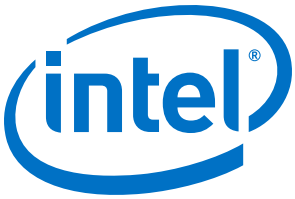


Avalon Verification IP Suite

User Guide

Updated for Intel® Quartus® Prime Design Suite: **20.4**

Subscribe
Send Feedback

UG-01073 | 2021.01.29
Latest document on the web: PDF | HTML

Contents

1. Introduction to Avalon Verification IP Suite 10
1.1. Advantages of Using BFMs and Monitors 10
1.2. BFM Implementation 10
1.3. Application Programming Interface 12
1.4. Application Example of BFMs 13

2. Clock Source BFM 14
2.1. Parameters 14
2.2. Clock Source API 14
2.2.1. Clock_stop() 14
2.2.2. get_run_state() 14
2.2.3. get_version() 15

3. Reset Source BFM 16
3.1. Parameters 16
3.2. Reset Source API 16
3.2.1. reset_deassert 16
3.2.2. get_version() 16

4. Avalon Interrupt Source and Interrupt Sink BFMs 18
4.1. Parameters 18
4.2. Interrupt Source and Sink API 18
4.2.1. get_irq() 19
4.2.2. get_version() 19
4.2.3. set_irq() 19

5. Avalon-MM Master BFM 20
5.1. Timing 20
5.2. Block Diagram 24
5.3. Parameters 26
5.4. Avalon-MM Master BFM API 28
5.4.1. event_all_transactions_complete() 28
5.4.2. event_command_issued() 28
5.4.3. event_max_command_queue_size() 28
5.4.4. event_min_command_queue_size() 29
5.4.5. event_read_response_complete() 29
5.4.6. event_response_complete() 29
5.4.7. event_write_response_complete() 29
5.4.8. get_command_issued_queue_size() 29
5.4.9. get_command_pending_queue_size() 30
5.4.10. get_read_response_queue_size() 30
5.4.11. get_response_address() 30
5.4.12. get_response_byte_enable() 30
5.4.13. get_response_burst_size() 31
5.4.14. get_response_data() 31
5.4.15. get_response_latency() 31
5.4.16. get_response_queue_size() 31
5.4.17. get_response_read_id() 32

5.4.18. get_response_read_response() 32
5.4.19. get_response_request() 32
5.4.20. get_response_wait_time() 32
5.4.21. get_response_write_id() 33
5.4.22. get_write_response_status() 33
5.4.23. get_write_response_queue_size() 33
5.4.24. get_version() 33
5.4.25. init() 34
5.4.26. pop_response() 34
5.4.27. push_command() 34
5.4.28. set_clken() 34
5.4.29. set_command_address() 35
5.4.30. set_command_arbiterlock() 35
5.4.31. set_command_byte_enable() 35
5.4.32. set_command_burst_count() 35
5.4.33. set_command_burst_size() 36
5.4.34. set_command_data() 36
5.4.35. set_command_debugaccess() 36
5.4.36. set_command_idle() 36
5.4.37. set_command_init_latency() 37
5.4.38. set_command_lock() 37
5.4.39. set_command_request() 37
5.4.40. set_command_timeout() 37
5.4.41. set_command_transaction_id() 38
5.4.42. set_command_write_response_request() 38
5.4.43. set_max_command_queue_size() 38
5.4.44. set_min_command_queue_size() 38
5.4.45. set_response_timeout() 38
5.4.46. signal_all_transactions_complete 39
5.4.47. signal_command_issued 39
5.4.48. signal_fatal_error 39
5.4.49. signal_max_command_queue_size 39
5.4.50. signal_min_command_queue_size 40
5.4.51. signal_read_response_complete 40
5.4.52. signal_response_complete 40
5.4.53. signal_write_response_complete 40

