to 1, the default is overridden and write burst commands yield one command transaction per burst cycle.
Language support:	Verilog HDL, VHDL

set_interface_wait_time()

Prototype:	`void set_interface_wait_time(int wait_cycles, int index)`
Arguments:	Verilog HDL: `wait_cycles`, `index` VHDL: `wait_cycles`, `index`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Specifies zero or more wait states to assert in each Avalon burst cycle by driving `waitrequest` active. With write burst commands, each write data cycle is forced to wait the number of cycles corresponding to the cycle index. With read burst commands, there is only one command cycle corresponding to index 0 which can be forced to wait.
Language support:	Verilog HDL, VHDL

set_max_response_queue_size()

Prototype:	`void set_max_response_queue_size(int size)`
Arguments:	Verilog HDL: `int size`
	VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the maximum pending response queue size threshold.
Language support:	Verilog HDL, VHDL

set_min_response_queue_size()

Prototype:	`void set_min_response_queue_size(int size)`
Arguments:	Verilog HDL: `int size`
	VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the minimum pending response queue size threshold.
Language support:	Verilog HDL, VHDL

set_read_response_id()

Prototype:	`void set_read_respose_id(AvalonTransactionId_t id)`
Arguments:	Verilog HDL: `AvalonTransactionId_t id`
	VHDL: `AvalonTransactionId_t id`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the transaction ID on the `avs_readid` pin.
Language support:	Verilog HDL, VHDL

set_read_response_status()

Prototype:	`void set_read_respose_status(AvalonReadResponse_t status, int index)`
Arguments:	Verilog HDL: `AvalonReadResponse_t status`, `int index`
	VHDL: `AvalonReadResponse_t status`, `int index`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the read response status code.
Language support:	Verilog HDL, VHDL

set_response_burst_size()

Prototype: `void set_response_burst_size(bit [AV_BURSTCOUNT_W-1:0] burst_size).`

Arguments: Verilog HDL: `burst_size`

VHDL: `burst_size`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the transaction burst count in the response descriptor.

Language support: Verilog HDL, VHDL

set_response_data()

Prototype: `void set_response_data(bit [AV_DATA_W-1:0] data, int index).`

Arguments: Verilog HDL: `data`, `index`

VHDL: `data`, `index`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the transaction read data in the response descriptor. For burst transactions, the command descriptor holds an array of data, with each element individually set by this method.

Language support: Verilog HDL, VHDL

set_response_latency()

Prototype: `void set_response_latency(bit [31:0]latency, int index)`

Arguments: Verilog HDL: `latency`, `index`

VHDL: `latency`, `index`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the response latency for read commands. The response is driven out the specified number of cycles after receiving the read command.

Language support: Verilog HDL, VHDL

set_response_request()

Prototype: `void set_response_request(Request_t request)`

Arguments: Verilog HDL: `Request_t request`

VHDL: `Request_t request`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the transaction type to read or write in the response descriptor. The enumeration type defines `REQ_READ` = `0` and `REQ_WRITE` = `1`.

Language support: Verilog HDL, VHDL

set_response_timeout()

Prototype:	`void set_response_timeout(int cycles)`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the number of cycles that may elapse before timing out.
Language support:	Verilog HDL, VHDL

set_write_response_id()

Prototype:	`void set_write_respose_id(AvalonTransactionId_t id)`
Arguments:	Verilog HDL: `AvalonTransactionId_t id` VHDL: `AvalonTransactionId_t id`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the transaction ID on the `avs_writeid` pin.
Language support:	Verilog HDL, VHDL

set_write_response_status()

Prototype:	`void set_write_respose_status(AvalonWriteResponse_t status, int index)`
Arguments:	Verilog HDL: `AvalonWriteResponse_t status`, `int index` VHDL: `AvalonWriteResponse_t status`, `int index`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the write response status code.
Language support:	Verilog HDL, VHDL

signal_command_received

Prototype:	`signal_command_received`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a command has been detected on an Avalon-MM port. The testbench can respond with a `set_command_wait_time` call on receiving this event to dynamically back pressure the driving Avalon-MM master. Alternatively, the previously set `wait_time` might be used continuously for a set of transactions.
Language support:	Verilog HDL

signal_error_exceed_max_pending_reads

Prototype:	`signal_error_exceed_max_pending_reads`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench of the error condition, in which the slave has more than `max_pending_reads` pipelined read commands queued and waiting to be processed.
Language support:	Verilog HDL

signal_max_response_queue_size

Prototype:	`signal_max_response_queue_size`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that the maximum pending transaction queue size threshold has been exceeded.
Language support:	Verilog HDL

signal_min_command_queue_size

Prototype:	`signal_min_response_queue_size`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that the pending transaction queue size is below the minimum threshold.
Language support:	Verilog HDL

signal_fatal_error

Prototype:	`signal_fatal_error`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a fatal error has occurred in this module.
Language support:	Verilog HDL

signal_response_issued

Prototype:	`signal_response_issued`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a response has been driven out on the Avalon bus.
Language support:	Verilog HDL

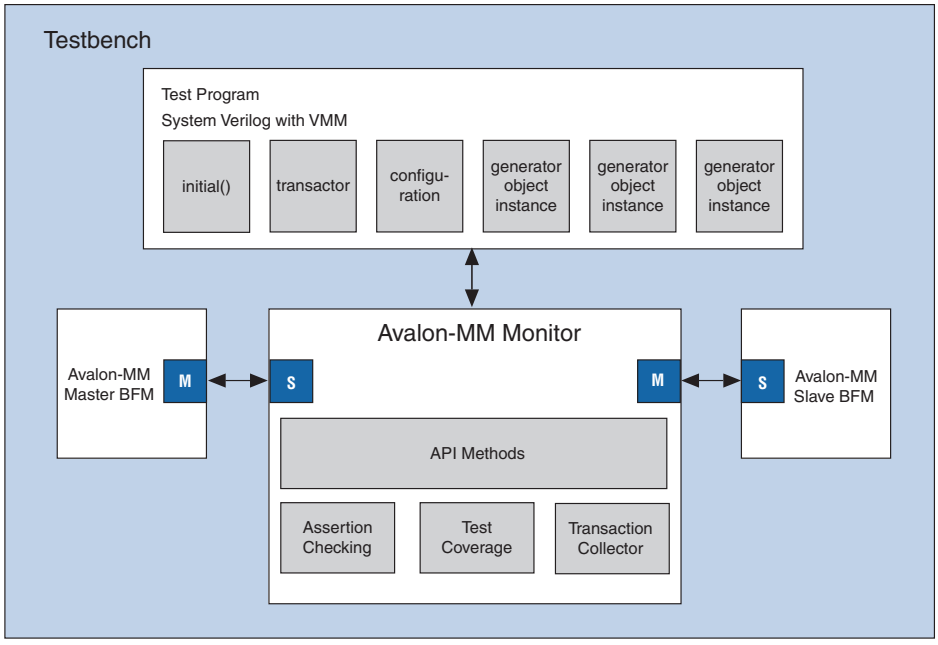
3. Avalon-MM Monitor

The Avalon-MM Monitor verifies Avalon-MM interfaces using SystemVerilog assertions. In addition, it provides test coverage reports so that you can determine when your test vectors provide sufficient test coverage for your component's functionality.

The Avalon-MM Monitor is implemented in SystemVerilog and uses the SystemVerilog Assertion (SVA) language. The SVA language is supported by the Synopsys VCS, and Mentor Graphics Questa simulators. If you are using ModelSim, the monitor component still compiles and simulates, but the assertion checking is disabled.

Figure 3–1 shows a testbench that uses an Avalon-MM Monitor to test components with Avalon-MM interfaces. The monitor's Avalon-MM Master interface is connected to a component's Avalon-MM slave interface, and an Avalon-MM Slave interface is connected to a component's Avalon-MM master interface. The test program communicates with the monitor. The test program can use the monitor's assertion checking and coverage groups to ensure that all legal parameter values for the DUT's Avalon-MM interface are tested. The Avalon-MM Monitor also includes a transaction collector feature to collect and monitor transaction status.

Figure 3–1. Testbench Using an Avalon-MM Monitor with Avalon-MM Interfaces



Parameters

The Avalon-MM Monitor supports the full range of signals defined for the Avalon-MM master and slave interfaces. You can customize the Avalon-MM master and slave interfaces using the parameters described in Table 3–1.

Table 3–1. Parameters for the Avalon-MM Monitor (Part 1 of 2)

Parameter	Default Value	Legal Values	Description
Port Widths			
Address width	**32**	—	Address width in bits.
Symbol width	**8**	—	Data symbol width in bits. The symbol width should be 8 for byte-oriented interfaces.
Number of symbols	**4**	—	Numbers of symbols per word.
Burstcount width	**3**	—	The width of the burst count in bits.
Readresponse width	**8**	—	Read response signal width in bits.
Writeresponse width	**8**	—	Write response signal width in bits.
Port Enables			
Use the read signal	**On**	**On/Off**	When **On**, the interface includes a `read` pin.
Use the write signal	**On**	**On/Off**	When **On**, the interface includes a `write` pin.
Use the address signal	**On**	**On/Off**	When **On**, the interface includes `address` pins.
Use the byte enable signal	**On**	**On/Off**	When **On**, the interface includes `byte_enable` pins.
Use the burstcount signal	**On**	**On/Off**	When **On**, the interface includes `burstcount` pins.
Use the readdata signal	**On**	**On/Off**	When **On**, the interface includes a `readdata` pin.
Use the readdatavalid signal	**On**	**On/Off**	When **On**, the interface includes a `readdatavalid` pin.
Use the writedata signal	**On**	**On/Off**	When **On**, the interface includes a `writedata` pin.
Use the begintransfer signal	**Off**	**On/Off**	When **On**, the interface includes `writedata` pins.
Use the beginbursttransfer signal	**Off**	**On/Off**	When **On**, the interface includes a `beginbursttransfer` pins.
Use the waitrequest signal	**On**	**On/Off**	When **On**, the interface includes a `waitrequest` pin.
Use the arbiterlock signal	**Off**	**On/Off**	When **On**, the interface includes an `arbiterlock` pin.
Use the lock signal	**Off**	**On/Off**	When **On**, the interface includes a `lock` pin.
Use the debugaccess signal	**Off**	**On/Off**	When **On**, the interface includes a `debugaccess` pin.
Use the transactionid signal	**Off**	**On/Off**	When **On**, the interface includes a `transactionid` pin.
Use the writeresponse signal	**Off**	**On/Off**	When **On**, the interface includes a `writeresponse` pin.
Use the readresponse signal	**Off**	**On/Off**	When **On**, the interface includes a `readresponse` pin.
Use the clken signals	**Off**	**On/Off**	When **On**, the interface includes a `clken` pin.

Table 3–1. Parameters for the Avalon-MM Monitor (Part 2 of 2)

Parameter	Default Value	Legal Values	Description
Burst Attributes			
Linewrap burst	**On**	**On/Off**	When **On**, the address for bursts wraps instead of an incrementing. With a wrapping burst, when the address reaches a burst boundary, it wraps back to the previous burst boundary such that only the low order bits need to be used for addressing.
Burst on burst boundaries only	**On**	**On/Off**	When **On**, memory bursts are aligned to the address size.
Miscellaneous			
Read response timeout (cycles)	**100**	—	Specifies when a timeout occurs if `readdatavalid` is not asserted.
Avalon write timeout (cycles)	**100**	—	Specifies when a timeout occurs if a burst write transfer has not completed.
Waitrequest timeout (cycles)	**1024**	—	Timeout period for the continuous assertion of `waitrequest`.
Maximum pending reads	**1**	—	Specifies the maximum number of pipelined reads that can be pending.
Fixed read latency (cycles)	**0**	—	Sets the read latency for fixed-latency slaves. Not used on interfaces that include the `readdatavalid` signal.
Maximum read latency (cycles)	**100**	—	Specifies the maximum read latency in cycle for test coverage function
Maximum waitrequest read cycles (for coverage)	**100**	—	Specifies the maximum wait time allowed for read cycle for coverage.
Maximum waitrequest write cycles (for coverage)	**100**	—	Maximum wait time allowed for write cycle for coverage.
Maximum continuous read (cycles)	**5**	—	Maximum continuous read time allowed for coverage.
Maximum continuous write (cycles)	**5**	—	Maximum continuous write time allowed for coverage.
Maximum continuous waitrequest (cycles)	**5**	—	Maximum continuous wait request time allowed for coverage.
Maximum continuous readdatavalid (cycles)	**5**	—	Maximum continuous readdatavalid time allowed for coverage.
VHDL BFM ID	**0**	**0–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.
Timing			
Fixed read wait time (cycles)	**1**	—	For master interfaces that do not use the `waitrequest` signal, the read wait time indicates the number of cycles before the master responds to a read. The timing is as if the master asserted `waitrequest` for this number of cycles.
Fixed write wait time (cycles)	**0**	—	For master interfaces that do not use the `waitrequest` signal, the write wait time indicates the number of cycles before the master accepts a write.
Registered waitrequest	**Off**	**On/Off**	Specifies whether to turn on the register stage.
Registered Incoming Signals	**Off**	**On/Off**	Specifies whether to register incoming signals.

Application Program Interface

This section describes the API for the Avalon-MM Monitor.

Assertion Checking

For assertion checking, the `enable_waitrequest_timeout` method enables the logic that verifies that the `waitrequest` signal is asserted for fewer cycles than the `waitrequest` timeout period. If the timeout period is violated, an error message displays on the console running the simulation. Error flags are also displayed in the waveform viewer. By default all assertions are enabled. However, depending on the parameterization of the Avalon-MM interface, some assertions are automatically disabled. For example, you might have to turn off some assertion checking to avoid the monitors generating error messages when injecting protocol errors to test the Avalon-MM component's error handling capability. The names of all methods that enable assertions begin with `set_enable_a`. By default, if your testbench includes the Avalon-MM monitor, the checking function is enabled. You can disable checking with the `DISABLE_ALTERA_AVALON_SIM_SVA` macro.

set_enable_a_address_align_with_data_width()

Prototype:	`set_enable_a_address_align_with_data_width()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures the byte address that the master uses is aligned with the data width.
Language support:	Verilog HDL

set_enable_a_beginbursttransfer_exist()

Prototype:	`set_enable_a_beginbursttransfer_exist()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `beginbursttransfer` is asserted during a transfer. It is disabled when `beginbursttransfer` is not used.
Language support:	Verilog HDL

set_enable_a_beginbursttransfer_legal()

Prototype:	`set_enable_a_beginbursttransfer_legal()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `beginbursttransfer` is asserted with a `read` or `write` signal. It is disabled when `beginbursttransfer` is not used.
Language support:	Verilog HDL

set_enable_a_beginbursttransfer_single_cycle()

Prototype:	`set_enable_a_beginbursttransfer_single_cycle()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `beginbursttransfer` is asserted for a single cycle regardless of the behavior of the `waitrequest` signal. It is disabled when `beginbursttransfer` is not used.
Language support:	Verilog HDL

set_enable_a_begintransfer_exist()

Prototype:	`set_enable_a_begintransfer_exist()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `begintransfer` is asserted during any single transfer. Disabled when either `begintransfer` is not supported.
Language support:	Verilog HDL

set_enable_a_begintransfer_legal()

Prototype:	`set_enable_a_begintransfer_legal()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `begintransfer` is asserted together with either `read` or `write`. Disabled when either `begintransfer` is not supported.
Language support:	Verilog HDL

set_enable_a_begintransfer_single_cycle()

Prototype:	`set_enable_a_begintransfer_single_cycle()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `begintransfer` is asserted for only 1 cycle and not reasserted for any single transfer, regardless of the status of the `waitrequest` signal.
Language support:	Verilog HDL

set_enable_a_burst_legal()

Prototype:	`set_enable_a_burst_legal()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that the total number of assertions for the `write` and `readdatavalid` is the same as the `burstcount` for any burst transfer. Disabled when burst transfers are not supported.
Language support:	Verilog HDL

set_enable_a_byteenable_legal()

Prototype:	`set_enable_a_byteenable_legal()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures the `byteenable` value is legal value as specified by the *Avalon Interface Specifications*. Disabled when `byteenable` is not supported.
Language support:	Verilog HDL

set_enable_a_constant_during_burst()

Prototype:	`set_enable_a_constant_during_burst()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that `address` and `burstcount`, and `byteenable` are held constant if a write burst transfer. Disabled when `waitrequest` is not supported. It is disabled when burst transfers are not supported.
Language support:	Verilog HDL

set_enable_a_constant_during_clk_disabled()

Prototype:	`set_enable_a_constant_during_clk_disabled()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that all signals are held constant if `clken` is deasserted.
Language support:	Verilog HDL

set_enable_a_constant_during_waitrequest()

Prototype:	`set_enable_a_constant_during_waitrequest()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that `read`, `write`, `writedata`, `address`, `burstcount`, and `byteenable` are held constant if `waitrequest` is asserted. Disabled when `waitrequest` is not supported.
Language support:	Verilog HDL

set_enable_a_exclusive_read_write()

Prototype:	`set_enable_a_exclusive_read_write()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `read` and `write` are not asserted simultaneously. Disabled when either `read` or `write` is not supported.
Language support:	Verilog HDL

set_enable_a_half_cycle_reset_legal()

Prototype:	`set_enable_a_half_cycle_reset_legal()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `reset` is asserted correctly.
Language support:	Verilog HDL

set_enable_a_less_than_burstcount_max_size()

Prototype:	`set_enable_a_less_than_burstcount_max_size()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `burstcount` size is less than or equal to the maximum burst size, `2**(AV_BURSTCOUNT_W-1)`. It is disabled when either burst transfers are not supported or the bust size is less than 1.
Language support:	Verilog HDL

set_enable_a_less_than_maximumpendingreadtransactions()

Prototype:	`set_enable_a_less_than_maximumpendingreadtransactions()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that the number of pending read transfers is less than `maximumPendingReadTransactions`. Disabled when either `read` is not supported or `maximumPendingReadTransactions` is less than 1.
Language support:	Verilog HDL

set_enable_a_no_readdatavalid_during_reset()

Prototype:	`set_enable_a_no_readdatavalid_during_reset()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that `readdatavalid` is deasserted if `reset` is asserted. Disabled when `readdatavalid` is not supported.
Language support:	Verilog HDL

set_enable_a_no_read_during_reset()

Prototype:	`set_enable_a_no_read_during_reset()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `read` is deasserted if `reset` is asserted. Disabled when `read` is not supported.
Language support:	Verilog HDL

set_enable_a_no_write_during_reset()

Prototype:	`set_enable_a_no_write_during_reset()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `write` is deasserted if `reset` is asserted. Disabled when `write` is not supported.
Language support:	Verilog HDL

set_enable_a_readid_sequence()

Prototype:	`set_enable_a_readid_sequence()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that verifies if the `readid` sequence follows the sequence of the `transactionid`.
Language support:	Verilog HDL

set_enable_a_read_response_sequence()

Prototype:	`set_enable_a_read_response_sequence()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `readdatavalid` is asserted while read is asserted for the same read transfer.
Language support:	Verilog HDL

set_enable_a_read_response_timeout()

Prototype:	`set_enable_a_read_response_timeout()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `readdatavalid` is asserted within maximum allowed timeout period. Disabled when either `readdatavalid` is not supported or the maximum allowed timeout period is less than 1.
Language support:	Verilog HDL

set_enable_a_register_incoming_signals()

Prototype:	`set_enable_a_register_incoming_signals()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `waitrequest` is asserted at all times and deasserts a single clock cycle after a read or write transaction.
Language support:	Verilog HDL

set_enable_a_waitrequest_during_reset()

Prototype:	`set_enable_a_waitrequest_during_resetl()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that `waitrequest` is asserted if `reset` is asserted. Disabled when `waitrequest` is not supported.
Language support:	Verilog HDL

set_enable_a_waitrequest_timeout()

Prototype:	`set_enable_a_waitrequest_timeout()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `waitrequest` is not asserted continuously for more than maximum allowed timeout period. Disabled when either `waitrequest` is not supported or the maximum timeout period is less than 1.
Language support:	Verilog HDL

set_enable_a_write_burst_timeout()

Prototype:	`set_enable_a_write_burst_timeout()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that the write burst transfer is completed within maximum allowed timeout period. Disabled when either write burst transfers are not supported or the write burst timeout period is less than 1 cycle.
Language support:	Verilog HDL

set_enable_a_writeid_sequence()

Prototype:	`set_enable_a_writeid_sequence()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that verifies if the `writeid` sequence follows the sequence of the `transactionid`.
Language support:	Verilog HDL

Coverage Group

Coverage group ensures that the verification suite tests all expected functionality of the interface. For example, the `cover_b2b_read_write` method ensures that the verification suite includes a test for sequential read and write commands. The Avalon-MM Monitor includes 30 coverage groups. By default all coverage groups are enabled. However, depending on the parameterization of a the Avalon-MM interface, some coverage groups are automatically disabled. For example, if the interface does not allow burst transfers, the coverage groups that test burst transfers are automatically disabled. The names of all methods that enable coverage functionality begin with `set_enable_c`.

