



Avalon Verification IP Suite

User Guide



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

UG-01073-3.1

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

12.0
June 2012



[Subscribe](#)

© 2012 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX are Reg. U.S. Pat. & Tm. Off. and/or trademarks of Altera Corporation in the U.S. and other countries. All other trademarks and service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



Section I. Introduction to Avalon Verification IP Suite

Advantages of Using BFM's and Monitors	1-1
Implementation of BFM's	1-1
Application Programming Interface	1-2
Application Example of BFM's	1-2
In This User Guide	1-3

Section II. Clock, Reset, and Interrupt BFM's

Chapter 1. Clock Source BFM

Parameters	1-1
Application Program Interface	1-1
clock_start()	1-1
clock_stop()	1-1
get_run_state()	1-1
get_version()	1-2

Chapter 2. Reset Source BFM

Parameters	2-1
Application Program Interface	2-1
reset_assert	2-1
reset_deassert	2-1
get_version()	2-1

Chapter 3. Avalon Interrupt Source and Interrupt Sink BFM's

Parameters	3-1
Application Program Interface	3-1
clear_irq()	3-1
get_irq()	3-2
get_version()	3-2
set_irq()	3-2

Section III. Avalon-MM BFM's

Chapter 1. Avalon-MM Master BFM

Functional Description	1-1
Timing	1-2
Block Diagram	1-5
Parameters	1-7
Application Program Interface	1-9
all_transactions_complete()	1-9
get_command_issued_queue_size()	1-9
get_command_pending_queue_size()	1-9
get_read_response_queue_size()	1-9
get_response_address()	1-9
get_response_byte_enable()	1-10
get_response_burst_size()	1-10

get_response_data()	1-10
get_response_latency()	1-10
get_response_queue_size()	1-10
get_response_read_id()	1-11
get_response_read_response()	1-11
get_response_request()	1-11
get_response_wait_time()	1-11
get_response_write_id()	1-11
get_response_write_response()	1-12
get_write_response_queue_size()	1-12
get_version()	1-12
init()	1-12
pop_response()	1-12
push_command()	1-13
set_clken()	1-13
set_command_address()	1-13
set_command_arbiterlock()	1-13
set_command_byte_enable()	1-13
set_command_burst_count()	1-14
set_command_burst_size()	1-14
set_command_data()	1-14
set_command_debugaccess()	1-14
set_command_idle()	1-15
set_command_init_latency()	1-15
set_command_lock()	1-15
set_command_request()	1-15
set_command_timeout()	1-15
set_command_transaction_id()	1-16
set_command_write_response_request()	1-16
set_max_command_queue_size()	1-16
set_min_command_queue_size()	1-16
set_response_timeout()	1-16
signal_all_transactions_complete	1-16
signal_command_issued	1-17
signal_fatal_error	1-17
signal_max_command_queue_size	1-17
signal_min_command_queue_size	1-17
signal_read_response_complete()	1-17
signal_response_complete()	1-18
signal_write_response_complete()	1-18

Chapter 2. Avalon-MM Master BFM with Avalon-ST API Wrapper

Chapter 3. Avalon-MM Slave BFM

Functional Description	3-2
Timing	3-3
Block Diagram	3-6
Parameters	3-8
Application Program Interface	3-10
get_clken()	3-10
get_command_address()	3-10
get_command_arbiterlock()	3-10
get_command_burst_count()	3-10

get_command_burst_cycle()	3-11
get_command_byte_enable()	3-11
get_command_data()	3-11
get_command_debugaccess()	3-11
get_command_queue_size()	3-11
get_command_lock()	3-12
get_command_request()	3-12
get_command_transaction_id()	3-12
get_command_write_response_request()	3-12
get_pending_read_latency_cycle()	3-12
get_pending_write_latency_cycle()	3-13
get_response_queue_size()	3-13
get_slave_bfm_status	3-13
get_version()	3-13
init()	3-13
pop_command()	3-14
push_response()	3-14
set_command_transaction_mode()	3-14
set_interface_wait_time()	3-14
set_max_response_queue_size()	3-14
set_min_response_queue_size()	3-15
set_read_response_id()	3-15
set_read_response_status()	3-15
set_response_burst_size()	3-15
set_response_data()	3-15
set_response_latency()	3-16
set_response_request()	3-16
set_response_timeout()	3-16
set_write_response_id()	3-16
set_write_response_status()	3-16
signal_command_received	3-17
signal_error_exceed_max_pending_reads	3-17
signal_max_response_queue_size	3-17
signal_min_command_queue_size	3-17
signal_fatal_error	3-17
signal_response_issued	3-18

Chapter 4. Avalon-MM Slave BFM with Avalon-ST API Wrapper

Chapter 5. Avalon-MM Monitor

Parameters	5-2
Application Program Interface	5-4
Assertion Checking	5-4
set_enable_a_address_align_with_data_width()	5-4
set_enable_a_beginbursttransfer_exist()	5-4
set_enable_a_beginbursttransfer_legal()	5-4
set_enable_a_beginbursttransfer_single_cycle()	5-5
set_enable_a_begintransfer_exist()	5-5
set_enable_a_begintransfer_legal()	5-5
set_enable_a_begintransfer_single_cycle()	5-5
set_enable_a_burst_legal()	5-5
set_enable_a_byteenable_legal()	5-6
set_enable_a_constant_during_burst()	5-6

set_enable_a_constant_during_clk_disabled()	5-6
set_enable_a_constant_during_waitrequest()	5-6
set_enable_a_exclusive_read_write()	5-6
set_enable_a_half_cycle_reset_legal()	5-7
set_enable_a_less_than_burstcount_max_size()	5-7
set_enable_a_less_than_maximumpendingreadtransactions()	5-7
set_enable_a_no_readdatavalid_during_reset()	5-7
set_enable_a_no_read_during_reset()	5-7
set_enable_a_no_write_during_reset()	5-8
set_enable_a_readid_sequence()	5-8
set_enable_a_read_response_sequence()	5-8
set_enable_a_read_response_timeout()	5-8
set_enable_a_register_incoming_signals()	5-8
set_enable_a_waitrequest_during_reset()	5-9
set_enable_a_waitrequest_timeout()	5-9
set_enable_a_write_burst_timeout()	5-9
set_enable_a_writeid_sequence()	5-9
Coverage Group	5-10
set_enable_c_b2b_read_read()	5-10
set_enable_c_b2b_read_write()	5-10
set_enable_c_b2b_write_read()	5-10
set_enable_c_b2b_write_write()	5-11
set_enable_c_continuous_read()	5-11
set_enable_c_continuous_readdatavalid()	5-11
set_enable_c_continuous_waitrequest()	5-11
set_enable_c_continuous_waitrequest_from_idle_to_read()	5-11
set_enable_c_continuous_waitrequest_from_idle_to_write()	5-12
set_enable_c_continuous_write()	5-12
set_enable_c_idle_before_transaction()	5-12
set_enable_c_idle_in_read_response()	5-12
set_enable_c_idle_in_write_burst()	5-12
set_enable_c_pending_read()	5-13
set_enable_c_read()	5-13
set_enable_c_read_after_reset()	5-13
set_enable_c_read_burstcount()	5-13
set_enable_c_read_byteenable()	5-13
set_enable_c_read_latency()	5-14
set_enable_c_read_response()	5-14
set_enable_c_waitrequest_in_write_burst()	5-14
set_enable_c_waitrequested_read()	5-14
set_enable_c_waitrequest_without_command()	5-14
set_enable_c_waitrequested_write()	5-15
set_enable_c_write()	5-15
set_enable_c_write_with_and_without_writeresponserequest()	5-15
set_enable_c_write_after_reset()	5-15
set_enable_c_write_burstcount()	5-15
set_enable_c_write_byteenable()	5-16
set_enable_c_write_response()	5-16
Transaction Monitoring	5-16
get_clken()	5-16
get_version()	5-16
get_command_address()	5-17
get_command_arbiterlock()	5-17
get_command_burst_count()	5-17

get_command_burst_cycle()	5-17
get_command_byte_enable()	5-17
get_command_data()	5-18
get_command_debugaccess()	5-18
get_command_issued_queue_size()	5-18
get_command_queue_size()	5-18
get_command_lock()	5-18
get_command_request()	5-19
get_command_transaction_id()	5-19
get_command_write_response_request()	5-19
get_read_response_queue_size()	5-19
get_response_address()	5-19
get_response_byte_enable()	5-20
get_response_burst_size()	5-20
get_response_data()	5-20
get_response_latency()	5-20
get_response_queue_size()	5-20
get_response_read_id()	5-21
get_response_read_response()	5-21
get_response_request()	5-21
get_response_wait_time()	5-21
get_response_write_id()	5-21
get_response_write_response()	5-22
get_transaction_fifo_max()	5-22
get_transaction_fifo_threshold()	5-22
get_write_response_queue_size()	5-22
init()	5-22
pop_command()	5-23
pop_response()	5-23
set_command_transaction_mode()	5-23
set_transaction_fifo_max()	5-23
set_transaction_fifo_threshold()	5-23
signal_command_received	5-24
signal_fatal_error	5-24
signal_read_response_complete	5-24
signal_response_complete	5-24
signal_transaction_fifo_overflow	5-24
signal_transaction_fifo_threshold	5-25
signal_write_response_complete	5-25

Section IV. Avalon-ST BFM

Chapter 1. Avalon-ST Source BFM

Functional Description	1-1
Timing	1-2
Block Diagram	1-3
Parameters	1-4
Application Program Interface	1-5
get_response_latency()	1-5
get_response_queue_size()	1-5
get_src_ready()	1-5
get_src_transaction_complete()	1-5
get_transaction_queue_size()	1-5

get_version()	1-6
init()	1-6
pop_response()	1-6
push_transaction()	1-6
set_max_transaction_queue_size()	1-6
set_min_transaction_queue_size()	1-7
set_response_timeout()	1-7
set_transaction_channel()	1-7
set_transaction_data()	1-7
set_transaction_idles()	1-7
set_transaction_eop()	1-7
set_transaction_empty()	1-8
set_transaction_error()	1-8
set_transaction_sop()	1-8
signal_fatal_error	1-8
signal_max_transaction_queue_size	1-8
signal_min_transaction_queue_size	1-8
signal_response_done	1-9
signal_src_driving_transaction	1-9
signal_src_not_ready	1-9
signal_src_ready	1-9
signal_src_transaction_complete	1-9

Chapter 2. Avalon-ST Source BFM with Avalon-ST API Wrapper

Chapter 3. Avalon-ST Sink BFM

Functional Description	3-2
Timing	3-2
Block Diagram	3-3
Parameters	3-4
Application Program Interface	3-5
get_transaction_channel()	3-5
get_transaction_data()	3-5
get_transaction_idles()	3-5
get_transaction_eop()	3-5
get_transaction_empty()	3-5
get_transaction_error()	3-6
get_transaction_queue_size()	3-6
get_transaction_sop()	3-6
get_version()	3-6
init()	3-6
pop_transaction()	3-6
set_ready()	3-7
signal_fatal_error	3-7
signal_sink_ready_assert	3-7
signal_sink_ready_deassert	3-7
signal_transaction_received	3-7

Chapter 4. Avalon-ST Sink BFM with Avalon-ST API Wrapper

Chapter 5. Avalon-ST Monitor

Parameters	5-2
Application Program Interface	5-3

Assertion Checking	5-3
set_enable_a_empty_legal()	5-3
set_enable_a_less_than_max_channel()	5-3
set_enable_a_no_data_outside_packet()	5-3
set_enable_a_non_missing_endofpacket()	5-4
set_enable_a_non_missing_startofpacket()	5-4
set_enable_a_valid_legal()	5-4
Coverage Group	5-5
set_enable_c_all_idle_beats()	5-5
set_enable_c_all_valid_beats()	5-5
set_enable_c_b2b_data_different_channel()	5-5
set_enable_c_b2b_data_same_channel()	5-6
set_enable_c_b2b_packet_different_channel()	5-6
set_enable_c_b2b_packet_in_different_transaction()	5-6
set_enable_c_b2b_packet_same_channel()	5-6
set_enable_c_b2b_packet_within_single_cycle()	5-6
set_enable_c_channel_change_in_packet()	5-7
set_enable_c_empty()	5-7
set_enable_c_error()	5-7
set_enable_c_error_in_middle_of_packet()	5-7
set_enable_c_idle_beat_between_packet()	5-7
set_enable_c_multiple_packet_per_cycle()	5-8
set_enable_c_non_valid_ready()	5-8
set_enable_c_non_valid_non_ready()	5-8
set_enable_c_packet()	5-8
set_enable_c_packet_no_idles_no_back_pressure()	5-8
set_enable_c_packet_size()	5-9
set_enable_c_packet_with_back_pressure()	5-9
set_enable_c_packet_with_idles()	5-9
set_enable_c_partial_valid_beats()	5-9
set_enable_c_single_packet_per_cycle()	5-9
set_enable_c_transfer()	5-10
set_enable_c_transaction_after_reset()	5-10
set_enable_c_valid_non_ready()	5-10
Transaction Monitoring	5-10
get_transaction_channel()	5-10
get_transaction_data()	5-11
get_transaction_empty()	5-11
get_transaction_eop()	5-11
get_transaction_error()	5-11
get_transaction_idles()	5-11
get_transaction_queue_size()	5-11
get_transaction_sop()	5-12
get_version()	5-12
pop_transaction()	5-12
set_transaction_fifo_max()	5-12
set_transaction_fifo_threshold()	5-12
signal_fatal_error	5-13
signal_transaction_fifo_overflow	5-13
signal_transaction_fifo_threshold	5-13
signal_transaction_received	5-13

Section V. Conduit and External Memory BFM

Chapter 1. Conduit BFM

Block Diagram	1-1
Parameters	1-2
Application Program Interface	1-3
get_<role name>()	1-3
get_version()	1-3
set_<role name>()	1-3
set_<role name>_oe()	1-3
signal_input_<role name>_change	1-3

Chapter 2. Tri-State Conduit BFM

Block Diagram	2-1
Parameters	2-2
Application Program Interface	2-3
get_input_transaction_queue_size()	2-3
get_output_transaction_queue_size()	2-3
get_transaction_<role name>_in()	2-3
get_transaction_latency()	2-3
get_version()	2-3
pop_transaction()	2-4
push_transaction()	2-4
set_max_transaction_queue_size()	2-4
set_min_transaction_queue_size()	2-4
set_num_of_transactions()	2-4
set_transaction_<role name>_out()	2-5
set_transaction_<role name>_outen()	2-5
set_transaction_idles()	2-5
set_valid_transaction_<role name>_out()	2-5
signal_all_transactions_complete	2-5
signal_fatal_error	2-5
signal_grant_deasserted_while_request_remain_asserted()	2-6
signal_interface_granted	2-6
signal_max_transaction_queue_size	2-6
signal_min_transaction_queue_size	2-6

Chapter 3. External Memory BFM

Functional Description	3-1
Block Diagram	3-1
Initializing the Memory Content	3-2
Reading and Writing to the Memory Content	3-2
Reading from the Memory	3-2
Writing to the Memory	3-3
Parameters	3-3
Application Program Interface	3-5
fill()	3-5
read()	3-5
signal_api_call	3-5
write()	3-5

Section VI. Nios II Custom Instruction BFM

Chapter 1. Nios II Custom Instruction Master BFM

Block Diagram	1-1
Parameters	1-2
Application Program Interface	1-3
get_instruction_queue_size()	1-3
get_result_delay()	1-3
get_result_queue_size()	1-3
get_result_value()	1-3
get_version()	1-4
insert_instruction()	1-4
pop_result()	1-4
push_instruction()	1-4
retrive_result()	1-5
set_ci_clk_en()	1-5
set_clock_enable_timeout()	1-5
set_instruction_a()	1-5
set_instruction_b()	1-5
set_instruction_c()	1-5
set_instruction_dataaa()	1-6
set_instruction_datab()	1-6
set_instruction_err_inject()	1-6
set_instruction_idle()	1-6
set_instruction_n()	1-6
set_instruction_readra()	1-6
set_instruction_readrb()	1-7
set_instruction_timeout()	1-7
set_instruction_writerc()	1-7
set_max_instruction_queue_size()	1-7
set_max_result_queue_size()	1-7
set_min_instruction_queue_size()	1-7
set_min_result_queue_size()	1-8
set_result_timeout()	1-8
signal_unexpected_result_received	1-8
signal_fatal_error	1-8
signal_instructions_completed	1-8
signal_instruction_start	1-8
signal_max_instruction_queue_size	1-9
signal_max_result_queue_size	1-9
signal_min_instruction_queue_size	1-9
signal_min_result_queue_size	1-9
signal_result_received	1-9

Chapter 2. Nios II Custom Instruction Slave BFM

Block Diagram	2-1
Parameters	2-2
Application Program Interface	2-3
get_ci_clk_en()	2-3
get_instruction_a()	2-3
get_instruction_b()	2-3
get_instruction_c()	2-3
get_instruction_dataaa()	2-3

get_instruction_datab()	2-4
get_instruction_idle()	2-4
get_instruction_n()	2-4
get_instruction_readra()	2-4
get_instruction_readrb()	2-4
get_instruction_writerc()	2-4
get_version()	2-5
insert_result()	2-5
retrieve_instruction()	2-5
set_clock_enable_timeout()	2-5
set_instruction_a()	2-6
set_instruction_b()	2-6
set_instruction_c()	2-6
set_instruction_timeout()	2-6
set_result_delay()	2-6
set_result_err_inject()	2-6
set_result_value()	2-7
signal_fatal_error	2-7
signal_instructions_inconsistent	2-7
signal_known_instruction_received	2-7
signal_result_done	2-7
signal_result_driven	2-7
signal_unknown_instruction_received	2-8

Section VII. Tutorials

Chapter 1. SOPC Builder Tutorial

Software Requirements	1-1
Verifying Avalon-MM Slave DUT	1-1
Setting up the Test	1-3
Creating an SOPC Builder Testbench for the DUT	1-3
Connecting and Generating the SOPC Builder System	1-5
Running the Simulation	1-5
Observing the Results	1-6
Verifying Avalon-MM Master DUT	1-7
Setting Up the Test	1-7
Creating an SOPC Builder Testbench for the DUT	1-7
Connecting and Generating the SOPC Builder System	1-9
Running the Simulation	1-10
Observing the Results	1-10

Chapter 2. Qsys Tutorial

Software Requirements	2-1
Verifying Avalon-ST DUT	2-1
Setting up the Test	2-2
Creating a Qsys System for the DUT	2-2
Generating a Qsys Testbench System	2-3
Setting up the Simulation Environment	2-5
Running the Simulation	2-5
Observing the Results	2-6

Additional Information

Document Revision History Info-1

How to Contact Altera Info-1

Typographic Conventions Info-2

The Avalon® Verification IP Suite provides bus functional models (BFMs) to simulate the behavior 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.

Implementation of BFMs

The Avalon Verification IP Suite BFMs (excluding 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 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.

The Avalon Verification IP Suite also includes wrapper components so that the BFM s can also be used in VHDL verification environments with simulators that support mixed language simulation. These wrapper components are generated in SOPC Builder only. Qsys does not support VHDL simulation with any BFM s other than the Clock Source and Reset Source BFM s.

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.

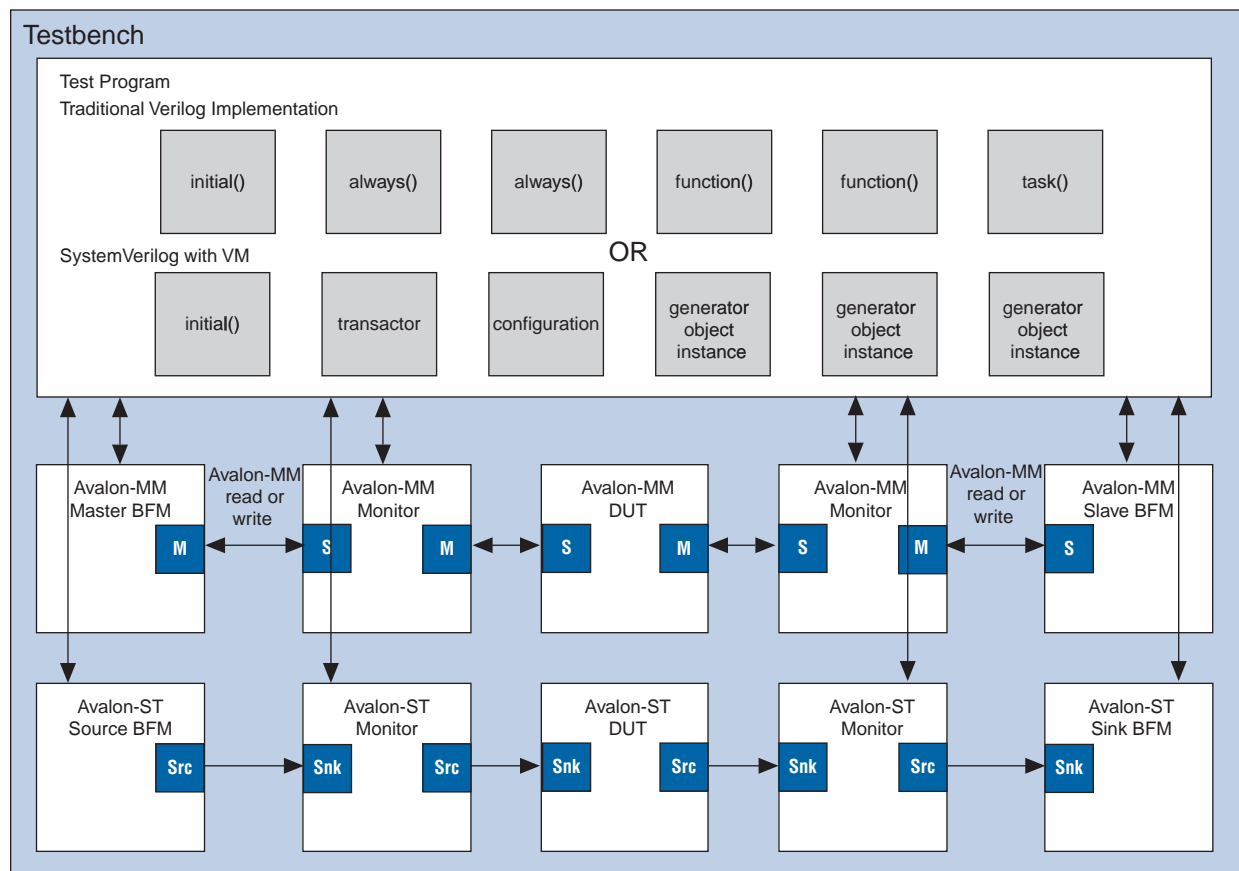


While you can use methods other than the API, Altera does not guarantee continued support or backwards compatibility of custom methods.

Application Example of BFM s

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 tutorials on how to use the BFMs to verify IP interfaces and components in SOPC Builder and Qsys.

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

- [Chapter 1, Clock Source BFM](#)
- [Chapter 2, Reset Source BFM](#)
- [Chapter 3, Avalon Interrupt Source and Interrupt Sink 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: None.
Returns: void.
Description: Turns on the clock.

clock_stop()

Prototype: `clock_stop()`.
Arguments: None.
Returns: void.
Description: Turns off the clock.

get_run_state()

Prototype: `get_run_state()`.
Arguments: None.
Returns: bit.
Description: Returns the state of the clock source; 1=running, 0=stop.

get_version()

Prototype: `string get_version().`

Arguments: None.

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".

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: None.
Returns: `void.`
Description: Asserts the reset signal.

reset_deassert

Prototype: `reset_deassert.`
Arguments: None.
Returns: `void.`
Description: Deasserts the reset signal.

get_version()

Prototype: `string get_version().`
Arguments: None.
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".

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



The Avalon Interrupt Source and Sink BFM 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 BFM 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 <code>irq</code> to <code>irq_n</code> .
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.
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 <code>irq</code> to <code>irq_n</code> .
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 BFM.

clear_irq()

Prototype: `int clear_irq().`

Arguments: `int interrupt_bit.`

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.

get_irq()

Prototype:	<code>get_irq()</code> .
Arguments:	None.
Returns:	<code>logic[AV_IRQ_W-1:0]</code> .
Description:	Returns the current value of the register holding the latched interrupt signal.

get_version()

Prototype:	<code>string get_version()</code> .
Arguments:	None.
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".

set_irq()

Prototype:	<code>set_irq()</code> .
Arguments:	<code>int interrupt_bit</code> .
Returns:	<code>void</code> .
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.

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

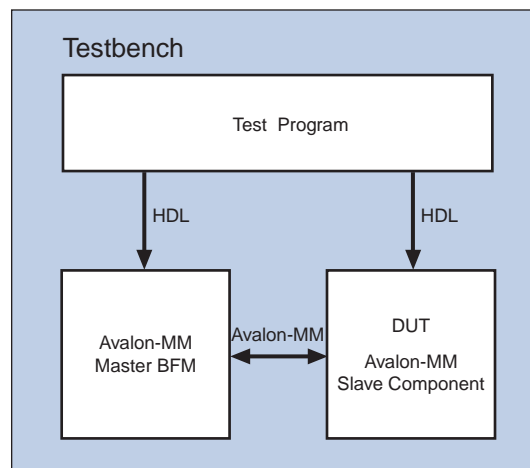
- [Chapter 1, Avalon-MM Master BFM](#)
- [Chapter 2, Avalon-MM Master BFM with Avalon-ST API Wrapper](#)
- [Chapter 3, Avalon-MM Slave BFM](#)
- [Chapter 4, Avalon-MM Slave BFM with Avalon-ST API Wrapper](#)
- [Chapter 5, Avalon-MM Monitor](#)

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 Monitors 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 SOPC Builder, refer to the *Avalon Interface Specifications (version 1.3)*.



For more information about the Avalon-MM specification supported in Qsys, refer to the *Avalon Interface Specifications (version 2.0)*.