6. Avalon-MM Slave BFM 41

6.1. Timing 42
6.2. Block Diagram 46
6.3. Parameters 48
6.4. Avalon-MM Slave BFM API 49
6.4.1. event_command_received() 50
6.4.2. event_response_issued() 50
6.4.3. event_max_response_queue_size() 50
6.4.4. event_min_response_queue_size() 50
6.4.5. get_clken() 51
6.4.6. get_command_address() 51
6.4.7. get_command_arbiterlock() 51
6.4.8. get_command_burst_count() 51
6.4.9. get_command_burst_cycle() 51

6.4.10. get_command_byte_enable() 52
6.4.11. get_command_data() 52
6.4.12. get_command_debugaccess() 52
6.4.13. get_command_queue_size() 52
6.4.14. get_command_lock() 53
6.4.15. get_command_request() 53
6.4.16. get_command_transaction_id() 53
6.4.17. get_command_write_response_request() 53
6.4.18. get_pending_read_latency_cycle() 54
6.4.19. get_pending_write_latency_cycle() 54
6.4.20. get_response_queue_size() 54
6.4.21. vget_slave_bfm_status 54
6.4.22. get_version() 55
6.4.23. init() 55
6.4.24. pop_command() 55
6.4.25. push_response() 55
6.4.26. set_command_transaction_mode() 56
6.4.27. set_interface_wait_time() 56
6.4.28. vset_max_response_queue_size() 56
6.4.29. set_min_response_queue_size() 56
6.4.30. set_read_response_id() 57
6.4.31. set_read_response_status() 57
6.4.32. set_response_burst_size() 57
6.4.33. set_response_data() 57
6.4.34. set_response_latency() 57
6.4.35. set_response_request() 58
6.4.36. set_response_timeout() 58
6.4.37. set_write_response_id() 58
6.4.38. set_write_response_status() 59
6.4.39. signal_command_received() 59
6.4.40. signal_error_exceed_max_pending_reads 59
6.4.41. signal_max_response_queue_size 59
6.4.42. signal_min_command_queue_size 60
6.4.43. signal_fatal_error 60
6.4.44. signal_response_issued 60

7. Avalon-MM Monitor 61

7.1. Parameters 62
7.2. Avalon-MM Monitor Assertion Checking API 64
7.2.1. set_enable_a_address_align_with_data_width() 64
7.2.2. set_enable_a_beginbursttransfer_exist() 64
7.2.3. set_enable_a_beginbursttransfer_legal() 64
7.2.4. set_enable_a_beginbursttransfer_single_cycle() 65
7.2.5. set_enable_a_begintransfer_exist() 65
7.2.6. set_enable_a_begintransfer_legal() 65
7.2.7. set_enable_a_begintransfer_single_cycle() 65
7.2.8. set_enable_a_burst_legal() 66
7.2.9. set_enable_a_byteenable_legal() 66
7.2.10. set_enable_a_constant_during_burst() 66
7.2.11. set_enable_a_constant_during_clk_disabled() 66
7.2.12. set_enable_a_constant_during_waitrequest() 67

7.2.13. set_enable_a_exclusive_read_write() 67
7.2.14. set_enable_a_half_cycle_reset_legal() 67
7.2.15. set_enable_a_less_than_burstcount_max_size() 67
7.2.16. set_enable_a_less_than_maximumpendingreadtransactions() 68
7.2.17. set_enable_a_no_readdatavalid_during_reset() 68
7.2.18. set_enable_a_no_read_during_reset() 68
7.2.19. set_enable_a_no_write_during_reset() 68
7.2.20. set_enable_a_readid_sequence() 69
7.2.21. set_enable_a_read_response_sequence() 69
7.2.22. set_enable_a_read_response_timeout() 69
7.2.23. set_enable_a_register_incoming_signals() 69
7.2.24. set_enable_a_waitrequest_during_reset() 70
7.2.25. set_enable_a_waitrequest_timeout() 70
7.2.26. set_enable_a_write_burst_timeout() 70
7.2.27. set_enable_a_writeid_sequence() 70
7.2.28. Coverage Group 71
7.2.29. Transaction Monitoring 78