To generate the coverage report when using the Synopsys VCS simulator, use the following command:

```
urg -dir simv.vdb ↵
```

To generate the coverage report when using the ModelSim-Altera software, use the following command:

```
run -all ↵
coverage report -details -file report.rpt ↵
```

set_enable_c_b2b_read_read()

Prototype:	`set_enable_c_b2b_read_read()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test back-to-back read transfers. This method is disabled when reads are not supported.
Language support:	Verilog HDL

set_enable_c_b2b_read_write()

Prototype:	`set_enable_c_b2b_read_write()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test a read transfer immediately followed by a write transfer. This method is disabled when reads or writes are not supported.
Language support:	Verilog HDL

set_enable_c_b2b_write_read()

Prototype:	`set_enable_c_b2b_write_read()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test a write transfer immediately followed by a read. This method is disabled if either reads or writes are not supported.
Language support:	Verilog HDL

set_enable_c_b2b_write_write()

Prototype:	`set_enable_c_b2b_write_write()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test back-to-back write transfers. This method is disabled if writes are not supported.
Language support:	Verilog HDL

set_enable_c_continuous_read()

Prototype:	`set_enable_c_continuous_read()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test continuous read transfers from 2 cycles until `AV_MAX_CONTINUOUS_READ`. Continuous read cycles of more than `AV_MAX_CONTINUOUS_READ` goes to another bin.
Language support:	Verilog HDL

set_enable_c_continuous_readdatavalid()

Prototype:	`set_enable_c_continuous_readdatavalid()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test continuous readdatavalid transfers from 2 cycles until `AV_MAX_CONTINUOUS_READDATAVALID`. Continuous read cycles of more than `AV_MAX_CONTINUOUS_READDATAVALID` goes to another bin.
Language support:	Verilog HDL

set_enable_c_continuous_waitrequest()

Prototype:	`set_enable_c_continuous_waitrequest()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test continuous waitrequest transfers from 2 cycles until `AV_MAX_CONTINUOUS_WAITREQUEST`. Continuous read cycles of more than `AV_MAX_CONTINUOUS_WAITREQUEST` goes to another bin.
Language support:	Verilog HDL

set_enable_c_continuous_waitrequest_from_idle_to_read()

Prototype:	`set_enable_c_continuous_waitrequest_from_idle_to_read()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test `waitrequest` transfers from their idle state until a `waitrequest` read.
Language support:	Verilog HDL

set_enable_c_continuous_waitrequest_from_idle_to_write()

Prototype:	`set_enable_c_continuous_waitrequest_from_idle_to_write()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test `waitrequest` transfers from their idle state until a `waitrequest` write.
Language support:	Verilog HDL

set_enable_c_continuous_write()

Prototype:	`set_enable_c_continuous_write()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test continuous write transfers from two cycles until `AV_MAX_CONTINUOUS_WRITE`. Continuous write cycles of more than `AV_MAX_CONTINUOUS_WRITE` goes to another bin.
Language support:	Verilog HDL

set_enable_c_idle_before_transaction()

Prototype:	`set_enable_c_idle_before_transaction()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to count idle cycles before read or write transactions.
Language support:	Verilog HDL

set_enable_c_idle_in_read_response()

Prototype:	`set_enable_c_idle_in_read_response()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to count idle cycles during a read burst response. This method is disabled if reads or `readdatavalids` are not supported.
Language support:	Verilog HDL

set_enable_c_idle_in_write_burst()

Prototype:	`set_enable_c_idle_in_write_burst()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to count idle cycles during a write burst transaction. This method is disabled if writes are not supported.
Language support:	Verilog HDL

set_enable_c_pending_read()

Prototype:	`set_enable_c_pending_read()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test pending read support. It covers all values for up to the maximum number of pending reads. This method is disabled when either reads or pipelined reads are not supported.
Language support:	Verilog HDL

set_enable_c_read()

Prototype:	`set_enable_c_read()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test read transfers. This method is disabled when reads are not supported.
Language support:	Verilog HDL

set_enable_c_read_after_reset()

Prototype:	`set_enable_c_read_after_reset()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test read transfers after reset.
Language support:	Verilog HDL

set_enable_c_read_burstcount()

Prototype:	`set_enable_c_read_burstcount()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group tests different sizes of `burstcount` during read burst transfers. It tests all possible values of `burstcount`. This method is disabled when either burst transfers or reads are not supported, or the maximum burst is less than 1.
Language support:	Verilog HDL

set_enable_c_read_byteenable()

Prototype:	`set_enable_c_read_byteenable()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group ensures all legal values of the `byteenable` signal are asserted during read transfers. It is disabled when either `byteenable` or `read` is not supported.
Language support:	Verilog HDL

set_enable_c_read_latency()

Prototype:	`set_enable_c_read_latency()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test all values of the read latency parameter. This method is disabled if `read` or `readdatavalids` are not supported, or if the maximum read latency is less than 1.
Language support:	Verilog HDL

set_enable_c_read_response()

Prototype:	`set_enable_c_read_response()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test each bit of the valid readresponse that represent dfferent status.
Language support:	Verilog HDL

set_enable_c_waitrequest_in_write_burst()

Prototype:	`set_enable_c_waitrequest_in_write_burst()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test the values of the `waitrequest` parameter during write burst transfers.
Language support:	Verilog HDL

set_enable_c_waitrequested_read()

Prototype:	`set_enable_c_waitrequested_read()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test all values of the wait request timeout parameter during read transfers. This method is disabled if `read` or `waitrequest` are not supported, or if the wait request timeout period is less than 1.
Language support:	Verilog HDL

set_enable_c_waitrequest_without_command()

Prototype:	`set_enable_c_waitrequest_without_command()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to verify that no command is asserted between the time when waitrequest is asserted until waitrequest is deasserted.
Language support:	Verilog HDL

set_enable_c_waitrequested_write()

Prototype:	`set_enable_c_waitrequested_write()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test all values of the wait request timeout parameter. This method is disabled if `write` or `waitrequest` are not supported, or if the wait request timeout period is less than 1.
Language support:	Verilog HDL

set_enable_c_write()

Prototype:	`set_enable_c_write()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test write transfers. This method is disabled when writes are not supported.
Language support:	Verilog HDL

set_enable_c_write_with_and_without_writeresponserequest()

Prototype:	`set_enable_c_write_with_and_without_writeresponserequest()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test write transactions with or without `writeresponserequest`.
Language support:	Verilog HDL

set_enable_c_write_after_reset()

Prototype:	`set_enable_c_write_after_reset()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test write transfers after reset.
Language support:	Verilog HDL

set_enable_c_write_burstcount()

Prototype:	`set_enable_c_write_burstcount()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test different sizes of `burstcount` during write burst transfers. It tests all possible values of `burstcount`. This method is disabled when either burst transfers or writes are not supported, or the maximum burst is less than 1.
Language support:	Verilog HDL

set_enable_c_write_byteenable()

Prototype:	`set_enable_c_write_byteenable()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group ensures all legal values of the `byteenable` signal are asserted during write transfers. It is disabled when either `byteenable` or `write` is not supported.
Language support:	Verilog HDL

set_enable_c_write_response()

Prototype:	`set_enable_c_write_response()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage group to test each bit of the valid writeresponse that represent dfferent status.
Language support:	Verilog HDL

Transaction Monitoring

Transaction monitoring is carried out through the transaction collector module. The transaction collector collects the transactions, encapsulates them into descriptors, and inserts the transactions into queue. The API provides the mechanism to query the transactions in queue and disposes them as they are processed. By default the transaction collector module is disabled. You must define the `ENABLE_ALTERA_AVALON_TRANSACTION_RECORDING` Verilog macro to enable this feature. This macro is required to ensure backward compatibility and to avoid breaking existing test cases.

event_transaction_fifo_threshold()

Prototype:	`event_transaction_fifo_threshold()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the transaction FIFO threshold level was exceeded.
Language support:	VHDL

event_transaction_fifo_overflow()

Prototype:	`event_transaction_fifo_overflow()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the transaction FIFO is full and further transactions will be dropped.
Language support:	VHDL

event_command_received()

Prototype:	`event_command_received()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that a command was received.
Language support:	VHDL

event_read_response_complete()

Prototype:	`event_read_response_complete()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that a read response was received.
Language support:	VHDL

event_write_response_complete()

Prototype:	`event_write_response_complete()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that a write response was received.
Language support:	VHDL

event_response_complete()

Prototype:	`event_response_complete()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that a read/write response was received.
Language support:	VHDL

get_clken()

Prototype:	`logic get_clken()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`logic`
Description:	Returns the clock enable signal status.
Language support:	Verilog HDL, VHDL

get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`String`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".
Language support:	Verilog HDL

get_command_address()

Prototype:	`bit [AV_ADDRESS_W-1:0] get_command_address()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit [AV_ADDRESS_W-1:0]`
Description:	Queries the received command descriptor for the transaction address.
Language support:	Verilog HDL, VHDL

get_command_arbiterlock()

Prototype:	`bit get_command_arbiterlock()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit`
Description:	Queries the received command descriptor for the transaction arbiterlock.
Language support:	Verilog HDL, VHDL

get_command_burst_count()

Prototype:	`[AV_BURSTCOUNT_W-1:0] get_command_burst_count()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`[AV_BURSTCOUNT_W-1:0]`
Description:	Queries the received command descriptor for the transaction burst count.
Language support:	Verilog HDL, VHDL

get_command_burst_cycle()

Prototype:	`int get_command_burst_cycle()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`Int`
Description:	The slave BFM receives and processes write burst commands as a sequence of discrete commands. The number of commands corresponds to the burst count. A separate command descriptor is constructed for each write burst cycle, corresponding to a partially completed burst. This method returns a burst cycle field that tells the testbench which burst cycle was active when this descriptor was constructed. This facility enables the testbench to query partially completed write burst operations. In other words, the testbench can query the write data word on each burst cycle as it arrives and begin to process it immediately rather than waiting until the entire burst has been received, making it possible to perform pipelined write burst processing in the testbench.
Language support:	Verilog HDL, VHDL

get_command_byte_enable()

Prototype:	`bit [AV_NUMSYMBOLS-1:0] get_command_byte_enable (int index)`
Arguments:	Verilog HDL: `index` VHDL: `index`, `bfm_id`, `req_if`
Returns:	`bit[AV_NUMSYMBOLS-1:0]`
Description:	Queries the received command descriptor for the transaction byte enable. For burst commands with burst count greater than 1, the index selects the data cycle.
Language support:	Verilog HDL, VHDL

get_command_data()

Prototype:	`bit [AV_DATA_W-1:0] get_command_data(int index)`
Arguments:	Verilog HDL: `index` VHDL: `index`, `bfm_id`, `req_if`
Returns:	`bit[AV_DATA_W-1:0]`
Description:	Queries the received command descriptor for the transaction write data. For burst commands with burst count greater than 1, the index selects the write data cycle.
Language support:	Verilog HDL, VHDL

get_command_debugaccess()

Prototype:	`bit get_command_debugaccess()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit`
Description:	Queries the received command descriptor for the transaction debugaccess.
Language support:	Verilog HDL, VHDL

get_command_issued_queue_size()

Prototype:	`int get_command_issued_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the command issued queue to determine number of pending commands.
Language support:	Verilog HDL, VHDL

get_command_queue_size()

Prototype:	`int get_command_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the command queue to determine number of pending commands.
Language support:	Verilog HDL, VHDL

get_command_lock()

Prototype:	`bit get_command_lock()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit`
Description:	Queries the received command descriptor for the transaction lock.
Language support:	Verilog HDL, VHDL

get_command_request()

Prototype:	`Request_t get_command_request()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`Request_t` (enumerated type)
Description:	Gets the received command descriptor to determine command request type. A command type may be `REQ_READ` or `REQ_WRITE`. These type values are defined in the enumerated type called `Request_t`, which is imported with the package named `altera_avalon_mm_pkg`.
Language support:	Verilog HDL, VHDL

get_command_transaction_id()

Prototype:	`AvalonTransactionId_t get_command_transaction_id()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`AvalonTransactionId_t`
Description:	Queries the received command descriptor for the transaction ID.
Language support:	Verilog HDL, VHDL

get_command_write_response_request()

Prototype:	`AvalonTransactionId_t get_command_write_response_request()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`AvalonTransactionId_t`
Description:	Queries the received command descriptor for the `write_response_request` field value. A value of 1 indicates that the master has requested for a write response.
Language support:	Verilog HDL, VHDL

get_read_response_queue_size()

Prototype:	`int get_read_response_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the read response queue to determine number of response descriptors currently stored in the BFM. This is the number of responses the test program can immediately remove from the response queue for further processing.
Language support:	Verilog HDL, VHDL

get_response_address()

Prototype:	`bit [AV_ADDRESS_W-1:0] get_response_address()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit[AV_ADDRESS_W-1:0]`
Description:	Returns the transaction address in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_byte_enable()

Prototype:	`bit [AV_NUMSYMBOLS-1:0] get_response_byte_enable(int index)`
Arguments:	Verilog HDL: `index` VHDL: `index`, `bfm_id`, `req_if`
Returns:	`bit[AV_NUMSYMBOLS-1:0]`
Description:	Returns the value of the byte enables in the response descriptor that has been removed from the response queue. Each cycle of a burst response is addressed individually by the specified index.
Language support:	Verilog HDL, VHDL

get_response_burst_size()

Prototype:	`bit [AV_BURSTCOUNT_W-1:0]get_response_burst_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit[AV_BURSTCOUNT_W-1:0]`
Description:	Returns the size of the response transaction burst count in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_data()

Prototype:	`bit [AV_DATA_W-1:0] get_response_data(int index)`
Arguments:	Verilog HDL: `index` VHDL: `index`, `bfm_id`, `req_if`
Returns:	`bit[AV_DATA_W-1:0]`
Description:	Returns the transaction read data in the response descriptor that has been removed from the response queue. Each cycle in a burst response is addressed individually by the specified index. In the case of read responses, the data is the data captured on the `avm_readdata` interface pin. In the case of write responses, the data on the driven `avm_writedata` pin is captured and reflected here.
Language support:	Verilog HDL, VHDL

get_response_latency()

Prototype:	`int get_response_latency(int index)`
Arguments:	Verilog HDL: `index` VHDL: `index`, `bfm_id`, `req_if`
Returns:	`int`
Description:	Returns the transaction read latency in the response descriptor that has been removed from the response queue. Each cycle in a burst read has its own latency entry.
Language support:	Verilog HDL, VHDL

get_response_queue_size()

Prototype:	`int get_response_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`automatic int`
Description:	Queries the response queue to determine number of response descriptors currently stored in the BFM. This is the number of responses the test program can immediately remove from the response queue for further processing.
Language support:	Verilog HDL, VHDL

get_response_read_id()

Prototype:	`AvalonTransactionId_t get_response_read_id()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`AvalonTransactionId_t`
Description:	Returns the read id of the transaction in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_read_response()

Prototype:	`AvalonReadResponse_t get_response_read_response(int index)`
Arguments:	Verilog HDL: `int index` VHDL: `int index`, `bfm_id`, `req_if`
Returns:	`AvalonReadResponse_t`
Description:	Returns the transaction read status in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_request()

Prototype:	`Request_t get_response_request()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`Request_t`
Description:	Returns the transaction command type in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_wait_time()

Prototype:	`int get_response_wait_time(int index)`
Arguments:	Verilog HDL: `index` VHDL: `index`, `bfm_id`, `req_if`
Returns:	`int`
Description:	Returns the wait latency for transaction in the response descriptor that has been removed from the response queue. Each cycle in a burst has its own wait latency entry.
Language support:	Verilog HDL, VHDL

get_response_write_id()

Prototype:	`AvalonTransactionId_t get_response_write_id()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`AvalonTransactionId_t`
Description:	Returns the write id of the transaction in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_response_write_response()

Prototype:	`AvalonWriteResponse_t get_response_write_response(int index)`
Arguments:	Verilog HDL: `index` VHDL: `index`, `bfm_id`, `req_if`
Returns:	`AvalonWriteResponse_t`
Description:	Returns the transaction write status in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

get_transaction_fifo_max()

Prototype:	`int get_transaction_fifo_max()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Gets the maximum transaction FIFO depth.
Language support:	Verilog HDL, VHDL

get_transaction_fifo_threshold()

Prototype:	`int get_transaction_fifo_threshold()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Gets the transaction FIFO threshold level.
Language support:	Verilog HDL, VHDL

get_write_response_queue_size()

Prototype:	`int get_write_response_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Queries the write response queue to determine number of response descriptors currently stored in the BFM. This is the number of responses the test program can immediately remove from the response queue for further processing.
Language support:	Verilog HDL, VHDL

init()

Prototype:	`init()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Initializes the counters and clears the queue.
Language support:	Verilog HDL, VHDL

pop_command()

Prototype:	`pop_command()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`Void`
Description:	Removes the command descriptor from the queue so that the testbench can query it with the `get_command` methods.
Language support:	Verilog HDL, VHDL

pop_response()

Prototype:	`void pop_response()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Removes the transaction descriptor from the queue so that the testbench can query it with the `get_command` methods. Sequence counter is initialized to 1.
Language support:	Verilog HDL, VHDL

set_command_transaction_mode()

Prototype:	`set_command_transaction_mode()`
Arguments:	Verilog HDL: `int mode` VHDL: `int mode`, `bfm_id`, `req_if`
Returns:	`void`
Description:	By default, write burst commands are consolidated into a single command transaction containing the write data for all burst cycles in that command. This mode is set when the mode argument equals 0. When the mode argument is set to 1, the default is overridden and write burst commands yield one command transaction per burst cycle.
Language support:	Verilog HDL, VHDL

set_transaction_fifo_max()

Prototype:	`set_transaction_fifo_max()`
Arguments:	Verilog HDL: `int level` VHDL: `int level`, `bfm_id`, `req_if`
Returns:	`void`.
Description:	Sets the maximum transaction level of the FIFO. The event `signal_transaction_fifo_max` is triggered when this level is exceeded.
Language support:	Verilog HDL, VHDL

set_transaction_fifo_threshold()

Prototype:	`set_transaction_fifo_threshold()`
Arguments:	Verilog HDL: `int level` VHDL: `int level`, `bfm_id`, `req_if`
Returns:	`void`.
Description:	Sets the threshold alert level of the FIFO. The event `signal_transaction_fifo_threshold` is triggered when this level is exceeded.
Language support:	Verilog HDL, VHDL

signal_command_received

Prototype:	`signal_command_received`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a command has been detected on the Avalon port. The testbench responds with a `set_interface_wait_time` call on receiving this event to dynamically backpressure the driving Avalon master.
Language support:	Verilog HDL

signal_fatal_error

Prototype:	`signal_fatal_error`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a fatal error has occured in this module.
Language support:	Verilog HDL

signal_read_response_complete

Prototype:	`signal_read_response_complete`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that the read response has been received and inserted into the response queue.
Language support:	Verilog HDL

signal_response_complete

Prototype:	`signal_response_complete`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Triggers when either `signal_read_response_complete` or `signal_write_response_complete` is triggered indicating that either a read or a write response has been received and inserted into the response queue.
Language support:	Verilog HDL

signal_transaction_fifo_overflow

Prototype:	`signal_transaction_fifo_overflow`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that the FIFO is full and further transactions are dropped.
Language support:	Verilog HDL

signal_transaction_fifo_threshold

Prototype:	`signal_transaction_fifo_threshold`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that the transaction FIFO threshold level has exceeded.
Language support:	Verilog HDL

signal_write_response_complete

Prototype:	`signal_write_response_complete`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that the write response has been received and inserted into the response queue.
Language support:	Verilog HDL

Section IV. Avalon-ST BFMs

This section provides information about Avalon-ST BFMs. This section includes the following chapters:

- Chapter 1, Avalon-ST Source BFM
- Chapter 2, Avalon-ST Sink BFM
- Chapter 3, Avalon-ST Monitor

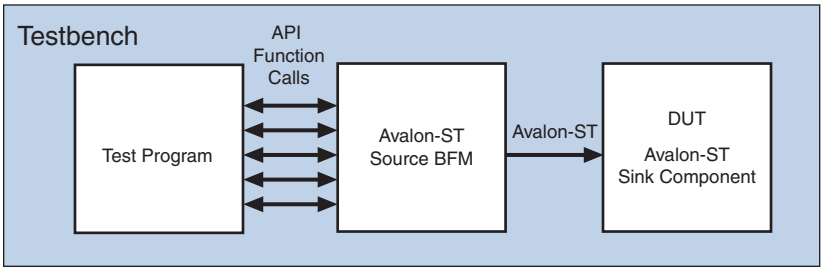
1. Avalon-ST Source BFM

The Avalon-ST Source BFM implements the Avalon-ST interface protocol, a protocol that is point-to-point, packet oriented, and drives unidirectional data. This BFM component includes a procedural interface to control signals on the Avalon-ST interface, including: ready, start of packet, and end of packet.

Figure 1–1 shows the top-level modules for a testbench that uses the Avalon-ST Source BFM to verify an Avalon-ST sink component. In addition to the Altera-provided Avalon-ST Source BFM component, the testbench typically includes a test program and the DUT.