Functional Description

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

- “Timing” on page 1–2
- “Block Diagram” on page 1–5

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

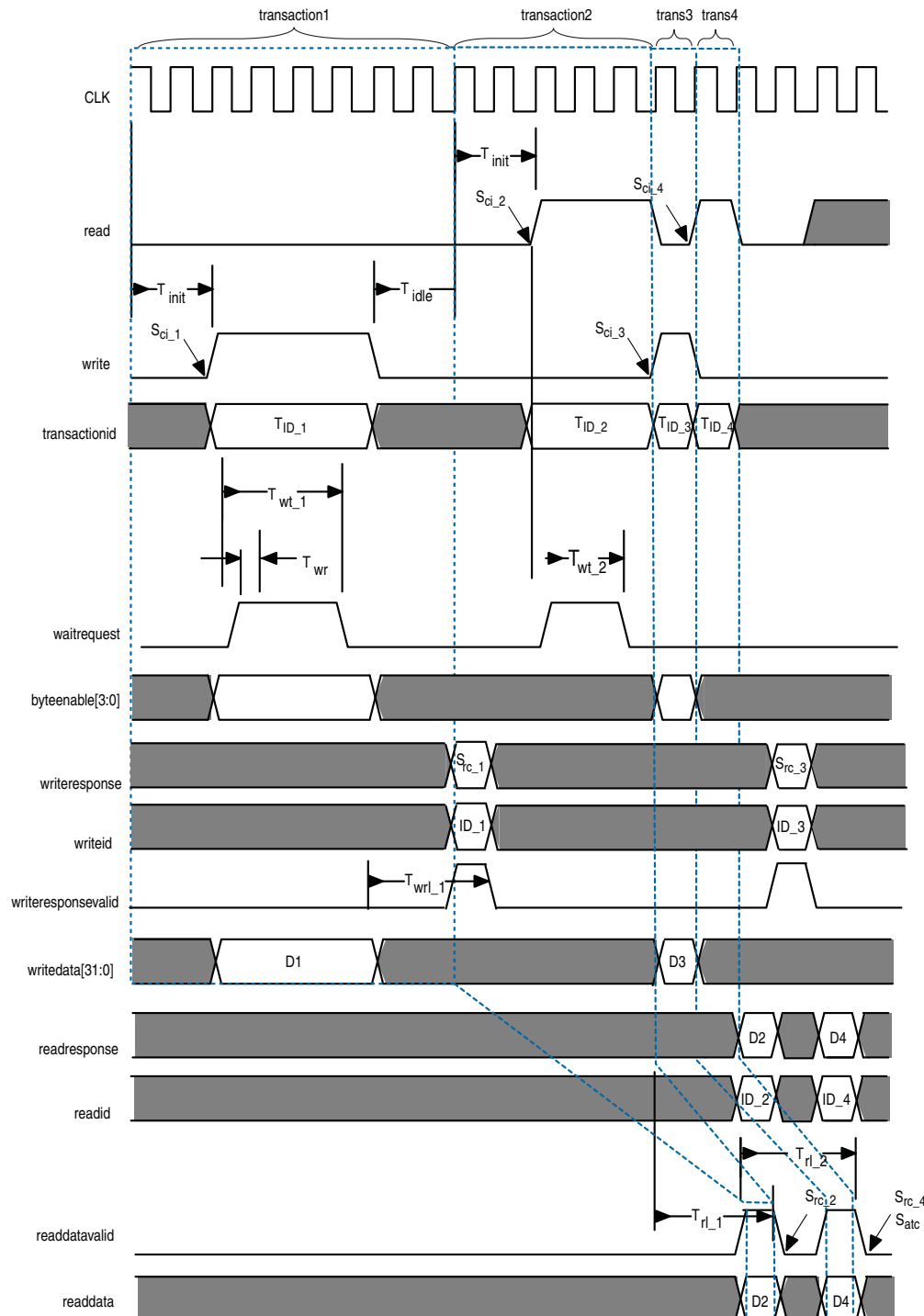


Table 1-1 lists the annotations used in Figure 1-2.

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

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 <code>set_command_init_latency</code> .
T_{wt_1}	The response wait time, which is three cycles. This time is determined by the number of cycles that the <code>waitrequest</code> signal is asserted by the slave. The program gets this value using the <code>get_response_wait_time</code> command.
T_{wr}	<code>waitrequest</code> is always sampled #1 after the falling edge of <code>clk</code> .
T_{idle}	The idle time after each transaction. This time is set by the command <code>set_command_idle</code> .
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 <code>get_response_latency</code> command. Note if the Avalon-MM slave component has defined a fixed read latency by defining the <code>readLatency</code> interface property, the <code>readdatavalid</code> 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 <code>get_response_latency</code> 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 <code>get_response_latency</code> command.
S_{ci_1} – S_{ci_4}	Signals when write or read commands are presented on the interface. The event name is <code>signal_command_issued</code> .
S_{rc_1} , S_{rc_3}	Signals write responses. The event name is <code>signal_response_complete</code> .
S_{rc_2} , S_{rc_4}	Signals read responses. The event name is <code>signal_response_complete</code> .
S_{atc}	Signals the end of the test. The event name is <code>signal_all_transactions_complete</code>
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

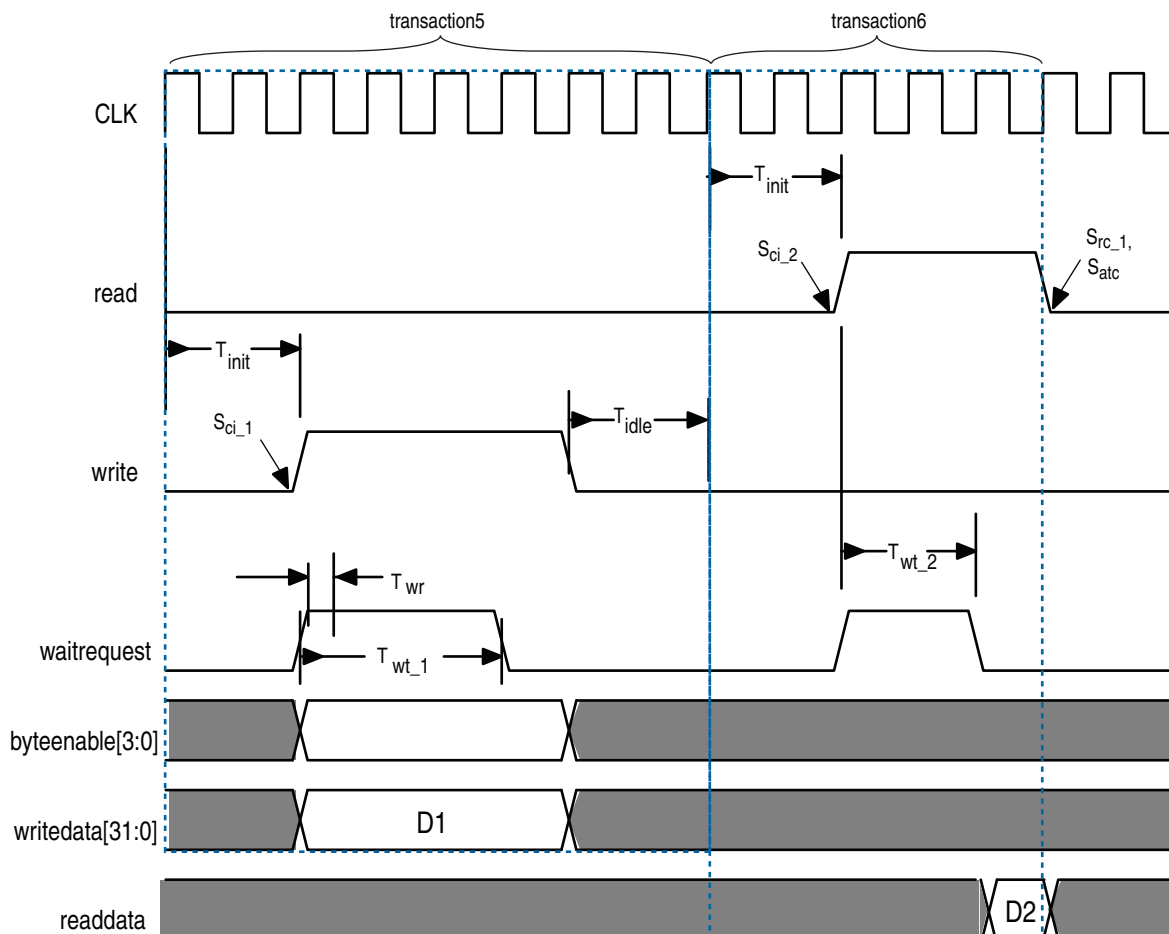


Table 1-2 lists the annotations used in Figure 1-3.

Table 1-2. Key to Annotations in Figure 1-3 (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 <code>set_command_init_latency</code> .
T_{wt_1}	The response wait time, which is three cycles. This time is determined by the number of cycles that the <code>waitrequest</code> signal is asserted by the slave. The program gets this value using the <code>get_response_wait_time</code> 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 <code>waitrequest</code> signal is asserted by the slave. The program gets this value using the <code>get_response_wait_time</code> command.
T_{wr}	<code>waitrequest</code> is always sampled #1 after the falling edge of <code>clk</code> .
T_{idle}	The idle time after a transaction. This time is set by the command <code>set_command_idle</code> .

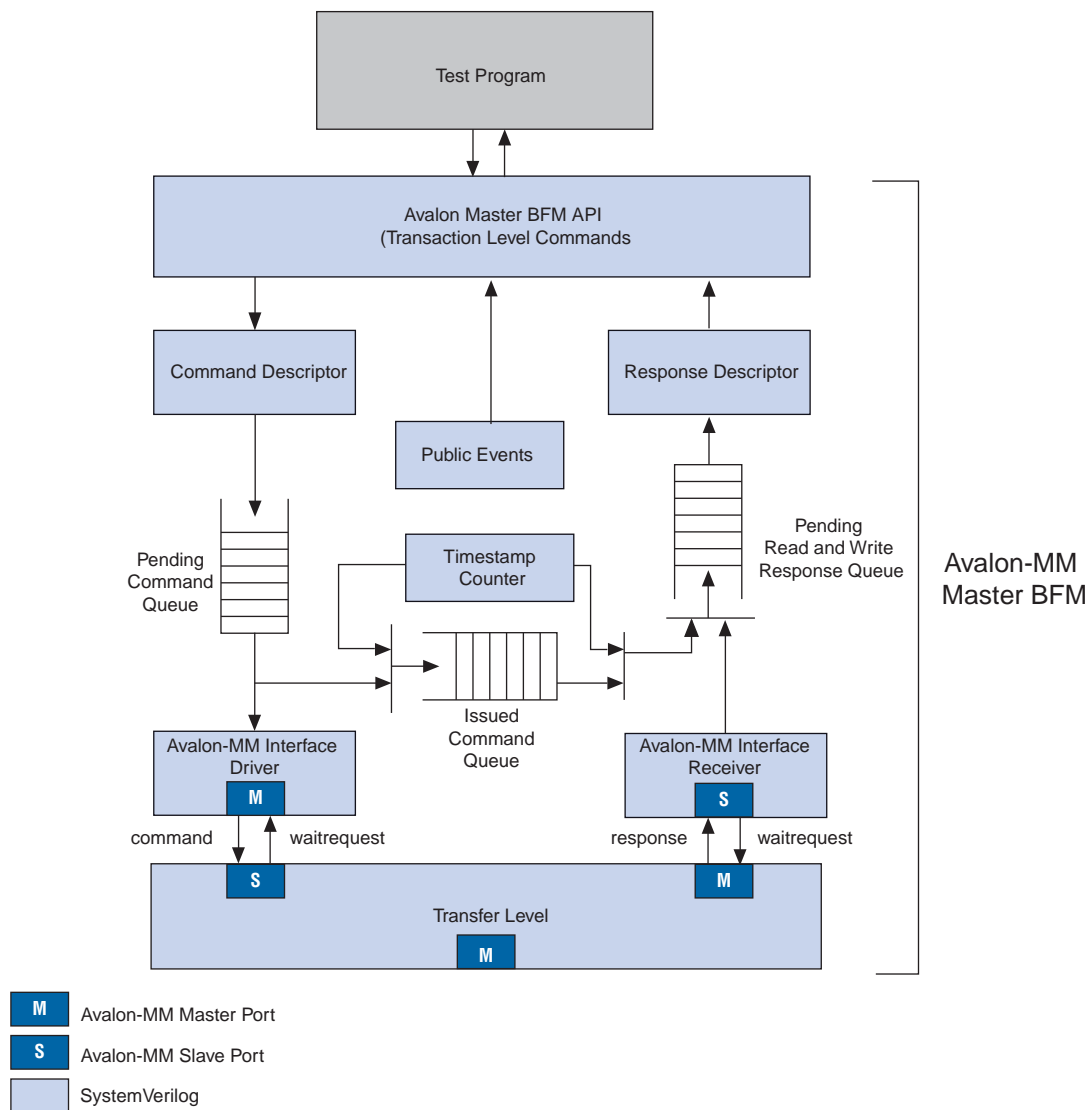
Table 1–2. Key to Annotations in Figure 1–3 (Part 2 of 2)

Symbol	Description
S_{ci_1} – S_{ci_2}	Signals when write and read commands are presented on the interface. The event name is <code>signal_command_issued</code> .
S_{rc_1}	Signals the first read response. The event name is <code>signal_response_complete</code> .
S_{atc}	Signals the end of the test. The event name is <code>signal_all_transactions_complete</code> .

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 <code>read</code> pin.
Use the write signal	On	On/Off	When On , the interface includes a <code>write</code> 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 <code>byteenable</code> pins.
Use the burstcount signal	On	On/Off	When On , the interface includes <code>burstcount</code> pins.
Use the readdata signal	On	On/Off	When On , the interface includes a <code>readdata</code> pin.
Use the readdatavalid signal	On	On/Off	When On , the interface includes a <code>readdatavalid</code> pin.
Use the writedata signal	On	On/Off	When On , the interface includes a <code>writedata</code> pin.
Use the begintransfer signal	Off	On/Off	When On , the interface includes <code>writedata</code> pins
Use the beginbursttransfer signal	Off	On/Off	When On , the interface includes a <code>beginbursttransfer</code> pins.
Use the arbiterlock signal	Off	On/Off	When On , the interface includes an <code>arbiterlock</code> pin.
Use the lock signal	Off	On/Off	When On , the interface includes a <code>lock</code> pin.
Use the debugaccess signal	Off	On/Off	When On , the interface includes a <code>debugaccess</code> pin.
Use the waitrequest signal	On	On/Off	When On , the interface includes a <code>waitrequest</code> pin.
Use the transactionid signal	Off	On/Off	When On , the interface includes a <code>transactionid</code> pin.
Use the write response signals	Off	On/Off	When On , the interface includes a <code>writeresponse</code> pin.
Use the read response signals	Off	On/Off	When On , the interface includes a <code>readresponse</code> pin.
Use the clken signals	Off	On/Off	When On , the interface includes a <code>clken</code> pin.
Port Polarity			
Assert reset high	On	On/Off	When On , <code>reset</code> is asserted high.
Assert waitrequest high	On	On/Off	When On , <code>waitrequest</code> is asserted high.
Assert read high	On	On/Off	When On , <code>read</code> is asserted high.

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

Parameter	Default Value	Legal Values	Description
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.
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.
API Streaming Interface (Note 1)			
Width of API interface data signal	64	—	The width of the data signal.
Width of API return interface data signal	64	—	The width of the return interface data signal.

Note to Table 1–3:

- (1) This interface is required only for the Avalon-MM Master BFM with Avalon-ST API Wrapper that is used in mixed language simulations.

Application Program Interface

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

all_transactions_complete()

Prototype: `bit all_transactions_complete();`
Arguments: None.
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.

get_command_issued_queue_size()

Prototype: `int get_command_issued_queue_size();`
Arguments: None.
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.

get_command_pending_queue_size()

Prototype: `int get_command_pending_queue_size();`
Arguments: None.
Returns: `int`.
Description: Queries the command queue to determine number of pending commands waiting to be driven out as Avalon requests.

get_read_response_queue_size()

Prototype: `int get_read_response_queue_size();`
Arguments: None.
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.

get_response_address()

Prototype: `bit [AV_ADDRESS_W-1:0] get_response_address();`
Arguments: None.
Returns: `bit`.
Description: Returns the transaction address in the response descriptor that has been removed from the response queue.

get_response_byte_enable()

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

get_response_burst_size()

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

get_response_data()

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

get_response_latency()

Prototype: `int get_response_latency(int index).`
Arguments: `index.`
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.

get_response_queue_size()

Prototype: `int get_response_queue_size().`
Arguments: `None.`
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.

get_response_read_id()

Prototype: [AV_TRANSACTIONID_W-1:0] get_response_read_id().
Arguments: None.
Returns: AvalonTransactionId_t.
Description: Returns the read id of the transaction in the response descriptor that has been removed from the response queue.

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: int index.
Returns: AvalonReadResponse_t.
Description: Returns the transaction read status in the response descriptor that has been removed from the response queue.

get_response_request()

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

get_response_wait_time()

Prototype: int get_response_wait_time(int index).
Arguments: index.
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.

get_response_write_id()

Prototype: bit [AV_TRANSACTIONID_W-1:0] get_response_write_id().
Arguments: None.
Returns: AvalonTransactionId_t.
Description: Returns the write id of the transaction in the response descriptor that has been removed from the response queue.

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: `int index.`

Returns: `AvalonWriteResponse_t.`

Description: Returns the transaction write status in the response descriptor that has been removed from the response queue.

get_write_response_queue_size()

Prototype: `int get_write_response_queue_size().`

Arguments: `None.`

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.

get_version()

Prototype: `string get_version().`

Arguments: `None.`

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".

init()

Prototype: `init.`

Arguments: `None.`

Returns: `void.`

Description: Initializes the Avalon-MM master interface.

pop_response()

Prototype: `void pop_response().`

Arguments: `None.`

Returns: `void.`

Description: Removes the oldest response descriptor from the response queue, such that transaction information is available using the `get_response_<rolename>` commands.

push_command()

Prototype:	<code>void push_command()</code> .
Arguments:	None.
Returns:	void.
Description:	Inserts the fully populated transaction descriptor onto the pending transaction command queue.

set_clken()

Prototype:	<code>void set_clken(bit state)</code> .
Arguments:	<code>bit state</code> .
Returns:	void.
Description:	Sets the assertion and deassertion of the clock enable signal.

set_command_address()

Prototype:	<code>void set_command_address(bit[AV_ADDRESS_W-1:0]addr)</code>
Arguments:	<code>addr</code> .
Returns:	void.
Description:	Sets the transaction address in the command descriptor.

set_command_arbiterlock()

Prototype:	<code>void set_command_arbiterlock (bit state)</code> .
Arguments:	<code>bit state</code> .
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.

set_command_byte_enable()

Prototype:	<code>void set_command_byte_enable(bit[AV_NUMSYMBOLS-1:0] byte_enable, int index)</code> .
Arguments:	<code>byte_enable</code> . <code>index</code> .
Returns:	void.
Description:	Sets the transaction byte enable field for the cycle of the burst command descriptor indicated by <code>index</code> . This field applies to both read and write operations.

set_command_burst_count()

Prototype: void set_command_burst_count(bit[AV_BURSTCOUNT_W-1:0] burst_count).
Arguments: burst_count.
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.

set_command_burst_size()

Prototype: void set_command_burst_size (bit[AV_BURSTCOUNT_W-1:0] burst_size).
Arguments: burst_size.
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.

set_command_data()

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

set_command_debugaccess()

Prototype: void set_command_debugaccess.
Arguments: bit state.
Returns: void.
Description: Controls the assertion or deassertion of the debugaccess interface signal. The debugaccess control is on transaction boundaries.

set_command_idle()

Prototype: void set_command_idle(int idle, int index).
Arguments: int idle.
int index.
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.

set_command_init_latency()

Prototype: void set_command_init_latency(int cycles).
Arguments: cycles.
Returns: void.
Description: Sets the number of cycles to postpone the start of a command.

set_command_lock()

Prototype: void set_command_lock (bit state).
Arguments: bit state.
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.

set_command_request()

Prototype: void set_command_request(Request_t request).
Arguments: Request_t request.
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.

set_command_timeout()

Prototype: void set_command_timeout(int cycles).
Arguments: int cycles.
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.

set_command_transaction_id()

Prototype: void set_command_transaction_id(bit[AV_TRANSACTIONID_W-1:0] id).
Arguments: AvalonTransactionId_t id.
Returns: void.
Description: Sets the transaction id number in the command descriptor.

set_command_write_response_request()

Prototype: void set_command_write_response_request (logic request).
Arguments: logic request.
Returns: void.
Description: Sets the flag that enables or disables the write response requests in the command descriptor.

set_max_command_queue_size()

Prototype: void set_max_command_queue_size(int size).
Arguments: int size.
Returns: void.
Description: Sets the pending command queue size maximum threshold.

set_min_command_queue_size()

Prototype: void set_min_command_queue_size(int size).
Arguments: int size.
Returns: void.
Description: Sets the pending command queue size minimum threshold.

set_response_timeout()

Prototype: void set_response_timeout(int cycles).
Arguments: int cycles.
Returns: void.
Description: Sets the number of cycles that may elapse before response time out. Disable time-out by setting the value to 0.

signal_all_transactions_complete

Prototype: signal_all_transactions_complete.
Arguments: None.
Returns: void.
Description: Signals that all queued transactions have completed.

signal_command_issued

Prototype: `signal_command_issued.`
Arguments: None.
Returns: void.
Description: Signals that the currently pending command has been driven to the interface.

signal_fatal_error

Prototype: `signal_fatal_error.`
Arguments: None.
Returns: void.
Description: Notifies the testbench that a fatal error has occurred in this module.

signal_max_command_queue_size

Prototype: `signal_max_command_queue_size.`
Arguments: None.
Returns: void.
Description: Signals that the maximum pending transaction queue size threshold has been exceeded.

signal_min_command_queue_size

Prototype: `signal_min_command_queue_size.`
Arguments: None.
Returns: void.
Description: Signals that the pending transaction queue size is below the minimum threshold.

signal_read_response_complete()

Prototype: `signal_read_response_complete.`
Arguments: None.
Returns: void.
Description: Signals that the read response has been received and inserted into the response queue.

signal_response_complete()

Prototype: `signal_response_complete.`
Arguments: None.
Returns: void.
Description: Triggers when either `signal_read_response_complete` or `signal_write_response_complete` is triggered.

signal_write_response_complete()

Prototype: `signal_write_response_complete.`
Arguments: None.
Returns: void.
Description: Signals that the write response has been received and inserted into the response queue.

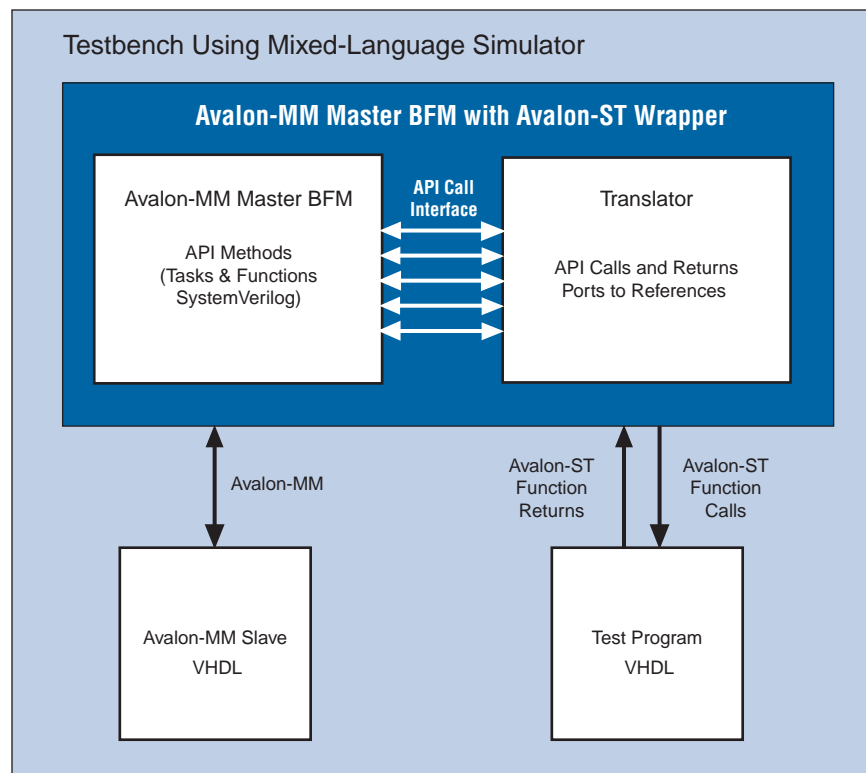
The Avalon-MM Master BFM with Avalon-ST API Wrapper provides an alternative way for the Avalon-MM Master BFM API to support VHDL testbenches. You can use the Avalon-MM Master BFM with Avalon-ST API Wrapper in HDL simulators that support mixed language simulation.



The API wrapper is only supported in SOPC Builder. The API wrapper cannot be generated in Qsys to create VHDL simulation models.

The Avalon-MM Master BFM with Avalon-ST API Wrapper component is implemented in SystemVerilog and uses an API wrapper to cast the Avalon-MM Master BFM's method calls and returns into signals that are carried on the call and return interface ports. To call a method, the method identifier is inserted into the BFM wrapper component via the channel field; the data is the arguments for the method. After the method is complete, the data field transports the arguments for the method call. The response is returned on the response Avalon-ST interface, and that Avalon-ST data signal carries the return value. The wrapper is necessary because VHDL can only access ports and does not support the method calls across hierarchical boundaries used in the Avalon-MM Master BFM field. [Figure 2–1](#) provides a high-level view the VHDL testbench communicating with the BFM.

Figure 2–1. Avalon-MM Master BFM with Avalon-ST Wrapper



In [Figure 2-1](#), the API call interface and Avalon-ST call and return interface operate in separate clock domains with `av_clk` synchronizing the FPGA logic and `api_clk` synchronizing the Avalon-ST translation interface. The Avalon-ST interface, which is not part of the actual hardware design, operates at much higher frequencies than the Avalon-MM Master BFM interface, enabling 1000 API calls and returns to be issued to the BFM per Avalon clock cycle.

For every function call in the BFM, there is a channel identifier that stores the fixed mapping between channel number and the function.