8. Avalon-ST Source BFM 91
8.1. Timing 91
8.2. Block Diagram 92
8.3. Parameters 93
8.4. Avalon-ST Source API 94
8.4.1. event_min_transaction_queue_size() 94
8.4.2. event_response_done() 95
8.4.3. event_src_driving_transaction() 95
8.4.4. event_src_not_ready() 95
8.4.5. event_src_ready() 95
8.4.6. event_src_transaction_complete() 96
8.4.7. get_response_latency() 96
8.4.8. get_response_queue_size() 96
8.4.9. get_src_ready() 96
8.4.10. get_src_transaction_complete() 96
8.4.11. get_transaction_queue_size() 97
8.4.12. get_version() 97
8.4.13. init() 97
8.4.14. pop_response() 97
8.4.15. push_transaction() 98
8.4.16. set_max_transaction_queue_size() 98
8.4.17. set_min_transaction_queue_size() 98
8.4.18. set_response_timeout() 98
8.4.19. set_transaction_channel() 99
8.4.20. set_transaction_data() 99
8.4.21. set_transaction_idles() 99
8.4.22. set_transaction_eop() 99
8.4.23. set_transaction_empty() 99
8.4.24. set_transaction_error() 100
8.4.25. set_transaction_sop() 100
8.4.26. signal_fatal_error 100
8.4.27. signal_max_transaction_queue_size 100
8.4.28. signal_min_transaction_queue_size 101

8.4.29. signal_response_done 101
8.4.30. signal_src_driving_transaction 101
8.4.31. signal_src_not_ready 101
8.4.32. signal_src_ready 101
8.4.33. signal_src_transaction_complete 102

9. Avalon-ST Sink BFM 103
9.1. Timing 103
9.2. Block Diagram 104
9.3. Parameters 105
9.4. Application Program Interface 106
9.4.1. event_sink_ready_assert() 107
9.4.2. event_sink_ready_deassert() 107
9.4.3. get_transaction_channel() 107
9.4.4. get_transaction_data() 107
9.4.5. get_transaction_idles() 107
9.4.6. get_transaction_eop() 108
9.4.7. get_transaction_empty() 108
9.4.8. get_transaction_error() 108
9.4.9. get_transaction_queue_size() 108
9.4.10. get_transaction_sop() 109
9.4.11. get_version() 109
9.4.12. init() 109
9.4.13. pop_transaction() 109
9.4.14. set_ready() 109
9.4.15. signal_fatal_error 110
9.4.16. signal_sink_ready_assert 110
9.4.17. signal_sink_ready_deassert 110
9.4.18. signal_transaction_received 110

10. Avalon-ST Monitor 111
10.1. Parameters 112
10.2. Avalon-ST Monitor Assertion Checking API 112
10.2.1. set_enable_a_empty_legal() 113
10.2.2. set_enable_a_less_than_max_channel() 113
10.2.3. set_enable_a_no_data_outside_packet() 113
10.2.4. set_enable_a_non_missing_endofpacket() 114
10.2.5. set_enable_a_non_missing_startofpacket() 114
10.2.6. set_enable_a_valid_legal() 114
10.2.7. Coverage Group 114
10.2.8. Transaction Monitoring 121

11. Avalon Streaming Credit Source BFM 127
11.1. Parameters 127
11.2. Implementation 128
11.3. Avalon Streaming Interface Credit Source APIs 128
11.3.1. initialize() 128
11.3.2. rst() 128
11.3.3. get_src_transaction_complete() 129
11.3.4. get_outstanding_credit() 129
11.3.5. push_transaction() 129
11.3.6. set_max_transaction_queue_size() 129