The BFMs allow illegal transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitor components verify protocol compliance.

Figure 1–1. Top-Level Module to Verify an Avalon-ST Sink Device



For more information about the Avalon-ST specification supported in Qsys, refer to the *Avalon Interface Specifications (version 2.1)*.

Functional Description

This section provides a functional description of the Avalon-ST Source BFM. It includes the following topics:

"Timing" on page 1–2

"Block Diagram" on page 1–3

Timing

The timing diagram shown in Figure 1–2 illustrates the timing for an Avalon-ST Source BFM sending data to a sink. In the first instance the sink is not ready when the source has data. In the second instance, the sink is ready but the source does not initially have valid data.

The Avalon-ST BFM behaves differently depending on whether the sink's `READY_LATENCY = 0` or `READY_LATENCY > 0`. When the ready latency is 0, the source BFM holds its current transaction until the sink is ready. When the ready latency is greater than 0, the BFM drives idles until the sink is ready, then it drives the transaction. Figure 1–2 illustrates the timing when `READY_LATENCY = 0`.

Figure 1–2. Avalon-ST Source Sending Data to a Sink

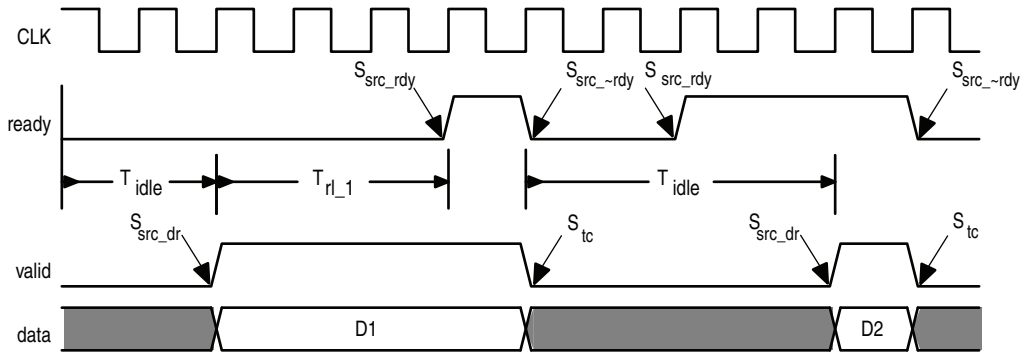


Table 1–1 explains the annotations used in Figure 1–2.

Table 1–1. Key to Annotations in Figure 1–2

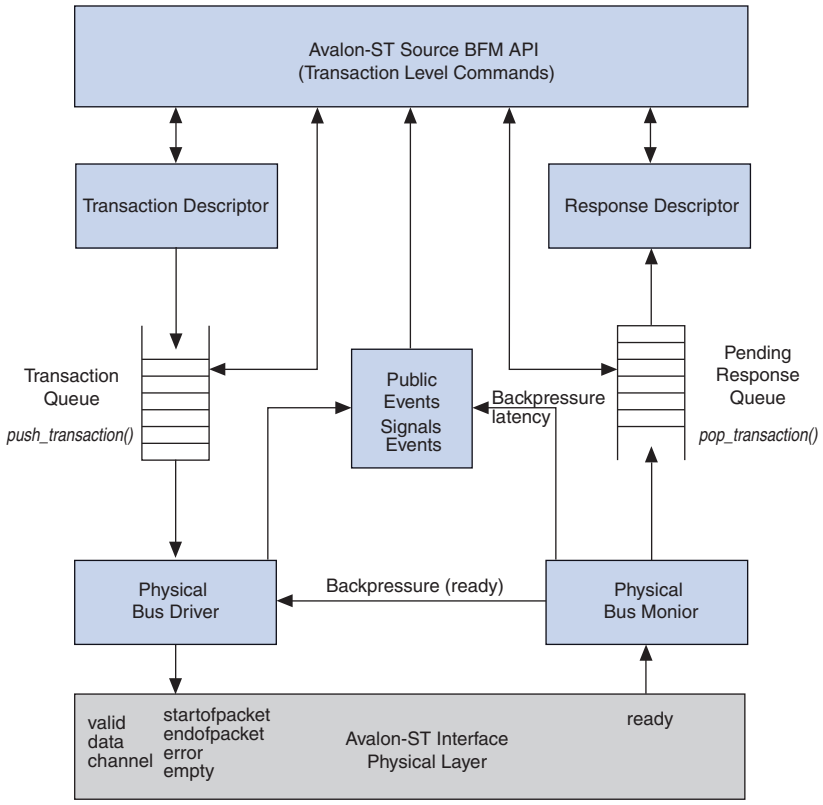
Symbol	Description
T_{idle}	The idle time before a transactions. This time is set by the command `set_transaction_idles`.
T_{rl_1}	The response latency for the first source to sink transaction, which is three cycles. The source gets this time using the `get_response_latency` command.
S_{src_dr}	Signals that the source is driving valid data. The event name is `signal_src_driving_transaction`.
S_{src_rdy}	Signals the source has received the assertion of `ready` from the sink. The event name is `signal_src_ready`.
S_{tc}	Signals the first transaction is complete. The event name is `signal_src_transaction_complete`.
$S_{src\_\sim rdy}$	Signals the source has received the deassertion of `ready` from the sink. The event name is `signal_src_not_ready`.

Block Diagram

Figure 1–3 shows a block diagram of the Avalon-ST Source BFM. This figure illustrates, the BFM includes the following six major blocks:

- *Avalon-ST Source API*—Provides methods to create Avalon-ST transactions and query the state of all queues.
- *Transaction Descriptor*—Accumulates the fields of an Avalon-ST command and inserts completed commands onto the pending command queue.
- *Avalon-ST Physical Driver*—Issues transfers and holds each transfer until `ready` is asserted.
- *Physical Bus Monitor*—Monitors the physical layer and reports on the status of the ready signal to the Physical Bus Driver and the Public Events module.
- *Public Events*—Signals the events described in the API.
- *Response Descriptor*—Collects information about completed transactions.

Figure 1–3. Block Diagram of the Avalon-ST Source BFM



Parameters

The Avalon-ST Source BFM supports all the of the signals defined for the Avalon-MM source interface. You can customize the Avalon-ST Source interface using the parameters described in Table 1–2.

Table 1–2. Parameters for the Avalon-ST Source BFM

Parameter	Default Value	Legal Values	Description
Port Enables			
Include the signals to support packets	**Off**	**On/Off**	When On, the interface includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use the channel port	**Off**	**On/Off**	When **On**, the interface includes `channel` pin or pins.
Use the error port	**Off**	**On/Off**	When **On**, the interface includes `error` pin or pins.
Use the ready port	**On**	**On/Off**	When **On**, the interface includes a `ready` pin.
Use the valid port	**On**	**On/Off**	When **On**, the interface includes a `valid` pin.
Use the empty port	**Off**	**On/Off**	When **On**, the interface includes `empty` pins.
Port Widths			
Symbol Width	**8**	**1–1024**	Data symbol width in bits. The symbol width should be 8 for byte-oriented interfaces.
Number of symbols	**4**	**1–1024**	Specifies the number of symbols that are transferred per beat.
Width of the channel port	**1**	**1–32**	Specifies the width of the `channel` signal.
Width of the error port	**1**	**1–1024**	Specifies the width of the `error` signal.
Width of the empty port	**1**	**1–1024**	Specifies the width of the `empty` signal.
Timing Attributes			
Ready latency	**0**	**0–8**	Specifies the delay between the `ready` and `valid` signals. Refer to the *Avalon Interface Specification* for more information.
Number of beats per cycle	**1**	**1–1024**	Specifies the number of beats per cycle.
Channel Attributes			
Max channel number	**1**	—	Specifies the maximum number of channels that the interface supports.
Miscellaneous			
VHDL BFM ID	**0**	**0–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.

Application Program Interface

This section describes the API for the Avalon-ST source BFM.

event_max_transaction_queue_size()

Prototype:	`event_max_transaction_queue_size()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the transaction queue exceeds the maximum level.
Language support:	VHDL

event_min_transaction_queue_size()

Prototype:	`event_min_transaction_queue_size()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the transaction queue is below the minimum level.
Language support:	VHDL

event_response_done()

Prototype:	`event_response_done()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the sink interface accepted the transaction.
Language support:	VHDL

event_src_driving_transaction()

Prototype:	`event_src_driving_transaction()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that a transaction was driven to the interface.
Language support:	VHDL

event_src_not_ready()

Prototype: `event_src_not_ready()`

Arguments: Verilog HDL: N.A.

VHDL: `bfm_id`, `req_if`

Returns: `void`

Description: Notifies the testbench that the ready signal was deasserted.

Language support: VHDL

event_src_ready()

Prototype: `event_src_ready()`

Arguments: Verilog HDL: N.A.

VHDL: `bfm_id`, `req_if`

Returns: `void`

Description: Notifies the testbench that the ready signal was asserted.

Language support: VHDL

event_src_transaction_complete()

Prototype: `event_src_transaction_complete()`

Arguments: Verilog HDL: N.A.

VHDL: `bfm_id`, `req_if`

Returns: `void`

Description: Notifies the testbench that all transactions were accepted.

Language support: VHDL

get_response_latency()

Prototype: `get_response_latency()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `int`

Description: Returns the response latency in cycles due to back pressure for the most recently removed transaction.

Language support: Verilog HDL, VHDL

get_response_queue_size()

Prototype: `get_response_queue_size()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `int`
Description: Returns the number of transactions in the response queues.
Language support: Verilog HDL, VHDL

get_src_ready()

Prototype: `get_src_ready()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `bit`
Description: Returns the value of the source ready port.
Language support: Verilog HDL, VHDL

get_src_transaction_complete()

Prototype: `get_src_transaction_complete()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `bit`
Description: Returns the transaction complete status.
Language support: Verilog HDL, VHDL

get_transaction_queue_size()

Prototype: `get_transaction_queue_size()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `int`
Description: Returns the number of transactions in the local queues.
Language support: Verilog HDL, VHDL

get_version()

Prototype:	`get_version()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`String`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 SP1 is encoded as "10.1.1".
Language support:	Verilog HDL

init()

Prototype:	`init()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Drives the interface to the idle state.
Language support:	Verilog HDL, VHDL

pop_response()

Prototype:	`pop_response()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Removes the response transaction from the queue before querying contents.
Language support:	Verilog HDL, VHDL

push_transaction()

Prototype:	`push_transaction()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Inserts the out-going transaction into the local transaction queue. The BFM drives the appropriate signals to the Avalon-ST interface based on the transactions in its local queue.
Language support:	Verilog HDL, VHDL

set_max_transaction_queue_size()

Prototype:	`void set_max_transaction_queue_size(int size)`
Arguments:	Verilog HDL: `int size` VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the pending transaction queue size maximum threshold. The public event `signal_max_transaction_queue_size` triggers when the threshold is exceeded.
Language support:	Verilog HDL, VHDL

set_min_transaction_queue_size()

Prototype:	`void set_min_transaction_queue_size(int size)`
Arguments:	Verilog HDL: `int size` VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the pending transaction minimum queue size threshold. The public event `signal_min_transaction_queue_size` triggers when the queue size level is below the minimum threshold.
Language support:	Verilog HDL, VHDL

set_response_timeout()

Prototype:	`set_response_timeout(int cycles)`
Arguments:	Verilog HDL: `cycles` VHDL: `cycles`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the number of cycles that have to elapse before a response timeout is asserted. Disable the time-out by setting the cycles argument to zero.
Language support:	Verilog HDL, VHDL

set_transaction_channel()

Prototype:	`set_transaction_channel(STChannel_t channel)`
Arguments:	Verilog HDL: `channel` VHDL: `channel`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the channel identifier in the out-going transaction.
Language support:	Verilog HDL, VHDL

set_transaction_data()

Prototype: `set_transaction_data(STData_t data)`
Arguments: Verilog HDL: `data`
VHDL: `data`, `bfm_id`, `req_if`
Returns: `void`
Description: Sets the value of `data` in the out-going transaction.
Language support: Verilog HDL, VHDL

set_transaction_idles()

Prototype: `set_transaction_idles(bit[31:0] idle_cycles)`
Arguments: Verilog HDL: `idle_cycles`
VHDL: `idle_cycles`, `bfm_id`, `req_if`
Returns: `void`
Description: Sets the number of idle cycles to elapse before driving the out-going transaction.
Language support: Verilog HDL, VHDL

set_transaction_eop()

Prototype: `set_transaction_eop(bit eop)`
Arguments: Verilog HDL: `eop`
VHDL: `eop`, `bfm_id`, `req_if`
Returns: `void`
Description: Sets the status of the end of packet signal in the out-going transaction.
Language support: Verilog HDL, VHDL

set_transaction_empty()

Prototype: `set_transaction_empty(STEmpty_t empty)`
Arguments: Verilog HDL: `empty`
VHDL: `empty`, `bfm_id`, `req_if`
Returns: `void`
Description: Sets the out-going transaction empty value.
Language support: Verilog HDL, VHDL

set_transaction_error()

Prototype:	`set_transaction_error(STError_t error)`
Arguments:	Verilog HDL: `error`
	VHDL: `error`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the out-going transaction error value.
Language support:	Verilog HDL, VHDL

set_transaction_sop()

Prototype:	`set_transaction_sop(bit sop)`
Arguments:	Verilog HDL: `sop`
	VHDL: `sop`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the status of the start of packet signal in the out-going transaction.
Language support:	Verilog HDL, VHDL

signal_fatal_error

Prototype:	`signal_fatal_error`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Signals that a fatal error has occurred. It terminates the simulation.
Language support:	Verilog HDL

signal_max_transaction_queue_size

Prototype:	`signal_max_transaction_queue_size`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Signals that the pending transaction queue size threshold has been exceeded.
Language support:	Verilog HDL

signal_min_transaction_queue_size

Prototype: `signal_min_transaction_queue_size`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Signals that the pending transaction queue size is below the minimum threshold.
Language support: Verilog HDL

signal_response_done

Prototype: `signal_response_done`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Signals that the response to a driven data beat is available.
Language support: Verilog HDL

signal_src_driving_transaction

Prototype: `signal_src_driving_transaction`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Signals when the source begins to drive a transaction to the interface.
Language support: Verilog HDL

signal_src_not_ready

Prototype: `signal_src_not_ready`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Signals that the `ready` signal is not asserted.
Language support: Verilog HDL

signal_src_ready

Prototype:	`signal_src_ready`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that the `ready` signal is asserted.
Language support:	Verilog HDL

signal_src_transaction_complete

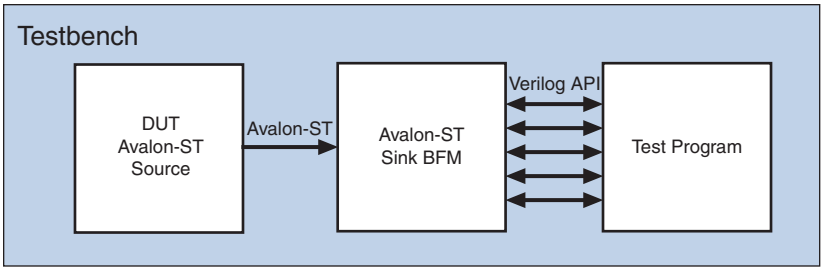
Prototype:	`signal_src_transaction_complete`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that all pending transactions have completed.
Language support:	Verilog HDL

2. Avalon-ST Sink BFM

The Avalon-ST Sink BFM implements the Avalon-ST interface protocol, a protocol that is point-to-point, packet oriented, and drives unidirectional data. This BFM component also includes a procedural interface to respond to the DUT that includes an Avalon-ST source interface. Figure 2–1 shows the top-level modules for testbench that uses the Avalon-ST Sink BFM to verify an Avalon-ST source device. In addition to the Altera-provided Avalon-ST Sink BFM component, the testbench includes a test program and the DUT.

The BFMs allow illegal transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitor components verify protocol compliance.

Figure 2–1. Top-Level Module to Verify an Avalon-ST Source Device



For more information about the Avalon-ST specification supported in Qsys, refer to the *Avalon Interface Specifications (version 2.1)*.

Functional Description

This section provides a functional description of the Avalon-ST Sink BFM. It includes the following topics:

Timing

The timing diagram shown in Figure 2–2 illustrates the timing for an Avalon-ST Sink BFM signalling when it is ready to receive data from an Avalon-ST source. In the first instance, the sink is not ready when the source has data. In the second instance, the sink is ready but the source does not initially have valid data.

Figure 2–2. Avalon-ST Source and Sink Timing

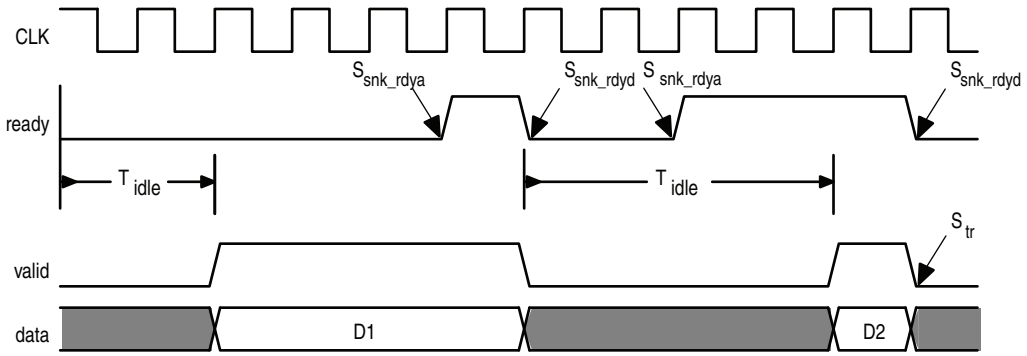


Table 2–1 describes the annotations used in Figure 2–2.

Table 2–1. Key to Annotations in Figure 2–2

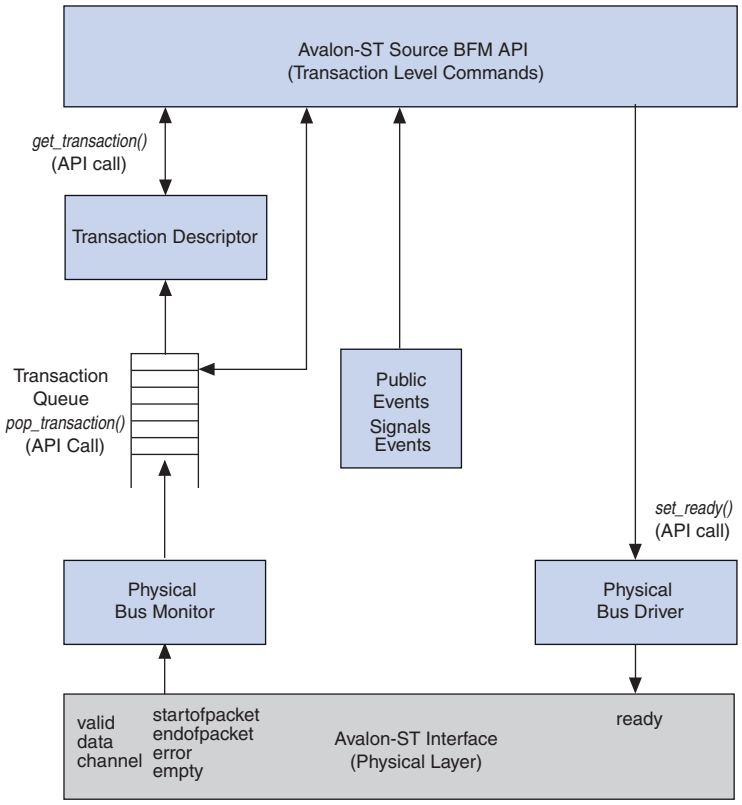
Symbol	Description
T_{idle}	The idle time between transactions. This time is reported by the command `get_transaction_idles`.
S_{snk_rdya}	Signals the sink has asserted `ready`. The event name is `signal_snk_ready_assert`.
S_{tr}	Signals the transaction has been received and queued. The event name is `signal_transaction_received`.
S_{snk_rdyd}	Signals the sink is not `ready`. The event name is `signal_snk_ready_deassert`.

Block Diagram

Figure 2–3 provides a block diagram of the Avalon-ST Sink BFM. This figure illustrates that the BFM includes the following five major blocks:

- *Avalon-ST Sink API*—Provides methods to get Avalon-ST transactions and control the `ready` signal.
- *Transaction Descriptor*—Accumulates the fields of an Avalon-ST command.
- *Avalon-ST Physical Driver*—Asserts and deasserts the `ready` signal to the system interconnect fabric.
- *Physical Bus Monitor*—Monitors the physical layer and collects transactions.
- *Public Events*—Signals the events described in the API.

Figure 2–3. Block Diagram of the Avalon-ST Sink BFM



Parameters

The Avalon-ST Sink BFM supports all of the of signals defined for the Avalon-MM sink interface. You can customize the Avalon-ST sink interface using the parameters described in Table 2–2.