`<$install_dir>/ip/altera/sopc_builder_ip/verification/lib/`

`altera_avalon_components_pkg.vhd` defines the following function calls:

- `MM_MSTR_INIT`
- `MM_MSTR_SET_RESP_TIMEOUT`
- `MM_MSTR_SET_CMD_TIMEOUT`
- `MM_MSTR_ALL_TRANS_COMPLETE`
- `MM_MSTR_GET_CMD_ISSUE_QUEUE_SIZE`
- `MM_MSTR_GET_CMD_PEND_QUEUE_SIZE`
- `MM_MSTR_GET_RESP_QUEUE_SIZE`
- `MM_MSTR_PUSH_CMD`
- `MM_MSTR_POP_RESP`
- `MM_MSTR_SET_CMD_DATA`
- `MM_MSTR_SET_CMD_ADDRESS`
- `MM_MSTR_SET_CMD_BYTE_ENABLE`
- `MM_MSTR_SET_CMD_BURST_COUNT`
- `MM_MSTR_SET_CMD_IDLE`
- `MM_MSTR_SET_CMD_REQUEST`
- `MM_MSTR_SET_CMD_RESERVED_1`
- `MM_MSTR_GET_RESP_REQUEST`
- `MM_MSTR_GET_RESP_DATA`
- `MM_MSTR_GET_RESP_ADDRESS`
- `MM_MSTR_GET_RESP_BYTE_ENABLE`
- `MM_MSTR_GET_RESP_BURST_SIZE`
- `MM_MSTR_GET_RESP_LATENCY`
- `MM_MSTR_GET_RESP_WAIT_TIME`
- `MM_MSTR_SET_CMD_INIT_LATENCY`
- `MM_MSTR_SET_CMD_BURST_SIZE`

With the exception of the API wrapper, the Avalon-MM Master BFM with Avalon-ST API Wrapper component is identical to the Avalon-MM Master BFM. For more information about this component, refer to [Chapter 1, Avalon-MM Master BFM](#).

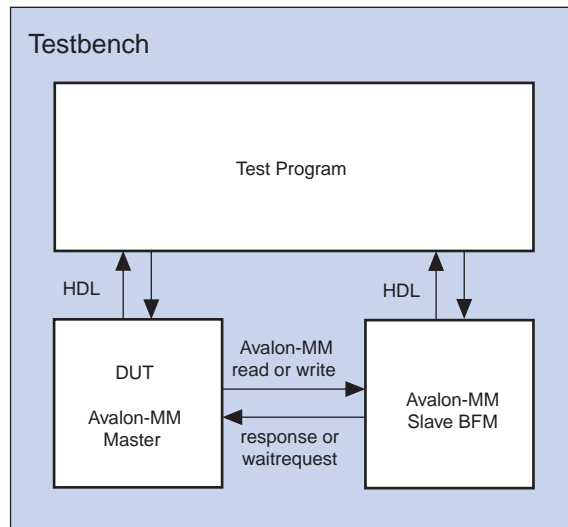
The Avalon-MM Slave BFM implements the slave side of the Avalon-MM interface protocol. This is a standard memory-mapped protocol including reads and writes typical of simple peripherals and the reads, writes, burst reads, and burst writes for typical memory devices. This BFM also includes a procedural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

Figure 3–1 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 3–1 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 Monitors components verify protocol compliance.

Figure 3–1. Top-Level Module to Verify an Avalon-MM Master



For more information about the Avalon-MM specification supported in SOPC Builder, refer to the [Avalon Interface Specifications \(version 1.3\)](#).



For more information about the Avalon-MM specification supported in Qsys, refer to the [Avalon Interface Specifications \(version 2.0\)](#).

Functional Description

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

- [“Timing” on page 3-3](#)
- [“Block Diagram” on page 3-6](#)

Timing

The timing diagram in [Figure 3-2](#) illustrates the sequence of events for an Avalon-MM Slave BFM responding to interleaved writes and reads when the `readdatavalid` signal is present.

Figure 3-2. Avalon-MM Slave Responding to Interleaved Write and Read Transactions

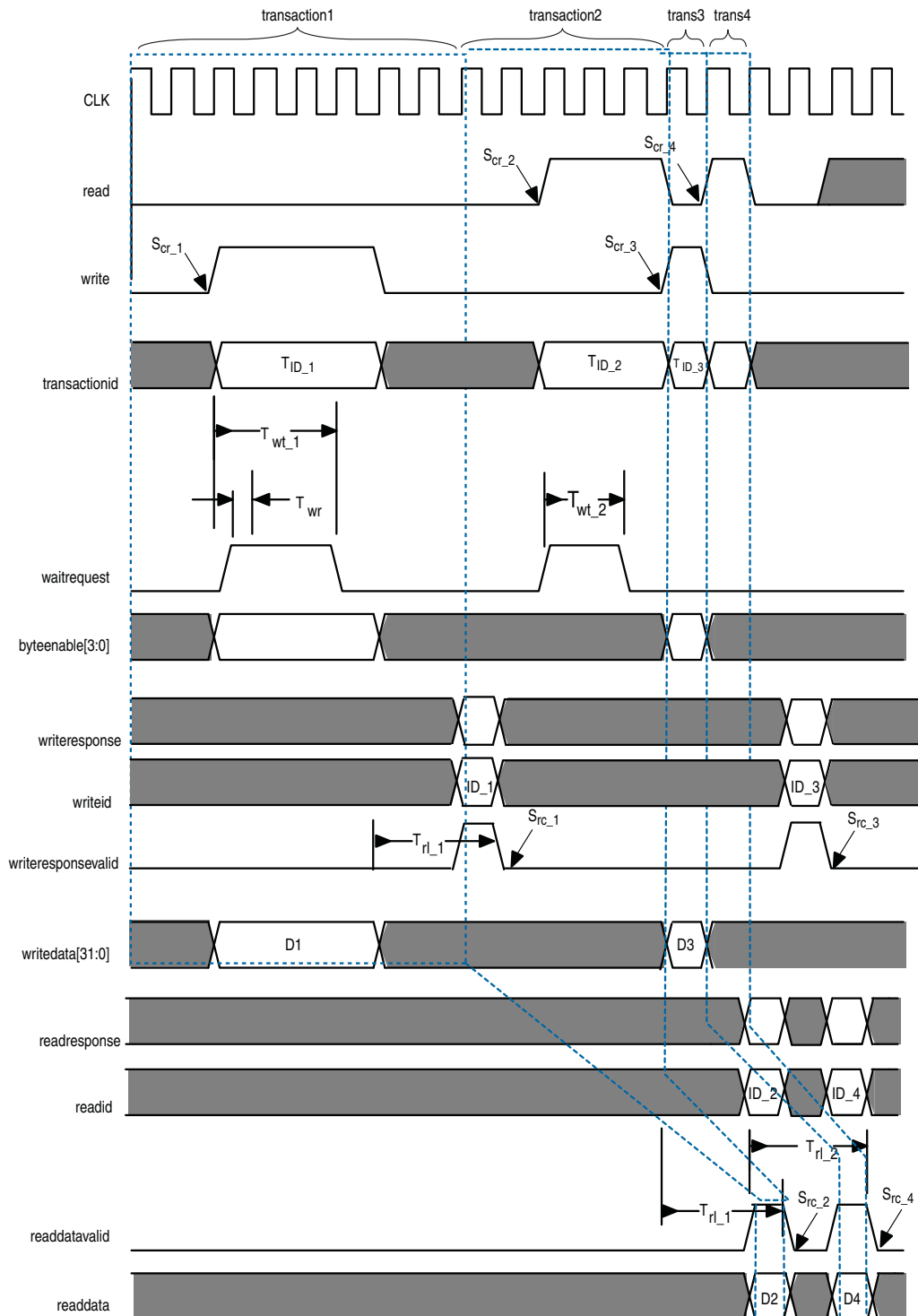


Table 3–1 lists the annotations used in Figure 3–2.

Table 3–1. Key to Annotations in Figure 3–2

Symbol	Description
T_{wt_1}	The response wait time, which is three cycles. The slave sets this value using the <code>set_interface_wait_time</code> command.
T_{wr}	waitrequest is sampled #1 after the falling edge of <code>clk</code> .
T_{wt_2}	The response wait time for the first read, which is two cycles. The slave sets this value using the <code>set_interface_wait_time</code> command.
S_{cr_1} – S_{cr_2}	Signals when read commands were received. The event name is <code>signal_command_received</code> .
T_{rl_1} , T_{rl_2}	The response latency for the reads, which is three cycles. The slave sets this time using the <code>set_response_latency</code> 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 <code>signal_response_issued</code> .
S_{rc_2} , S_{rc_4}	Signals read responses. The event name is <code>signal_response_issued</code> .
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.

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

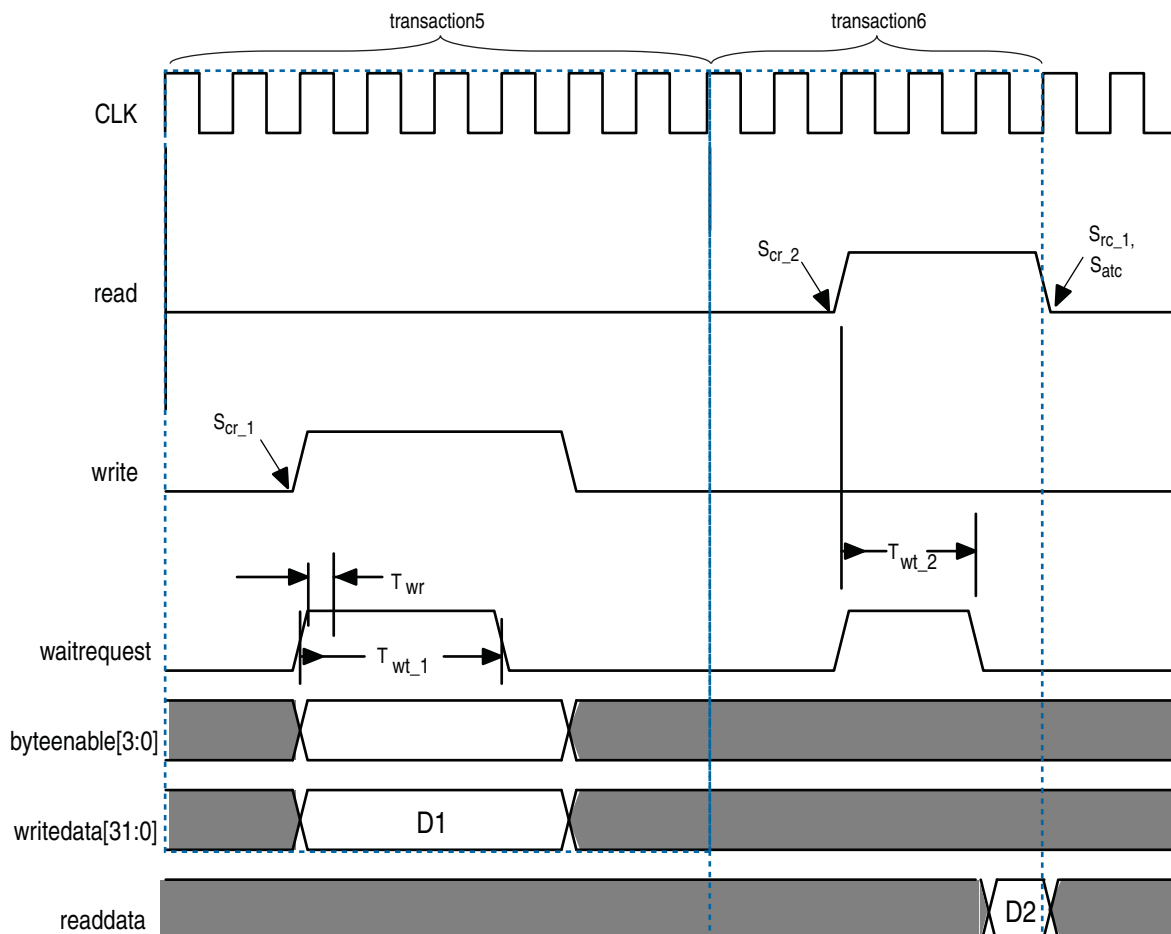


Table 3–2. Key to Annotations in Figure 3–3 (Part 1 of 2)

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

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

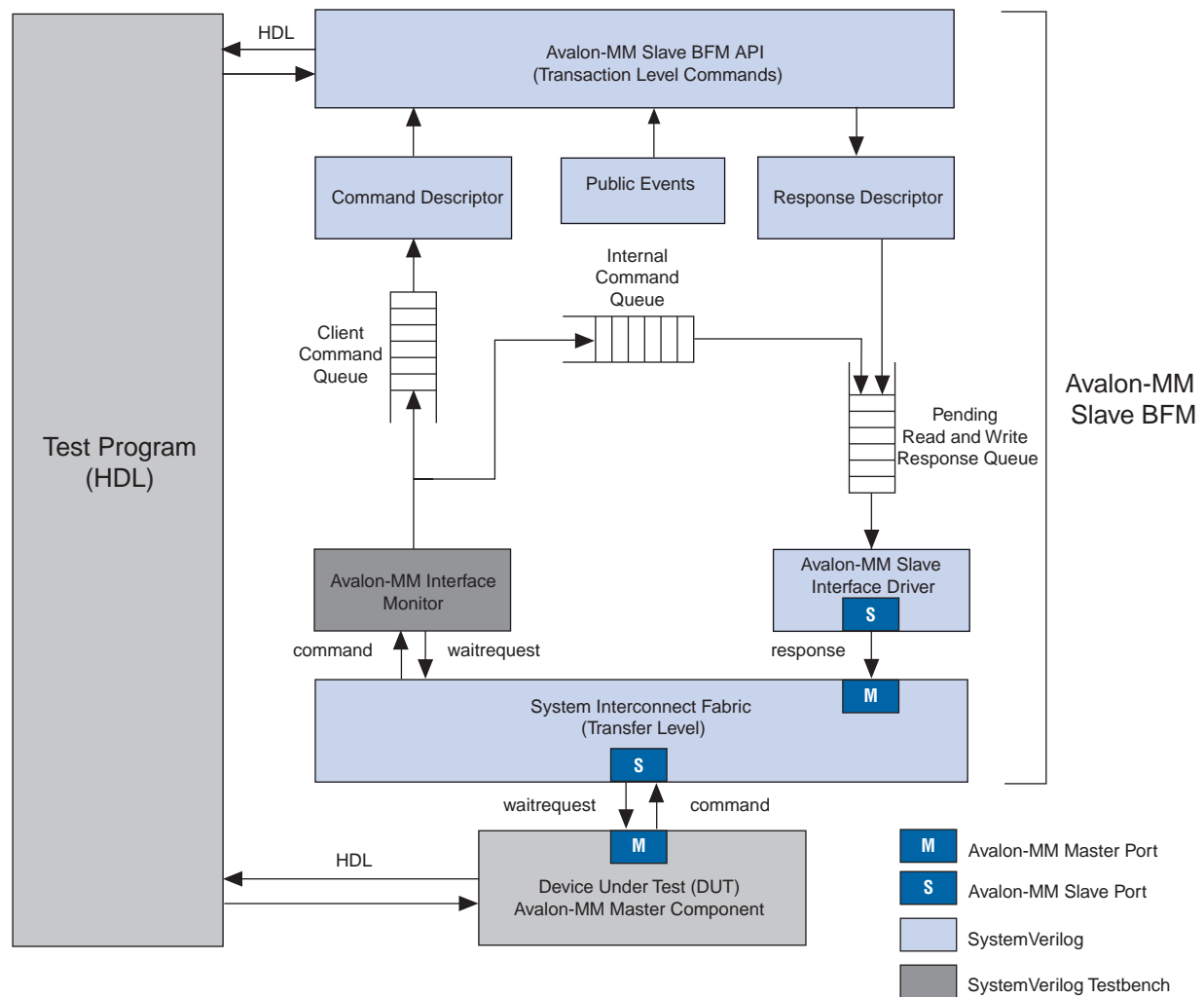
Symbol	Description
S _{rc_1}	Signals the first read response. The event name is <code>signal_response_complete</code> .
S _{atc}	Signals the end of the test. The event name is <code>signal_all_transactions_complete</code>

Block Diagram

Figure 3–4 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 3-4. 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 3-3](#).

Table 3-3. 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 <code>AV_SYMBOL_W</code> 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 <code>read</code> pin.
Use the write signal	On	On/Off	When On , the interface includes a <code>write</code> 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 <code>byte_enable</code> pins.
Use the burstcount signal	On	On/Off	When On , the interface includes <code>burstcount</code> pins.
Use the readdata signal	On	On/Off	When On , the interface includes a <code>readdata</code> pin.
Use the readdatavalid signal	On	On/Off	When On , the interface includes a <code>readdatavalid</code> pin.
Use the writedata signal	On	On/Off	When On , the interface includes a <code>writedata</code> pin.
Use the begintransfer signal	Off	On/Off	When On , the interface includes <code>writedata</code> pins.
Use the beginbursttransfer signal	Off	On/Off	When On , the interface includes a <code>beginbursttransfer</code> pin.
Use the arbiterlock signal	Off	On/Off	When On , the interface includes an <code>arbiterlock</code> pin.
Use the lock signal	Off	On/Off	When On , the interface includes a <code>lock</code> pin.
Use the debugaccess signal	Off	On/Off	When On , the interface includes a <code>debugaccess</code> pin.
Use the waitrequest signal	On	On/Off	When On , the interface includes a <code>waitrequest</code> pin.
Use the transactionid signal	Off	On/Off	When On , the interface includes a <code>transactionid</code> pin.
Use the write response signals	Off	On/Off	When On , the interface includes a <code>writeresponse</code> pin.
Use the read response signals	Off	On/Off	When On , the interface includes a <code>readresponse</code> pin.
Use the clken signals	Off	On/Off	When On , the interface includes a <code>clken</code> pin.
Port Polarity			
Assert reset high	On	On/Off	When On , <code>reset</code> is asserted high.
Assert waitrequest high	On	On/Off	When On , <code>waitrequest</code> is asserted high.
Assert read high	On	On/Off	When On , <code>read</code> is asserted high.
Assert write high	On	On/Off	When On , <code>write</code> is asserted high.

Table 3–3. 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 , <code>byteenable</code> is asserted high.
Assert readdatavalid high	On	On/Off	When On , <code>readdatavalid</code> is asserted high.
Assert arbiterlock high	On	On/Off	When On , <code>arbiterlock</code> is asserted high.
Assert lock high	On	On/Off	When On , <code>lock</code> 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.
Timing			
Fixed read latency (cycles)	0	—	Sets the read latency for fixed-latency slaves. Not used on interfaces that include the <code>readdatavalid</code> signal.
Fixed read wait time (cycles)	1	—	For slave interfaces that do not use the <code>waitrequest</code> 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 <code>waitrequest</code> for this number of cycles.
Fixed write wait time (cycles)	0	—	For slave interfaces that do not use the <code>waitrequest</code> 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.
API Streaming Interface (Note 1)			
Width of API interface data signal	64	—	The width of the data signal.
Width of API return interface data signal	64	—	The width of the return interface data signal.

Note to Table 3–3:

- (1) This interface is required only for the Avalon-MM Slave BFM with Avalon-ST API Wrapper that is used in mixed language simulations.

Application Program Interface

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

get_clken()

Prototype: `logic get_clken().`
Arguments: None.
Returns: `logic.`
Description: Returns the clock enable signal status.

get_command_address()

Prototype: `bit [AV_ADDRESS_W-1:0] get_command_address().`
Arguments: None.
Returns: `bit [AV_ADDRESS_W-1:0].`
Description: Queries the received command descriptor for the transaction address.

get_command_arbiterlock()

Prototype: `bit get_command_arbiterlock().`
Arguments: None.
Returns: `bit.`
Description: Queries the received command descriptor for the transaction arbiterlock.

get_command_burst_count()

Prototype: `[AV_BURSTCOUNT_W-1:0] get_command_burst_count().`
Arguments: None.
Returns: `[AV_BURSTCOUNT_W-1:0].`
Description: Queries the received command descriptor for the transaction burst count.

get_command_burst_cycle()

Prototype:	<code>int get_command_burst_cycle()</code> .
Arguments:	None.
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.

get_command_byte_enable()

Prototype:	<code>bit [AV_NUMSYMBOLS-1:0] get_command_byte_enable (int index).</code>
Arguments:	<code>index.</code>
Returns:	<code>bit [AV_NUMSYMBOLS-1:0].</code>
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.

get_command_data()

Prototype:	<code>bit [AV_DATA_W-1:0] get_command_data(int index).</code>
Arguments:	<code>index.</code>
Returns:	<code>bit [AV_DATA_W-1:0].</code>
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.

get_command_debugaccess()

Prototype:	<code>bit get_command_debugaccess()</code> .
Arguments:	None.
Returns:	bit.
Description:	Queries the received command descriptor for the transaction debugaccess.

get_command_queue_size()

Prototype:	<code>int get_command_queue_size()</code> .
Arguments:	None.
Returns:	int.
Description:	Queries the command queue to determine number of pending commands.

get_command_lock()

Prototype: `bit get_command_lock()`.
Arguments: None.
Returns: `bit`.
Description: Queries the received command descriptor for the transaction lock.

get_command_request()

Prototype: `Request_t get_command_request()`.
Arguments: None.
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`.

get_command_transaction_id()

Prototype: `AvalonTransactionId_t get_command_transaction_id()`.
Arguments: None.
Returns: `AvalonTransactionId_t`.
Description: Queries the received command descriptor for the transaction ID.

get_command_write_response_request()

Prototype: `AvalonTransactionId_t get_command_write_response_request()`.
Arguments: None.
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.

get_pending_read_latency_cycle()

Prototype: `int get_pending_read_latency_cycle()`.
Arguments: None.
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.

get_pending_write_latency_cycle()

Prototype:	<code>int get_pending_write_latency_cycle().</code>
Arguments:	None.
Returns:	<code>int</code> .
Description:	Queries the write command queue to determine the number of cycles needed for the Slave BFM to complete the current write response.

get_response_queue_size()

Prototype:	<code>int get_response_queue_size().</code>
Arguments:	None.
Returns:	<code>int</code> .
Description:	Queries the response queue to determine number of response descriptors pending.

get_slave_bfm_status

Prototype:	<code>bit get_slave_bfm_status.</code>
Arguments:	None.
Returns:	<code>bit</code> .
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.

get_version()

Prototype:	<code>string get_version().</code>
Arguments:	None.
Returns:	<code>String</code> .
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".

init()

Prototype:	<code>init().</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Initializes the Avalon-MM slave interface.

pop_command()

Prototype:	<code>void pop_command();</code>
Arguments:	None.
Returns:	void.
Description:	Removes the command descriptor from the queue so that the testbench can query it using the <code>get_command</code> methods.

push_response()

Prototype:	<code>void push_response();</code>
Arguments:	None.
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.

set_command_transaction_mode()

Prototype:	<code>void set_command_transaction_mode (int mode);</code>
Arguments:	mode.
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.

set_interface_wait_time()

Prototype:	<code>void set_interface_wait_time(int wait_cycles, int index).</code>
Arguments:	wait_cycles. index.
Returns:	void.
Description:	Specifies zero or more wait states to assert in each Avalon burst cycle by driving <code>waitrequest</code> 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.

set_max_response_queue_size()

Prototype:	<code>void set_max_response_queue_size(int size).</code>
Arguments:	int size.
Returns:	void.
Description:	Sets the maximum pending response queue size threshold.

set_min_response_queue_size()

Prototype: void set_min_response_queue_size(int size).
Arguments: int size.
Returns: void.
Description: Sets the minimum pending response queue size threshold.

set_read_response_id()

Prototype: void set_read_response_id(AvalonTransactionId_t id).
Arguments: AvalonTransactionId_t id.
Returns: void.
Description: Sets the transaction ID on the avs_readid pin.

set_read_response_status()

Prototype: void set_read_response_status(AvalonReadResponse_t status, int index).
Arguments: AvalonReadResponse_t status.
int index.
Returns: void.
Description: Sets the read response status code.

set_response_burst_size()

Prototype: void set_response_burst_size(bit [AV_BURSTCOUNT_W-1:0] burst_size).
Arguments: burst_size.
Returns: void.
Description: Sets the transaction burst count in the response descriptor.

set_response_data()

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

set_response_latency()

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

set_response_request()

Prototype: `void set_response_request(Request_t request).`
Arguments: Request_t request.
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.

set_response_timeout()

Prototype: `void set_response_timeout(int cycles).`
Arguments: None.
Returns: void.
Description: Sets the number of cycles that may elapse before timing out.

set_write_response_id()

Prototype: `void set_write_respose_id(AvalonTransactionId_t id).`
Arguments: AvalonTransactionId_t id.
Returns: void.
Description: Sets the transaction ID on the avs_writeid pin.

set_write_response_status()

Prototype: `void set_write_respose_status(AvalonWriteResponse_t status, int index).`
Arguments: AvalonWriteResponse_t status.
int index.
Returns: void.
Description: Sets the write response status code.

signal_command_received

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

signal_error_exceed_max_pending_reads

Prototype:	<code>signal_error_exceed_max_pending_reads.</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Notifies the testbench of the error condition, in which the slave has more than <code>max_pending_reads</code> pipelined read commands queued and waiting to be processed.

signal_max_response_queue_size

Prototype:	<code>signal_max_response_queue_size.</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Signals that the maximum pending transaction queue size threshold has been exceeded.

signal_min_command_queue_size

Prototype:	<code>signal_min_response_queue_size.</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Signals that the pending transaction queue size is below the minimum threshold.

signal_fatal_error

Prototype:	<code>signal_fatal_error.</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Notifies the testbench that a fatal error has occurred in this module.

signal_response_issued

Prototype:	<code>signal_response_issued.</code>
Arguments:	None.
Returns:	<code>void.</code>
Description:	Notifies the testbench that a response has been driven out on the Avalon bus.

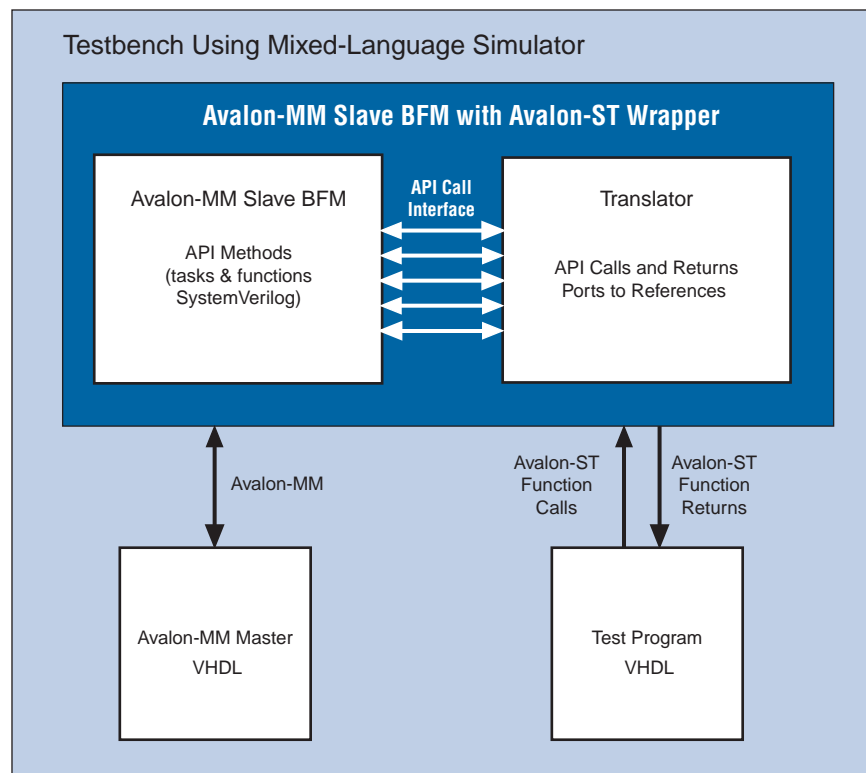
The Avalon-MM Slave BFM with Avalon-ST API Wrapper provides an alternative way for the Avalon-MM Slave BFM API to support VHDL testbenches. You can use the Avalon-MM Slave BFM with Avalon-ST API Wrapper in HDL simulators that support mixed language simulation.