11.3.7. set_min_transaction_queue_size()....129
11.3.8. set_transaction_data()....130
11.3.9. set_transaction_channel()....130
11.3.10. set_transaction_error()....130
11.3.11. set_transaction_idles()....130
11.3.12. set_transaction_sop()....130
11.3.13. set_transaction_eop()....131
11.3.14. set_transaction_empty()....131
11.3.15. return_credit()....131
11.3.16. set_user_signal_per_symbol_data()....131
11.3.17. push_user_signal_per_symbol_transaction()....131
11.3.18. set_user_signal_per_packet_data()....132
11.3.19. push_user_signal_per_packet_transaction()....132
11.3.20. signal_credit_arrived....132
11.3.21. signal_fatal_error....132
11.3.22. signal_src_transaction_complete....133
11.3.23. signal_src_driving_transaction....133
11.3.24. signal_max_transaction_queue_size....133
11.3.25. signal_min_transaction_queue_size....133

12. Avalon Streaming Credit Sink BFM....134

12.1. Parameters....134
12.2. Implementation....135
12.3. Avalon Streaming Interface Credit Sink APIs....135
12.3.1. initialize()....135
12.3.2. rst()....135
12.3.3. get_outstanding_credit()....136
12.3.4. get_transaction_idles()....136
12.3.5. get_transaction_data()....136
12.3.6. get_transaction_channel()....136
12.3.7. get_transaction_error()....136
12.3.8. get_transaction_sop()....137
12.3.9. get_transaction_eop()....137
12.3.10. get_transaction_empty()....137
12.3.11. pop_transaction()....137
12.3.12. send_credit()....137
12.3.13. get_user_signal_per_symbol_data()....138
12.3.14. pop_user_signal_per_symbol_transaction()....138
12.3.15. get_user_signal_per_packet_data()....138
12.3.16. pop_user_signal_per_packet_transaction()....138
12.3.17. signal_fatal_error....139
12.3.18. signal_has_max_credits....139
12.3.19. signal_transaction_received....139

13. Conduit BFM....140

13.1. Parameters....141
13.2. Conduit BFM API....141
13.2.1. event_reset_asserted....141
13.2.2. get_<role name>()....142
13.2.3. get_version()....142
13.2.4. set_<role name>()....142

13.2.5. set_<role name>_oe() 142
13.2.6. signal_input_<role name>_change 143

14. Tri-State Conduit BFM 144
14.1. Parameters 144
14.2. Tri-State Conduit BFM API 145
14.2.1. event_interface_granted() 145
14.2.2. event_grant_deasserted_while_request_remain_asserted 145
14.2.3. event_max_transaction_queue_size() 146
14.2.4. event_min_transaction_queue_size() 146
14.2.5. get_input_transaction_queue_size() 146
14.2.6. get_output_transaction_queue_size() 146
14.2.7. get_transaction_<role name>_in() 147
14.2.8. get_transaction_latency() 147
14.2.9. get_version() 147
14.2.10. pop_transaction() 147
14.2.11. push_transaction() 147
14.2.12. set_max_transaction_queue_size() 148
14.2.13. set_min_transaction_queue_size() 148
14.2.14. set_num_of_transactions() 148
14.2.15. set_transaction_<role name>_out() 148
14.2.16. set_transaction_<role name>_outen() 149
14.2.17. set_transaction_idles() 149
14.2.18. set_valid_transaction_<role name>_out() 149
14.2.19. signal_all_transactions_complete 149
14.2.20. signal_fatal_error 149
14.2.21. signal_grant_deasserted_while_request_remain_asserted 150
14.2.22. signal_interface_granted 150
14.2.23. signal_max_transaction_queue_size 150
14.2.24. signal_min_transaction_queue_size 150

15. External Memory BFM 152
15.1. Using the External Memory BFM 153
15.2. Parameters 153
15.3. External Memory BFM API 155
15.3.1. read() 156
15.3.2. signal_api_call 156
15.3.3. write() 156

16. Nios II Custom Instruction Master BFM 157
16.1. Parameters 159
16.2. Nios II Custom Instruction API 160
16.2.1. event_result_received() 160