Table 2–2. Parameters for the Avalon-ST Sink BFM

Parameter	Default Value	Legal Values	Description
Port Enables			
Include the signals to support packets	**Off**	**On/Off**	When On, the interface includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use the channel port	**Off**	**On/Off**	When **On**, the interface includes `channel` pin or pins.
Use the error port	**Off**	**On/Off**	When **On**, the interface includes `error` pin or pins.
Use the ready port	**On**	**On/Off**	When **On**, the interface includes a `ready` pin.
Use the valid port	**On**	**On/Off**	When **On**, the interface includes a `valid` pin.
Use the empty port	**Off**	**On/Off**	When **On**, the interface includes `empty` pins.
Port Widths			
Symbol Width	**8**	**1–1024**	Data symbol width in bits. The symbol width should be 8 for byte-oriented interfaces.
Number of symbols	**4**	**1–1024**	Specifies the number of symbols that are transferred per beat.
Width of the channel port	**1**	**1–32**	Specifies the width of the `channel` signal.
Width of the error port	**1**	**1–1024**	Specifies the width of the `error` signal.
Width of the empty port	**1**	**1–1024**	Specifies the width of the `empty` signal.
Timing Attributes			
Ready latency	**0**	**0–8**	Specifies the delay between the `ready` and `valid` signals. Refer to the *Avalon Interface Specification* for more information.
Number of beats per cycle	1	**1–1024**	Specifies the number of beats per cycle.
Channel Attributes			
Max channel number	**1**	—	Specifies the maximum number of channels that the interface supports.
Miscellaneous			
VHDL BFM ID	**0**	**0–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.

Application Program Interface

This section describes the API for the Avalon-ST Sink BFM.

event_transaction_received()

Prototype:	`event_transaction_received()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Signals that the transaction was received.
Language support:	VHDL

event_sink_ready_assert()

Prototype:	`event_sink_ready_assert()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Signals that the `ready` signal was asserted.
Language support:	VHDL

event_sink_ready_deassert()

Prototype:	`event_sink_ready_deassert()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Signals that the `ready` signal was deasserted.
Language support:	VHDL

get_transaction_channel()

Prototype:	`get_transaction_channel()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`STChannel_t`
Description:	Returns the channel identifier for the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_transaction_data()

Prototype: `get_transaction_data()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `STData_t`
Description: Returns the data in the most recently removed transaction.
Language support: Verilog HDL, VHDL

get_transaction_idles()

Prototype: `get_transaction_idles()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `bit[31:0]`
Description: Returns the number of idle cycles in the most recently removed transaction.
Language support: Verilog HDL, VHDL

get_transaction_eop()

Prototype: `get_transaction_eop()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `bit`
Description: Returns the transaction end of packet status in the most recently removed transaction.
Language support: Verilog HDL, VHDL

get_transaction_empty()

Prototype: `get_transaction_empty()`
Arguments: Verilog HDL: None
VHDL: `bfm_id`, `req_if`
Returns: `STEmpty_t`
Description: Returns the number of empty symbols in the most recently removed transaction.
Language support: Verilog HDL, VHDL

get_transaction_error()

Prototype:	`get_transaction_error()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`STError_t`
Description:	Returns the error in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_transaction_queue_size()

Prototype:	`get_transaction_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`
Description:	Returns the length of the queue holding received transactions.
Language support:	Verilog HDL, VHDL

get_transaction_sop()

Prototype:	`get_transaction_sop()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit`
Description:	Returns the transaction start of packet status in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_version()

Prototype:	`get_version()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`string`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 SP1 is encoded as "10.1.1".
Language support:	Verilog HDL

init()

Prototype:	`init()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Drives the interface to the idle state.
Language support:	Verilog HDL, VHDL

pop_transaction()

Prototype:	`pop_transaction()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Removes the transaction descriptor from the queue so that the testbench can query it using the `get_transaction` methods.
Language support:	Verilog HDL, VHDL

set_ready()

Prototype:	`set_ready()`
Arguments:	Verilog HDL: bit VHDL: bit, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the value of the interface's `ready` signal. To assert back pressure, deassert this signal. The parameter `USE_READY` must be set to 1 to enable the `ready` signal.
Language support:	Verilog HDL, VHDL

signal_fatal_error

Prototype:	`signal_fatal_error`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that a fatal error has occurred. It terminates the simulation.
Language support:	Verilog HDL

signal_sink_ready_assert

Prototype:	`signal_sink_ready_assert`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that `sink_ready` is asserted, turning off back pressure.
Language support:	Verilog HDL

signal_sink_ready_deassert

Prototype:	`signal_sink_ready_deassert`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that `sink_ready` is deasserted, turning on back pressure.
Language support:	Verilog HDL

signal_transaction_received

Prototype:	`signal_transaction_received`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that the transaction has been received and queued.
Language support:	Verilog HDL

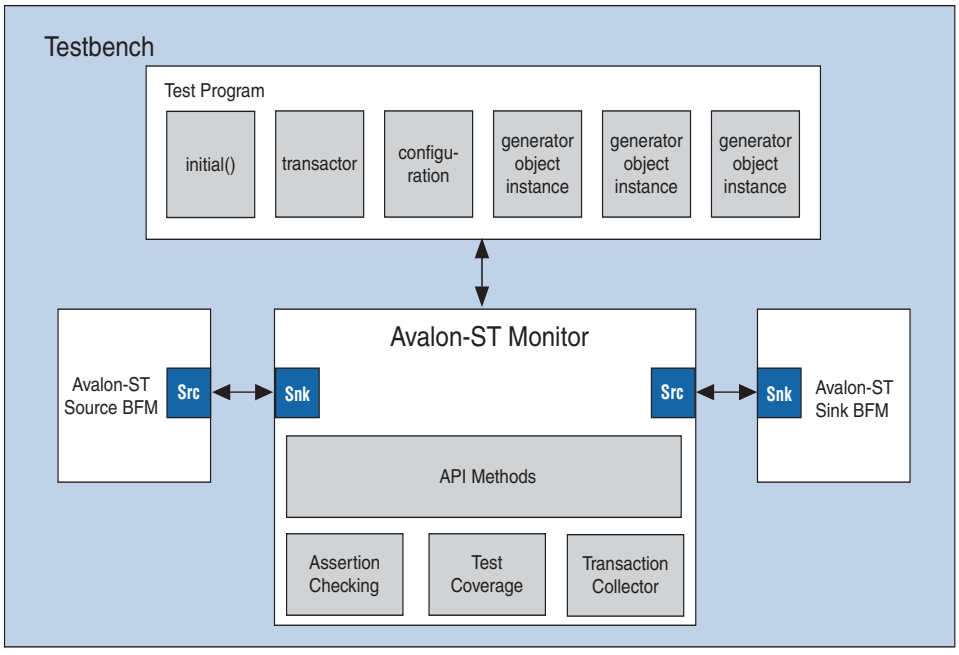
3. Avalon-ST Monitor

The Avalon-ST Monitor verifies Avalon-ST interfaces using SystemVerilog assertions. In addition, it provides test coverage reports so that you can determine when your test vectors provide sufficient test coverage for your DUT functionality.

The Avalon-ST Monitor is implemented in SystemVerilog and uses the SystemVerilog Assertion (SVA) language. The SVA language is supported by the Synopsys VCS, and Mentor Graphics Questa. If you are using ModelSim, the monitor component still compiles and simulates, but the assertion checking is disabled.

Figure 3–1 shows a testbench that uses an Avalon-ST Monitor to test components with Avalon-ST interfaces. This figure illustrates that the monitor's Avalon-ST source interface is connected to the DUT's Avalon-ST sink interface, and an Avalon-ST sink interface is connected to the DUT's Avalon-ST source interface. The test program communicates with the monitor. It uses the monitor's assertion checking and coverage groups to assure that all legal parameter values for the DUT's Avalon-ST interfaces are verified.

Figure 3–1. Testbench Using an Avalon-ST Monitor with Avalon-ST Interfaces



Parameters

The Avalon-ST monitor supports the full range of signals defined for the Avalon-ST source and sink interfaces. You can customize the Avalon-ST source and sink interfaces using the parameters described in Table 3–1.

Table 3–1. Parameters for the Avalon-ST Monitor BFM

Parameter	Default Value	Legal Values	Description
Port Widths			
Symbol width	**8**	—	Data symbol width in bits. The symbol width should be 8 for byte-oriented interfaces.
Number of symbols	**4**	—	Numbers of symbols per word.
Width of the channel signal	**1**	—	Specifies the width of the `channel` signal in bits.
Width of the error port	**1**	—	Specifies the width of the `error` signal in bits.
Width of the empty port	**1**	—	Specifies the width of the `empty` signal in bits.
Port Enables			
Include the signals to support packets	**On**	**On/Off**	When **On**, the interface includes a the `startofpacket`, `endofpacket`, and `empty` signals.
Use the channel port	**On**	**On/Off**	When **On**, the interface includes a `channel` pin.
Use the error port	**On**	**On/Off**	When **On**, the interface includes `error` pins.
Use the ready port	**On**	**On/Off**	When **On**, the interface includes `ready` pins.
Use the valid port	**On**	**On/Off**	When **On**, the interface includes `valid` pins.
Use the empty port	**On**	**On/Off**	When **On**, the interface includes a `empty` pin.
Timing Attributes			
Ready latency	**0**	—	Specifies the `readyLatency` parameter for data interfaces that support backpressure. Refer to the Avalon Interface Specifications for more information.
Number of beats per cycle	**1**	**1–1024**	Specifies the number of beats per cycle.
Channel Attributes			
Max Channel Number	**1**	—	Specifies when a timeout will occur if a burst write transfer has not completed.
Miscellaneous Properties			
Max Packet Size Covered	**1**	—	Specifies the maximum packet size.
VHDL BFM ID	**0**	**0–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.

Application Program Interface

This section describes the API for the Avalon-ST Monitor.

Assertion Checking

Assertion checking methods enable and disable protocol assertions that are used to ensure protocol compliance. For example, the `enable_a_no_data_outside_packet` method enables the assertion that verifies that no data is transmitted between the assertion of the `endofpacket` and the next `startofpacket` signals. If a violation is found, an error message is displayed on the console running the simulation. Error flags also are displayed in the waveform viewer. By default all assertions are enabled. However, depending on the parameterization of a the Avalon-ST interface, some assertions are automatically disabled. For example, you might have to disable some assertion checking to avoid generating error messages when injecting protocol errors to test the Avalon-ST component's error handling capability. The names of all methods that implement assertions begin with `set_enable_a`. By default, if your testbench includes the Avalon-ST monitor, the checking function is enabled. You can disable checking with the `DISABLE_ALTERA_AVALON_SIM_SVA` macro.

set_enable_a_empty_legal()

Prototype:	`set_enable_a_empty_legal()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `empty` is 0 except when `endofpacket` is asserted and that `empty` is always less than the number of symbols in a packet.
Language support:	Verilog HDL

set_enable_a_less_than_max_channel()

Prototype:	`set_enable_a_less_than_max_channel()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that the value of the `channel` signal is less than the maximum number of channels.
Language support:	Verilog HDL

set_enable_a_no_data_outside_packet()

Prototype:	`set_enable_a_no_data_outside_packet()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `valid` data is not transferred outside of a packet when the interface uses packet transmission.
Language support:	Verilog HDL

set_enable_a_non_missing_endofpacket()

Prototype:	`set_enable_a_non_missing_endofpacket()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that the `startofpacket` signal is asserted between each two assertions of an `endofpacket` signal.
Language support:	Verilog HDL

set_enable_a_non_missing_startofpacket()

Prototype:	`set_enable_a_non_missing_startofpacket()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures that each assertion of the `startofpacket` signal is followed by the assertion of an `endofpacket` signal.
Language support:	Verilog HDL

set_enable_a_valid_legal()

Prototype:	`set_enable_a_valid_legal()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures `valid` is deasserted `readyLatency` cycles after `ready` is deasserted if the `readyLatency` is greater than 0.
Language support:	Verilog HDL

Coverage Group

Coverage group ensures that the verification suite tests all expected functionality of the interface. For example, the `cover_b2b_packet_different_channel` method allows each individual coverage point to be enabled or disabled. When coverage points are disabled, they do not show up as missing coverage in the coverage report. By default all coverage groups are enabled. However, depending on the parameterization of a the Avalon-MM interface, some coverage groups are automatically disabled. For example, if the interface does not use packets, the coverage groups that test packet transfers are automatically disabled. The names of all methods that enable coverage functionality begin with `set_enable_c`.

To generate the coverage report when using the Synopsys VCS simulator, use the following command:

```
urg -dir simv.vdb ↵
```

To generate the coverage report when using the ModelSim-Altera software, use the following command:

```
run -all ↵
coverage report -details -file report.rpt ↵
```

set_enable_c_all_idle_beats()

Prototype:	`set_enable_c_all_idle_beats()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for number of transaction with all idle beats.
Language support:	Verilog HDL

set_enable_c_all_valid_beats()

Prototype:	`set_enable_c_all_valid_beats()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for number of transaction with all valid beats.
Language support:	Verilog HDL

set_enable_c_b2b_data_different_channel()

Prototype:	`set_enable_c_b2b_data_different_channel()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures back-to-back `valid` signals for different channels. It is disabled when channels are not supported.
Language support:	Verilog HDL

set_enable_c_b2b_data_same_channel()

Prototype:	`set_enable_c_b2b_data_same_channel()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for back-to-back `valid` signals for the same channel. It is disabled when channels are not supported.
Language support:	Verilog HDL

set_enable_c_b2b_packet_different_channel()

Prototype:	`set_enable_c_b2b_packet_different_channel()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for back-to-back packet transmission for different channels. It is disabled when packet transmission or channels are not supported.
Language support:	Verilog HDL

set_enable_c_b2b_packet_in_different_transaction()

Prototype:	`set_enable_c_b2b_packet_in_different_transaction()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for back-to-back packet transmission of different transactions. It is disabled when packet transmission or channels are not supported.
Language support:	Verilog HDL

set_enable_c_b2b_packet_same_channel()

Prototype: `set_enable_c_b2b_packet_same_channel()`

Arguments: Verilog HDL: `Boolean`

VHDL: N.A.

Returns: `void`

Description: Enables a coverage point that ensures test coverage for back-to-back packet transmission for the same channel. It is disabled when packet transmission or channels are not supported.

Language support: Verilog HDL

set_enable_c_b2b_packet_within_single_cycle()

Prototype: `set_enable_c_b2b_packet_within_single_cycle()`

Arguments: Verilog HDL: `Boolean`

VHDL: N.A.

Returns: `void`

Description: Enables a coverage point that ensures test coverage for back-to-back packet transmission within a single cycle. It is disabled when packet transmission or channels are not supported.

Language support: Verilog HDL

set_enable_c_channel_change_in_packet()

Prototype: `set_enable_c_channel_change_in_packet()`

Arguments: Verilog HDL: `Boolean`

VHDL: N.A.

Returns: `void`

Description: Enables a coverage point that ensures test coverage of a change of channels within the packet transaction. It is disabled when either the `channel` signal or packet transmission is not supported.

Language support: Verilog HDL

set_enable_c_empty()

Prototype: `set_enable_c_empty()`

Arguments: Verilog HDL: `Boolean`

VHDL: N.A.

Returns: `void`

Description: Enables a coverage point that ensures test coverage of a `empty` signal. It is disabled when packet transmission is not supported.

Language support: Verilog HDL

set_enable_c_error()

Prototype:	`set_enable_c_error()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage of all bits of the `error` signal. It is disabled when the `error` signal is not supported.
Language support:	Verilog HDL

set_enable_c_error_in_middle_of_packet()

Prototype:	`set_enable_c_error_in_middle_of_packet()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for the assertion of the `error` signal in the middle of a packet. It is disabled when the `error` signal is not supported.
Language support:	Verilog HDL

set_enable_c_idle_beat_between_packet()

Prototype:	`set_enable_c_idle_beat_between_packet()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for packet transactions that own idle beats in between. It is disabled when packet transmission is not supported.
Language support:	Verilog HDL

set_enable_c_multiple_packet_per_cycle()

Prototype:	`set_enable_c_multiple_packet_per_cycle()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for number of transactions that carry multiple packets per single cycle. It is disabled when packet transmission is not supported.
Language support:	Verilog HDL

set_enable_c_non_valid_ready()

Prototype:	`set_enable_c_non_valid_ready()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for the assertion of `valid` signal with different values for readyLatency. Refer to the *Avalon Interface Specifications* for more information.
Language support:	Verilog HDL

set_enable_c_non_valid_non_ready()

Prototype:	`set_enable_c_non_valid_non_ready()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for the deassertion of both `ready` and `valid`. It is disabled when the `ready` signal is not supported.
Language support:	Verilog HDL

set_enable_c_packet()

Prototype:	`set_enable_c_packet()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage packet transmission for different values of the `channel` signal. It is disabled when packet transmission is not supported.
Language support:	Verilog HDL

set_enable_c_packet_no_idles_no_back_pressure()

Prototype:	`set_enable_c_packet_no_idles_no_back_pressure()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage of packet transaction without back pressure and idle cycles. It is disabled when packet transmission is not supported.
Language support:	Verilog HDL

set_enable_c_packet_size()

Prototype:	`set_enable_c_packet_size()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for different size of packets. It is disabled when packet transmission is not supported.
Language support:	Verilog HDL

set_enable_c_packet_with_back_pressure()

Prototype:	`set_enable_c_packet_with_back_pressure()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage of packet transaction with backpressure. It is disabled when either the `ready` signal or packet transmission is not supported.
Language support:	Verilog HDL

set_enable_c_packet_with_idles()

Prototype:	`set_enable_c_packet_with_idles()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage of packet transaction with idle cycles. It is disabled when packet transmission is not supported.
Language support:	Verilog HDL

set_enable_c_partial_valid_beats()

Prototype:	`set_enable_c_partial_valid_beats()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for number of transaction with partially valid beats.
Language support:	Verilog HDL

set_enable_c_single_packet_per_cycle()

Prototype:	`set_enable_c_single_packet_per_cycle()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for number of transactions that carry a single packet per cycle. It is disabled when packet transmission is not supported.
Language support:	Verilog HDL

set_enable_c_transfer()

Prototype:	`set_enable_c_transfer()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage of a `valid` signal is asserted correctly for different channels. It is disabled when the `ready` or `valid` signals are not supported.
Language support:	Verilog HDL

set_enable_c_transaction_after_reset()

Prototype:	`set_enable_c_transaction_after_reset()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for transaction on the first cycle after reset.
Language support:	Verilog HDL

set_enable_c_valid_non_ready()

Prototype:	`set_enable_c_valid_non_ready()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for `valid` signal when `ready` is deasserted. It is disabled when the `readyLatency` is greater than 0.
Language support:	Verilog HDL

Transaction Monitoring

Transaction monitoring is carried out through the transaction collector module. The transaction collector collects the transactions, encapsulates them into descriptors, and inserts the transactions into queue. The API provides the mechanism to query the transactions in queue and disposes them as they are processed. By default, the transaction collector module is disabled. You must define the `ENABLE_ALTERA_AVALON_TRANSACTION_RECORDING` Verilog macro to enable this feature. This macro is required to ensure backward compatibility and to avoid breaking existing test cases.

event_transaction_received()

Prototype:	`event_transaction_received()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that a transaction was received.
Language support:	VHDL

event_transaction_fifo_threshold()

Prototype:	`event_transaction_fifo_threshold()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the transaction FIFO exceeds the threshold level.
Language support:	VHDL

event_transaction_fifo_overflow()

Prototype:	`event_transaction_fifo_overflow()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Notifies the testbench that the transaction FIFO is full and transactions will be dropped.
Language support:	VHDL

get_transaction_channel()

Prototype:	`get_transaction_channel()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`STChannel_t`.
Description:	Returns the channel identifier for the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_transaction_data()

Prototype:	`get_transaction_data()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`STData_t`.
Description:	Returns the data in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_transaction_empty()

Prototype:	`get_transaction_empty()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`STEmpty_t`.
Description:	Returns the number of empty symbols in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_transaction_eop()

Prototype:	`get_transaction_eop()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit`.
Description:	Returns the transaction end of packet status in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_transaction_error()

Prototype:	`get_transaction_error()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`STError_t`.
Description:	Returns the error in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_transaction_fifo_max()

Prototype:	`int get_transaction_fifo_max()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`.
Description:	Gets the maximum transaction FIFO depth.
Language support:	Verilog HDL, VHDL

get_transaction_fifo_threshold()

Prototype:	`int get_transaction_fifo_threshold()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`.
Description:	Gets the transaction FIFO threshold level.
Language support:	Verilog HDL, VHDL

get_transaction_idles()

Prototype:	`get_transaction_idles()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit[31:0]`.
Description:	Returns the number of idle cycles in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_transaction_queue_size()

Prototype:	`get_transaction_queue_size()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`int`.
Description:	Returns the length of the queue holding received transactions.
Language support:	Verilog HDL, VHDL

get_transaction_sop()

Prototype:	`get_transaction_sop()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit`.
Description:	Returns the transaction start of packet status in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`String`.
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".
Language support:	Verilog HDL

pop_transaction()

Prototype:	`void pop_transaction()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Removes the transaction descriptor from the queue so that the testbench can query it with the `get_transaction` methods.
Language support:	Verilog HDL, VHDL

set_transaction_fifo_max()

Prototype:	`set_transaction_fifo_max()`
Arguments:	Verilog HDL: `int level` VHDL: `int level`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the maximum transaction level of the FIFO. The event `signal_transaction_fifo_max` is triggered when this level is exceeded.
Language support:	Verilog HDL, VHDL

set_transaction_fifo_threshold()

Prototype:	`set_transaction_fifo_threshold()`
Arguments:	Verilog HDL: `int level`
	VHDL: `int level`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the threshold alert level of the FIFO. The event `signal_transaction_fifo_threshold` is triggered when this level is exceeded.
Language support:	Verilog HDL, VHDL

signal_fatal_error

Prototype:	`signal_fatal_error`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a fatal error has occured in this module.
Language support:	Verilog HDL

signal_transaction_fifo_overflow

Prototype:	`signal_transaction_fifo_overflow`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that the FIFO is full and further transactions are dropped.
Language support:	Verilog HDL

signal_transaction_fifo_threshold

Prototype:	`signal_transaction_fifo_threshold`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that the transaction FIFO threshold level has exceeded.
Language support:	Verilog HDL

signal_transaction_received

Prototype:	`signal_transaction_received`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a transaction has been received and queued.
Language support:	Verilog HDL

Section V. Conduit and External Memory BFMs

This section provides information about conduit and external memory BFMs. This section includes the following chapters:

You can use Conduit BFMs to verify the following aspects of Avalon Conduit interfaces:

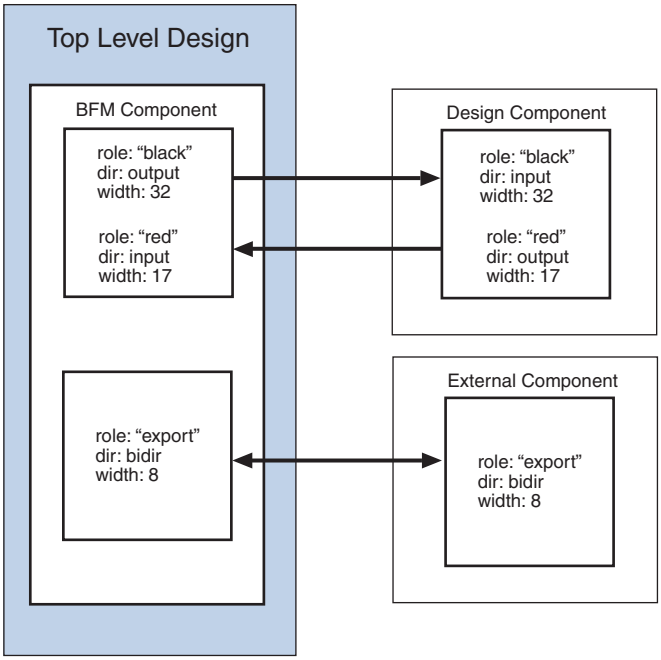
- Port compatibility and polarity
- Legal port widths

Conduit BFMs are only supported in Qsys.