The API wrapper is only supported in SOPC Builder. The API wrapper cannot be generated in Qsys to create VHDL simulation models.

The Avalon-MM Slave BFM with Avalon-ST API Wrapper component is implemented in SystemVerilog and uses an API wrapper to cast the Avalon-MM BFM's method calls and returns into signals that are carried on the call and return interface ports. To call a method, the method identifier is inserted into the wrapper component via the channel field; the data is the arguments for the method. After the method is complete, the data field transports the arguments for the method call. The response is returned on the response Avalon-ST interface, and that Avalon-ST data signal carries the return value. The wrapper is necessary because VHDL can only access ports and does not support the method calls across hierarchical boundaries used in the Avalon-MM Master BFM field. [Figure 4-1](#) provides a high-level view of this VHDL testbench communicating with the BFM.

Figure 4-1. Avalon-MM Slave BFM with Avalon-ST Wrapper



In [Figure 4-1](#), the API call interface and Avalon-ST call and return interface operate in separate clock domains with `av_clk` synchronizing the FPGA logic and `api_clk` synchronizing the Avalon-ST translation interface. The Avalon-ST interface, which is not part of the actual hardware design, operates at a much faster frequency than the Avalon-MM Slave BFM interface, enabling 1000 API calls and returns to be issued to the BFM per Avalon clock cycle.

For every function call in the BFM, there is a channel identifier that stores the fixed mapping between channel number and the function.

`<$install_dir>/ip/altera/sopc_builder_ip/verification/lib/`

`altera_avalon_components_pkg.vhd` defines the following function calls:

- `MM_SLV_SIGNAL_FATAL_ERROR`
- `MM_SLV_SIGNAL_ERROR_EXCEED_MAX_PENDING_READS`
- `MM_SLV_SIGNAL_COMMAND_RECEIVED`
- `MM_SLV_SIGNAL_RESP_ISSUED`
- `MM_SLV_SIGNAL_RESERVED_4`
- `MM_SLV_SIGNAL_RESERVED_5`
- `MM_SLV_SIGNAL_RESERVED_6`
- `MM_SLV_SIGNAL_RESERVED_7`

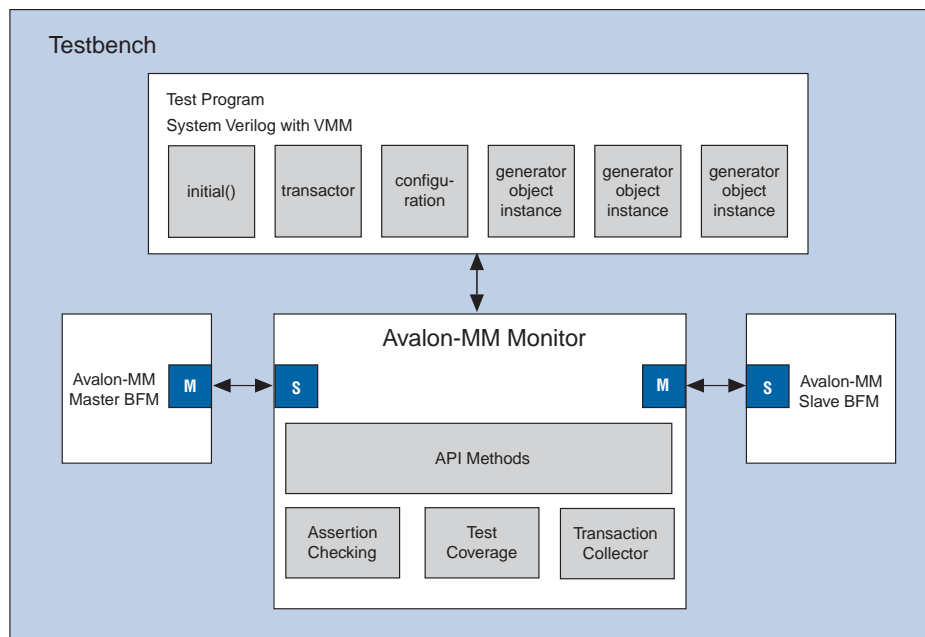
With the exception of the API wrapper, the Avalon-MM Slave BFM with Avalon-ST API Wrapper component is identical to the Avalon-MM Slave BFM. For more information about this component, refer to [Chapter 3, Avalon-MM Slave BFM](#).

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 5–1 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 5–1. 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 5-1](#).

Table 5-1. 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 5-1. 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 <code>readdatavalid</code> 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 <code>waitrequest</code> .
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 <code>readdatavalid</code> 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.
Timing			
Fixed read wait time (cycles)	1	—	For master interfaces that do not use the <code>waitrequest</code> 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 <code>waitrequest</code> for this number of cycles.
Fixed write wait time (cycles)	0	—	For master interfaces that do not use the <code>waitrequest</code> 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: Boolean.
Returns: void.
Description: Enables an assertion that ensures the byte address that the master uses is aligned with the data width.

set_enable_a_beginbursttransfer_exist()

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

set_enable_a_beginbursttransfer_legal()

Prototype: `set_enable_a_beginbursttransfer_legal()`.
Arguments: Boolean.
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.

set_enable_a_beginbursttransfer_single_cycle()

Prototype: `set_enable_a_beginbursttransfer_single_cycle()`.
Arguments: Boolean.
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.

set_enable_a_begintransfer_exist()

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

set_enable_a_begintransfer_legal()

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

set_enable_a_begintransfer_single_cycle()

Prototype: `set_enable_a_begintransfer_single_cycle()`.
Arguments: Boolean.
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.

set_enable_a_burst_legal()

Prototype: `set_enable_a_burst_legal()`.
Arguments: Boolean.
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.

set_enable_a_byteenable_legal()

Prototype: `set_enable_a_byteenable_legal()`.
Arguments: Boolean.
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.

set_enable_a_constant_during_burst()

Prototype: `set_enable_a_constant_during_burst()`.
Arguments: Boolean.
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.

set_enable_a_constant_during_clk_disabled()

Prototype: `set_enable_a_constant_during_clk_disabled()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that ensures that all signals are held constant if `clken` is deasserted.

set_enable_a_constant_during_waitrequest()

Prototype: `set_enable_a_constant_during_waitrequest()`.
Arguments: Boolean.
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.

set_enable_a_exclusive_read_write()

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

set_enable_a_half_cycle_reset_legal()

Prototype: `set_enable_a_half_cycle_reset_legal()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that ensures `reset` is asserted correctly.

set_enable_a_less_than_burstcount_max_size()

Prototype: `set_enable_a_less_than_burstcount_max_size()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that ensures `burstcount` size is less than or equal to the maximum burst size, $2^{**}(\text{AV_BURSTCOUNT_W}-1)$. It is disabled when either burst transfers are not supported or the bust size is less than 1.

set_enable_a_less_than_maximumpendingreadtransactions()

Prototype: `set_enable_a_less_than_maximumpendingreadtransactions()`.
Arguments: Boolean.
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.

set_enable_a_no_readdatavalid_during_reset()

Prototype: `set_enable_a_no_readdatavalid_during_reset()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that ensures that `readdatavalid` is deasserted if `reset` is asserted. Disabled when `readdatavalid` is not supported.

set_enable_a_no_read_during_reset()

Prototype: `set_enable_a_no_read_during_reset()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that ensures `read` is deasserted if `reset` is asserted. Disabled when `read` is not supported.

set_enable_a_no_write_during_reset()

Prototype: `set_enable_a_no_write_during_reset()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that ensures `write` is deasserted if `reset` is asserted. Disabled when `write` is not supported.

set_enable_a_readid_sequence()

Prototype: `set_enable_a_readid_sequence()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that verifies if the `readid` sequence follows the sequence of the `transactionid`.

set_enable_a_read_response_sequence()

Prototype: `set_enable_a_read_response_sequence()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that ensures `readdatavalid` is asserted while `read` is asserted for the same read transfer.

set_enable_a_read_response_timeout()

Prototype: `set_enable_a_read_response_timeout()`.
Arguments: Boolean.
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.

set_enable_a_register_incoming_signals()

Prototype: `set_enable_a_register_incoming_signals()`.
Arguments: Boolean.
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.

set_enable_a_waitrequest_during_reset()

Prototype: `set_enable_a_waitrequest_during_reset1()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that ensures that `waitrequest` is asserted if `reset` is asserted. Disabled when `waitrequest` is not supported.

set_enable_a_waitrequest_timeout()

Prototype: `set_enable_a_waitrequest_timeout()`.
Arguments: Boolean.
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.

set_enable_a_write_burst_timeout()

Prototype: `set_enable_a_write_burst_timeout()`.
Arguments: Boolean.
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.

set_enable_a_writeid_sequence()

Prototype: `set_enable_a_writeid_sequence()`.
Arguments: Boolean.
Returns: void.
Description: Enables an assertion that verifies if the `writeid` sequence follows the sequence of the `transactionid`.

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: Boolean.
Returns: void.
Description: Enables a coverage group to test back-to-back read transfers. This method is disabled when reads are not supported.

set_enable_c_b2b_read_write()

Prototype: `set_enable_c_b2b_read_write()`.
Arguments: Boolean.
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.

set_enable_c_b2b_write_read()

Prototype: `set_enable_c_b2b_write_read()`.
Arguments: Boolean.
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.

set_enable_c_b2b_write_write()

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

set_enable_c_continuous_read()

Prototype: `set_enable_c_continuous_read()`.
Arguments: Boolean.
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.

set_enable_c_continuous_readdatavalid()

Prototype: `set_enable_c_continuous_readdatavalid()`.
Arguments: Boolean.
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.

set_enable_c_continuous_waitrequest()

Prototype: `set_enable_c_continuous_waitrequest()`.
Arguments: Boolean.
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.

set_enable_c_continuous_waitrequest_from_idle_to_read()

Prototype: `set_enable_c_continuous_waitrequest_from_idle_to_read()`.
Arguments: Boolean.
Returns: void.
Description: Enables a coverage group to test waitrequest transfers from their idle state until a waitrequest read.

set_enable_c_continuous_waitrequest_from_idle_to_write()

Prototype: `set_enable_c_continuous_waitrequest_from_idle_to_write()`.
Arguments: Boolean.
Returns: void.
Description: Enables a coverage group to test waitrequest transfers from their idle state until a waitrequest write.

set_enable_c_continuous_write()

Prototype: `set_enable_c_continuous_write()`.
Arguments: Boolean.
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.

set_enable_c_idle_before_transaction()

Prototype: `set_enable_c_idle_before_transaction()`.
Arguments: Boolean.
Returns: void.
Description: Enables a coverage group to count idle cycles before read or write transactions.

set_enable_c_idle_in_read_response()

Prototype: `set_enable_c_idle_in_read_response()`.
Arguments: Boolean.
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.

set_enable_c_idle_in_write_burst()

Prototype: `set_enable_c_idle_in_write_burst()`.
Arguments: Boolean.
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.

set_enable_c_pending_read()

Prototype: `set_enable_c_pending_read()`.
Arguments: Boolean.
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.

set_enable_c_read()

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

set_enable_c_read_after_reset()

Prototype: `set_enable_c_read_after_reset()`.
Arguments: Boolean.
Returns: void.
Description: Enables a coverage group to test read transfers after reset.

set_enable_c_read_burstcount()

Prototype: `set_enable_c_read_burstcount()`.
Arguments: Boolean.
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.

set_enable_c_read_byteenable()

Prototype: `set_enable_c_read_byteenable()`.
Arguments: Boolean.
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.

set_enable_c_read_latency()

Prototype: `set_enable_c_read_latency()`.
Arguments: Boolean.
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.

set_enable_c_read_response()

Prototype: `set_enable_c_read_response()`.
Arguments: Boolean.
Returns: void.
Description: Enables a coverage group to test each bit of the valid readresponse that represent different status.

set_enable_c_waitrequest_in_write_burst()

Prototype: `set_enable_c_waitrequest_in_write_burst()`.
Arguments: Boolean.
Returns: void.
Description: Enables a coverage group to test the values of the `waitrequest` parameter during write burst transfers.

set_enable_c_waitrequested_read()

Prototype: `set_enable_c_waitrequested_read()`.
Arguments: Boolean.
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.

set_enable_c_waitrequest_without_command()

Prototype: `set_enable_c_waitrequest_without_command()`.
Arguments: Boolean.
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.

set_enable_c_waitrequested_write()

Prototype: `set_enable_c_waitrequested_write()`.
Arguments: Boolean.
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.

set_enable_c_write()

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

set_enable_c_write_with_and_without_writerresponserequest()

Prototype: `set_enable_c_write_with_and_without_writerresponserequest()`.
Arguments: Boolean.
Returns: void.
Description: Enables a coverage group to test write transactions with or without `writerresponserequest`.

set_enable_c_write_after_reset()

Prototype: `set_enable_c_write_after_reset()`.
Arguments: Boolean.
Returns: void.
Description: Enables a coverage group to test write transfers after reset.

set_enable_c_write_burstcount()

Prototype: `set_enable_c_write_burstcount()`.
Arguments: Boolean.
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.

set_enable_c_write_byteenable()

Prototype:	<code>set_enable_c_write_byteenable()</code> .
Arguments:	Boolean.
Returns:	void.
Description:	Enables a coverage group ensures all legal values of the <code>byteenable</code> signal are asserted during write transfers. It is disabled when either <code>byteenable</code> or <code>write</code> is not supported.

set_enable_c_write_response()

Prototype:	<code>set_enable_c_write_response()</code> .
Arguments:	Boolean.
Returns:	void.
Description:	Enables a coverage group to test each bit of the valid <code>writeresponse</code> that represent different status.

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.

get_clken()

Prototype:	<code>logic get_clken()</code> .
Arguments:	None.
Returns:	logic.
Description:	Returns the clock enable signal status.

get_version()

Prototype:	<code>string get_version()</code> .
Arguments:	None.
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".

get_command_address()

Prototype: bit [AV_ADDRESS_W-1:0] get_command_address().
Arguments: None.
Returns: bit [AV_ADDRESS_W-1:0].
Description: Queries the received command descriptor for the transaction address.

get_command_arbiterlock()

Prototype: bit get_command_arbiterlock().
Arguments: None.
Returns: bit.
Description: Queries the received command descriptor for the transaction arbiterlock.

get_command_burst_count()

Prototype: [AV_BURSTCOUNT_W-1:0] get_command_burst_count().
Arguments: None.
Returns: [AV_BURSTCOUNT_W-1:0].
Description: Queries the received command descriptor for the transaction burst count.

get_command_burst_cycle()

Prototype: int get_command_burst_cycle().
Arguments: None.
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.

get_command_byte_enable()

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

get_command_data()

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

get_command_debugaccess()

Prototype: `bit get_command_debugaccess().`
Arguments: `None.`
Returns: `bit.`
Description: Queries the received command descriptor for the transaction debugaccess.

get_command_issued_queue_size()

Prototype: `int get_command_issued_queue_size().`
Arguments: `None.`
Returns: `int.`
Description: Queries the command issued queue to determine number of pending commands.

get_command_queue_size()

Prototype: `int get_command_queue_size().`
Arguments: `None.`
Returns: `int.`
Description: Queries the command queue to determine number of pending commands.

get_command_lock()

Prototype: `bit get_command_lock().`
Arguments: `None.`
Returns: `bit.`
Description: Queries the received command descriptor for the transaction lock.

get_command_request()

Prototype: Request_t get_command_request().
Arguments: None.
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.

get_command_transaction_id()

Prototype: AvalonTransactionId_t get_command_transaction_id().
Arguments: None.
Returns: AvalonTransactionId_t.
Description: Queries the received command descriptor for the transaction ID.

get_command_write_response_request()

Prototype: AvalonTransactionId_t
get_command_write_response_request().
Arguments: None.
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.

get_read_response_queue_size()

Prototype: int get_read_response_queue_size().
Arguments: None.
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.

get_response_address()

Prototype: bit [AV_ADDRESS_W-1:0] get_response_address().
Arguments: None.
Returns: bit[AV_ADDRESS_W-1:0].
Description: Returns the transaction address in the response descriptor that has been removed from the response queue.

get_response_byte_enable()

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

Arguments: `index.`

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.

get_response_burst_size()

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

Arguments: `None.`

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.

get_response_data()

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

Arguments: `index.`

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.

get_response_latency()

Prototype: `int get_response_latency(int index).`

Arguments: `index.`

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.

get_response_queue_size()

Prototype: `int get_response_queue_size().`

Arguments: `None.`

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.

get_response_read_id()

Prototype: AvalonTransactionId_t get_response_read_id().
Arguments: None.
Returns: AvalonTransactionId_t.
Description: Returns the read id of the transaction in the response descriptor that has been removed from the response queue.

get_response_read_response()

Prototype: AvalonReadResponse_t get_response_read_response(int index).
Arguments: int index.
Returns: AvalonReadResponse_t.
Description: Returns the transaction read status in the response descriptor that has been removed from the response queue.

get_response_request()

Prototype: Request_t get_response_request().
Arguments: None.
Returns: Request_t.
Description: Returns the transaction command type in the response descriptor that has been removed from the response queue.

get_response_wait_time()

Prototype: int get_response_wait_time(int index).
Arguments: index.
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.

get_response_write_id()

Prototype: AvalonTransactionId_t get_response_write_id().
Arguments: None.
Returns: AvalonTransactionId_t.
Description: Returns the write id of the transaction in the response descriptor that has been removed from the response queue.

get_response_write_response()

Prototype: AvalonWriteResponse_t get_response_write_response(int index).
Arguments: index.
Returns: AvalonWriteResponse_t.
Description: Returns the transaction write status in the response descriptor that has been removed from the response queue.

get_transaction_fifo_max()

Prototype: int get_transaction_fifo_max().
Arguments: None.
Returns: int.
Description: Gets the maximum transaction FIFO depth.

get_transaction_fifo_threshold()

Prototype: int get_transaction_fifo_threshold().
Arguments: None.
Returns: int.
Description: Gets the transaction FIFO threshold level.

get_write_response_queue_size()

Prototype: int get_write_response_queue_size().
Arguments: None.
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.

init()

Prototype: init().
Arguments: None.
Returns: void.
Description: Initializes the counters and clears the queue.

pop_command()

Prototype:	<code>pop_command()</code> .
Arguments:	None.
Returns:	Void.
Description:	Removes the command descriptor from the queue so that the testbench can query it with the <code>get_command</code> methods.

pop_response()

Prototype:	<code>void pop_response()</code> .
Arguments:	None.
Returns:	void.
Description:	Removes the transaction descriptor from the queue so that the testbench can query it with the <code>get_command</code> methods. Sequence counter is initialized to 1.

set_command_transaction_mode()

Prototype:	<code>set_command_transaction_mode()</code> .
Arguments:	<code>int mode</code> .
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.

set_transaction_fifo_max()

Prototype:	<code>set_transaction_fifo_max()</code> .
Arguments:	<code>int level</code> .
Returns:	void.
Description:	Sets the maximum transaction level of the FIFO. The event <code>signal_transaction_fifo_max</code> is triggered when this level is exceeded.

set_transaction_fifo_threshold()

Prototype:	<code>set_transaction_fifo_threshold()</code> .
Arguments:	<code>int level</code> .
Returns:	void.
Description:	Sets the threshold alert level of the FIFO. The event <code>signal_transaction_fifo_threshold</code> is triggered when this level is exceeded.

signal_command_received

Prototype: `signal_command_received.`
Arguments: None.
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.

signal_fatal_error

Prototype: `signal_fatal_error.`
Arguments: None.
Returns: `void.`
Description: Notifies the testbench that a fatal error has occurred in this module.

signal_read_response_complete

Prototype: `signal_read_response_complete.`
Arguments: None.
Returns: `void.`
Description: Notifies the testbench that the read response has been received and inserted into the response queue.

signal_response_complete

Prototype: `signal_response_complete.`
Arguments: None.
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.

signal_transaction_fifo_overflow

Prototype: `signal_transaction_fifo_overflow.`
Arguments: None.
Returns: `void.`
Description: Notifies the testbench that the FIFO is full and further transactions are dropped.

signal_transaction_fifo_threshold

Prototype: `signal_transaction_fifo_threshold.`
Arguments: None.
Returns: `void.`
Description: Notifies the testbench that the transaction FIFO threshold level has exceeded.

signal_write_response_complete

Prototype: `signal_write_response_complete.`
Arguments: None.
Returns: `void.`
Description: Notifies the testbench that the write response has been received and inserted into the response queue.

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

- [Chapter 1, Avalon-ST Source BFM](#)
- [Chapter 2, Avalon-ST Source BFM with Avalon-ST API Wrapper](#)
- [Chapter 3, Avalon-ST Sink BFM](#)
- [Chapter 4, Avalon-ST Sink BFM with Avalon-ST API Wrapper](#)
- [Chapter 5, Avalon-ST Monitor](#)

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.


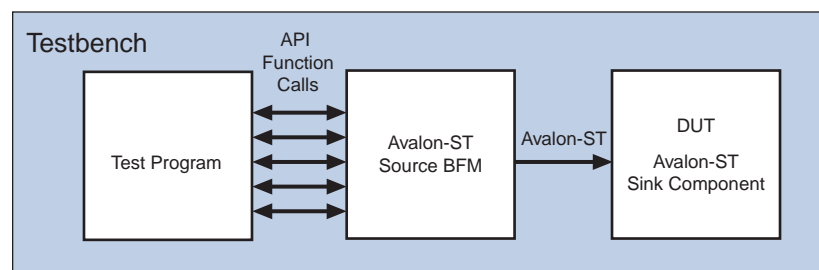


 The BFMs allow illegal transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitors components verify protocol compliance.

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



-  For more information about the Avalon-ST specification supported in SOPC Builder, refer to the [Avalon Interface Specifications \(version 1.3\)](#).
-  For more information about the Avalon-ST specification supported in Qsys, refer to the [Avalon Interface Specifications \(version 2.0\)](#).

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](#)

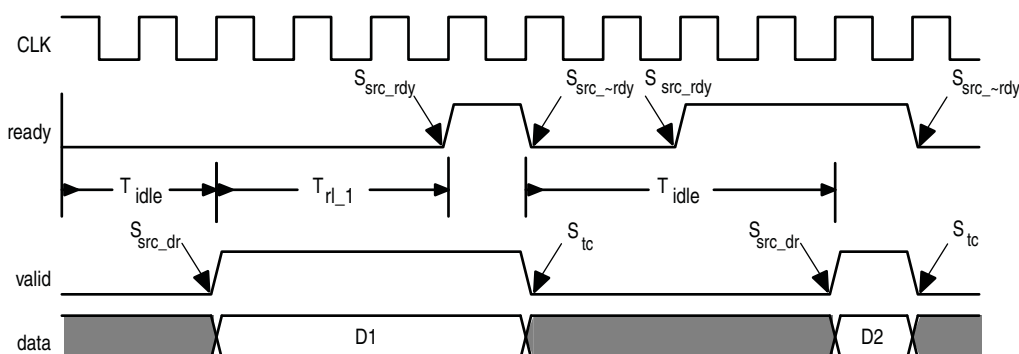
Timing

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](#)

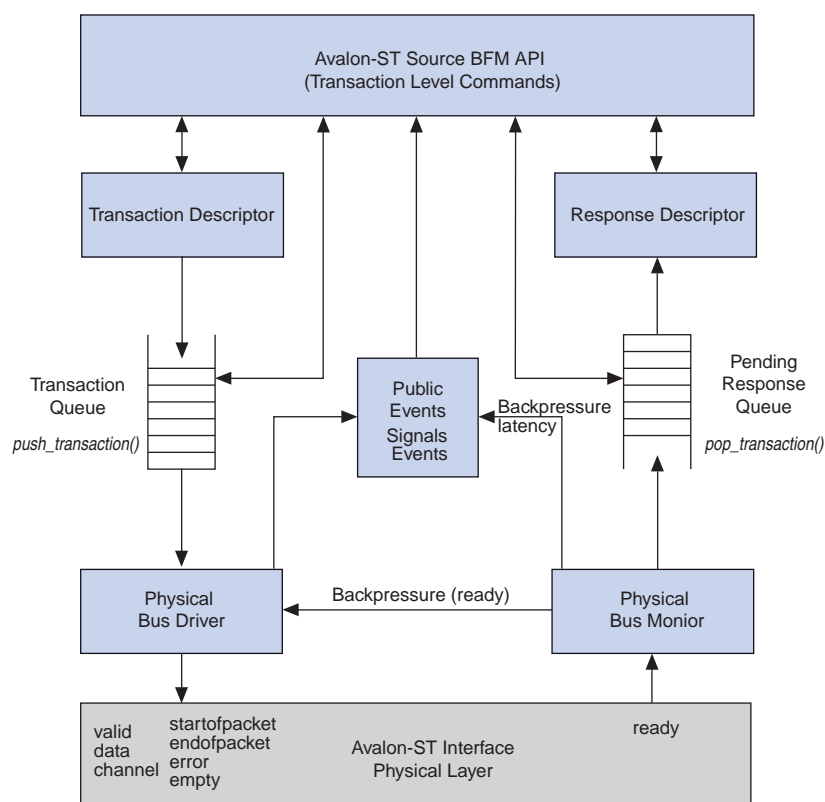
Symbol	Description
T_{idle}	The idle time before a transactions. This time is set by the command <code>set_transaction_idles</code> .
T_{rl_1}	The response latency for the first source to sink transaction, which is three cycles. The source gets this time using the <code>get_response_latency</code> command.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

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 <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals.
Use the channel port	Off	On/Off	When On, the interface includes <code>channel</code> pin or pins.
Use the error port	Off	On/Off	When On, the interface includes <code>error</code> pin or pins.
Use the ready port	On	On/Off	When On, the interface includes a <code>ready</code> pin.
Use the valid port	On	On/Off	When On, the interface includes a <code>valid</code> pin.
Use the empty port	Off	On/Off	When On, the interface includes <code>empty</code> 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 <code>channel</code> signal.
Width of the error port	1	1-1024	Specifies the width of the <code>error</code> signal.
Width of the empty port	1	1-1024	Specifies the width of the <code>empty</code> signal.
Timing Attributes			
Ready latency	0	0-8	Specifies the delay between the <code>ready</code> and <code>valid</code> 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.
API Streaming Interface (Note 1)			
Width of API interface data signal	64	—	The width of the data signal.
Width of API return interface data signal	64	—	The width of the return interface data signal.