17. Nios II Custom Instruction Slave BFM 171
17.1. Parameters 172
17.2. Nios II Custom Instruction Slave BFM API 172
17.2.1. event_instruction_inconsistent() 173
17.2.2. event_instruction_unchanged() 173
17.2.3. event_result_driven() 173
17.2.4. event_result_done() 173
17.2.5. event_unknown_instruction_received() 174

17.2.6. get_ci_clk_en() 174
17.2.7. get_instruction_a() 174
17.2.8. get_instruction_b() 174
17.2.9. get_instruction_c() 174
17.2.10. get_instruction_dataa() 175
17.2.11. get_instruction_datab() 175
17.2.12. get_instruction_idle() 175
17.2.13. get_instruction_n() 175
17.2.14. get_instruction_readra() 176
17.2.15. get_instruction_readrb() 176
17.2.16. get_instruction_writerc() 176
17.2.17. get_version() 176
17.2.18. insert_result() 176
17.2.19. retrieve_instruction() 177
17.2.20. set_clock_enable_timeout() 177
17.2.21. set_instruction_a() 178
17.2.22. set_instruction_b() 178
17.2.23. set_instruction_c() 178
17.2.24. set_instruction_timeout() 178
17.2.25. set_result_delay() 178
17.2.26. set_result_err_inject() 179
17.2.27. set_result_value() 179
17.2.28. signal_fatal_error 179
17.2.29. signal_instructions_inconsistent 179
17.2.30. signal_known_instruction_received 180
17.2.31. signal_result_done 180
17.2.32. signal_result_driven 180
17.2.33. signal_unknown_instruction_received 180

18. Avalon-ST Verilog HDL Testbench 181

18.1. **Verifying** Avalon-ST DUT 181
18.2. About the Test 182
18.2.1. Creating the Qsys Design 183
18.2.2. Generating a Qsys Testbench System 184
18.2.3. Running the Simulation 185
18.2.4. Observing the Results 185

19. Avalon-MM Verilog HDL and VHDL Testbenches 187

19.1. Avalon-MM Verilog HDL Testbench Description 187
19.1.1. Running the Verilog HDL Testbench for a Single Avalon-MM Master and Slave Pair 188
19.1.2. Running the Verilog HDL Testbench for the Two Avalon-MM Masters and Slaves 190
19.2. Avalon-MM VHDL Testbench Description 192
19.2.1. Running the Testbench for a Single Avalon-MM Master and Slave Pair 193
19.2.2. Running the Testbench for Two Avalon-MM Masters Slaves 194
19.3. **Using the VHDL BFMs** 196

20. Document Revision History 198

1. Introduction to Avalon Verification IP Suite

The Avalon® Verification IP Suite provides bus functional models (BFMs) to simulate the behavior and facilitate the verification of IP. The Verification IP Suite includes BFMs for 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

1.1. Advantages of Using BFMs and Monitors

Using these 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, the verification suite provides a platform that you can use to implement constraint-driven randomized tests. For example, you can implement the following modules for random testing:
  - Traffic scenario drivers
  - Scoreboard and coverage facilities
  - Assertion checkers