Block Diagram

Figure 1–1 shows a block diagram of a Conduit BFM.

Figure 1–1. Conduit BFM Block Diagram



An Avalon Conduit interface can have an arbitrary number of ports. Each port can be an input, output, or bidirectional port. Legal port widths range from 1 through 1024 bits in size. Each port has an associated role name. This role name is an arbitrary string. Qsys uses these names to check for conduit interconnect compatibility between components. A connection is legal when two conduit interconnected components have the same port role names and complementary directions. For example, when an input connects with an output, the connection is legal. A port can also have a specific role named `export`. Ports with this role name are exported from the current system design module to the Conduit BFM module I/O.

A set of functions forming the API are used to construct or deconstruct transactions. Outgoing transactions are driven out on the physical conduit interface and vice versa.

At the beginning of the simulation, registers that store the data that is sent to the output ports are empty. The Conduit BFM drives 'x' to the output port until you rewrite the registers by calling the `set_<role name>` API. Initially, bidirectional ports work as input ports. You can change its functionality by calling the `set_<role name>_oe` API. The Conduit BFM prints out a message when the behavior of the bidirectional port changes from an input port to an output port and vice versa. Bidirectional ports drive register values to the interface when this API is set to 1. Otherwise, bidirectional ports work as input ports. You can call the `get_<role name>` API to obtain the value coming from the input and bidirectional ports.

Parameters

The Conduit BFM supports signals that interface to external memory devices, such as address, data, and control signals that have the signal type `export`.

For more information about Avalon Conduit interfaces supported in Qsys, refer to the *Avalon Interface Specifications (version 2.1)*.

Table 1–1 lists the parameter settings for the Conduit BFM.

Table 1–1. Conduit BFM Parameter Settings

Option	Default Value	Legal Values	Description
Role	—	Any string	Specifies the role name of each port.
Width	**1**	**1–1024**	Specifies the port width.
Direction	**input**	**input, output, bidir**	Specifies the direction of the signal.

Application Program Interface

This section describes the API for the Conduit BFM.

get_<*role name*>()

Prototype:	`int` <*role name port width*> `get_`<*role name*>`()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`int` <*role name port width*>
Description:	Returns interface signal value from the input/bidirectional port.
Language support:	Verilog HDL

get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`string`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".
Language support:	Verilog HDL

set_<*role name*>()

Prototype:	`void set_`<*role name*>`()`
Arguments:	Verilog HDL: `new_value` VHDL: N.A.
Returns:	`void`
Description:	Rewrites the registers inside the BFMs that are driven to the <*role name*> output ports.
Language support:	Verilog HDL

set_<*role name*>_oe()

Prototype:	`void set_`<*role name*>`_oe()`
Arguments:	Verilog HDL: bit enable VHDL: N.A.
Returns:	`void`
Description:	Enables the bidirectional ports when the value is set to 1.
Language support:	Verilog HDL

signal_input_<*role name*>_change

Prototype: `signal_input_<role name>_change`

Arguments: Verilog HDL: None

VHDL: N.A.

Returns: `void`

Description: Triggers when the input signal for a particular port changes its value. For a bidirectional port, this event is only triggered if its input value defers from its last input value.

Language support: Verilog HDL

You can use the Tri-State Conduit BFM to verify the following aspects of Avalon-TC interfaces:

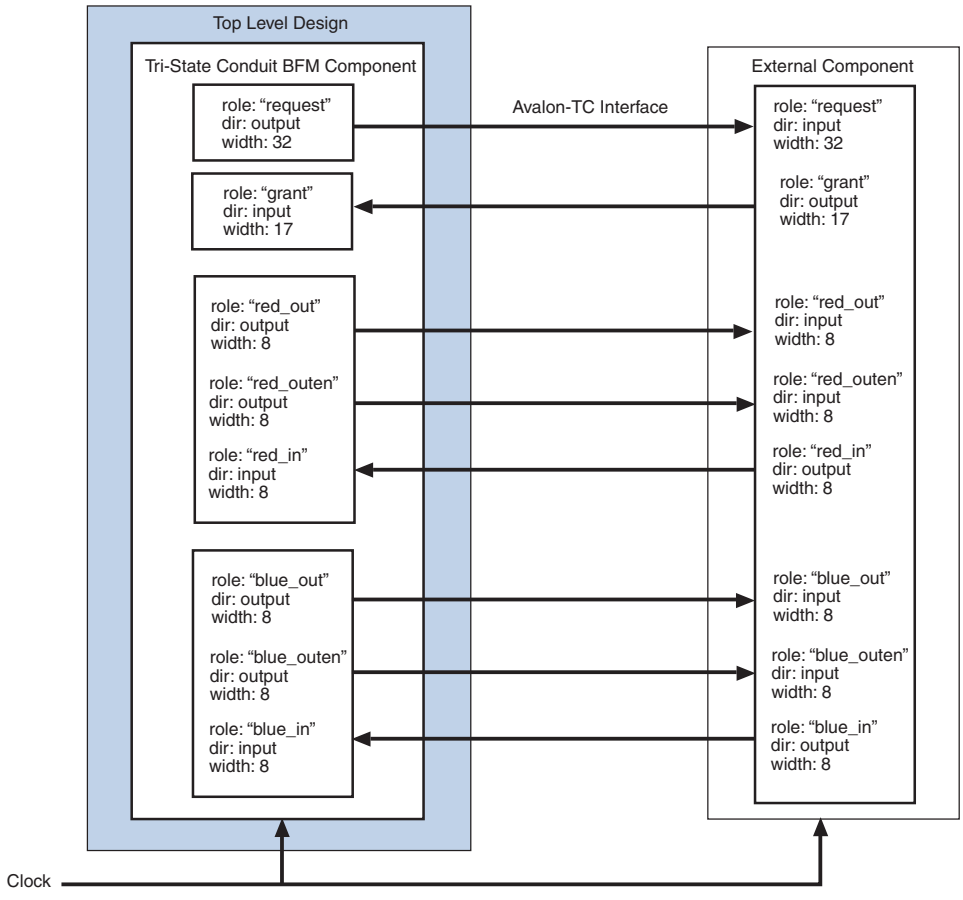
- Port compatibility and polarity
- Legal port widths

The Tri-State Conduit BFM is only supported in Qsys.

Block Diagram

Figure 2–1 shows a block diagram of a Tri-State Conduit BFM connected to an external component using an Avalon-TC interface.

Figure 2–1. Conduit BFM Block Diagram



An Avalon-TC interface can have an arbitrary number of ports. Each port has an associated role name. This role name is an arbitrary string. The difference between conduit interfaces and Avalon-TC interface is the way in which bidirectional ports are handled. In Avalon-TC interfaces, a bidirectional port is decomposed into three distinct unidirectional port signals with role names having the following suffixes:

- <*role name*>_in
- <*role name*>_out
- <*role name*>_outen

The set of bidirectional ports in the Avalon-TC interface are grouped together. The Avalon-TC interface also includes the request port, the grant port, and an associated clock. These request and grant signals are the control signals to and from the arbiter that controls access to the shared media.

The following port combinations are not legal:

- In and out roles (without a <*role name*>_outen role)
- In and outen roles (without a <*role name*>_out role)
- Only an outen role (without a <*role name*>_out role)

Parameters

The Tri-State Conduit BFM supports signals that interface to multiple external memory devices.

For more information about the Avalon-TC interface supported in Qsys, refer to the *Avalon Interface Specifications (version 2.1)*.

Table 2–1 lists the parameter settings for the Tri-State Conduit BFM.

Table 2–1. Tri-State Conduit BFM Parameter Settings

Option	Default Value	Legal Values	Description
Role	—	Any string	Specifies the role name of each port.
Width	**1**	**1–1024**	Specifies the port width.
USE_INPUT	**1**	**0** or **1**	Specifies an input port.
USE_OUTPUT	**1**	**0** or **1**	Specifies an output port.
USE_OUTPUTENABLE	**1**	**0** or **1**	Specifies an output enable port.
MAX_MULTIPLE_TRANSACTION	**1024**	—	Specifies the maximum transactions of data while request and grant signals are asserted. The value is constraint by the number of roles.

Application Program Interface

This section describes the API for the Tri-State Conduit BFM.

get_input_transaction_queue_size()

Prototype:	`int get_input_transaction_queue_size()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`int`
Description:	Returns the size of the queued input transaction in the BFM.
Language support:	Verilog HDL

get_output_transaction_queue_size()

Prototype:	`int get_output_transaction_queue_size()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`int`
Description:	Returns the size of the queued output transaction in the BFM.
Language support:	Verilog HDL

get_transaction_*<role name>*_in()

Prototype:	`int` *<role name port width>* `get_transaction_`*<role name>*`_in()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`int` *<role name port width>*
Description:	Returns the interface signal value from the *<role name>*`_in` input ports.
Language support:	Verilog HDL

get_transaction_latency()

Prototype:	`int get_transaction_latency()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`int`
Description:	Returns the latency field value from the input transaction.
Language support:	Verilog HDL

get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`string`
Description:	Returns the BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".
Language support:	Verilog HDL

pop_transaction()

Prototype:	`void pop_transaction()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Returns the input transaction queued inside the BFM. A fatal error triggers if you remove a transaction from an empty queue.
Language support:	Verilog HDL

push_transaction()

Prototype:	`void push_transaction()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Registers an output transaction into the BFM. All registered output transactions are put into transaction queue. A fatal error triggers if you insert a transaction while the BFM is reset.
Language support:	Verilog HDL

set_max_transaction_queue_size()

Prototype:	`void set_max_transaction_queue_size(int size)`
Arguments:	Verilog HDL: `int size` VHDL: N.A.
Returns:	`void`
Description:	Sets the maximum size of the queued transactions. The BFM triggers an event when the queued transactions goes above the maximum size.
Language support:	Verilog HDL

set_min_transaction_queue_size()

Prototype: `int set_min_transaction_queue_size()`

Arguments: Verilog HDL: None

VHDL: N.A.

Returns: `void`

Description: Sets the minimum size of the queued transactions. The BFM triggers an event when the queued transactions falls below the minimum size.

Language support: Verilog HDL

set_num_of_transactions()

Prototype: `int set_num_of_transactions()`

Arguments: Verilog HDL: `int multiple_transaction_num`

VHDL: N.A.

Returns: `void`

Description: Sets the number of transactions to the DUT.

Language support: Verilog HDL

set_transaction_*<role name>*_out()

Prototype: `void set_transaction_`*<role name>*`_out()`

Arguments: Verilog HDL: `int index`

VHDL: N.A.

Returns: `void`

Description: Sets the value of the transaction to the *<role name>*`_out` output ports.

Language support: Verilog HDL

set_transaction_*<role name>*_outen()

Prototype: `string set_transaction_`*<role name>*`_outen()`

Arguments: Verilog HDL: `int index, bit outen`

VHDL: N.A.

Returns: `void`

Description: Sets the value of the transaction to the *<role name>*`_outen` output ports.

Language support: Verilog HDL

set_transaction_idles()

Prototype: `void set_transaction_idles()`
Arguments: Verilog HDL: `bit[31:0] idle_cycles`
VHDL: N.A.
Returns: `void`
Description: Sets the number of idle cycles that elapse before driving the out-going transaction.
Language support: Verilog HDL

set_valid_transaction_<*role name*>_out()

Prototype: `void set_valid_transaction_<`*role name*`>_out()`
Arguments: Verilog HDL: `int index`
VHDL: N.A.
Returns: `void`
Description: Sets the value of the valid transaction to the <*role name*>`_out` output port.
Language support: Verilog HDL

signal_all_transactions_complete

Prototype: `signal_all_transactions_complete`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Triggers when all the queued output and input transactions are completely retrieved.
Language support: Verilog HDL

signal_fatal_error

Prototype: `signal_fatal_error`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Notifies the testbench that a fatal error has occurred in this module.
Language support: Verilog HDL

signal_grant_deasserted_while_request_remain_asserted

Prototype:	`signal_grant_deasserted_while_request_remain_asserted`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Arguments:	None.
Returns:	`void`
Description:	Triggers when the grant signal changes value from high to low while the request signal remains asserted.
Language support:	Verilog HDL

signal_interface_granted

Prototype:	`signal_interface_granted`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Arguments:	None.
Returns:	`void`
Description:	Triggers when the grant signal is asserted.
Language support:	Verilog HDL

signal_max_transaction_queue_size

Prototype:	`signal_max_transaction_queue_size`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Arguments:	None.
Returns:	`void`
Description:	Triggers when the size of the pending queue exceeds the maximum size.
Language support:	Verilog HDL

signal_min_transaction_queue_size

Prototype:	`signal_min_transaction_queue_size`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Arguments:	None.
Returns:	`void`
Description:	Triggers when the size of the pending queue falls below the minimum size.
Language support:	Verilog HDL

You can use external memory BFMs to verify the following aspects of external memory interfaces:

- Read and write operations
- Memory initialization

External Memory BFMs are only supported in Qsys.

Functional Description

This section provides a functional description of the external memory BFM. It includes the following topics:

- "Block Diagram"
- "Initializing the Memory Content" on page 3–2
- "Reading and Writing to the Memory Content" on page 3–2

Block Diagram

Figure 3–1 shows a block diagram of how to use the external memory BFM with tri-state components.

Figure 3–1. Usage of External Memory BFM with Tri-State Components

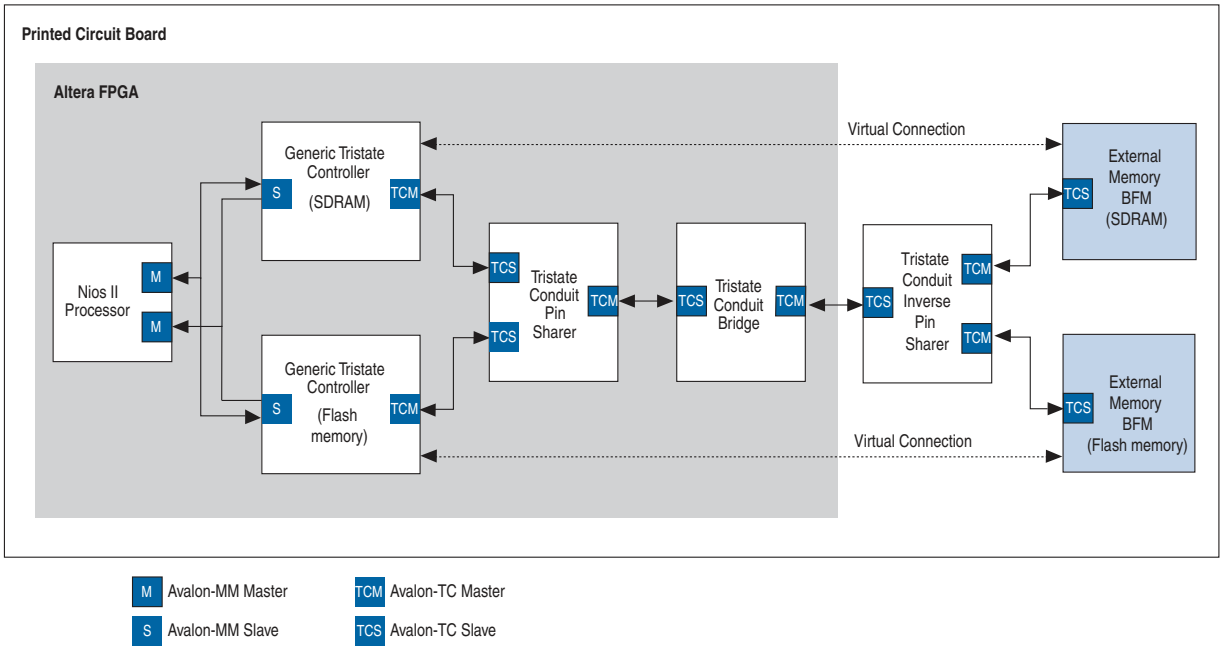


Table 3–2 lists the function of the external memory BFM and its related components.

Table 3–1. External Memory BFM and Related Components

Component	Description
External memory BFM	Represents the external RAM. The external memory BFM is a memory model with an Avalon-TC interface. The BFM also models a set of memories that are supported by the generic tri-state controller component.
Tri-State Conduit Bridge	Converts Avalon-TC signals into conduit signals.
Tri-State Conduit Pin Sharer	Carries the shared address bus and data.
Tri-State Conduit Inverse Pin Sharer	
Generic Tri-State Controller	Controls the external memory BFM. The generic tri-state controller accepts read and write requests and converts these requests into necessary SDRAM and bank management commands.

Refer to the following documents for more information:

- For more information about the tri-state conduit bridge, refer to *the "Tri-State Conduit Bridge"* section in the *Qsys Interconnect* chapter of the *Quartus II Handbook*.
- For more information about tri-state conduit pin sharer, refer to *"Tri-State Conduit Pin Sharer"* section in the *Qsys Interconnect* chapter of the *Quartus II Handbook*.
- For more information about generic tri-state controller, refer to *"Generic Tri-State Controller"* section in the *Qsys Interconnect* chapter of the *Quartus II Handbook*.

Initializing the Memory Content

At the beginning of the simulation, the external memory BFM loads the memory initialization file (`INIT_FILE`) to initialize its memory content. For example, if the memory file has a memory size of 50, the BFM fills its memory content with addresses 0–49. However, if you do not provide the memory initialization file, the memory content of the BFM remains blank.

Reading and Writing to the Memory Content

You can read or write to the memory content through the APIs or the interface signals.

Reading from the Memory

The BFM uses `cdt_data_io` as a bidirectional data port. During read transfers, this port acts as an output port and drives the corresponding address memory content when the BFM asserts or deasserts the following signals:

- Asserts `cdt_output_enable` signal
- Asserts `cdt_read` signal
- Deasserts `cdt_write` signal
- Asserts `cdt_chip_select` signal

Otherwise, the cdt_data_io port acts as an inactive input port and is held in high impedance state.

Writing to the Memory

The BFM overwrites its memory content when the BFM asserts the following signals:

- cdt_write signal
- cdt_chip_select signal

Parameters

Table 3–2 lists the parameter settings for the external memory BFM.