Note to Table 1-2:

- (1) This interface is required only for the Avalon-ST Source BFM with Avalon-ST API Wrapper which is used in mixed language simulations.

Application Program Interface

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

get_response_latency()

Prototype: `get_response_latency()`.
Arguments: None.
Returns: `int`.
Description: Returns the response latency in cycles due to back pressure for the most recently removed transaction.

get_response_queue_size()

Prototype: `get_response_queue_size()`.
Arguments: None.
Returns: `int`.
Description: Returns the number of transactions in the response queues.

get_src_ready()

Prototype: `get_src_ready()`.
Arguments: None.
Returns: `bit`.
Description: Returns the value of the source ready port.

get_src_transaction_complete()

Prototype: `get_src_transaction_complete()`.
Arguments: None.
Returns: `bit`.
Description: Returns the transaction complete status.

get_transaction_queue_size()

Prototype: `get_transaction_queue_size()`.
Arguments: None.
Returns: `int`.
Description: Returns the number of transactions in the local queues.

get_version()

Prototype:	<code>get_version()</code> .
Arguments:	None.
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".

init()

Prototype:	<code>init()</code> .
Arguments:	None.
Returns:	void.
Description:	Drives the interface to the idle state.

pop_response()

Prototype:	<code>pop_response()</code> .
Arguments:	None.
Returns:	void.
Description:	Removes the response transaction from the queue before querying contents.

push_transaction()

Prototype:	<code>push_transaction()</code> .
Arguments:	None.
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.

set_max_transaction_queue_size()

Prototype:	<code>void set_max_transaction_queue_size(int size).</code>
Arguments:	<code>int size.</code>
Returns:	void.
Description:	Sets the pending transaction queue size maximum threshold. The public event <code>signal_max_transaction_queue_size</code> triggers when the threshold is exceeded.

set_min_transaction_queue_size()

Prototype:	<code>void set_min_transaction_queue_size(int size).</code>
Arguments:	<code>int size.</code>
Returns:	<code>void.</code>
Description:	Sets the pending transaction minimum queue size threshold. The public event <code>signal_min_transaction_queue_size</code> triggers when the queue size level is below the minimum threshold.

set_response_timeout()

Prototype:	<code>set_response_timeout(int cycles).</code>
Arguments:	<code>cycles.</code>
Returns:	<code>void.</code>
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.

set_transaction_channel()

Prototype:	<code>set_transaction_channel(STChannel_t channel).</code>
Arguments:	<code>channel.</code>
Returns:	<code>void.</code>
Description:	Sets the channel identifier in the out-going transaction.

set_transaction_data()

Prototype:	<code>set_transaction_data(STData_t data).</code>
Arguments:	<code>data.</code>
Returns:	<code>void.</code>
Description:	Sets the value of data in the out-going transaction.

set_transaction_idles()

Prototype:	<code>set_transaction_idles(bit[31:0] idle_cycles).</code>
Arguments:	<code>idle_cycles.</code>
Returns:	<code>void.</code>
Description:	Sets the number of idle cycles to elapse before driving the out-going transaction.

set_transaction_eop()

Prototype:	<code>set_transaction_eop(bit eop).</code>
Arguments:	<code>eop.</code>
Returns:	<code>void.</code>
Description:	Sets the status of the end of packet signal in the out-going transaction.

set_transaction_empty()

Prototype: `set_transaction_empty(STEmpty_t empty).`
Arguments: `empty.`
Returns: `void.`
Description: Sets the out-going transaction empty value.

set_transaction_error()

Prototype: `set_transaction_error(STError_t error).`
Arguments: `error.`
Returns: `void.`
Description: Sets the out-going transaction error value.

set_transaction_sop()

Prototype: `set_transaction_sop(bit sop).`
Arguments: `sop.`
Returns: `void.`
Description: Sets the status of the start of packet signal in the out-going transaction.

signal_fatal_error

Prototype: `signal_fatal_error.`
Arguments: `None.`
Returns: `void.`
Description: Signals that a fatal error has occurred. It terminates the simulation.

signal_max_transaction_queue_size

Prototype: `signal_max_transaction_queue_size.`
Arguments: `None.`
Returns: `void.`
Description: Signals that the pending transaction queue size threshold has been exceeded.

signal_min_transaction_queue_size

Prototype: `signal_min_transaction_queue_size.`
Arguments: `None.`
Returns: `void.`
Description: Signals that the pending transaction queue size is below the minimum threshold.

signal_response_done

Prototype: `signal_response_done.`
Arguments: None.
Returns: `void.`
Description: Signals that the response to a driven data beat is available.

signal_src_driving_transaction

Prototype: `signal_src_driving_transaction.`
Arguments: None.
Returns: `void.`
Description: Signals when the source begins to drive a transaction to the interface.

signal_src_not_ready

Prototype: `signal_src_not_ready.`
Arguments: None.
Returns: `void.`
Description: Signals that the `ready` signal is not asserted.

signal_src_ready

Prototype: `signal_src_ready.`
Arguments: None.
Returns: `void.`
Description: Signals that the `ready` signal is asserted.

signal_src_transaction_complete

Prototype: `signal_src_transaction_complete.`
Arguments: None.
Returns: `void.`
Description: Signals that all pending transactions have completed.

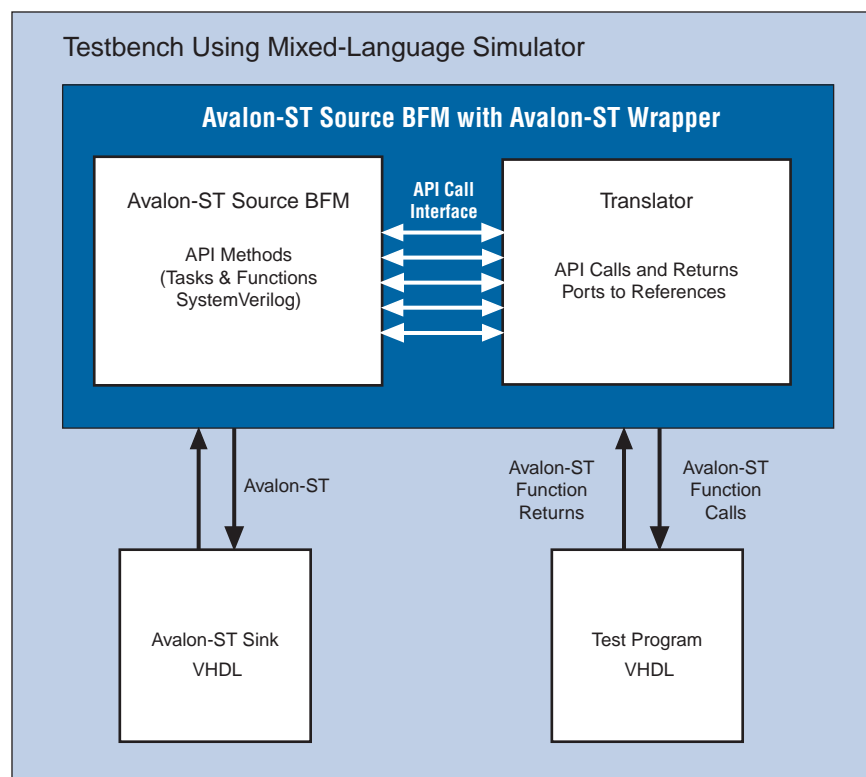
The Avalon-ST Source BFM with Avalon-ST API Wrapper provides VHDL support for the Avalon-ST Source BFM. You can use the Avalon-ST Source BFM with Avalon-ST API Wrapper in HDL simulators that support mixed language simulation.



The API wrapper is only supported in SOPC Builder. The API wrapper cannot be generated in Qsys to create VHDL simulation models.

The Avalon-ST Source BFM with Avalon-ST API Wrapper component is implemented in SystemVerilog and uses an API wrapper to cast the Avalon-ST Source BFM's method calls and returns into signals that are carried on the call and return interface ports. The wrapper is necessary because VHDL can only access ports and does not support the method calls used in the Avalon-ST Source BFM. [Figure 2-1](#) provides a high-level view of this component.

Figure 2-1. Avalon-ST Source BFM with Avalon-ST Wrapper



In [Figure 2-1](#), the API call interface and Avalon-ST call and return interface operate in separate clock domains with `av_clk` synchronizing the FPGA logic and `api_clk` synchronizing the Avalon-ST translation interface. The Avalon-ST interface, which is not part of the actual hardware design, operates at a much faster clock frequency than the Avalon-ST Source BFM interface.

For every function call in the BFM, there is a channel identifier, which stores the fixed mapping between channel number and the function.

`<$install_dir>/ip/altera/sopc_builder_ip/verification/lib/`

`altera_avalon_components_pkg.vhd` defines the following function calls:

- `ST_SRC_INIT`
- `ST_SRC_SET_RESP_TIMEOUT`
- `ST_SRC_PUSH_TRANS`
- `ST_SRC_GET_TRANS_QUEUE_SIZE`
- `ST_SRC_GET_RESP_QUEUE_SIZE`
- `ST_SRC_SET_TRANS_DATA`
- `ST_SRC_SET_TRANS_CHANNEL`
- `ST_SRC_SET_TRANS_IDLES`
- `ST_SRC_SET_TRANS_SOP`
- `ST_SRC_SET_TRANS_EOP`
- `ST_SRC_SET_TRANS_ERROR`
- `ST_SRC_SET_TRANS_EMPTY`
- `ST_SRC_POP_RESP`
- `ST_SRC_GET_RESP_LATENCY`
- `ST_SRC_GET_SRC_READY`
- `ST_SRC_GET_SRC_TRANS_COMPLETE`

With the exception of the API wrapper, the Avalon-ST Source BFM with Avalon-ST API Wrapper component is identical to the Avalon-ST Source BFM. For more information about this component refer to [Chapter 1, Avalon-ST Source 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 3–1](#) 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.


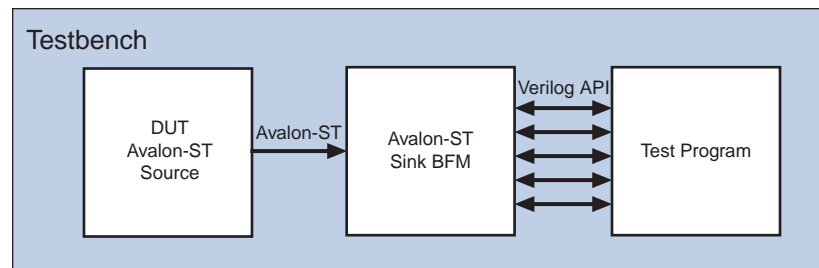


 The BFMs allow illegal transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitors components verify protocol compliance.

Figure 3–1. Top-Level Module to Verify an Avalon-ST Source Device



-  For more information about the Avalon-ST specification supported in SOPC Builder, refer to the [Avalon Interface Specifications \(version 1.3\)](#).
-  For more information about the Avalon-ST specification supported in Qsys, refer to the [Avalon Interface Specifications \(version 2.0\)](#).

Functional Description

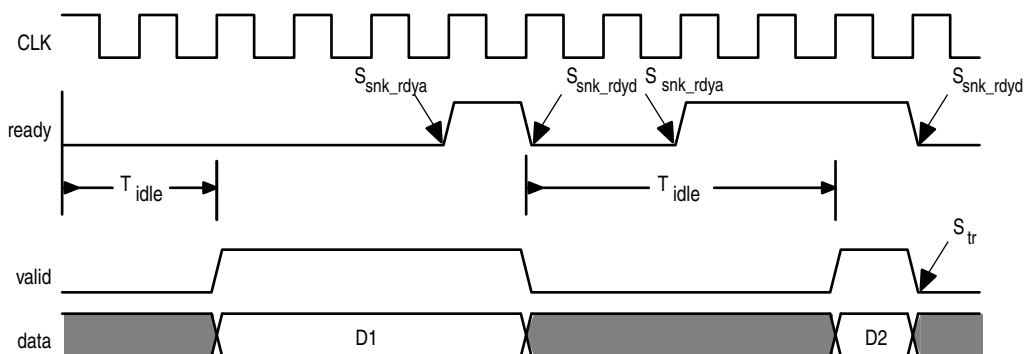
This section provides a functional description of the Avalon-ST Sink BFM. It includes the following topics:

- “Timing” on page 3-2
- “Block Diagram” on page 3-3

Timing

The timing diagram shown in [Figure 3-2](#) 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 3-2. Avalon-ST Source and Sink Timing



[Table 3-1](#) describes the annotations used in [Figure 3-2](#).

Table 3-1. Key to Annotations in [Figure 3-2](#)

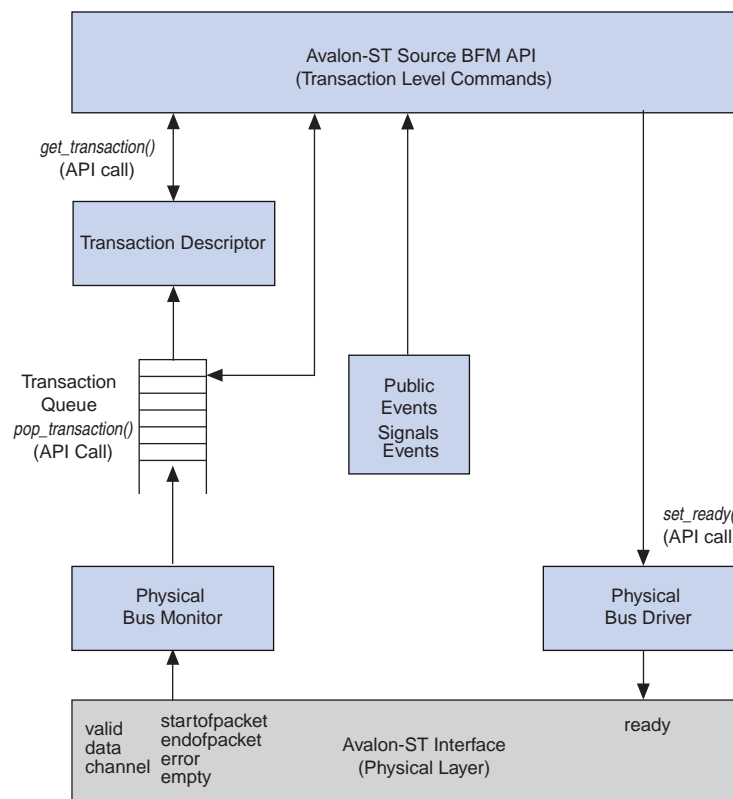
Symbol	Description
T_{idle}	The idle time between transactions. This time is reported by the command <code>get_transaction_idles</code> .
S_{snk_rdyd}	Signals the sink has asserted <code>ready</code> . The event name is <code>signal_snk_ready_assert</code> .
S_{tr}	Signals the transaction has been received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

Block Diagram

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



Parameters

The Avalon-ST Sink BFM supports all of the signals defined for the Avalon-MM sink interface. You can customize the Avalon-ST sink interface using the parameters described in [Table 3-2](#).

Table 3-2. 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 <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals.
Use the channel port	Off	On/Off	When On, the interface includes <code>channel</code> pin or pins.
Use the error port	Off	On/Off	When On, the interface includes <code>error</code> pin or pins.
Use the ready port	On	On/Off	When On, the interface includes a <code>ready</code> pin.
Use the valid port	On	On/Off	When On, the interface includes a <code>valid</code> pin.
Use the empty port	Off	On/Off	When On, the interface includes <code>empty</code> 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 <code>channel</code> signal.
Width of the error port	1	1–1024	Specifies the width of the <code>error</code> signal.
Width of the empty port	1	1–1024	Specifies the width of the <code>empty</code> signal.
Timing Attributes			
Ready latency	0	0–8	Specifies the delay between the <code>ready</code> and <code>valid</code> 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.
API Streaming Interface (Note 1)			
Width of API interface data signal	64	—	The width of the data signal.
Width of API return interface data signal	64	—	The width of the return interface data signal.

Note to Table 3-2:

- (1) This interface is required only for the Avalon-ST Sink BFM with Avalon-ST API Wrapper, which is used in mixed language simulations.

Application Program Interface

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

get_transaction_channel()

Prototype: `get_transaction_channel()`.
Arguments: None.
Returns: `STChannel_t`.
Description: Returns the channel identifier for the most recently removed transaction.

get_transaction_data()

Prototype: `get_transaction_data()`.
Arguments: None.
Returns: `STData_t`.
Description: Returns the data in the most recently removed transaction.

get_transaction_idles()

Prototype: `get_transaction_idles()`.
Arguments: None.
Returns: `bit[31:0]`.
Description: Returns the number of idle cycles in the most recently removed transaction.

get_transaction_eop()

Prototype: `get_transaction_eop()`.
Arguments: None.
Returns: `bit`.
Description: Returns the transaction end of packet status in the most recently removed transaction.

get_transaction_empty()

Prototype: `get_transaction_empty()`.
Arguments: None.
Returns: `STEmpty_t`.
Description: Returns the number of empty symbols in the most recently removed transaction.

get_transaction_error()

s

Prototype: `get_transaction_error()`.
Arguments: None.
Returns: `STError_t`.
Description: Returns the error in the most recently removed transaction.

get_transaction_queue_size()

s

Prototype: `get_transaction_queue_size()`.
Arguments: None.
Returns: `int`.
Description: Returns the length of the queue holding received transactions.

get_transaction_sop()

s

Prototype: `get_transaction_sop()`.
Arguments: None.
Returns: `bit`.
Description: Returns the transaction start of packet status in the most recently removed transaction.

get_version()

Prototype: `get_version()`.
Arguments: None.
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".

init()

Prototype: `init`.
Arguments: None.
Returns: `void`.
Description: Drives the interface to the idle state.

pop_transaction()

s

Prototype: `pop_transaction()`.
Arguments: None.
Returns: `void`.
Description: Removes the transaction descriptor from the queue so that the testbench can query it using the `get_transaction` methods.

set_ready()

Prototype:	<code>set_ready()</code> .
Arguments:	bit.
Returns:	void.
Description:	Sets the value of the interface <code>ready</code> signal. To assert back pressure, deassert this signal. The parameter <code>USE_READY</code> must be set to 1 to enable the <code>ready</code> signal.

signal_fatal_error

Prototype:	<code>signal_fatal_error</code> .
Arguments:	None.
Returns:	void.
Description:	Signals that a fatal error has occurred. It terminates the simulation.

signal_sink_ready_assert

Prototype:	<code>signal_sink_ready_assert</code> .
Arguments:	None.
Returns:	void.
Description:	Signals that <code>sink_ready</code> is asserted, turning off back pressure.

signal_sink_ready_deassert

Prototype:	<code>signal_sink_ready_deassert</code> .
Arguments:	None.
Returns:	void.
Description:	Signals that <code>sink_ready</code> is deasserted, turning on back pressure.

signal_transaction_received

Prototype:	<code>signal_transaction_received</code> .
Arguments:	None.
Returns:	void.
Description:	Signals that the transaction has been received and queued.

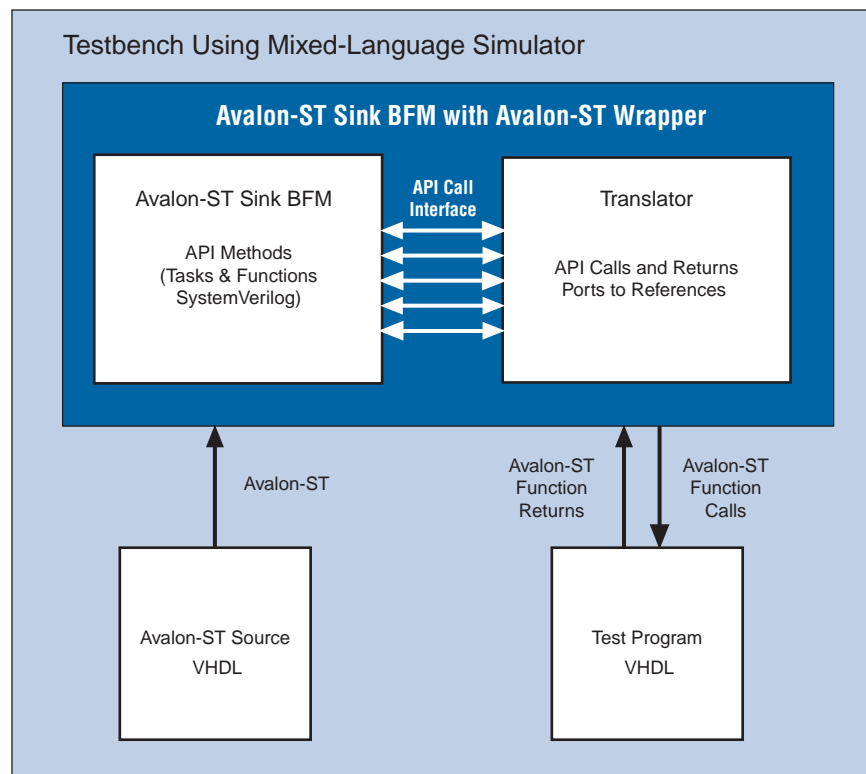
The Avalon-ST Sink BFM with Avalon-ST API Wrapper provides VHDL support for the Avalon-ST Sink BFM. You can use the Avalon-ST Sink BFM with Avalon-ST API Wrapper in HDL simulators that support mixed language simulation.



The API wrapper is only supported in SOPC Builder. The API wrapper cannot be generated in Qsys to create VHDL simulation models.

The Avalon-ST Sink BFM with Avalon-ST API Wrapper component is implemented in SystemVerilog and uses an API wrapper to cast the Avalon-ST Sink BFM's method calls and returns into signals that are carried on the call and return interface ports. The wrapper is necessary because VHDL can only access ports and does not support the method calls used in the Avalon-ST Sink BFM. [Figure 4-1](#) provides a high-level view of this component.

Figure 4-1. Avalon-ST Sink BFM with Avalon-ST Wrapper



In [Figure 4-1](#), the API call interface and Avalon-ST call and return interface operate in separate clock domains with `av_clk` synchronizing the FPGA logic and `api_clk` synchronizing the Avalon-ST translation interface. The Avalon-ST interface, which is not part of the actual hardware design, operates at a much higher frequency than the Avalon-ST Sink BFM interface.

For every function call in the BFM, there is a channel identifier, which stores the fixed mapping between channel number and the function.

`<$install_dir>/ip/altera/sopc_builder_ip/verification/lib/`

`altera_avalon_components_pkg.vhd` defines the following function calls:

- `ST_SINK_INIT`
- `ST_SINK_SET_READY`
- `ST_SINK_POP_TRANS`
- `ST_SINK_GET_TRANS_IDLES`
- `ST_SINK_GET_TRANS_DATA`
- `ST_SINK_GET_TRANS_CHANNEL`
- `ST_SINK_GET_TRANS_SOP`
- `ST_SINK_GET_TRANS_EOP`
- `ST_SINK_GET_TRANS_ERROR`
- `ST_SINK_GET_TRANS_EMPTY`
- `ST_SINK_GET_TRANS_QUEUE_SIZE`

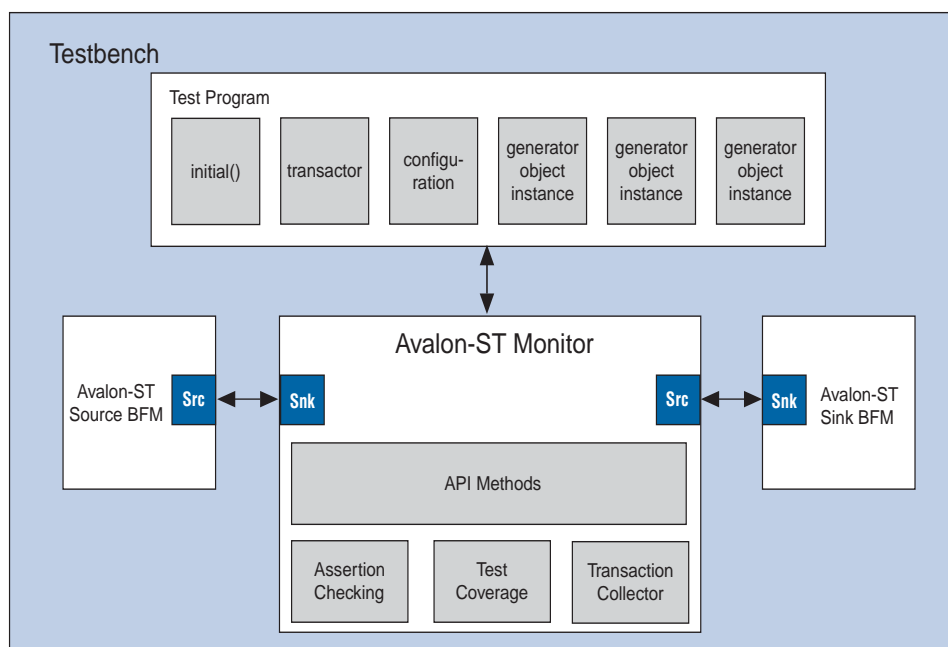
With the exception of the API wrapper, the Avalon-ST Sink BFM with Avalon-ST API Wrapper component is identical to the Avalon-ST Sink BFM. For more information about this component, refer to [Chapter 3, Avalon-ST Sink BFM](#).

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 5–1 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 5–1. 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 5-1](#).

Table 5-1. 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 <code>channel</code> signal in bits.
Width of the error port	1	—	Specifies the width of the <code>error</code> signal in bits.
Width of the empty port	1	—	Specifies the width of the <code>empty</code> signal in bits.
Port Enables			
Include the signals to support packets	On	On/Off	When On , the interface includes a the <code>startofpacket</code> , <code>endofpacket</code> , and <code>empty</code> signals.
Use the channel port	On	On/Off	When On , the interface includes a <code>channel</code> pin.
Use the error port	On	On/Off	When On , the interface includes <code>error</code> pins.
Use the ready port	On	On/Off	When On , the interface includes <code>ready</code> pins.
Use the valid port	On	On/Off	When On , the interface includes <code>valid</code> pins.
Use the empty port	On	On/Off	When On , the interface includes a <code>empty</code> pin.
Timing Attributes			
Ready latency	0	—	Specifies the <code>readyLatency</code> 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.