1.2. BFM Implementation

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

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

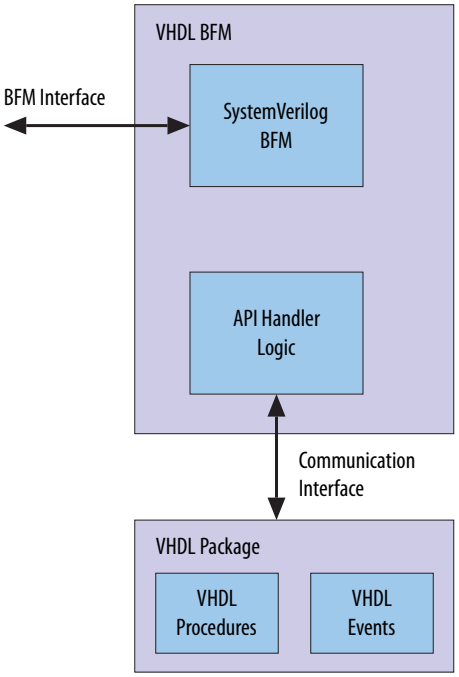
Table 1. BFM Language Support

BFM	Verilog HDL Support	VHDL Support
Clock Source and Reset Source	Yes	Yes
Avalon Interrupt Source and Sink	Yes	Version 13.0 and higher
Avalon-MM Master, Slave, and Monitor	Yes	Version 13.0 and higher
Avalon-ST Source, Sink, and Monitor	Yes	Version 13.0 and higher
Conduit and Tri-State Conduit	Yes	Version 14.0 and higher
External Memory	Yes	Version 13.0 and higher
Nios II Custom Instruction Master and Slave	Yes	Version 13.0 and higher

The VHDL BFM has four parts as shown in the figure below.

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

Figure 1. VHDL Component BFM



The monitor components use the SystemVerilog Assertion (SVA) language and are supported only by simulators that support SVA, including:

- ModelSim - Intel FPGA Edition
- Synopsys VCS
- Mentor Graphics® Questa.

1.3. Application Programming Interface

Intel provides you with a set of application programming interfaces (API) for each Avalon Verification IP Suite BFM. You can use the APIs 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.

Note: You can design custom verification environments that do not take advantage of the API. However, Altera does not guarantee continued support or backwards compatibility custom methods.

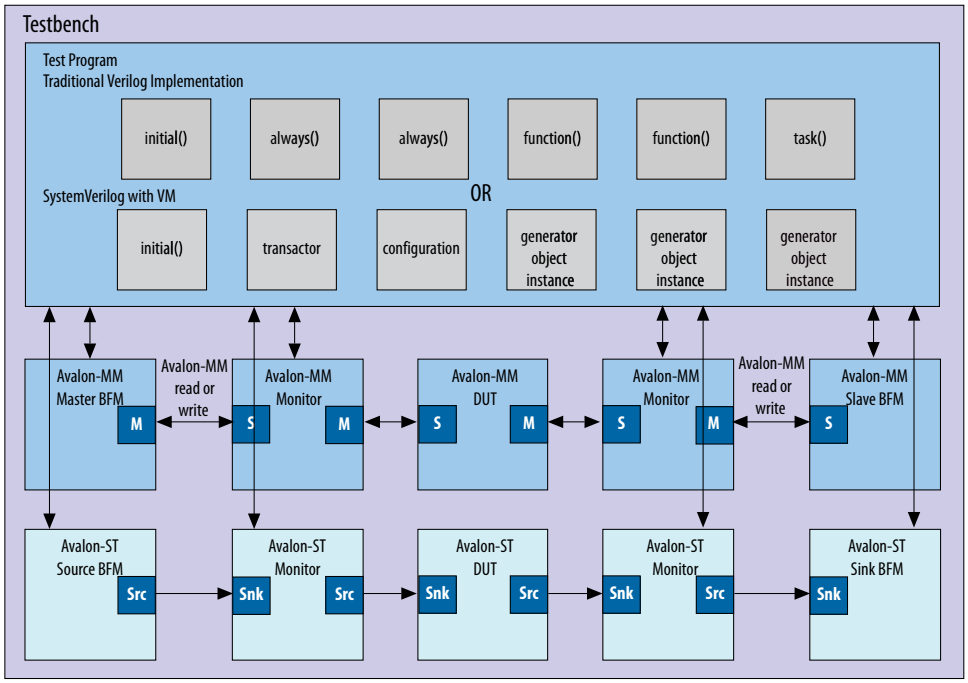
1.4. Application Example of BFMs

The figure below shows an Avalon-MM design with the following components:

- An Avalon-MM device under test (DUT) that includes both Avalon-MM master and slave interfaces
- An Avalon-ST DUT that includes both source and sink interfaces, although typical components might include a single Avalon interface.