Table 3–2. External Memory BFM Parameter Settings (Part 1 of 2)

Option	Default Value	Legal Values	Description
Port Enables			
Use the byteenable signal	**On**	**On/Off**	When **On**, the interface includes a byteenable pin to enable specific byte lanes during transfer.
Use the chip select signal	**On**	**On/Off**	When **On**, the interface includes a chipselect pin. When present, the slave port ignores all Avalon-MM signals unless chipselect is asserted. chipselect is always present in combination with read or write.
Use the write signal	**On**	**On/Off**	When **On**, the interface includes a write pin that enables the write-request signal.
Use the read signal	**On**	**On/Off**	When **On**, the interface includes a read pin that enables the read-request signal.
Use the output enable signal	**On**	**On/Off**	When **On**, the interface includes an outputenable pin.
Use the begintransfer signal	**On**	**On/Off**	When **On**, the interface includes a begintransfer pin.
Use the reset input signal	**Off**	**On/Off**	When **On**, the interface includes a reset pin.
Use the active low byteenable signal	**Off**	**On/Off**	When **On**, the interface includes an active low byteenable pin.
Use the active low chipselect signal	**Off**	**On/Off**	When **On**, the interface includes an active low chipselect_n pin.
Use the active low write signal	**Off**	**On/Off**	When **On**, the interface includes an active low write_n pin.
Use the active low read signal	**Off**	**On/Off**	When **On**, the interface includes an active low read_n pin.
Use the active low outputenable signal	**Off**	**On/Off**	When **On**, the interface includes an active low outputenable_n pin.
Use the active low begintransfer signal	**Off**	**On/Off**	When **On**, the interface includes an active low begintransfer_n pin.
Use the active low reset signal	**Off**	**On/Off**	When **On**, the interface includes an active low reset_n pin.

Table 3–2. External Memory BFM Parameter Settings (Part 2 of 2)

Option	Default Value	Legal Values	Description
Interface Signals Name			
Address Role	cdt_address	—	Specifies the conduit interface role name that matches the role name on the external memory device.
Data Role	cdt_data_io	—	
Write Role	cdt_write	—	
Read Role	cdt_read	—	
Byteenable Role	cdt_byteenable	—	
Chip Select Role	cdt_chipselect	—	
Outputenable Role	cdt_outputenable	—	
Begintransfer Role	cdt_begintransfer	—	
Reset Role	cdt_reset	—	
Port Widths			
Address width	8	1–32	Specifies the address width in bits.
Symbol width	8	1–1024	Specifies the data symbol width in bits.
Number of symbols	4	1, 2, 4, 8, 16, 32, 64, 128	Specifies the number of symbols in a data.
Memory Contents			
Memory Initialization	altera_external_memory_bfm.hex	—	Specifies the file to initialize the memory content at the beginning of the simulation. The BFM supports only one memory file.
Interface Timing			
Read Latency of Interface	0	—	Specifies the read latency of the interface.
Miscellaneous Properties			
VHDL BFM ID	0	0–1023	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.

Application Program Interface

This section describes the API for the external memory BFM.

fill()

Prototype: `fill()`

Arguments:

Verilog HDL:	VHDL:
`logic[DATA_W-1:0] data`	`logic[DATA_W-1:0] data`
`bit[DATA_W-1:0] increment`	`bit[DATA_W-1:0] increment`
`bit[CDT_ADDRESS_W-1:0] address low`	`bit[CDT_ADDRESS_W-1:0] address low`
`bit[CDT_ADDRESS_W-1:0] address high`	`bit[CDT_ADDRESS_W-1:0] address high`
	`bfm_id`
	`req_if`

Returns: `void`

Description: Overwrites the memory content at the starting address specified by `address_low` until the ending address specified by `address_high`. The `data` field indicates the data value. The `increment` field indicates the data value increment from one address to the next address. For example, fill (`data[1]`, `increment[2]`, `address_low[10]`, `address_high[12]`) fills the memory as follows:

- `memory[address=10]` is filled with data value 1
- `memory[address=11]` is filled with data value 3
- `memory[address=12]` is filled with data value 5

Language support: Verilog HDL, VHDL

read()

Prototype: `read()`

Arguments: Verilog HDL: `bit[CDT_ADDRESS_W-1:0] address`

VHDL: `bit[CDT_ADDRESS_W-1:0] address`, `bfm_id`, `req_if`

Returns: `logic[DATA_W-1:0]`

Description: Retrieves the memory content from an address you specify.

Language support: Verilog HDL, VHDL

signal_api_call

Prototype: `signal_api_call`

Arguments: Verilog HDL: None

VHDL: N.A.

Returns: `void`

Description: Triggers when a client make an API call.

Language support: Verilog HDL

write()

Prototype:	`write()`	
Arguments:	Verilog HDL:	VHDL:
	`bit[CDT_ADDRESS_W-1:0] address`	`bit[CDT_ADDRESS_W-1:0] address`
	`logic[DATA_W-1:0] data`	`logic[DATA_W-1:0] data`
		`bfm_id`
		`req_if`
Returns:	`void`	
Description:	Overwrites the memory content at an address you specify.	
Language support:	Verilog HDL, VHDL	

Section VI. Nios II Custom Instruction BFMs

This section provides information about Nios II Custom Instruction Master and Slave BFMs. This section includes the following chapters:

- Chapter 1, Nios II Custom Instruction Master BFM
- Chapter 2, Nios II Custom Instruction Slave BFM

1. Nios II Custom Instruction Master BFM

You can use Nios II Custom Instruction Master BFM to verify the following aspects of the Nios II custom instruction master interface:

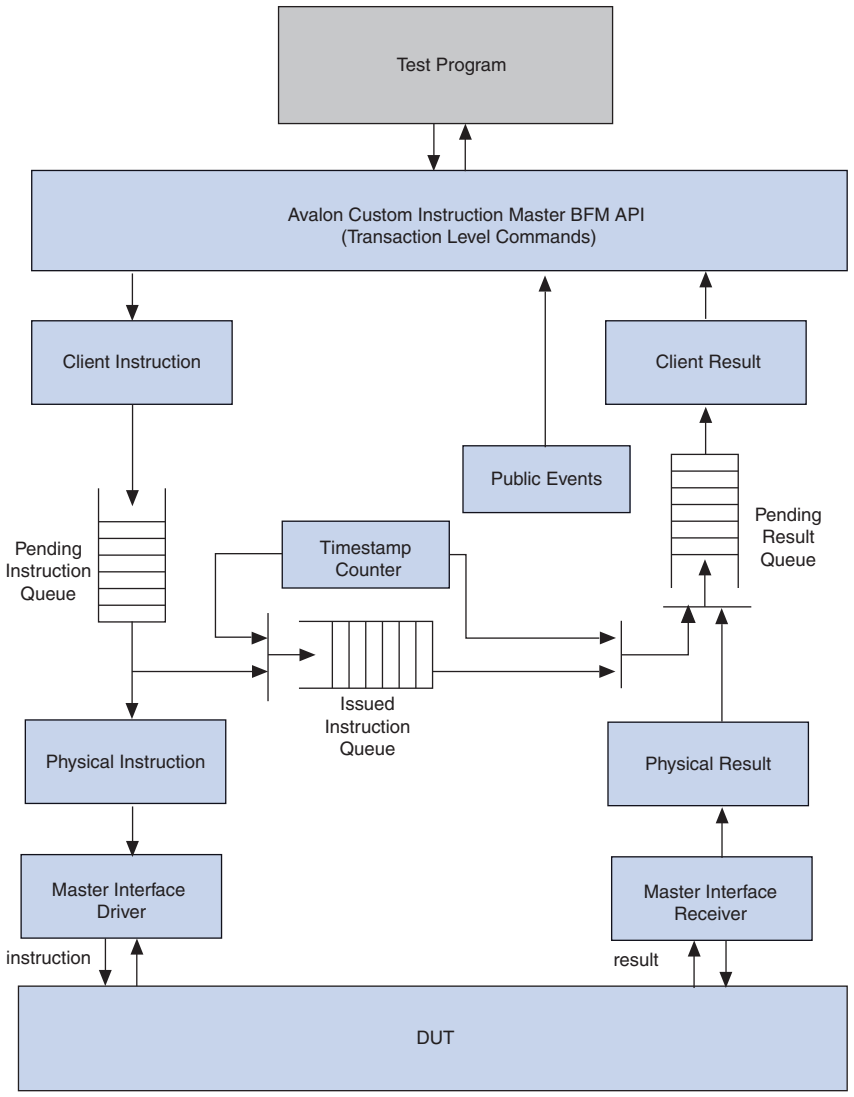
- Combinational and multicycle master custom instructions
- Extended instructions

The Nios II Custom Instruction Master BFM is only supported in Qsys.

Block Diagram

Figure 1–1 shows a block diagram of a Nios II Custom Instruction Master BFM.

Figure 1–1. Custom Instructions Master BFM Block Diagram



The Nios II Custom Instruction Master BFM uses queues to manage instructions. You can create instructions and push them into the instruction queue. The BFM then removes the instructions out one-by-one and drives them on the interface. You can insert the instructions simultaneously at the beginning of the simulation. If there is no instruction to execute, the BFM drives unknown (X), except on the `readra`, `readrb`, and `writerc` control ports which are driven high.

The result is sampled based on the driven instruction and inserted into a result queue. You can remove the result on an event basis, or at the end of the simulation.

Parameters

Table 1–1 lists the parameter settings for the custom instruction master BFM interface.

Table 1–1. Custom Instruction Master BFM Parameter Settings

Option	Default Value	Legal Values	Description
General			
Number of Operands to Use	**2**	**0,1,2**	Specifies the number of operands to use. 0: no operands are used 1: use dataa port only 2: use dataa and datab ports
Fixed Length for Multi-cycle Mode	**2**	—	Specifies the fixed length for multi-cycle mode.
Port Enables			
Use Result Port	**On**	**On/Off**	When **On**, the interface includes a `result` pin.
Use Multi-cycle Mode	**Off**	**On/Off**	When **On,** the interface can include a `start` pin, a `done` pin , both pins, or neither pins. The result returns in any of the following conditions: ■ With a `start` signal—Result returns together with an instruction. ■ Without a `start` signal—Result returns with instruction on the bus at every clock cycle. ■ With a `done` signal—Result returns at any time. ■ Without a `done` signal—Result returns at a fixed cycle.
Using start port	**On**	**On/Off**	When **On**, the interface includes a `start` pin.
Using done port	**On**	**On/Off**	When **On**, the interface includes a `done` pin.
Use Extended Port	**Off**	**On/Off**	When **On**, the interface includes a `n` pin.
Extended Port Width	**1**	—	Specifies the width of the extended `n` port.
Use Internal Register a	**Off**	**On/Off**	When **On**, the interface includes the `readra` and `a` pins.
Use Internal Register b	**Off**	**On/Off**	When **On**, the interface includes the `readrb` and `b` pins.
Use Internal Register c	**Off**	**On/Off**	When **On**, the interface includes the `readrc` and `c` pins.
Miscellaneous Properties			
VHDL BFM ID	**0**	**0–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.

Application Program Interface

This section describes the API for the Nios II Custom Instruction Master BFM.

event_instruction_start()

Prototype:	`event_instruction_start()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates the instruction to be driven to the interface.
Language support:	VHDL

event_result_received()

Prototype:	`event_result_received()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates that a result was received.
Language support:	VHDL

event_unexpected_result_received()

Prototype:	`event_unexpected_result_received()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates that a result was received without an instruction.
Language support:	VHDL

event_instructions_completed()

Prototype:	`event_instructions_completed()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates that all instructions were executed.
Language support:	VHDL

event_max_instruction_queue_size()

Prototype: `event_max_instruction_queue_size()`
Arguments: Verilog HDL: N.A.
VHDL: `bfm_id`, `req_if`
Returns: `void`
Description: Indicates that pending instructions exceed the maximum level.
Language support: VHDL

event_min_instruction_queue_size()

Prototype: `event_min_instruction_queue_size()`
Arguments: Verilog HDL: N.A.
VHDL: `bfm_id`, `req_if`
Returns: `void`
Description: Indicates that pending instructions are below the minimum level.
Language support: VHDL

event_max_result_queue_size()

Prototype: `event_max_result_queue_size()`
Arguments: Verilog HDL: N.A.
VHDL: `bfm_id`, `req_if`
Returns: `void`
Description: Indicates that the received result exceeds the maximum level.
Language support: VHDL

event_min_result_queue_size()

Prototype: `event_min_result_queue_size()`
Arguments: Verilog HDL: N.A.
VHDL: `bfm_id`, `req_if`
Returns: `void`
Description: Indicates that the received result is below the minimum level.
Language support: VHDL

get_instruction_queue_size()

Prototype: `int get_instruction_queue_size(int size)`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `int size`.

Description: Returns the number of instructions in the queue.

Language support: Verilog HDL, VHDL

get_result_delay()

Prototype: `int get_result_delay()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: Width of the data `(ci_data_t)` that can contain the following variables:

- `[Word_width-1:0]`
- `[Ext_width-1:0]`
- `[Addr_width-1:0]`

Description: Returns the result delay.

Language support: Verilog HDL, VHDL

get_result_queue_size()

Prototype: `int get_result_queue_size(int size)`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `int size`.

Description: Returns the number of results in the queue.

Language support: Verilog HDL, VHDL

get_result_value()

Prototype: `string get_result_value()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: Width of the data `(ci_data_t)` that can contain the following variables:

- `[Word_width-1:0]`
- `Ext_width-1:0]`
- `[Addr_width-1:0]`

Description: Returns the instruction result.

Language support: Verilog HDL, VHDL

get_version()

Prototype: `string get_version()`

Arguments: Verilog HDL: None

VHDL: N.A.

Returns: `string`

Description: Returns the BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".

Language support: Verilog HDL

insert_instruction()

Prototype: `void insert_instruction()`

Arguments:

Verilog HDL:	VHDL:
`ci_data_t dataa`	`ci_data_t dataa`
`ci_data_t datab`	`ci_data_t datab`
`ci_n_t n`	`ci_n_t n`
`ci_addr_t a`	`ci_addr_t a`
`ci_addr_t b`	`ci_addr_t b`
`ci_addr_t c`	`ci_addr_t c`
`logic readra`	`logic readra`
`logic readrb`	`logic readrb`
`logic writerc`	`logic writerc`
`ci_data_t idle`	`ci_data_t idle`
`int err_inj`	`int err_inj`
	`bfm_id`
	`req_if`

Returns: `void`

Description: A simplified API to set and push instructions.

Language support: Verilog HDL, VHDL

pop_result()

Prototype: `void pop_result()`

Arguments: Verilog HDL: None

VHDL: `bfm_id`, `req_if`

Returns: `void`.

Description: Removes the result instruction from the queue before querying the contents.

Language support: Verilog HDL, VHDL

push_instruction()

Prototype:	`void push_instruction()`
Arguments:	Verilog HDL: None
	VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Inserts a new instruction into the queue.
Language support:	Verilog HDL, VHDL

retrive_result()

Prototype:	`void retrive_result()`	
Arguments:	Verilog HDL:	VHDL:
	`output ci_data_t value.`	`output ci_data_t value.`
	`output ci_data_t delay.`	`output ci_data_t delay.`
		`bfm_id`
		`req_if`
Returns:	`void`	
Description:	A simplified API to remove and retrieve results.	
Language support:	Verilog HDL, VHDL	

set_ci_clk_en()

Prototype:	`void set_ci_clk_en()`
Arguments:	Verilog HDL: `bit enable`
	VHDL: `bit enable`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the `ci_clk_en` signal synchronously with the clock.
Language support:	Verilog HDL, VHDL

set_clock_enable_timeout()

Prototype:	`void set_clock_enable_timeout()`
Arguments:	Verilog HDL: `int timeout`
	VHDL: `int timeout`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the timeout value for the clock enable. Sets the value to `0` (zero)to disable timeouts.
Language support:	Verilog HDL, VHDL

set_instruction_a()

Prototype: `void set_instruction_a()`
Arguments: Verilog HDL: `ci_addr_t address`
VHDL: `ci_addr_t address`, `bfm_id`, `req_if`
Returns: `void`
Description: Sets the instruction register file address `a` value.
Language support: Verilog HDL, VHDL

set_instruction_b()

Prototype: `void set_instruction_b()`
Arguments: Verilog HDL: `ci_addr_t address`
VHDL: `ci_addr_t address`, `bfm_id`, `req_if`
Returns: `void`
Description: Sets the instruction register file address `b` value.
Language support: Verilog HDL, VHDL

set_instruction_c()

Prototype: `void set_instruction_c()`
Arguments: Verilog HDL: `ci_addr_t address`
VHDL: `ci_addr_t address`, `bfm_id`, `req_if`
Returns: `void`
Description: Sets the instruction register file address `c` value.
Language support: Verilog HDL, VHDL

set_instruction_dataa()

Prototype: `void set_instruction_dataa()`
Arguments: Verilog HDL: `ci_data_t data`
VHDL: `ci_data_t data`, `bfm_id`, `req_if`
Returns: `void`
Description: Sets the instruction `dataa` operand value.
Language support: Verilog HDL, VHDL

set_instruction_datab()

Prototype:	`void set_instruction_datab()`
Arguments:	Verilog HDL: `ci_data_t data`
	VHDL: `ci_data_t data`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the instruction `datab` operand value.
Language support:	Verilog HDL, VHDL

set_instruction_err_inject()

Prototype:	`void set_instruction_err_inject()`
Arguments:	Verilog HDL: `int err_inj`
	VHDL: `int err_inj`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the instruction to execute in pre-defined error.
Language support:	Verilog HDL, VHDL

set_instruction_idle()

Prototype:	`void set_instruction_idle()`
Arguments:	Verilog HDL: `ci_data_t idle`
	VHDL: `ci_data_t idle`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the instruction `idle` value.
Language support:	Verilog HDL, VHDL

set_instruction_n()

Prototype:	`void set_instruction_n()`
Arguments:	Verilog HDL: `ci_n_t code`
	VHDL: `ci_n_t code`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the instruction extended opcode value `n`.
Language support:	Verilog HDL, VHDL

set_instruction_readra()

Prototype: `void set_instruction_readra()`

Arguments: Verilog HDL: `logic enable`

VHDL: `logic enable`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the instruction register file read `a` value.

Language support: Verilog HDL, VHDL

set_instruction_readrb()

Prototype: `void set_instruction_readrb()`

Arguments: Verilog HDL: `logic enable`

VHDL: `logic enable`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the instruction register file read `b` value.

Language support: Verilog HDL, VHDL

set_instruction_timeout()

Prototype: `void set_instruction_timeout()`

Arguments: Verilog HDL: `int timeout`

VHDL: `int timeout`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the timeout value for an instruction. Sets the value to `0` (zero) to disable the timeout.

Language support: Verilog HDL, VHDL

set_instruction_writerc()

Prototype: `void set_instruction_writerc()`

Arguments: Verilog HDL: `logic enable`

VHDL: `logic enable`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the instruction register file write `c` value.