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: Boolean.
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.

set_enable_a_less_than_max_channel()

Prototype: `set_enable_a_less_than_max_channel()`.
Arguments: Boolean.
Returns: Void.
Description: Enables an assertion that ensures that the value of the `channel` signal is less than the maximum number of channels.

set_enable_a_no_data_outside_packet()

Prototype: `set_enable_a_no_data_outside_packet()`.
Arguments: Boolean.
Returns: Void.
Description: Enables an assertion that ensures `valid` data is not transferred outside of a packet when the interface uses packet transmission.

set_enable_a_non_missing_endofpacket()

Prototype: `set_enable_a_non_missing_endofpacket()`.
Arguments: Boolean.
Returns: Void.
Description: Enables an assertion that ensures that the `startofpacket` signal is asserted between each two assertions of an `endofpacket` signal.

set_enable_a_non_missing_startofpacket()

Prototype: `set_enable_a_non_missing_startofpacket()`.
Arguments: Boolean.
Returns: Void.
Description: Enables an assertion that ensures that each assertion of the `startofpacket` signal is followed by the assertion of an `endofpacket` signal.

set_enable_a_valid_legal()

Prototype: `set_enable_a_valid_legal()`.
Arguments: Boolean.
Returns: Void.
Description: Enables an assertion that ensures `valid` is deasserted `readyLatency` cycles after `ready` is deasserted if the `readyLatency` is greater than 0.

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: Boolean.
Returns: Void.
Description: Enables a coverage point that ensures test coverage for number of transaction with all idle beats.

set_enable_c_all_valid_beats()

Prototype: `set_enable_c_all_valid_beats()`.
Arguments: Boolean.
Returns: Void.
Description: Enables a coverage point that ensures test coverage for number of transaction with all valid beats.

set_enable_c_b2b_data_different_channel()

Prototype: `set_enable_c_b2b_data_different_channel()`.
Arguments: Boolean.
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.

set_enable_c_b2b_data_same_channel()

Prototype: `set_enable_c_b2b_data_same_channel()`.
Arguments: Boolean.
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.

set_enable_c_b2b_packet_different_channel()

Prototype: `set_enable_c_b2b_packet_different_channel()`.
Arguments: Boolean.
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.

set_enable_c_b2b_packet_in_different_transaction()

Prototype: `set_enable_c_b2b_packet_in_different_transaction()`.
Arguments: Boolean.
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.

set_enable_c_b2b_packet_same_channel()

Prototype: `set_enable_c_b2b_packet_same_channel()`.
Arguments: Boolean.
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.

set_enable_c_b2b_packet_within_single_cycle()

Prototype: `set_enable_c_b2b_packet_within_single_cycle()`.
Arguments: Boolean.
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.

set_enable_c_channel_change_in_packet()

Prototype: `set_enable_c_channel_change_in_packet()`.
Arguments: Boolean.
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.

set_enable_c_empty()

Prototype: `set_enable_c_empty()`.
Arguments: Boolean.
Returns: Void.
Description: Enables a coverage point that ensures test coverage of a `empty` signal. It is disabled when packet transmission is not supported.

set_enable_c_error()

Prototype: `set_enable_c_error()`.
Arguments: Boolean.
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.

set_enable_c_error_in_middle_of_packet()

Prototype: `set_enable_c_error_in_middle_of_packet()`.
Arguments: Boolean.
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.

set_enable_c_idle_beat_between_packet()

Prototype: `set_enable_c_idle_beat_between_packet()`.
Arguments: Boolean.
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.

set_enable_c_multiple_packet_per_cycle()

Prototype: `set_enable_c_multiple_packet_per_cycle()`.
Arguments: Boolean.
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.

set_enable_c_non_valid_ready()

Prototype: `set_enable_c_non_valid_ready()`.
Arguments: Boolean.
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.

set_enable_c_non_valid_non_ready()

Prototype: `set_enable_c_non_valid_non_ready()`.
Arguments: Boolean.
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.

set_enable_c_packet()

Prototype: `set_enable_c_packet()`.
Arguments: Boolean.
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.

set_enable_c_packet_no_idles_no_back_pressure()

Prototype: `set_enable_c_packet_no_idles_no_back_pressure()`.
Arguments: Boolean.
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.

set_enable_c_packet_size()

Prototype: `set_enable_c_packet_size()`.
Arguments: Boolean.
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.

set_enable_c_packet_with_back_pressure()

Prototype: `set_enable_c_packet_with_back_pressure()`.
Arguments: Boolean.
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.

set_enable_c_packet_with_idles()

Prototype: `set_enable_c_packet_with_idles()`.
Arguments: Boolean.
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.

set_enable_c_partial_valid_beats()

Prototype: `set_enable_c_partial_valid_beats()`.
Arguments: Boolean.
Returns: Void.
Description: Enables a coverage point that ensures test coverage for number of transaction with partially valid beats.

set_enable_c_single_packet_per_cycle()

Prototype: `set_enable_c_single_packet_per_cycle()`.
Arguments: Boolean.
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.

set_enable_c_transfer()

Prototype:	<code>set_enable_c_transfer()</code> .
Arguments:	Boolean.
Returns:	Void.
Description:	Enables a coverage point that ensures test coverage of a <code>valid</code> signal is asserted correctly for different channels. It is disabled when the <code>ready</code> or <code>valid</code> signals are not supported.

set_enable_c_transaction_after_reset()

Prototype:	<code>set_enable_c_transaction_after_reset()</code> .
Arguments:	Boolean.
Returns:	Void.
Description:	Enables a coverage point that ensures test coverage for transaction on the first cycle after reset.

set_enable_c_valid_non_ready()

Prototype:	<code>set_enable_c_valid_non_ready()</code> .
Arguments:	Boolean.
Returns:	Void.
Description:	Enables a coverage point that ensures test coverage for <code>valid</code> signal when <code>ready</code> is deasserted. It is disabled when the <code>readyLatency</code> is greater than 0.

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.

get_transaction_channel()

Prototype:	<code>get_transaction_channel()</code> .
Arguments:	None.
Returns:	<code>STChannel_t</code> .
Description:	Returns the channel identifier for the most recently removed transaction.

get_transaction_data()

Prototype: `get_transaction_data()`.
Arguments: None.
Returns: `STData_t`.
Description: Returns the data in the most recently removed transaction.

get_transaction_empty()

s
Prototype: `get_transaction_empty()`.
Arguments: None.
Returns: `STEmpty_t`.
Description: Returns the number of empty symbols in the most recently removed transaction.

get_transaction_eop()

s
Prototype: `get_transaction_eop()`.
Arguments: None.
Returns: `bit`.
Description: Returns the transaction end of packet status in the most recently removed transaction.

get_transaction_error()

s
Prototype: `get_transaction_error()`.
Arguments: None.
Returns: `STError_t`.
Description: Returns the error in the most recently removed transaction.

get_transaction_idles()

Prototype: `get_transaction_idles()`.
Arguments: None.
Returns: `bit[31:0]`.
Description: Returns the number of idle cycles in the most recently removed transaction.

get_transaction_queue_size()

s
Prototype: `get_transaction_queue_size()`.
Arguments: None.
Returns: `int`.
Description: Returns the length of the queue holding received transactions.

get_transaction_sop()

s

Prototype:	<code>get_transaction_sop()</code> .
Arguments:	None.
Returns:	bit.
Description:	Returns the transaction start of packet status in the most recently removed transaction.

get_version()

Prototype:	<code>string get_version()</code> .
Arguments:	None.
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".

pop_transaction()

Prototype:	<code>void pop_transaction()</code> .
Arguments:	None.
Returns:	void.
Description:	Removes the transaction descriptor from the queue so that the testbench can query it with the <code>get_transaction</code> methods.

set_transaction_fifo_max()

Prototype:	<code>set_transaction_fifo_max()</code> .
Arguments:	<code>int level</code> .
Returns:	Void.
Description:	Sets the maximum transaction level of the FIFO. The event <code>signal_transaction_fifo_max</code> is triggered when this level is exceeded.

set_transaction_fifo_threshold()

Prototype:	<code>set_transaction_fifo_threshold()</code> .
Arguments:	<code>int level</code> .
Returns:	Void.
Description:	Sets the threshold alert level of the FIFO. The event <code>signal_transaction_fifo_threshold</code> is triggered when this level is exceeded.

signal_fatal_error

Prototype: `signal_fatal_error.`
Arguments: None.
Returns: `void.`
Description: Notifies the testbench that a fatal error has occurred in this module.

signal_transaction_fifo_overflow

Prototype: `signal_transaction_fifo_overflow.`
Arguments: None.
Returns: `void.`
Description: Notifies the testbench that the FIFO is full and further transactions are dropped.

signal_transaction_fifo_threshold

Prototype: `signal_transaction_fifo_threshold.`
Arguments: None.
Returns: `void.`
Description: Notifies the testbench that the transaction FIFO threshold level has exceeded.

signal_transaction_received

Prototype: `signal_transaction_received.`
Arguments: None.
Returns: `void.`
Description: Notifies the testbench that a transaction has been received and queued.

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

- [Chapter 1, Conduit BFM](#)
- [Chapter 2, Tri-State Conduit BFM](#)
- [Chapter 3, External Memory BFM](#)

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

- Port compatibility and polarity
- Legal port widths

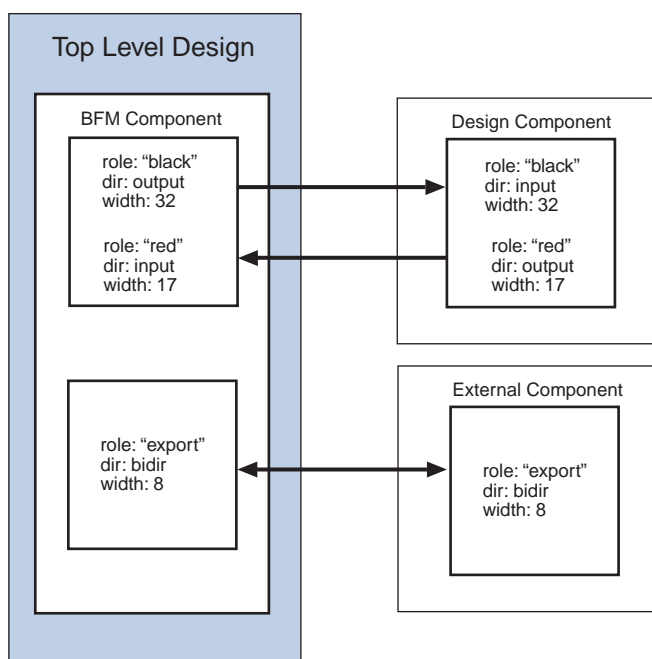


Conduit BFM are only supported in Qsys.

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 \(version 2.0\)](#).

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 <width of the role name port> get_<role name>().`
Arguments: None.
Returns: `Int <width of the role name port>.`
Description: Returns interface signal value from the input/bidirectional port.

get_version()

Prototype: `string get_version().`
Arguments: None.
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".

set_<role name>()

Prototype: `void set_<role name>().`
Arguments: `new_value.`
Returns: `void.`
Description: Rewrites the registers inside the BFM that are driven to the <role name> output ports.

set_<role name>_oe()

Prototype: `void set_<role name>_oe().`
Arguments: `bit enable.`
Returns: `void`
Description: Enables the bidirectional ports when the value is set to 1.

signal_input_<role name>_change

Prototype: `signal_input_<role name>_change.`
Arguments: None.
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.

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

- Port compatibility and polarity
- Legal port widths

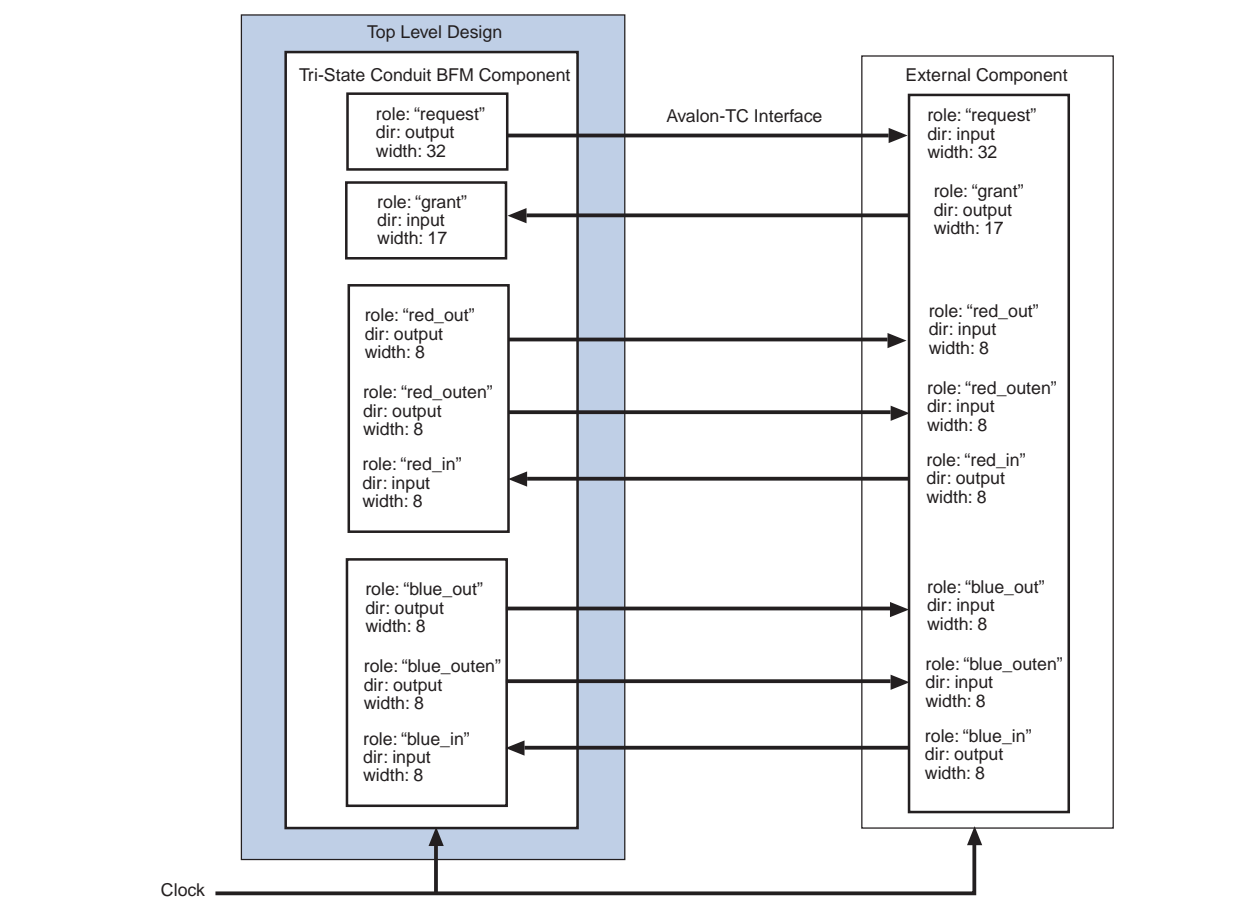


The Tri-State Conduit BFM is only supported in Qsys.

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 \(version 2.0\)](#).

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: None.
Returns: Int.
Description: Returns the size of the queued input transaction in the BFM.

get_output_transaction_queue_size()

Prototype: `int get_output_transaction_queue_size().`
Arguments: None.
Returns: Int.
Description: Returns the size of the queued output transaction in the BFM.

get_transaction_<role name>_in()

Prototype: `int <width of the role name port>get_transaction_<role name>_in().`
Arguments: None.
Returns: Int <width of the role name port>.
Description: Returns the interface signal value from the <role name>_in input ports.

get_transaction_latency()

Prototype: `int get_transaction_latency().`
Arguments: None.
Returns: Int.
Description: Returns the latency field value from the input transaction.

get_version()

Prototype: `string get_version().`
Arguments: None.
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".

pop_transaction()

Prototype:	<code>void pop_transaction();</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Returns the input transaction queued inside the BFM. A fatal error triggers if you remove a transaction from an empty queue.

push_transaction()

Prototype:	<code>void push_transaction();</code>
Arguments:	None.
Returns:	<code>void</code> .
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.

set_max_transaction_queue_size()

Prototype:	<code>void set_max_transaction_queue_size(int size);</code>
Arguments:	<code>int size</code> .
Returns:	<code>void</code> .
Description:	Sets the maximum size of the queued transactions. The BFM triggers an event when the queued transactions goes above the maximum size.

set_min_transaction_queue_size()

Prototype:	<code>int set_min_transaction_queue_size();</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Sets the minimum size of the queued transactions. The BFM triggers an event when the queued transactions falls below the minimum size.

set_num_of_transactions()

Prototype:	<code>int set_num_of_transactions();</code>
Arguments:	<code>int multiple_transaction_num</code> .
Returns:	<code>void</code> .
Description:	Sets the number of transactions to the DUT.

set_transaction_<role name>_out()

Prototype: void set_transaction_<role name>_out().
Arguments: int index.
Returns: void.
Description: Sets the value of the transaction to the <role name>_out output ports.

set_transaction_<role name>_outen()

Prototype: string set_transaction_<role name>_outen().
Arguments: int index.
bit outen.
Returns: void.
Description: Sets the value of the transaction to the <role name>_outen output ports.

set_transaction_idles()

Prototype: void set_transaction_idles().
Arguments: bit[31:0] idle_cycles.
Returns: void.
Description: Sets the number of idle cycles that elapse before driving the out-going transaction.

set_valid_transaction_<role name>_out()

Prototype: void set_valid_transaction_<role name>_out().
Arguments: int index.
Returns: void.
Description: Sets the value of the valid transaction to the <role name>_out output port.

signal_all_transactions_complete

Prototype: signal_all_transactions_complete.
Arguments: None.
Returns: void
Description: Triggers when all the queued output and input transactions are completely retrieved.

signal_fatal_error

Prototype: signal_fatal_error.
Arguments: None.
Returns: void.
Description: Notifies the testbench that a fatal error has occurred in this module.

signal_grant_deasserted_while_request_remain_asserted()

Prototype:	signal_grant_deasserted_while_request_remain_asserted.
Arguments:	None.
Returns:	void.
Description:	Triggers when the grant signal changes value from high to low while the request signal remains asserted.

signal_interface_granted

Prototype:	signal_interface_granted.
Arguments:	None.
Returns:	void
Description:	Triggers when the grant signal is asserted.

signal_max_transaction_queue_size

Prototype:	signal_max_transaction_queue_size.
Arguments:	None.
Returns:	void.
Description:	Triggers when the size of the pending queue exceeds the maximum size.

signal_min_transaction_queue_size

Prototype:	signal_min_transaction_queue_size.
Arguments:	None.
Returns:	void.
Description:	Triggers when the size of the pending queue falls below the minimum size.

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

- Read and write operations
- Memory initialization



External Memory BFM 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 tristate components.

Figure 3–1. Usage of External Memory BFM with Tristate Components

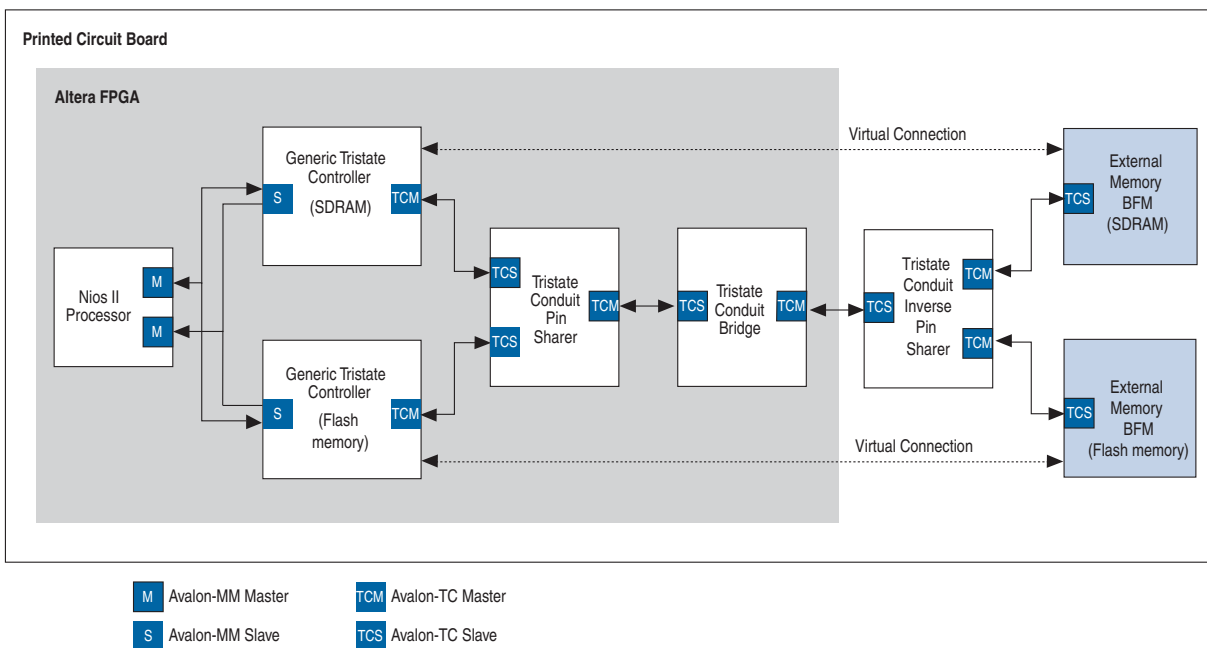


Table 3–2 lists the function of the external memory BFM and its related components.

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 tristate controller component.
Tristate Conduit Bridge	Converts Avalon-TC signals into conduit signals.
Tristate Conduit Pin Sharer	Carries the shared address bus and data.
Tristate Conduit Inverse Pin Sharer	
Generic Tristate Controller	Controls the external memory BFM. The generic tristate controller accepts read and write requests and converts these requests into necessary SDRAM and bank management commands.



For more information about tristate conduit bridge, refer to “Tristate Conduit Bridge” section in the *Qsys Interconnect* chapter of the *Quartus II Handbook*.

For more information about tristate conduit pin sharer, refer to “Tristate Conduit Pin Sharer” section in the *Qsys Interconnect* chapter of the *Quartus II Handbook*.

For more information about generic tristate controller, refer to “Generic Tristate 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 <code>byteenable</code> pin to enable specific byte lanes during transfer.
Use the chip select signal	On	On/Off	When On , the interface includes a <code>chipselect</code> pin. When present, the slave port ignores all Avalon-MM signals unless <code>chipselect</code> is asserted. <code>chipselect</code> is always present in combination with <code>read</code> or <code>write</code> .
Use the write signal	On	On/Off	When On , the interface includes a <code>write</code> pin that enables the write-request signal.
Use the read signal	On	On/Off	When On , the interface includes a <code>read</code> pin that enables the read-request signal.
Use the output enable signal	On	On/Off	When On , the interface includes an <code>outputenable</code> pin.
Use the begintransfer signal	On	On/Off	When On , the interface includes a <code>begintransfer</code> pin.
Use the reset input signal	Off	On/Off	When On , the interface includes a <code>reset</code> pin.
Use the active low byteenable signal	Off	On/Off	When On , the interface includes an active low <code>byteenable</code> pin.
Use the active low chipselect signal	Off	On/Off	When On , the interface includes an active low <code>chipselect_n</code> pin.
Use the active low write signal	Off	On/Off	When On , the interface includes an active low <code>write_n</code> pin.
Use the active low read signal	Off	On/Off	When On , the interface includes an active low <code>read_n</code> pin.
Use the active low outputenable signal	Off	On/Off	When On , the interface includes an active low <code>outputenable_n</code> pin.
Use the active low begintransfer signal	Off	On/Off	When On , the interface includes an active low <code>begintransfer_n</code> pin.
Use the active low reset signal	Off	On/Off	When On , the interface includes an active low <code>reset_n</code> 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.

Application Program Interface

This section describes the API for the external memory BFM.

fill()

Prototype:	<code>fill()</code> .
Arguments:	<code>logic[DATA_W-1:0]</code> data. <code>bit[DATA_W-1:0]</code> increment. <code>bit[CDT_ADDRESS_W-1:0]</code> address low. <code>bit[CDT_ADDRESS_W-1:0]</code> address high.
Returns:	<code>void</code> .
Description:	Overwrites the memory content at the starting address specified by <code>address_low</code> until the ending address specified by <code>address_high</code> . The <code>data</code> field indicates the data value. The <code>increment</code> field indicates the data value increment from one address to the next address. For example, <code>fill (data[1], increment[2], address_low[10], address_high[12])</code> fills the memory as follows: <ul style="list-style-type: none">■ <code>memory[address=10]</code> is filled with data value 1■ <code>memory[address=11]</code> is filled with data value 3■ <code>memory[address=12]</code> is filled with data value 5

read()

Prototype:	<code>read()</code> .
Arguments:	<code>bit[CDT_ADDRESS_W-1:0]</code> address.
Returns:	<code>logic[DATA_W-1:0]</code> .
Description:	Retrieves the memory content from an address you specify.

signal_api_call

Prototype:	<code>signal_api_call</code> .
Arguments:	None.
Returns:	<code>void</code> .
Description:	Triggers when a client make an API call.

write()

Prototype:	<code>write()</code> .
Arguments:	<code>bit[CDT_ADDRESS_W-1:0]</code> address. <code>logic[DATA_W-1:0]</code> data.
Returns:	<code>void</code> .
Description:	Overwrites the memory content at an address you specify.