This figure illustrates it is possible to write a testbench using a traditional VerilogHDL implementation or using SystemVerilog with VMM.

Figure 2. Avalon Verification IP Suite Testbench for Avalon-MM and Avalon-ST Interfaces



To verify a component with Avalon-MM interfaces, insert a monitor between the master BFM and the slave interface. To verify a component with Avalon-ST interfaces, insert a monitor between the source BFM and sink interface. You can insert a second monitor between the slave or sink BFM and the master or source interface of the DUT. You can inserted monitors anywhere in the system to provide protocol assertion checking and functional coverage reporting.

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

2. Clock Source BFM

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

Note: The Clock Source BFM is only supported in Qsys.

2.1. Parameters

Table 2. Clock Source BFM Parameter Settings

Option	Default Value	Legal Values	Description
Clock rate	**10**	N/A	Specifies the clock rate in MHz.

2.2. Clock Source API

clock_start()

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

2.2.1. Clock_stop()

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

2.2.2. get_run_state()

Prototype:	`get_run_state()`
Arguments:	Verilog HDL: None VHDL: N.A.

continued...

Returns:	`bit`
Description:	Returns the state of the clock source; 1=running, 0=stop.
Language support:	Verilog HDL

2.2.3. get_version()

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

3. Reset Source BFM

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

3.1. Parameters

Table 3. 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**	N/A	Specifies the number of cycles that the reset signal is asserted at the initial stage of the simulation.

3.2. Reset Source API

reset_assert

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

3.2.1. reset_deassert

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

3.2.2. get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None

continued...

	VHDL: N.A.
Returns:	`String`.
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 10.1 sp1 is encoded as "10.1.1".
Language support:	Verilog HDL

4. Avalon Interrupt Source and Interrupt Sink BFMs

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

4.1. Parameters

Table 4. Clock Source 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 `irq` to `irq_n`.
IRQ width	**1**	**1–32**	Specifies the width of the interrupt source signal.
Asynchronous IRQ	**Off**	**On/Off**	Specifies whether the interrupt signal is asserted or deasserted immediately after an API call or one clock cycle after an API call. Turn on this option to allow changes to the interrupt signal immediately after an API call. Turn off this option to allow changes to the interrupt signal on the next clock edge.
VHDL BFM ID	**0**	**1–1023**	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.
Interrupt Sink			
Assert IRQ high	**On**	**On/Off**	Specifies the polarity of the interrupt sink signal. Turn on this option to change the name of the interrupt source signal port from `irq` to `irq_n`.
IRQ width	**1**	**1–32**	Specifies the width of the interrupt source signal.

4.2. Interrupt Source and Sink API

clear_irq()

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

4.2.1. get_irq()

get_irq()

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

4.2.2. get_version()

get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`String`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 13.1 sp1 is encoded as "13.1.1".
Language Support:	Verilog HDL

4.2.3. set_irq()

set_irq()

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

5. Avalon-MM Master BFM

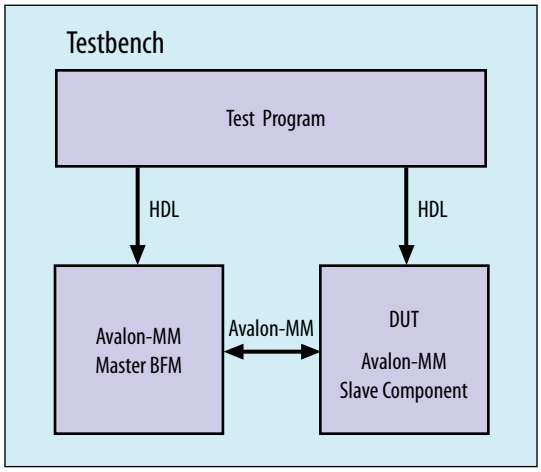
The Avalon-MM Master BFM implements the Avalon-MM interface protocol, including: read, write, burst read, and burst write. The figure below shows the top-level modules for a testbench using the Avalon-MM BFM to verify an Avalon-MM slave component. The typical testbench includes the following components:

- The Avalon-MM Master BFM
- A test program
- The DUT that includes an Avalon-MM slave interface

Using the Avalon-MM BFM has the following advantage. It highlights any misinterpretation of the Avalon-MM protocol that might be missed in a testbench designed by a single engineer.