Language support: Verilog HDL, VHDL

set_max_instruction_queue_size()

Prototype:	`void set_max_instruction_queue_size(int size).`
Arguments:	Verilog HDL: `int size` VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the pending instruction queue size maximum threshold.
Language support:	Verilog HDL, VHDL

set_max_result_queue_size()

Prototype:	`void set_max_result_queue_size(int size).`
Arguments:	Verilog HDL: `int size` VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the pending result queue size maximum threshold.
Language support:	Verilog HDL, VHDL

set_min_instruction_queue_size()

Prototype:	`void set_min_instruction_queue_size(int size).`
Arguments:	Verilog HDL: `int size` VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the pending instruction queue size minimum threshold.
Language support:	Verilog HDL, VHDL

set_min_result_queue_size()

Prototype:	`void set_min_result_queue_size(int size).`
Arguments:	Verilog HDL: `int size` VHDL: `int size`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the pending result queue size minimum threshold.
Language support:	Verilog HDL, VHDL

set_result_timeout()

Prototype:	`void set_result_timeout()`
Arguments:	Verilog HDL: `int timeout`
	VHDL: `int timeout`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the timeout value for a result. Set the value to `0` to disable timeout.
Language support:	Verilog HDL, VHDL

signal_unexpected_result_received

Prototype:	`signal_unexpected_result_received`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Signals that a result has been received without an instruction.
Language support:	Verilog HDL

signal_fatal_error

Prototype:	`signal_fatal_error`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a fatal error has occured in this module.
Language support:	Verilog HDL

signal_instructions_completed

Prototype:	`signal_instructions_completed`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Signals that all instructions in the BFM has been executed.
Language support:	Verilog HDL

signal_instruction_start

Prototype: `signal_instruction_start`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Signals that an instruction has been driven to the interface.
Language support: Verilog HDL

signal_max_instruction_queue_size

Prototype: `signal_max_instruction_queue_size`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Signals that the maximum pending instruction queue size threshold has been exceeded.
Language support: Verilog HDL

signal_max_result_queue_size

Prototype: `signal_max_result_queue_size`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Signals that the maximum pending result queue size threshold has been exceeded.
Language support: Verilog HDL

signal_min_instruction_queue_size

Prototype: `signal_min_instruction_queue_size`
Arguments: Verilog HDL: None
VHDL: N.A.
Returns: `void`
Description: Signals that the pending instruction queue size is below the minimum threshold.
Language support: Verilog HDL

signal_min_result_queue_size

Prototype:	`signal_min_result_queue_size`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that the pending result queue size is below the minimum threshold.
Language support:	Verilog HDL

signal_result_received

Prototype:	`signal_result_received`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that a result has been received.
Language support:	Verilog HDL

2. Nios II Custom Instruction Slave BFM

You can use Nios II Custom Instruction Slave BFM to verify the following aspects of the Nios II custom instruction slave interface:

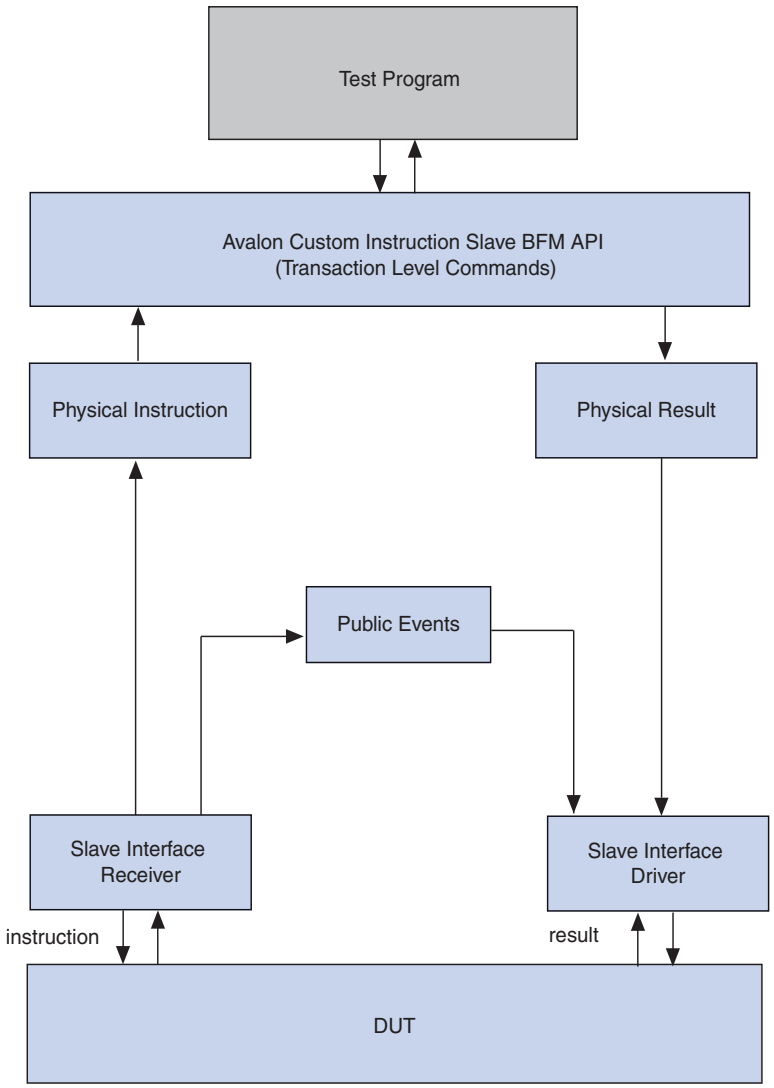
- Combinational and multicycle slave custom instructions
- Extended instructions

The Nios II Custom Instruction Slave BFM is only supported in Qsys.

Block Diagram

Figure 2–1 shows a block diagram of a Nios II Custom Instruction Slave BFM.

Figure 2–1. Custom Instructions Slave BFM Block Diagram



The Nios II Custom Instruction Slave BFM does not use queues to manage the instructions or results. Without queues, the BFM uses events to retrieve the instructions and to drive results. This method allows greater flexibility in controlling the output result (for example, driving a result when the interface is unknown). If there is an instruction and you do not provide the result, the BFM drives the old result onto the interface. If there is no instruction at all, the BFM drives unknown (X) on the interface.

Parameters

Table 2–1 lists the parameter settings for the custom instruction slave BFM interface.

Table 2–1. Custom Instruction Slave BFM Parameter Settings

Option	Default Value	Legal Values	Description
General			
Number of Operands to Use	**2**	**0,1,2**	Specifies the number of operands to use. 0: no operands are used. 1: use dataa port only. 2: use dataa and datab ports.
Fixed Length for Multi-cycle Mode	**2**	**—**	Specifies the fixed length for multi-cycle mode.
Port Enables			
Use Result Port	**On**	**On/Off**	When **On**, the interface includes a `result` pin.
Use Multi-cycle Mode	**Off**	**On/Off**	When **On,** the interface can include a `start` pin, a `done` pin, both pins, or neither pins. The result returns in any of the following conditions: ■ With a `start` signal—Result returns together with an instruction. ■ Without a `start` signal—Result returns with instruction on the bus at every clock cycle. ■ With a `done` signal—Result returns at any time. ■ Without a `done` signal—Result returns at a fixed cycle.
Using start port	**On**	**On/Off**	When **On**, the interface includes a `start` pin.
Using done port	**On**	**On/Off**	When **On**, the interface includes a `done` pin.
Use Extended Port	**Off**	**On/Off**	When **On**, the interface includes a `n` pin.
Extended Port Width	**1**	**—**	Specifies the width of the extended `n` port.
Use Internal Register a	**Off**	**On/Off**	When **On**, the interface includes the `readra` and `a` pins.
Use Internal Register b	**Off**	**On/Off**	When **On**, the interface includes the `readrb` and `b` pins.
Use Internal Register c	**Off**	**On/Off**	When **On**, the interface includes the `readrc` and `c` pins.
Miscellaneous Properties			
VHDL BFM ID	**0**	**0–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.

Application Program Interface

This section describes the API for the Nios II custom instruction slave BFM.

event_known_instruction_received()

Prototype:	`event_known_instruction_received()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates that a change occurred on the instruction interface and there is no unknown value.
Language support:	VHDL

event_instruction_inconsistent()

Prototype:	`event_instruction_inconsistent()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates that an instruction changed before the previous instruction finished.
Language support:	VHDL

event_instruction_unchanged()

Prototype:	`event_instruction_unchanged()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates that an instruction sampled on the interface has not changed from the previous instruction.
Language support:	VHDL

event_result_driven()

Prototype:	`event_result_driven()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates that the result will be driven out from the slave.
Language support:	VHDL

event_result_done()

Prototype:	`event_result_done()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates that the master accepted the result.
Language support:	VHDL

event_unknown_instruction_received()

Prototype:	`event_unknown_instruction_received()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`
Returns:	`void`
Description:	Indicates that a change occurred on the instruction interface and there is an unknown value.
Language support:	VHDL

get_ci_clk_en()

Prototype:	`void get_ci_clk_en(bit enable)`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`bit enable`
Description:	Retrieves the clock enable signal.
Language support:	Verilog HDL, VHDL

get_instruction_a()

Prototype:	`string get_instruction_a()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`ci_addr_t`
Description:	Retrieves the instruction register file address `a` value.
Language support:	Verilog HDL, VHDL

get_instruction_b()

Prototype:	`string get_instruction_b()`
Arguments:	Verilog HDL: None
	VHDL: `bfm_id`, `req_if`
Returns:	`ci_addr_t`
Description:	Retrieves the instruction register file address `b` value.
Language support:	Verilog HDL, VHDL

get_instruction_c()

Prototype:	`string get_instruction_c()`
Arguments:	Verilog HDL: None
	VHDL: `bfm_id`, `req_if`
Returns:	`ci_addr_t`
Description:	Retrieves the instruction register file address `c` value.
Language support:	Verilog HDL, VHDL

get_instruction_dataa()

Prototype:	`void get_instruction_dataa()`
Arguments:	Verilog HDL: None
	VHDL: `bfm_id`, `req_if`
Returns:	`ci_data_t data`
Description:	Retrieves the instruction `dataa` operand value.
Language support:	Verilog HDL, VHDL

get_instruction_datab()

Prototype:	`void get_instruction_datab()`
Arguments:	Verilog HDL: None
	VHDL: `bfm_id`, `req_if`
Returns:	`ci_data_t data`
Description:	Retrieves the instruction `datab` operand value.
Language support:	Verilog HDL, VHDL

get_instruction_idle()

Prototype:	`void get_instruction_idle()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`ci_data_t`
Description:	Retrieves the pre-instruction `idle` value.
Language support:	Verilog HDL, VHDL

get_instruction_n()

Prototype:	`void get_instruction_n()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`ci_n_t`
Description:	Retrieves the instruction extended opcode value `n`.
Language support:	Verilog HDL, VHDL

get_instruction_readra()

Prototype:	`logic get_instruction_readra()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`logic`
Description:	Retrieves the instruction register file read `a` value.
Language support:	Verilog HDL, VHDL

get_instruction_readrb()

Prototype:	`logic get_instruction_readrb()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`logic`
Description:	Retrieves the instruction register file read `b` value.
Language support:	Verilog HDL, VHDL

get_instruction_writerc()

Prototype:	`logic get_instruction_writerc()`
Arguments:	Verilog HDL: None VHDL: `bfm_id`, `req_if`
Returns:	`logic`
Description:	Retrieves the instruction register file write `c` value.
Language support:	Verilog HDL, VHDL

get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`string`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".
Language support:	Verilog HDL

insert_result()

Prototype: `void insert_result()`

Arguments:	Verilog HDL:	VHDL:
	`ci_data_t value`	`ci_data_t value`
	`ci_data_t delay`	`ci_data_t delay`
	`int err_inj`	`int err_inj`
		`bfm_id`
		`req_if`

Returns:	`void`
Description:	A simplified API to set results.
Language support:	Verilog HDL, VHDL

retrieve_instruction()

Prototype: `void retrieve_instruction.`

Arguments:

Verilog HDL:	VHDL:
`output ci_data_t dataa`	`output ci_data_t dataa`
`output ci_data_t datab`	`output ci_data_t datab`
`output ci_n_t n`	`output ci_n_t n`
`output ci_addr_t a`	`output ci_addr_t a`
`output ci_addr_t b`	`output ci_addr_t b`
`output ci_addr_t c`	`output ci_addr_t c`
`output logic readra`	`output logic readra`
`output logic readrb`	`output logic readrb`
`output logic writerc`	`output logic writerc`
`output ci_data_t idle`	`output ci_data_t idle`
	`bfm_id`
	`req_if`

Returns: `void`

Description: A simplified API to retrieve instruction.

Language support: Verilog HDL, VHDL

set_clock_enable_timeout()

Prototype: `void set_clock_enable_timeout()`

Arguments: Verilog HDL: `int timeout`

VHDL: `int timeout`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the timeout value for the clock enable. Set the value to `0` to disable timeout.

Language support: Verilog HDL, VHDL

set_instruction_a()

Prototype: `void set_instruction_a()`

Arguments: Verilog HDL: `ci_addr_t address`

VHDL: `ci_addr_t address`, `bfm_id`, `req_if`

Returns: `void`

Description: Sets the instruction register file address `a` value.

Language support: Verilog HDL, VHDL

set_instruction_b()

Prototype:	`void set_instruction_b()`
Arguments:	Verilog HDL: `ci_addr_t address`
	VHDL: `ci_addr_t address`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the instruction register file address `b` value.
Language support:	Verilog HDL, VHDL

set_instruction_c()

Prototype:	`void set_instruction_c()`
Arguments:	Verilog HDL: `ci_addr_t address`
	VHDL: `ci_addr_t address`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the instruction register file address `c` value.
Language support:	Verilog HDL, VHDL

set_instruction_timeout()

Prototype:	`void set_instruction_timeout()`
Arguments:	Verilog HDL: `int timeout`
	VHDL: `int timeout`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the timeout value for an instruction. Set the value to `0` to disable timeouts.
Language support:	Verilog HDL, VHDL

set_result_delay()

Prototype:	`void set_result_delay()`
Arguments:	Verilog HDL: `ci_data_t delay`
	VHDL: `ci_data_t delay`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the instruction result delay.
Language support:	Verilog HDL, VHDL

set_result_err_inject()

Prototype:	`void set_result_err_inject()`
Arguments:	Verilog HDL: `int err_inj`
	VHDL: `int err_inj`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the instruction result to execute in pre-defined error.
Language support:	Verilog HDL, VHDL

set_result_value()

Prototype:	`void set_result_value()`
Arguments:	Verilog HDL: `ci_data_t value`
	VHDL: `ci_data_t value`, `bfm_id`, `req_if`
Returns:	`void`
Description:	Sets the instruction result.
Language support:	Verilog HDL, VHDL

signal_fatal_error

Prototype:	`signal_fatal_error`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a fatal error has occured in this module.
Language support:	Verilog HDL

signal_instructions_inconsistent

Prototype:	`signal_instructions_inconsistent`
Arguments:	Verilog HDL: None
	VHDL: N.A.
Returns:	`void`
Description:	Signals that an instruction has changed while the previous instruction has not completed.
Language support:	Verilog HDL

signal_known_instruction_received

Prototype:	`signal_known_instruction_received`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that a change has occured on the instruction interface and there is no unknown value.
Language support:	Verilog HDL

signal_result_done

Prototype:	`signal_result_done`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that a result has been received by the master.
Language support:	Verilog HDL

signal_result_driven

Prototype:	`signal_result_driven`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that a result has been driven from the slave interface.
Language support:	Verilog HDL

signal_unknown_instruction_received

Prototype:	`signal_unknown_instruction_received`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that a change has occured on the instruction interface and there is an unknown value.
Language support:	Verilog HDL

Section VII. Tutorials

This section describes how to use the Avalon Verification IP. This section includes the following chapters:

- Chapter 1, Qsys Tutorial
- Chapter 2, Using the VHDL BFMs

1. Qsys Tutorial

This chapter demonstrates how to use the Avalon-ST Source and Sink BFMs to verify the functionality of an Avalon-ST component using a Qsys-generated testbench. In this example, the Avalon-ST Single-Clock FIFO buffer is the DUT. The testbench includes both the Avalon-ST Source and Sink BFMs to verify the DUT behavior.

Software Requirements

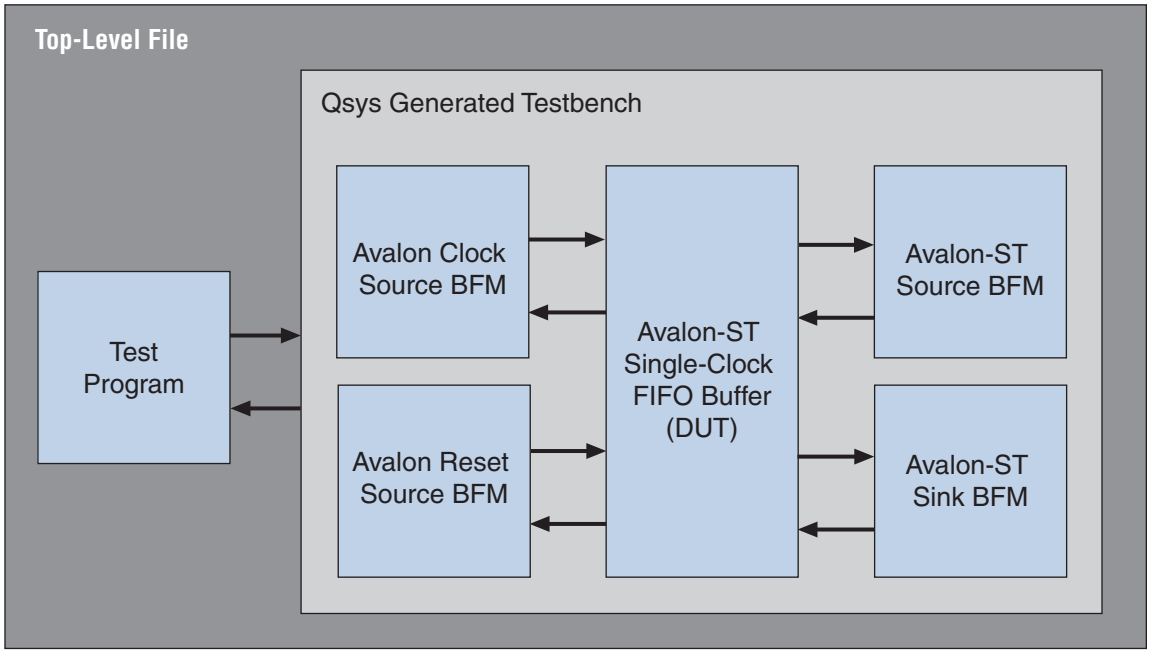
The following software and file are required to run the test:

- Quartus II software, version 12.0 or higher.
- ModelSim-AE software that you installed with the Quartus II software.
- The **ug_avalon_verification.zip** file. This design example file is available for download at www.altera.com/literature/ug/ug_avalon_verification.zip.

Verifying Avalon-ST DUT

Figure 1–1 shows the test setup to verify the Avalon-ST Single-Clock FIFO buffer using the Avalon-ST Source and Sink BFMs. The Avalon Clock Source and Reset Source BFMs provide clock and reset functions to the DUT. The Avalon-ST Source BFM connects to the DUT and drives transactions. The Avalon-ST Sink BFM monitors transactions from the Avalon-ST Single-Clock FIFO buffer. The test program controls the BFMs using the BFM API to drive and monitor transactions.

Figure 1–1. Top-Level Testbench for Avalon-ST DUT Component



The test flow includes the following steps:

1. The test program initializes the BFMs.
2. The test program runs the following three parallel processes:
   a. Creates and sends four test transactions to the source BFM. The transactions consists of six Avalon-ST signals—`data`, `channel`, `error`, `empty`, `startofpacket`, `endofpacket`, and a BFM-related parameter, `idle`. The Avalon-ST Source BFM drives the transactions to the Avalon-ST Single-Clock FIFO buffer. In addition, the Avalon-ST Source BFM keeps a local copy of the transactions for future reference, and prints the transaction values in the ModelSim transcript console.
   b. Controls the Avalon-ST Sink BFM. When the Avalon-ST Sink BFM receives a transaction, the Avalon-ST Sink BFM reads the transaction values, prints the transaction values on the ModelSim transcript console, and compares the values it receives to the values from the Avalon-ST Source BFM. The Avalon-ST Sink BFM reports any mismatch in values as failures. During this process, the Avalon-ST Sink BFM backpressures the Avalon-ST Single-Clock FIFO buffer.
   c. Measures the response latency when the Avalon-ST Single-Clock FIFO buffer backpressures the Avalon-ST Source BFM. The Avalon-ST Source BFM prints the transaction values on the ModelSim transcript console.
3. The parallel processes terminate when the Avalon-ST Source and Sink BFM transaction queues are empty and all four transactions are complete.
4. The test program prints a pass or fail message in the ModelSim transcript console. The test passes if all of the transactions that the Avalon-ST Source BFM sends to the Avalon-ST Single-Clock FIFO buffer match the transactions that the Avalon-ST Sink BFM receives from the Avalon-ST Single-Clock FIFO buffer.

Setting up the Test

In this section you generate a testbench system in Qsys for the DUT.

Creating a Qsys System for the DUT

Before you run the design file, unzip the **ug_avalon_verification.zip** file to a working directory on your hard drive. This location is referred to as <*working_directory*>.

1. On the Windows Start menu, point to **All Programs**, then **Altera**, and click **Quartus II**><*version number*> to run the Quartus II software.
2. On the File menu, click **Open**. Select **st_bfm_project.qpf** located in <*working_directory*>**\ug_avalon_verification\qsys**.
3. On the Tools menu, click **Qsys**.
4. When prompted to open a file, select **st_bfm_qsys_tutorial.qsys**, and click **Open** to open the blank Qsys system provided.
5. Type `fifo` in the search box located in the **Component Library** panel. From the search results, double-click on the **Avalon-ST Single Clock FIFO** component.

6. In the parameter editor, change the parameter values to match the values listed in Table 1–1.

Table 1–1. Avalon-ST Single Clock FIFO Parameter Values

Parameters	Value
Symbols per beat	4
Bits per symbol	8
FIFO depth	2
Channel width	3
Error width	3
Use packets	On
Use fill level	Off
Use store and forward	Off
Use almost full status	Off
Use almost empty status	Off

7. Click **Finish**.
8. Right-click on the `sc_fifo_0` component and select **Rename**. Rename the component to `dut`.
9. On the **System Contents** tab, in the **Export** column, rename the exported interface names to match the names listed in Table 1–2.