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

- [Chapter 1, Nios II Custom Instruction Master BFM](#)
- [Chapter 2, Nios II Custom Instruction Slave 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

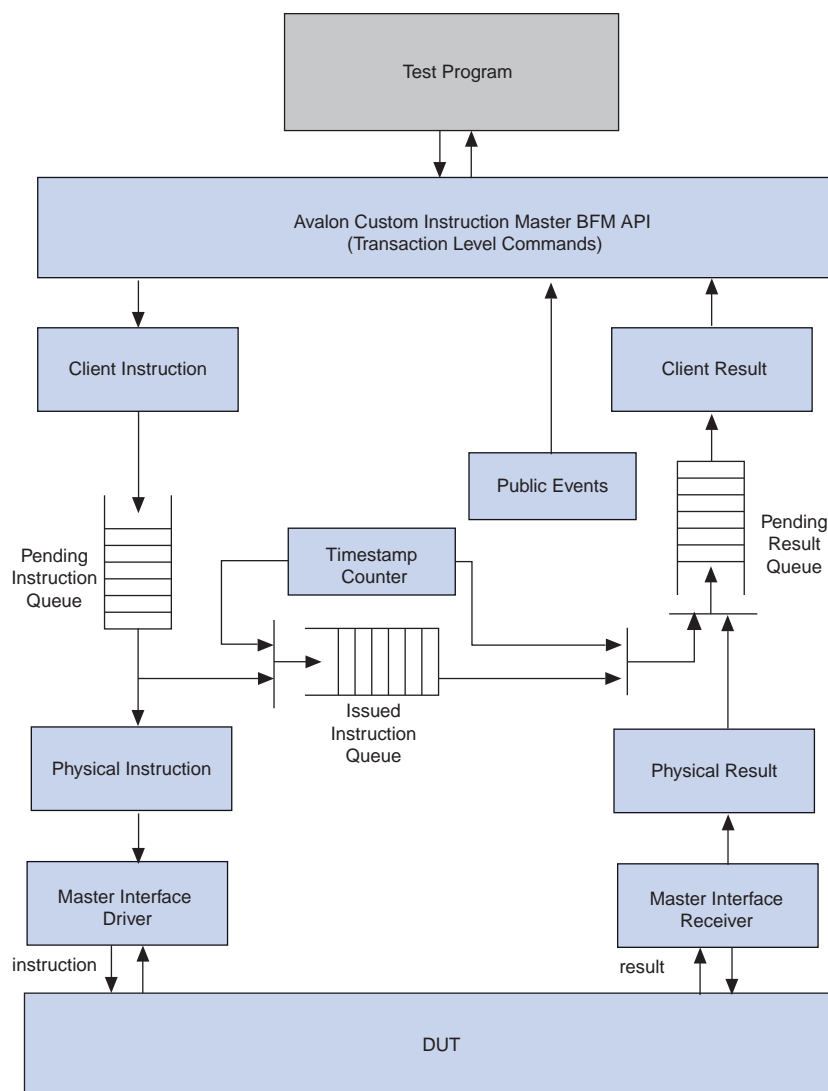


The Nios II Custom Instruction Master BFM is only supported in Qsys.

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 <code>result</code> pin.
Use Multi-cycle Mode	Off	On/Off	When On , the interface can include a <code>start</code> pin, a <code>done</code> pin, both pins, or neither pins. The result returns in any of the following conditions: <ul style="list-style-type: none"> ■ With a <code>start</code> signal—Result returns together with an instruction. ■ Without a <code>start</code> signal—Result returns with instruction on the bus at every clock cycle. ■ With a <code>done</code> signal—Result returns at any time. ■ Without a <code>done</code> signal—Result returns at a fixed cycle.
Using start port	On	On/Off	When On , the interface includes a <code>start</code> pin.
Using done port	On	On/Off	When On , the interface includes a <code>done</code> pin.
Use Extended Port	Off	On/Off	When On , the interface includes a <code>n</code> pin.
Extended Port Width	1	—	Specifies the width of the extended <code>n</code> port.
Use Internal Register a	Off	On/Off	When On , the interface includes the <code>readra</code> and <code>a</code> pins.
Use Internal Register b	Off	On/Off	When On , the interface includes the <code>readrb</code> and <code>b</code> pins.
Use Internal Register c	Off	On/Off	When On , the interface includes the <code>readrc</code> and <code>c</code> pins.

Application Program Interface

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

get_instruction_queue_size()

Prototype: `int get_instruction_queue_size(int size).`
Arguments: None.
Returns: `int size.`
Description: Returns the number of instructions in the queue.

get_result_delay()

Prototype: `int get_result_delay().`
Arguments: None.
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.

get_result_queue_size()

Prototype: `int get_result_queue_size(int size).`
Arguments: None.
Returns: `int size.`
Description: Returns the number of results in the queue.

get_result_value()

Prototype: `string get_result_value().`
Arguments: None.
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.

get_version()

Prototype:	<code>string get_version().</code>
Arguments:	None.
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".

insert_instruction()

Prototype:	<code>void insert_instruction().</code>
Arguments:	<code>ci_data_t dataa.</code> <code>ci_data_t datab.</code> <code>ci_n_t n.</code> <code>ci_addr_t a.</code> <code>ci_addr_t b.</code> <code>ci_addr_t c.</code> <code>logic readra.</code> <code>logic readrb.</code> <code>logic writerc.</code> <code>ci_data_t idle.</code> <code>int err_inj.</code>
Returns:	<code>void.</code>
Description:	A simplified API to set and push instructions.

pop_result()

Prototype:	<code>void pop_result().</code>
Arguments:	None.
Returns:	<code>void..</code>
Description:	Removes the result instruction from the queue before querying the contents.

push_instruction()

Prototype:	<code>void push_instruction().</code>
Arguments:	None.
Returns:	<code>void.</code>
Description:	Inserts a new instruction into the queue.

retrive_result()

Prototype: void retrive_result.
Arguments: output ci_data_t value.
output ci_data_t delay.
Returns: void.
Description: A simplified API to remove and retrieve results.

set_ci_clk_en()

Prototype: void set_ci_clk_en().
Arguments: bit enable.
Returns: void.
Description: Sets the ci_clk_en signal synchronously with the clock.

set_clock_enable_timeout()

Prototype: void set_clock_enable_timeout().
Arguments: int timeout.
Returns: void.
Description: Sets the timeout value for the clock enable. Sets the value to 0 (zero) to disable timeouts.

set_instruction_a()

Prototype: void set_instruction_a().
Arguments: ci_addr_t address.
Returns: void.
Description: Sets the instruction register file address a value.

set_instruction_b()

Prototype: void set_instruction_b().
Arguments: ci_addr_t address.
Returns: void.
Description: Sets the instruction register file address b value.

set_instruction_c()

Prototype: void set_instruction_c().
Arguments: ci_addr_t address.
Returns: void.
Description: Sets the instruction register file address c value.

set_instruction_dataa()

Prototype: void set_instruction_dataa().
Arguments: ci_data_t data.
Returns: void.
Description: Sets the instruction dataa operand value.

set_instruction_datab()

Prototype: void set_instruction_datab().
Arguments: ci_data_t data.
Returns: void.
Description: Sets the instruction datab operand value.

set_instruction_err_inject()

Prototype: void set_instruction_err_inject().
Arguments: int err_inj.
Returns: void.
Description: Sets the instruction to execute in pre-defined error.

set_instruction_idle()

Prototype: void set_instruction_idle().
Arguments: ci_data_t idle.
Returns: void.
Description: Sets the instruction idle value.

set_instruction_n()

Prototype: void set_instruction_n().
Arguments: ci_n_t code.
Returns: void.
Description: Sets the instruction extended opcode value n.

set_instruction_readra()

Prototype: void set_instruction_readra().
Arguments: logic enable.
Returns: void.
Description: Sets the instruction register file read a value.

set_instruction_readrb()

Prototype: `void set_instruction_readrb().`
Arguments: `logic enable.`
Returns: `void.`
Description: Sets the instruction register file read `b` value.

set_instruction_timeout()

Prototype: `void set_instruction_timeout().`
Arguments: `int timeout.`
Returns: `void.`
Description: Sets the timeout value for an instruction. Sets the value to 0 (zero) to disable the timeout.

set_instruction_writerc()

Prototype: `void set_instruction_writerc().`
Arguments: `logic enable.`
Returns: `void.`
Description: Sets the instruction register file write `c` value.

set_max_instruction_queue_size()

Prototype: `void set_max_instruction_queue_size(int size).`
Arguments: `int size.`
Returns: `void.`
Description: Sets the pending instruction queue size maximum threshold.

set_max_result_queue_size()

Prototype: `void set_max_result_queue_size(int size).`
Arguments: `int size.`
Returns: `void.`
Description: Sets the pending result queue size maximum threshold.

set_min_instruction_queue_size()

Prototype: `void set_min_instruction_queue_size(int size).`
Arguments: `int size.`
Returns: `void.`
Description: Sets the pending instruction queue size minimum threshold.

set_min_result_queue_size()

Prototype: void set_min_result_queue_size(int size).
Arguments: int size..
Returns: void.
Description: Sets the pending result queue size minimum threshold.

set_result_timeout()

Prototype: void set_result_timeout().
Arguments: int timeout.
Returns: void.
Description: Sets the timeout value for a result. Set the value to 0 to disable timeout.

signal_unexpected_result_received

Prototype: signal_unexpected_result_received.
Arguments: None.
Returns: void.
Description: Signals that a result has been received without an instruction.

signal_fatal_error

Prototype: signal_fatal_error.
Arguments: None.
Returns: void.
Description: Notifies the testbench that a fatal error has occurred in this module.

signal_instructions_completed

Prototype: signal_instructions_completed.
Arguments: None.
Returns: void.
Description: Signals that all instructions in the BFM have been executed.

signal_instruction_start

Prototype: signal_instruction_start.
Arguments: None.
Returns: void.
Description: Signals that an instruction has been driven to the interface.

signal_max_instruction_queue_size

Prototype:	<code>signal_max_instruction_queue_size.</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Signals that the maximum pending instruction queue size threshold has been exceeded.

signal_max_result_queue_size

Prototype:	<code>signal_max_result_queue_size.</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Signals that the maximum pending result queue size threshold has been exceeded.

signal_min_instruction_queue_size

Prototype:	<code>signal_min_instruction_queue_size.</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Signals that the pending instruction queue size is below the minimum threshold.

signal_min_result_queue_size

Prototype:	<code>signal_min_result_queue_size.</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Signals that the pending result queue size is below the minimum threshold.

signal_result_received

Prototype:	<code>signal_result_received.</code>
Arguments:	None.
Returns:	<code>void</code> .
Description:	Signals that a result has been received.

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

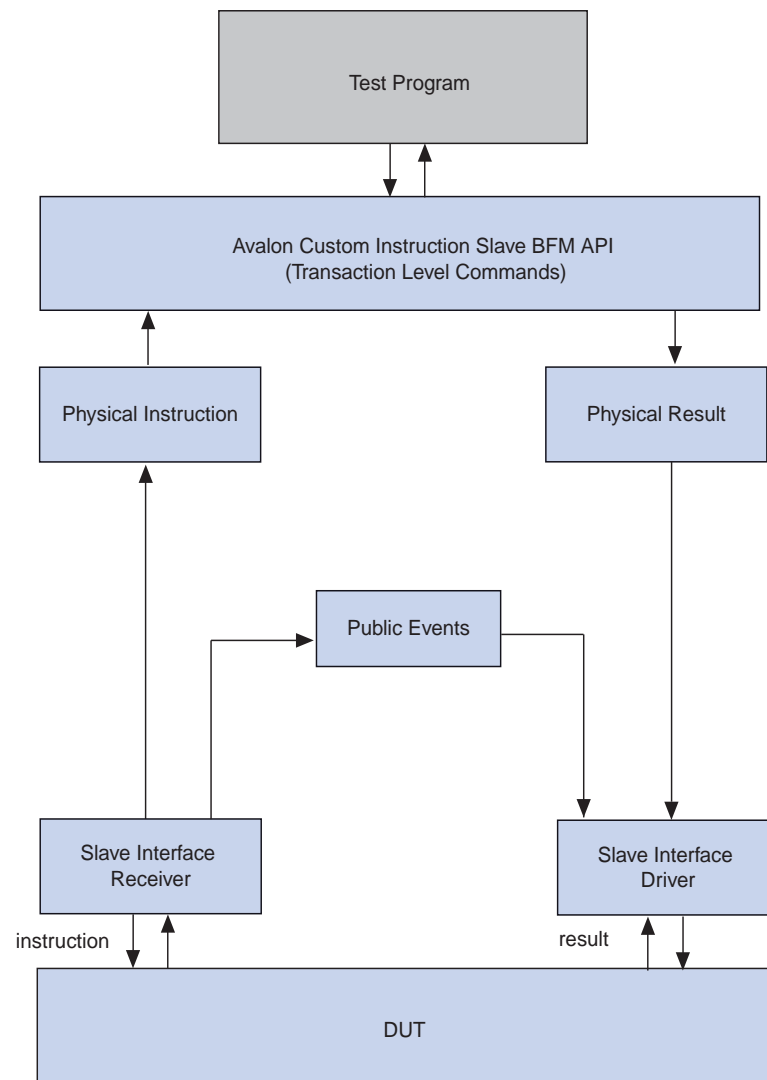


The Nios II Custom Instruction Slave BFM is only supported in Qsys.

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 master BFM interface.

Table 2–1. 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 <code>result</code> pin.
Use Multi-cycle Mode	Off	On/Off	When On , the interface can include a <code>start</code> pin, a <code>done</code> pin, both pins, or neither pins. The result returns in any of the following conditions: <ul style="list-style-type: none"> ■ With a <code>start</code> signal—Result returns together with an instruction. ■ Without a <code>start</code> signal—Result returns with instruction on the bus at every clock cycle. ■ With a <code>done</code> signal—Result returns at any time. ■ Without a <code>done</code> signal—Result returns at a fixed cycle.
Using start port	On	On/Off	When On , the interface includes a <code>start</code> pin.
Using done port	On	On/Off	When On , the interface includes a <code>done</code> pin.
Use Extended Port	Off	On/Off	When On , the interface includes a <code>n</code> pin.
Extended Port Width	1	—	Specifies the width of the extended <code>n</code> port.
Use Internal Register a	Off	On/Off	When On , the interface includes the <code>readra</code> and <code>a</code> pins.
Use Internal Register b	Off	On/Off	When On , the interface includes the <code>readrb</code> and <code>b</code> pins.
Use Internal Register c	Off	On/Off	When On , the interface includes the <code>readrc</code> and <code>c</code> pins.

Application Program Interface

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

get_ci_clk_en()

Prototype: void get_ci_clk_en(bit enable).
Arguments: None.
Returns: bit enable.
Description: Retrieves the clock enable signal.

get_instruction_a()

Prototype: string get_instruction_a().
Arguments: None.
Returns: ci_addr_t.
Description: Retrieves the instruction register file address a value.

get_instruction_b()

Prototype: string get_instruction_b().
Arguments: None.
Returns: ci_addr_t.
Description: Retrieves the instruction register file address b value.

get_instruction_c()

Prototype: string get_instruction_c().
Arguments: None.
Returns: ci_addr_t.
Description: Retrieves the instruction register file address c value.

get_instruction_dataa()

Prototype: void get_instruction_dataa().
Arguments: None.
Returns: ci_data_t data.
Description: Retrieves the instruction dataa operand value.

get_instruction_datab()

Prototype: void get_instruction_datab().
Arguments: None.
Returns: ci_data_t data.
Description: Retrieves the instruction datab operand value.

get_instruction_idle()

Prototype: void get_instruction_idle().
Arguments: None.
Returns: ci_data_t.
Description: Retrieves the pre-instruction idle value.

get_instruction_n()

Prototype: void get_instruction_n().
Arguments: None.
Returns: ci_n_t
Description: Retrieves the instruction extended opcode value n.

get_instruction_readra()

Prototype: logic get_instruction_readra().
Arguments: None.
Returns: logic.
Description: Retrieves the instruction register file read a value.

get_instruction_readrb()

Prototype: logic get_instruction_readrb().
Arguments: None.
Returns: logic.
Description: Retrieves the instruction register file read b value.

get_instruction_writerc()

Prototype: logic get_instruction_writerc().
Arguments: None.
Returns: logic.
Description: Retrieves the instruction register file write c value.

get_version()

Prototype:	<code>string get_version().</code>
Arguments:	None.
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".

insert_result()

Prototype:	<code>void insert_result().</code>
Arguments:	<code>ci_data_t value.</code> <code>ci_data_t delay.</code> <code>int err_inj.</code>
Returns:	<code>void.</code>
Description:	A simplified API to set results.

retrieve_instruction()

Prototype:	<code>void retrieve_instruction.</code>
Arguments:	<code>output ci_data_t dataaa.</code> <code>output ci_data_t datab.</code> <code>output ci_n_t n.</code> <code>output ci_addr_t a.</code> <code>output ci_addr_t b.</code> <code>output ci_addr_t c.</code> <code>output logic readra.</code> <code>output logic readrb.</code> <code>output logic writerc.</code> <code>output ci_data_t idle.</code>
Returns:	<code>void.</code>
Description:	A simplified API to retrieve instruction.

set_clock_enable_timeout()

Prototype:	<code>void set_clock_enable_timeout().</code>
Arguments:	<code>int timeout.</code>
Returns:	<code>void.</code>
Description:	Sets the timeout value for the clock enable. Set the value to 0 to disable timeout.

set_instruction_a()

Prototype: void set_instruction_a().
Arguments: ci_addr_t address.
Returns: void.
Description: Sets the instruction register file address a value.

set_instruction_b()

Prototype: void set_instruction_b().
Arguments: ci_addr_t address.
Returns: void.
Description: Sets the instruction register file address b value.

set_instruction_c()

Prototype: void set_instruction_c().
Arguments: ci_addr_t address.
Returns: void.
Description: Sets the instruction register file address c value.

set_instruction_timeout()

Prototype: void set_instruction_timeout().
Arguments: int timeout.
Returns: void.
Description: Sets the timeout value for an instruction. Set the value to 0 to disable timeouts.

set_result_delay()

Prototype: void set_result_delay().
Arguments: ci_data_t delay.
Returns: void.
Description: Sets the instruction result delay.

set_result_err_inject()

Prototype: void set_result_err_inject().
Arguments: int err_inj.
Returns: void.
Description: Sets the instruction result to execute in pre-defined error.

set_result_value()

Prototype: void set_result_value().
Arguments: ci_data_t value.
Returns: void.
Description: Sets the instruction result.

signal_fatal_error

Prototype: signal_fatal_error.
Arguments: None.
Returns: void.
Description: Notifies the testbench that a fatal error has occurred in this module.

signal_instructions_inconsistent

Prototype: signal_instructions_inconsistent.
Arguments: None.
Returns: void.
Description: Signals that an instruction has changed while the previous instruction has not completed.

signal_known_instruction_received

Prototype: signal_known_instruction_received.
Arguments: None.
Returns: void.
Description: Signals that a change has occurred on the instruction interface and there is no unknown value.

signal_result_done

Prototype: signal_result_done.
Arguments: None.
Returns: void.
Description: Signals that a result has been received by the master.

signal_result_driven

Prototype: signal_result_driven.
Arguments: None.
Returns: void.
Description: Signals that a result has been driven from the slave interface.

signal_unknown_instruction_received

Prototype: `signal_unknown_instruction_received.`

Arguments: None.

Returns: void.

Description: Signals that a change has occurred on the instruction interface and there is an unknown value.

This section describes the Avalon Verification IP tutorials for SOPC Builder and Qsys. This section includes the following chapters:

- [Chapter 1, SOPC Builder Tutorial](#)
- [Chapter 2, Qsys Tutorial](#)

This chapter demonstrates how to use the Avalon-MM Master and Slave BFM to verify Avalon-MM master and slave components in an SOPC Builder design. In the first example, the DUT is an on-chip RAM that includes an Avalon-MM slave port. Its behavior is verified using the Avalon-MM Master BFM component. The second example verifies an Avalon-MM master DUT using the Avalon-MM Slave BFM component.

Software Requirements

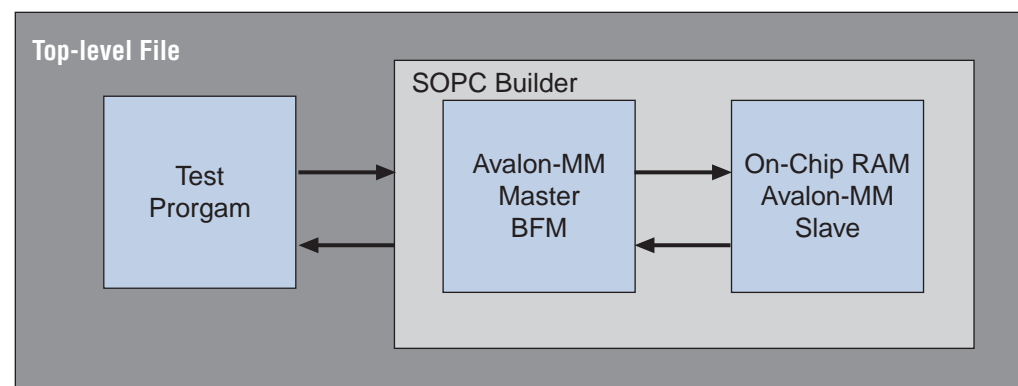
The following software and file are required to run the test:

- Quartus II software, version 12.0 or later.
- 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-MM Slave DUT

Figure 1–1 illustrates the top-level testbench to verify an Avalon-MM slave component. An on-chip RAM component is connected to the Avalon-MM Master BFM in SOPC Builder. The test program initializes the Avalon-MM Master BFM. After the initialization and system reset complete, the test program instructs the master BFM to write random data to the slave DUT. The write data is also saved into a local array for future reference. The Avalon-MM Master BFM reads back the data written, compares it to the data stored in the local array, and reports mismatches. The test passes if all the read data is correct.

Figure 1–1. Top-Level Testbench for an Avalon-MM Slave Component



Example 1-1 shows an excerpt from the test program that demonstrates the use of the Avalon-MM Master API. **Example 1-1** shows the following two tasks:

- `master_set_and_push_commands`—Sets the fields of the command descriptor and inserts it on to the command queue.
- `master_pop_and_get_response`—Pops or removes the response received by the Avalon-MM Master BFM.

As these tasks illustrate, use the `set_command_<field>` methods to define the command and the `push_command` method to add the command to the queue. Use the `pop_response` method to get a response and the `get_response_<field>` to retrieve the fields of the response.

Example 1-1. Verilog Tasks Illustrating the Avalon-MM Master BFM API

```
//this task sets the command descriptor for master BFM and push it to the queue
task master_set_and_push_command;
// . . .
begin
    `MSTR_BFM.set_command_request(request);
    `MSTR_BFM.set_command_address(addr);
    `MSTR_BFM.set_command_byte_enable(byte_enable, `INDEX_ZERO);
    `MSTR_BFM.set_command_idle(idle, `INDEX_ZERO);
    `MSTR_BFM.set_command_init_latency(init_latency);

    if (request == REQ_WRITE)
    begin
        `MSTR_BFM.set_command_data(data, `INDEX_ZERO);
    end

    `MSTR_BFM.push_command();
end
endtask

//this task pops the response received by master BFM from queue
task master_pop_and_get_response;
// . . .
begin
    `MSTR_BFM.pop_response();
    request = Request_t' (`MSTR_BFM.get_response_request());
    addr = `MSTR_BFM.get_response_address();
    data = `MSTR_BFM.get_response_data(`INDEX_ZERO);
end
endtask
```



For more information about the methods that the Avalon-MM Master BFM uses, refer to the “[Application Program Interface](#)” on page 1-9 in the *Avalon Memory-Mapped Master BFM*.



Although this testbench is written in Verilog HDL, the Avalon Verification IP Suite supports VHDL by providing wrappers for the Avalon-MM Master and Slave BFMs. You can include the BFMs with wrappers in simulators that support mixed language simulation. For more information, refer to [Chapter 2, Avalon-MM Master BFM with Avalon-ST API Wrapper](#) and [Chapter 4, Avalon-MM Slave BFM with Avalon-ST API Wrapper](#).

Setting up the Test

This section describes the steps to build a test system in the SOPC Builder to verify the on-chip RAM using the Avalon-MM Master BFM.

Creating an SOPC Builder Testbench 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>*.

Follow these steps to create an SOPC Builder testbench:

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. Open the **master_bfm_project.qpf** file located in *<working_directory>\ug_avalon_verification\sopc_builder\tutorial_master_bfm*.
3. On the Tools menu, click **SOPC Builder** to launch the SOPC Builder tool.
4. Type "Avalon MM Master BFM" in the search box located in the **Component Library** panel. From the search results, double-click on the **Avalon MM Master BFM** component.
5. In the parameter editor, change the parameter values to match the values listed in [Table 1-1](#).

Table 1-1. Master BFM Parameter Values (Part 1 of 2)

Parameter	Value
Port Widths	
Address width	16
Symbol width	8
Read Response width	8
Write Response width	8
Parameters	
Number of symbols	4
Burstcount width	3
Port Enables	
Use the read signal	On
Use the write signal	On
Use the address signal	On
Use the byteenable signal	On
Use the burstcount signal	Off
Use the readdata signal	On
Use the readdatavalid signal	On
Use the writedata signal	On
Use the begintransfer signal	Off
Use the beginbursttransfer signal	Off
Use the arbiterlock signal	Off

Table 1–1. Master BFM Parameter Values (Part 2 of 2)

Parameter	Value
Use the lock signal	Off
Use the debugaccess signal	Off
Use the waitrequest signal	On
Use the clken signals	Off
Port Polarity	
Assert reset high	On
Assert waitrequest high	On
Assert read high	On
Assert write high	On
Assert byteenable high	On
Assert readdatavalid high	On
Assert arbiterlock high	Off
Assert lock high	Off
Burst Attributes	
Constant burst behavior	Off
Linewrap burst	Off
Burst on burst boundaries only	Off
Miscellaneous	
Maximum pending reads	1
Fixed read latency (cycles)	0
Timing	
Fixed read wait time (cycles)	0
Fixed write wait time (cycles)	0
Registered waitrequest	Off
Registered Incoming Signals	Off
Interface Address Type	
Set master interface address type to symbols or words	SYMBOLS

6. Click **Finish**.
7. Right-click on the component and select **Rename**. Rename the component name to `master_bfm`.
8. In the search box located in the Component Library panel, type `onchip memory`. From the search results, double-click the **On-Chip Memory (RAM or ROM)** component.
9. Retain the default settings for the on-chip RAM, and click **Finish**.
10. Right-click on the RAM and click **Rename**. Rename the component name to `ram`.

Connecting and Generating the SOPC Builder System

To connect and generate the SOPC Builder system, follow these steps:

1. Connect the master_bfm m0 Avalon-MM master port to the onchip_mem s1 Avalon-MM slave port using the following procedure:
 - a. Click on the m0 port then hover in the Connections column to display possible connections.
 - b. Click on the open dot at the intersection of the onchip_mem s1 port and the master_bfm m0 to create a connection.
2. Click **Generate**. Save the system if you are prompted to do so.

Running the Simulation

In this section you run a simulation in the ModelSim-Altera software for the testbench that you created. To complete this simulation you use the test program provided in the design files to provide simulation stimulus.