Table 1–2. Avalon-ST Single Clock FIFO Exported Interface Names

Interface Name	Description	Export Name
clk	Clock Input	clk
clk_reset	Reset Input	reset
in	Avalon Streaming Sink	st_in
out	Avalon Streaming Source	st_out

Generating a Qsys Testbench System

Follow these steps to generate a testbench system for the DUT:

1. On the **Generation** tab, change the parameter values to match the values listed in Table 1–3.

Table 1–3. Generation Tab Parameter Values (Part 1 of 2)

Parameters	Value
Simulation	
Create simulation model	None
Create testbench Qsys system	Standard, BFMs for standard Avalon Interfaces
Create testbench simulation model	Verilog
Synthesis	
Create HDL design files for synthesis	Turned off
Create block symbol file (.bsf)	Turned off

Table 1–3. Generation Tab Parameter Values (Part 2 of 2)

Parameters	Value
Output Directory	
Path	*<working_directory>*\ug_avalon_verification\qsys\st_bfm_qsys_tutorial

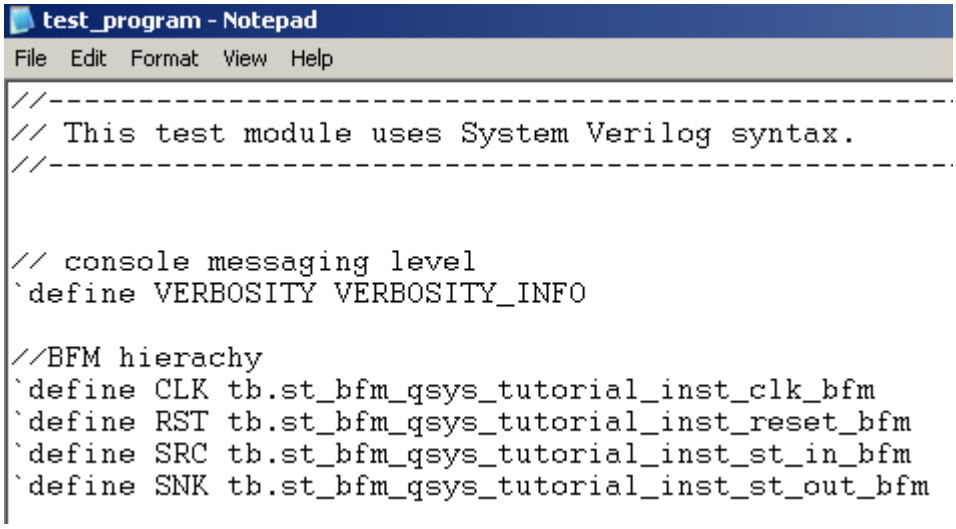
2. Click **Generate**. Save the system if you are prompted to do so. Do not close the Qsys window after successful generation.
3. To view information about the generated testbench file, open **st_bfm_qsys_tutorial_tb.html** located in the following directory: *<working_directory>***ug_avalon_verification\qsys\st_bfm_qsys_tutorial\testbench**.
4. In the **st_bfm_qsys_tutorial_tb.html** file, verify that the names of the generated BFMs match the instance names in Table 1–4.

Table 1–4. Generated BFM Instance Names

BFM Type	Instance Name
altera_avalon_clock_source	st_bfm_qsys_tutorial_inst_clk_bfm
altera_avalon_reset_source	st_bfm_qsys_tutorial_inst_reset_bfm
altera_avalon_st_source_bfm	st_bfm_qsys_tutorial_inst_st_in_bfm
altera_avalon_st_sink_bfm	st_bfm_qsys_tutorial_inst_st_out_bfm

5. Use the instance names listed in Table 1–4 to define and access the APIs of the corresponding BFMs in your test program. Figure 1–2 shows a code example that uses instance names to define a particular BFM in the test program.

Figure 1–2. Using Instance Names to Define BFMs



The test program for this tutorial is located in *<working_directory>***ug_avalon_verification\qsys\user_test_program**.

Setting up the Simulation Environment

To set up the simulation environment for your test program, open your ModelSim script file (**.tcl** or **.do**) and set the hierarchy variables used in the Qsys-generated simulation script (**msim_setup.tcl**). The ModelSim script file (**load_sim.tcl**) included with this tutorial has the correct hierarchy variable settings. However, if you would like to know how to set up the correct hierarchical variables used in the Qsys-generated simulation model, refer to Table 1–5 for the coding examples.

Table 1–5. Coding Examples to Set Hierarchy Variables

Hierarchy Variables Coding Example	Description
`set TOP_LEVEL_NAME "top"`	Sets the name of the top level file that instantiates the Qsys-generated testbench system and the test program.
`set QSYS_SIMDIR"../st_bfm_qsys_tutorial/testbench`	Sets the Qsys simulation path to the directory that includes the ModelSim script. You must set this path when your ModelSim script file (**msim_setup.tcl**) and test program are located in different directories.

The hierarchy variables enable the ModelSim script to source the **msim_setup.tcl** and use the command aliases defined in the Qsys-generated simulation script to compile the device library files and SystemVerilog design files (**test_program.sv** and **top.sv**) that instantiate the test program and the Qsys-generated testbench simulation model. The ModelSim script (**load_sim.tcl**) then uses the command alias to elaborate the top-level simulation design and loads the **wave.do** file that sets up the waveform view in the ModelSim-Altera software.

Running the Simulation

In this section, you run a simulation in the ModelSim-Altera software on the testbench that you created. To complete this simulation, use the test program provided in the design files to provide the stimulus. By default, **msim_setup.tcl** compiles the BFM source files into different libraries. In this tutorial, the BFM source files must be in a single library.

Complete the following steps to compile the source files to a single directory:

1. In Qsys, on the **Tools** menu click **Nios II Command Shell**.
2. In Nios II Command Shell, change the directory to *<working_directory>***\ug_avalon_verification\qsys**
3. Type the following command and hit enter:

```
ip-make-simscript --spd=st_bfm_qsys_tutorial_tb.spd --output-
directory=./st_bfm_qsys_tutorial/testbench/ --compile-to-work
```

To run the simulation, follow these steps:

1. Start the ModelSim-Altera software.
2. On the File menu click **Change Directory**.

3. Navigate to *<working_directory>***ug_avalon_verification****qsys****user_test_program** directory, and click **OK**.
4. On the Compile menu, click **Compile Options**.
5. Click the **Verilog & System Verilog** tab.
6. In the **Language Syntax** box, select **Use SystemVerilog** and click **OK**.
7. On the File menu, click **Load**.

   ☞ Ensure you activate your cursor on the ModelSim-Altera Transcript window, otherwise the **Load** function is disabled.

8. Select **load_sim.tcl,** and click **Open**. The Tcl file creates a new working library, compiles all source files, runs simulation, and loads signals into the ModelSim waveform viewer.
9. To run the simulation, type the following command in the ModelSim-Altera transcript console:

   ```
   run 1200 ns ↵
   ```

   ☞ You can run the `h` command to show the available options for the **msim_setup.tcl** macro script.

Observing the Results

You can view the simulation results in the following two ways:

- In the ModelSim transcript console
- In the waveforms window

Example 1–1 shows an extract of the simulation results.

Example 1–1. Extract of the Simulation Results in the ModelSim Transcript Console

```
# 990000: INFO: top.tb.st_bfm_qsys_tutorial_inst_reset_bfm.reset_deassert: Reset
deasserted
# 990000: INFO: top.pgm.print_transaction: Source BFM: Send transaction 0
# 990000: INFO: top.pgm.print_transaction:   Data:  0
# 990000: INFO: top.pgm.print_transaction:   Idles: 0
# 990000: INFO: top.pgm.print_transaction:   SOP:   1
# 990000: INFO: top.pgm.print_transaction:   EOP:   0
# 990000: INFO: top.pgm.print_transaction:   Channel: 0
# 990000: INFO: top.pgm.print_transaction:   Error:   0
# 990000: INFO: top.pgm.print_transaction:   Empty:   0
# 990000: INFO: top.pgm.print_transaction: Source BFM: Send transaction 1
# 990000: INFO: top.pgm.print_transaction:   Data:  1
# 990000: INFO: top.pgm.print_transaction:   Idles: 0
# 990000: INFO: top.pgm.print_transaction:   SOP:   0
# 990000: INFO: top.pgm.print_transaction:   EOP:   0
# 990000: INFO: top.pgm.print_transaction:   Channel: 0
# 990000: INFO: top.pgm.print_transaction:   Error:   0
# 990000: INFO: top.pgm.print_transaction:   Empty:   0
# 990000: INFO: top.pgm.print_transaction: Source BFM: Send transaction 2
# 990000: INFO: top.pgm.print_transaction:   Data:  2
# 990000: INFO: top.pgm.print_transaction:   Idles: 0
# 990000: INFO: top.pgm.print_transaction:   SOP:   0
# 990000: INFO: top.pgm.print_transaction:   EOP:   0
# 990000: INFO: top.pgm.print_transaction:   Channel: 0
# 990000: INFO: top.pgm.print_transaction:   Error:   0
# 990000: INFO: top.pgm.print_transaction:   Empty:   0
# 990000: INFO: top.pgm.print_transaction: Source BFM: Send transaction 3
# 990000: INFO: top.pgm.print_transaction:   Data:  3
# 990000: INFO: top.pgm.print_transaction:   Idles: 0
# 990000: INFO: top.pgm.print_transaction:   SOP:   0
# 990000: INFO: top.pgm.print_transaction:   EOP:   1
# 990000: INFO: top.pgm.print_transaction:   Channel: 0
# 990000: INFO: top.pgm.print_transaction:   Error:   0
# 990000: INFO: top.pgm.print_transaction:   Empty:   0
# 1030000: INFO: top.pgm.test_threads.source_response_thread: Source response latency 0
# 1050000: INFO: top.pgm.test_threads.source_response_thread: Source response latency 0
# 1090000: INFO: top.pgm.test_threads.source_response_thread: Source response latency 1
# 1090000: INFO: top.pgm.print_transaction: Sink BFM: Receive transaction 0
# 1090000: INFO: top.pgm.print_transaction:   Data:  0
# 1090000: INFO: top.pgm.print_transaction:   Idles: 3
# 1090000: INFO: top.pgm.print_transaction:   SOP:   1
# 1090000: INFO: top.pgm.print_transaction:   EOP:   0
# 1090000: INFO: top.pgm.print_transaction:   Channel: 0
# 1090000: INFO: top.pgm.print_transaction:   Error:   0
# 1090000: INFO: top.pgm.print_transaction:   Empty:   0
# 1090000: INFO: top.pgm.compare_transaction: Transaction 0 compare OK
# 1110000: INFO: top.pgm.test_threads.source_response_thread: Source response latency 0
# 1130000: INFO: top.pgm.print_transaction: Sink BFM: Receive transaction 1
# 1130000: INFO: top.pgm.print_transaction:   Data:  1
# 1130000: INFO: top.pgm.print_transaction:   Idles: 0
# 1130000: INFO: top.pgm.print_transaction:   SOP:   0
```

```
# 1130000: INFO: top.pgm.print_transaction:   EOP:   0
# 1130000: INFO: top.pgm.print_transaction:   Channel: 0
# 1130000: INFO: top.pgm.print_transaction:   Error:   0
# 1130000: INFO: top.pgm.print_transaction:   Empty:   0
# 1130000: INFO: top.pgm.compare_transaction: Transaction 1 compare OK
# 1150000: INFO: top.pgm.print_transaction: Sink BFM: Receive transaction 2
# 1150000: INFO: top.pgm.print_transaction:   Data:  2
# 1150000: INFO: top.pgm.print_transaction:   Idles: 0
# 1150000: INFO: top.pgm.print_transaction:   SOP:   0
# 1150000: INFO: top.pgm.print_transaction:   EOP:   0
# 1150000: INFO: top.pgm.print_transaction:   Channel: 0
# 1150000: INFO: top.pgm.print_transaction:   Error:   0
# 1150000: INFO: top.pgm.print_transaction:   Empty:   0
# 1150000: INFO: top.pgm.compare_transaction: Transaction 2 compare OK
# 1190000: INFO: top.pgm.print_transaction: Sink BFM: Receive transaction 3
# 1190000: INFO: top.pgm.print_transaction:   Data:  3
# 1190000: INFO: top.pgm.print_transaction:   Idles: 0
# 1190000: INFO: top.pgm.print_transaction:   SOP:   0
# 1190000: INFO: top.pgm.print_transaction:   EOP:   1
# 1190000: INFO: top.pgm.print_transaction:   Channel: 0
# 1190000: INFO: top.pgm.print_transaction:   Error:   0
# 1190000: INFO: top.pgm.print_transaction:   Empty:   0
# 1190000: INFO: top.pgm.compare_transaction: Transaction 3 compare OK
# 1190000: INFO: top.pgm: Test Passed
```

As Example 1–1 illustrates, when the Avalon-ST source BFM drives a transaction, it also prints the transaction to the ModelSim transcript window, creating a record of the test. The Avalon-ST Sink BFM also prints the transactions it receives on the transcript window. The Avalon-ST Sink BFM compares the transaction it receives with the one sent by the Avalon-ST Source BFM, and the results of the comparison are printed on the transcript window.

In Example 1–1 the `idles` values for the source and sink are different. The Avalon-ST Source BFM sets the number of idle cycles to zero using the `set_transaction_idles` function. The Avalon-ST Sink BFM waits for three cycles before receiving the first transaction because it takes three cycles for the transaction to propagate from the input port to the output port of the Avalon-ST Single-Clock FIFO buffer. The difference in values for the `idle` field is not an error because the Avalon-ST interface protocol allows source and sink components to have different latencies.

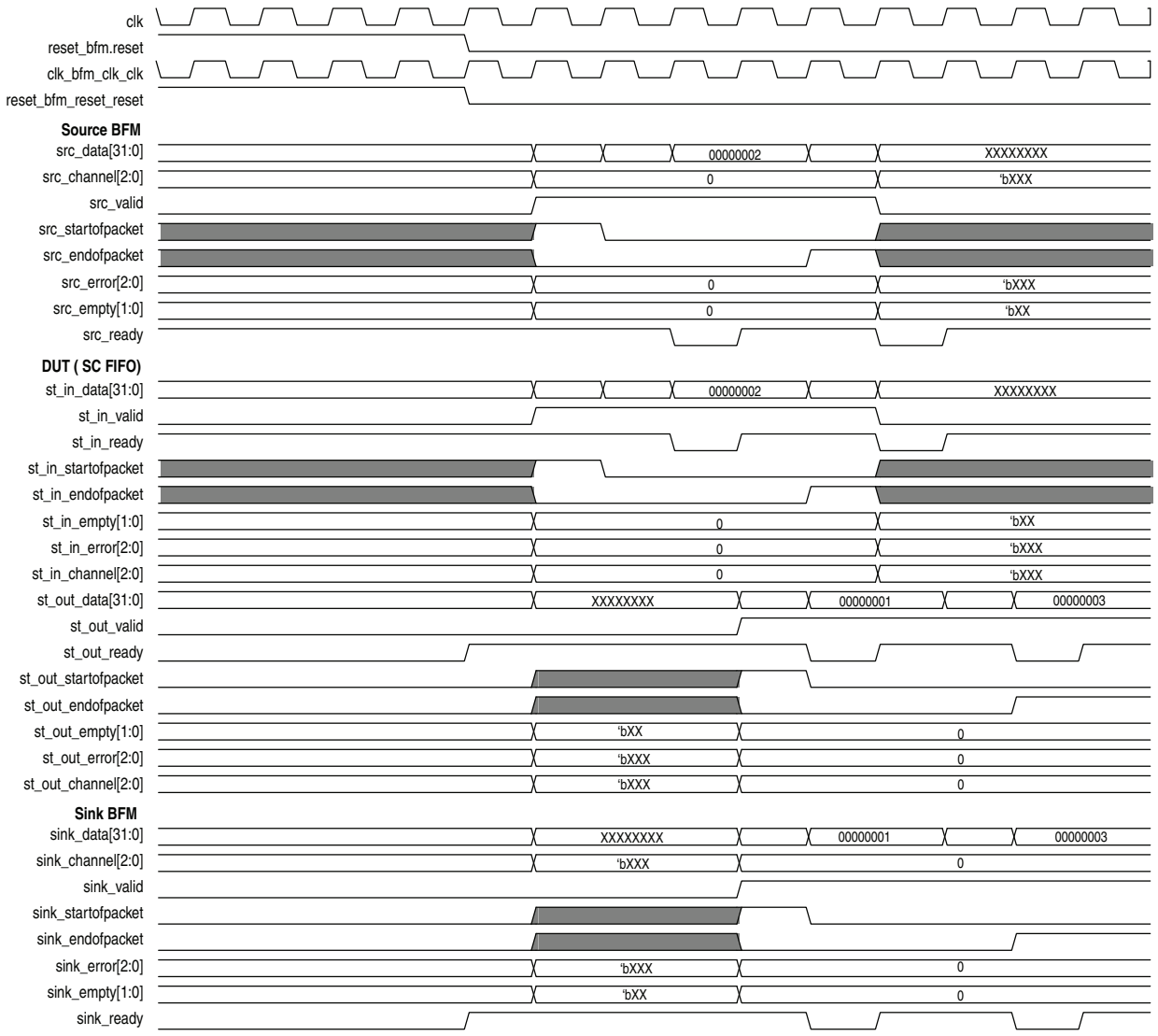
Example 1–2 shows the ModelSim transcript for the source response latency, which is the number of clock cycles the Avalon-ST Single-Clock FIFO buffer takes when the Avalon-ST Single-Clock FIFO buffer backpressures the Avalon-ST Source BFM. The third response shows a non-zero response latency. During the third transaction, the Avalon-ST Single-Clock FIFO buffer is full so it is not able to receive the transaction. As a result, the Avalon-ST Single-Clock FIFO buffer backpressures the Avalon-ST Source BFM.

Example 1–2. Response Latency

```
# 1030000: INFO: top.pgm.test_threads.source_response_thread: Source response latency 0
# 1050000: INFO: top.pgm.test_threads.source_response_thread: Source response latency 0
# 1090000: INFO: top.pgm.test_threads.source_response_thread: Source response latency 1
# 1110000: INFO: top.pgm.test_threads.source_response_thread: Source response latency 0
```

Figure 1–3 shows the simulation waveforms in the ModelSim-Altera software wave window.

Figure 1–3. Using Instance Names to Define BFMs



2. Using the VHDL BFMs

The Quartus II software version 13.0 and higher provides VHDL BFM support in Qsys. To use a VHDL BFM, your test program must include the appropriate VHDL package. For example, to use the Avalon-MM master BFM, include the package for this BFM in your test program. Packages are named:

*<BFM type or component name>*_**vhdl_pkg**

The VHDL BFM design and simulation flow is similar to the Verilog HDL flow, and involves the following steps:

1. Create the system design in Qsys.
2. Generate the testbench design in the Qsys **Generation** tab. Qsys automatically assigns a unique ID (0 to 1023) to each VHDL BFM instance in the testbench design.

   ☞ You can implement up to 1,024 instances of a particular BFM component.

3. Open the testbench system in Qsys.
4. (Optional) Make changes as needed to the BFM, such as changing the BFM instance name or the VHDL BFM ID. You change the ID with the **VHDL BFM ID** option.

   ☞ The VHDL BFM ID is only applicable for VHDL BFMs. The parameter appears in the top-level HDL for both Verilog HDL and VHDL files. However, Verilog HDL systems ignore this setting.

5. Generate a VHDL simulation model of the testbench design.
6. Create a custom test program.
7. Compile and load the Qsys design and testbench in a simulator.
8. Run the simulation.

This chapter provides additional information about the document and Altera.

Document Revision History

The following table shows the revision history for this document.

Date	Version	Changes
May 2013	3.2	Added information on VHDL support for verification IP. Removed information on the following wrappers. These wrappers are not supported in the Quartus II software version 13.0 and higher. ■ Avalon-MM master BFM with Avalon-ST API wrapper ■ Avalon-MM slave BFM with Avalon-ST API wrapper ■ Avalon-ST source BFM with Avalon-ST API wrapper ■ Avalon-ST sink BFM with Avalon-ST API wrapper Removed references to SOPC Builder.
June 2012	3.1	■ Updated SOPC Tutorial chapter. ■ Updated Qsys Tutorial chapter.
May 2011	3.0	■ Added External Memory BFM chapter. ■ Updated Avalon-MM Master and Slave BFMs chapters. ■ Updated Avalon-MM Monitor chapter. ■ Updated SOPC Tutorial chapter. ■ Added Qsys Tutorial chapter.
January 2011	2.0	■ Added Clock Source BFM and Reset Source BFM chapters. ■ Added Interrupt Source BFM and Interrupt Sink BFM chapters. ■ Added Conduit BFM and Tri-State Conduit BFM chapters. ■ Added Custom Instructions Master and Custom Instructions Slave BFMs chapters. ■ Updated Avalon-MM Master and Slave BFMs chapters. ■ Updated Avalon-ST Source and Sink BFMs chapters. ■ Updated Avalon-MM and Avalon-ST Monitor chapters. ■ Updated Avalon-MM and Avalon-ST Tutorial chapters.
August 2010	1.2	Updated Avalon Verification IP Suite Design Files for the Quartus II 10.0 release.
December 2009	1.1	Added Avalon-ST Tutorial chapter.
November 2009	1.0	Initial release covering 9.1 *Avalon Verification IP Suite User Guide*.

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact *(1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, *<file name>* and *<project name>***.pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix `n` denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
↵	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.

Visual Cue	Meaning
	A question mark directs you to a software help system with related information.
	The feet direct you to another document or website with related information.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.