1. Start the ModelSim-Altera software.
2. On the File menu click **Change Directory**.
3. Navigate to
`<working_directory>\ug_avalon_verification\sopc_builder\tutorial_master_bfm`
and click **OK**.
4. On the Compile menu, click **Compile Options**.
5. Click the **Verilog & SystemVerilog** 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 **script.do**, and click **Open**. The script creates a new working library, compiles all source files, runs simulation, and loads signals into the ModelSim waveform viewer.



If you are running ModelSim-SE you must use the `-novopt` option to prevent ModelSim from optimizing the design, making the signals specified in for the wave viewer unavailable.

Observing the Results

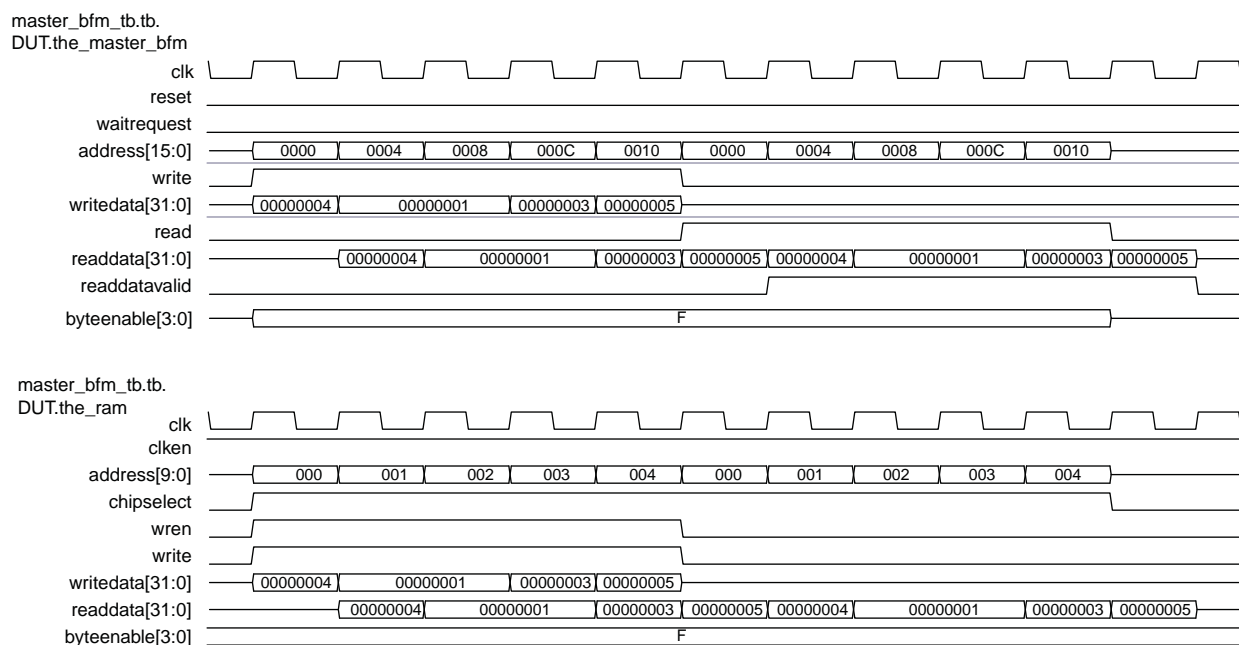
In this test, the Avalon-MM Master BFM writes five words of random data to the on-chip memory (DUT). The Avalon-MM Master BFM then reads back the five words and compares the data read to the expected values. If simulation is successful, the message shown in [Example 1-2](#) appears.

Example 1-2. Message in ModelSim Transcript Console when Running Simulation for Avalon-MM Slave DUT

```
960000: INFO: master_bfm_tb: Test has completed. 5 pass, 0 fail
```

[Figure 1-2](#) shows the waveform when the Avalon-MM Master BFM writes and reads to the slave DUT.

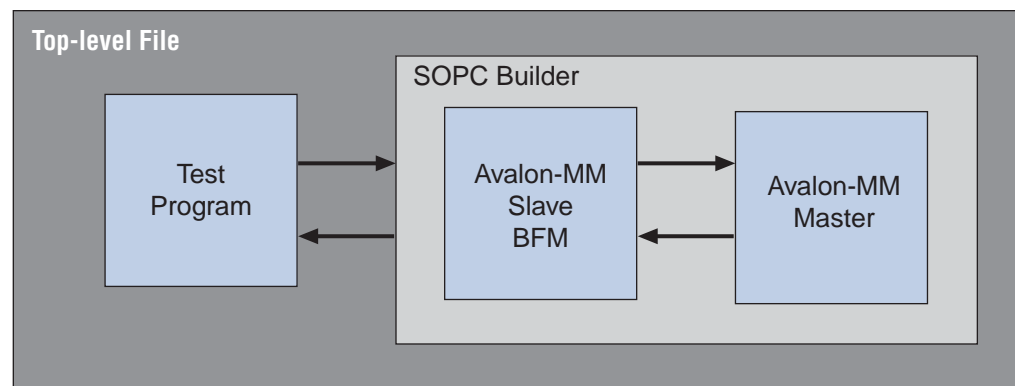
Figure 1-2. Master BFM writing to and reading from the Slave DUT



Verifying Avalon-MM Master DUT

Figure 1-3 illustrates the top-level testbench to verify an Avalon-MM master component using an Avalon-MM Slave BFM. The Avalon-MM master DUT is a simple write-read master that writes data to a slave component and reads back the data written.

Figure 1-3. Top-Level Testbench for Avalon-MM Master Component



The amount of data written is specified by the master's `BLOCKSIZE` parameter. The default value for this parameter is 4. When all data is written, the master DUT reads the data back from the slave BFM and checks it against the expected data. If a mismatch occurs, the master DUT asserts its exported error signal.

The Avalon-MM Slave BFM component responds to the master's commands when the `signal_command_received` event is triggered. The test program takes the master command from the slave BFM component's client command queue. If the command is a write, the write data is saved to a local array. For read commands, data is read out of the local array. The test program then constructs a response descriptor with the read data. The slave BFM drives the response to the master DUT. The test ends after the master DUT has received all responses from the slave BFM. The test passes if the master DUT does not assert its error signal.



For more information on the methods used by the Avalon-MM Slave BFM to construct commands, refer to the [“Application Program Interface”](#) on page 3-10 of the *Avalon Memory-Mapped Slave BFM*.

Setting Up the Test

This section describes the steps to build a test system in the SOPC Builder to verify the Avalon-MM master using the Avalon-MM Slave BFM.

Creating an SOPC Builder Testbench 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>`.

Follow these steps to create an SOPC Builder testbench:

1. Open the **slave_bfm_project.qpf** file located in
`<working_directory>\ug_avalon_verification\sopc_builder\tutorial_slave_bfm`.
2. On the Tools menu, click **SOPC Builder**.
3. To create the design, in the **System Contents** tab, expand **BFM Tutorial** and click **Write-Read Master** and then click **Add**.
4. Retain the default values given in the configuration wizard and click **Finish** to add this component to your system.
5. Right-click on the component and click **Rename**. Rename the component name to master.
6. In the search box located in the Component Library panel, type **Avalon mm slave bfm**. From the search results, double-click the **Altera Avalon-MM Slave BFM** component.
7. In the parameter editor, change the parameter values to match the values listed in [Table 1-2](#).

Table 1-2. Avalon-MM Slave BFM Parameter Values (Part 1 of 2)

Parameter	Value
Port Widths	
Address width	16
Symbol width	8
Read Response width	8
Write Response width	8
Parameters	
Number of symbols	4
Burstcount width	3
Port Enables	
Use the read signal	On
Use the write signal	On
Use the address signal	On
Use the bytenable signal	On
Use the burstcount signal	Off
Use the readdata signal	On
Use the readdatavalid signal	On
Use the writedata signal	On
Use the begintransfer signal	Off
Use the beginbursttransfer signal	Off
Use the arbiterlock signal	Off
Use the lock signal	Off
Use the debugaccess signal	Off
Use the waitrequest signal	On

Table 1-2. Avalon-MM Slave BFM Parameter Values (Part 2 of 2)

Parameter	Value
Use the clken signals	Off
Port Polarity	
Assert reset high	On
Assert waitrequest high	On
Assert read high	On
Assert write high	On
Assert byteenable high	On
Assert readdatavalid high	On
Assert arbiterlock high	Off
Assert lock high	Off
Burst Attributes	
Linewrap burst	Off
Burst on burst boundaries only	Off
Miscellaneous	
Maximum pending reads	2
Timing	
Fixed read latency (cycles)	0
Fixed read wait time (cycles)	1
Fixed write wait time (cycles)	0
Registered waitrequest	Off
Registered Incoming Signals	Off
Interface Address Type	
Set slave interface address type to symbols or words	WORDS

8. Click **Finish**.
9. Right-click on the component and select **Rename**. Rename the component name to slave_bfm.

Connecting and Generating the SOPC Builder System

To connect and generate the SOPC Builder system, follow these steps:

1. Connect the master m0 port to the slave_bfm s0 Avalon slave port using the following procedure:
 - a. Click on the master m0 port then hover in the **Connections** column to display possible connections.
 - b. Click on the open dot at the intersection of the slave_bfm s0 port and the m0 port to create a connection.
2. Click **Generate**. Save the system if you are prompted to do so.

Running the Simulation

Follow these steps, to run the simulation:

1. Start the ModelSim-Altera software.
2. On the File menu, click **Change Directory**.
3. Navigate to
`<working_directory>\ug_avalon_verification\sopc_builder\tutorial_slave_bfm`
 and click **OK**.
4. On the Compile menu, click **Compile Options**.
5. Click the **Select Verilog & SystemVerilog** tab.
6. In the **Language Syntax** box, select **Use SystemVerilog** and click **OK**.
7. On the File menu, click **Load** to open the ModelSim script file, **script.do**.

The script file creates a new working library, compiles all source files, runs simulation, and loads signals into the ModelSim wave viewer.



If you are running ModelSim-SE you must use the `-novopt` option to prevent ModelSim from optimizing the design, making the signals specified in for the wave viewer unavailable.

Observing the Results

In this example, the master DUT writes four data words to the Avalon-MM Slave BFM component and reads them back. The test program displays the simulation results in the ModelSim transcript console every time the Avalon-MM Slave BFM component receives master command. [Example 1-3](#) shows a partial transcript from a successful run.

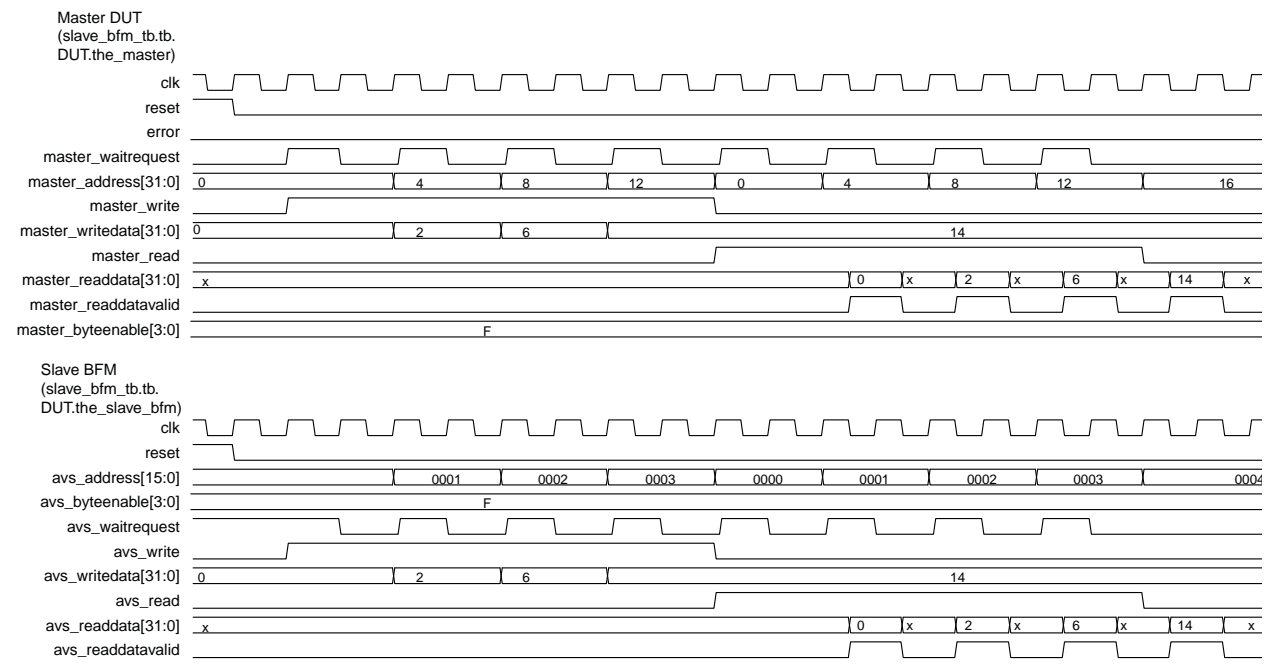
####

Example 1-3. Simulation Results in the ModelSim Transcript Console when Running Simulation for Avalon-MM Master DUT

```
# 251000: INFO: slave_bfm_tb: Master Write request to address 0000 with data 00000000
# 291000: INFO: slave_bfm_tb: Master Write request to address 0001 with data 00000002
# 331000: INFO: slave_bfm_tb: Master Write request to address 0002 with data 00000006
# 371000: INFO: slave_bfm_tb: Master Write request to address 0003 with data 0000000e
# 411000: INFO: slave_bfm_tb: Master Read request from address 0000
# 451000: INFO: slave_bfm_tb: Master Read request from address 0001
# 491000: INFO: slave_bfm_tb: Master Read request from address 0002
# 531000: INFO: slave_bfm_tb: Master Read request from address 0003
```

Figure 1-4 shows the waveforms for the Avalon-MM master DUT write and reads to the Avalon-MM Slave BFM component.

Figure 1-4. Avalon-MM Master Writes and Reads to Avalon-MM Slave BFM



This chapter demonstrates how to use the Avalon-ST Source and Sink BFM 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 BFM to verify the DUT behavior.

Software Requirements

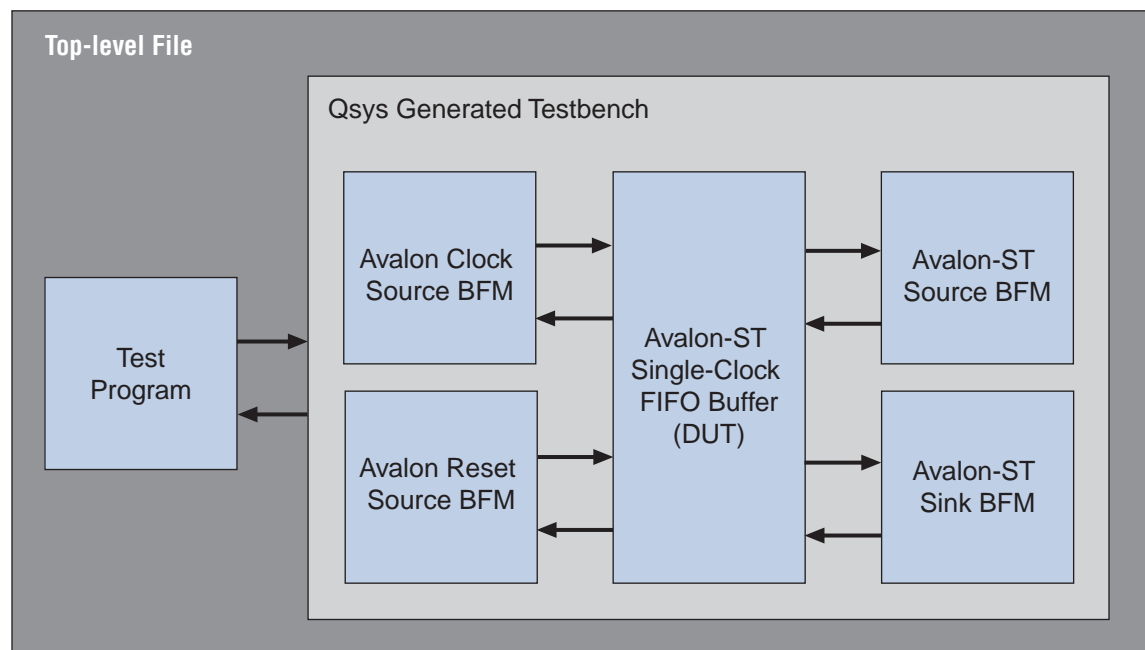
The following software and file are required to run the test:

- Quartus II software, version 12.0 or later.
- 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 2–1 shows the test setup to verify the Avalon-ST Single-Clock FIFO buffer using the Avalon-ST Source and Sink BFM. The Avalon Clock Source and Reset Source BFM 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 BFM using the BFM API to drive and monitor transactions.

Figure 2–1. Top-Level Testbench for Avalon-ST DUT Component



The test flow includes the following steps:

1. The test program initializes the BFM.
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.

- In the parameter editor, change the parameter values to match the values listed in [Table 2-1](#).

Table 2-1. 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

- Click **Finish**.
- Right-click on the `sc_fifo_0` component and select **Rename**. Rename the component to `dut`.
- On the **System Contents** tab, in the **Export** column, rename the exported interface names to match the names listed in [Table 2-2](#).

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

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

Generating a Qsys Testbench System

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

- On the **Generation** tab, change the parameter values to match the values listed in [Table 2-3](#).

Table 2-3. Generation Tab Parameter Values

Parameters	Value
Simulation	
Create simulation model	None
Create testbench Qsys system	Standard, BFM 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 2-3. Generation Tab Parameter Values

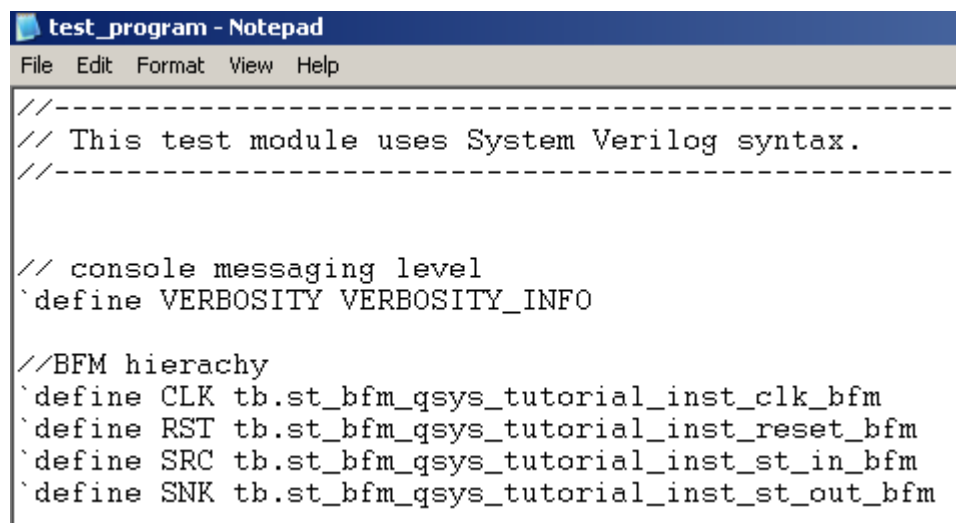
Parameters	Value
Output Directory	
Path	<code><working_directory>\ug_avalon_verification\qsys\st_bfm_qsys_tutorial</code>

- Click **Generate**. Save the system if you are prompted to do so. Do not close the Qsys window after successful generation.
- 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`.
- In the `st_bfm_qsys_tutorial_tb.html` file, verify that the names of the generated BFM s match the instance names in [Table 2-4](#).

Table 2-4. Generated BFM Instance Names

BFM Type	Instance Name
<code>altera_avalon_clock_source</code>	<code>st_bfm_qsys_tutorial_inst_clk_bfm</code>
<code>altera_avalon_reset_source</code>	<code>st_bfm_qsys_tutorial_inst_reset_bfm</code>
<code>altera_avalon_st_source_bfm</code>	<code>st_bfm_qsys_tutorial_inst_st_in_bfm</code>
<code>altera_avalon_st_sink_bfm</code>	<code>st_bfm_qsys_tutorial_inst_st_out_bfm</code>

- Use the instance names listed in [Table 2-4](#) to define and access the APIs of the corresponding BFM s in your test program. [Figure 2-2](#) shows a code example that uses instance names to define a particular BFM in the test program.

Figure 2-2. Using Instance Names to Define BFM s


```

test_program - Notepad
File Edit Format View Help

//-----
// This test module uses System Verilog syntax.
//-----

// console messaging level
`define VERBOSITY VERBOSITY_INFO

//BFM hierachy
`define CLK tb.st_bfm_qsys_tutorial_inst_clk_bfm
`define RST tb.st_bfm_qsys_tutorial_inst_reset_bfm
`define SRC tb.st_bfm_qsys_tutorial_inst_st_in_bfm
`define SNK tb.st_bfm_qsys_tutorial_inst_st_out_bfm

```



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 2-5](#) for the coding examples.

Table 2-5. Coding Examples to Set Hierarchy Variables

Hierarchy Variables Coding Example	Description
<code>set TOP_LEVEL_NAME "top"</code>	Sets the name of the top level file that instantiates the Qsys-generated testbench system and the test program.
<code>set QSYS_SIMDIR "../st_bfm_qsys_tutorial/testbench</code>	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 2-1 shows an extract of the simulation results.

Example 2-1. 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 2-1](#) 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 2-1](#) 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.

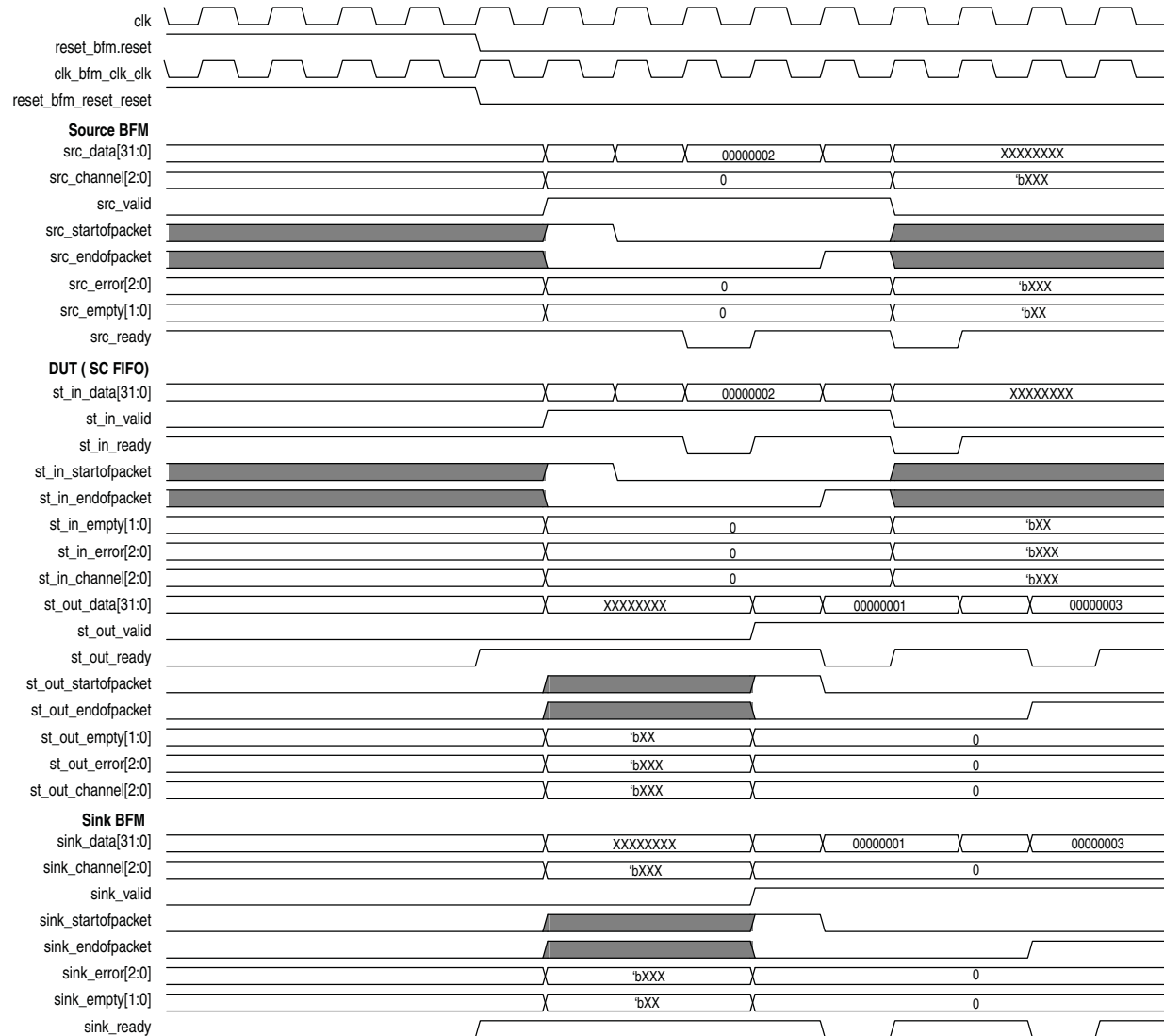
[Example 2-2](#) shows the ModelSim transcript for the source response latency, which is the number of clock cycles the Avalon-ST Single-Clock FIFO buffer takes when the Avalon-ST Single-Clock FIFO buffer backpressures the Avalon-ST Source BFM. The third response shows a non-zero response latency. During 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 2-2. 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 2–3 shows the simulation waveforms in the ModelSim-Altera software wave window.

Figure 2–3. Using Instance Names to Define BFM



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
June 2012	3.1	<ul style="list-style-type: none"> ■ Updated SOPC Tutorial chapter. ■ Updated Qsys Tutorial chapter.
May 2011	3.0	<ul style="list-style-type: none"> ■ Added External Memory BFM chapter. ■ Updated Avalon-MM Master and Slave BFM chapters. ■ Updated Avalon-MM Monitor chapter. ■ Updated SOPC Tutorial chapter. ■ Added Qsys Tutorial chapter.
January 2011	2.0	<ul style="list-style-type: none"> ■ 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 BFM chapters. ■ Updated Avalon-MM Master and Slave BFM chapters. ■ Updated Avalon-ST Source and Sink BFM 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 <i>Avalon Verification IP Suite User Guide</i> .

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








Contact (1)	Contact Method	Address
(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.
<i>Italic Type with Initial Capital Letters</i>	Indicate document titles. For example, <i>Stratix IV Design Guidelines</i> .
<i>italic type</i>	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.
	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.
 CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
 WARNING	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.