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

Figure 3. Top-Level Module to Verify an Avalon-MM Slave Device



Related Information

Avalon Interface Specifications

5.1. Timing

The following timing diagram illustrates the sequence of events for an Avalon-MM Master BFM. The Master BFM drives 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 4. Avalon-MM Master Driving Interleaved Write and Read Transactions

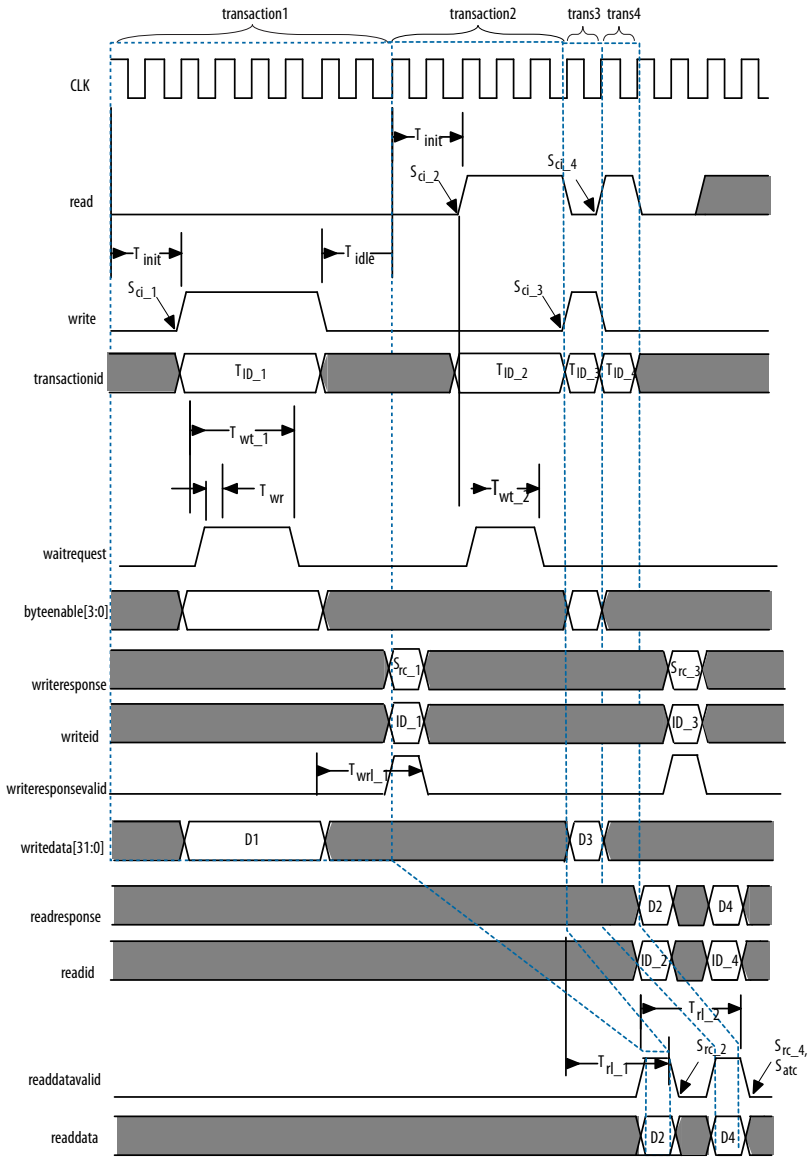


Table 5. Key to the Annotations

The following table lists the annotations used in the figure.

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

continued...

Symbol	Description
T_{idle}	The idle time after each transaction. This time is set by the command `set_command_idle`.
T_{rl_1}	The response latency for the first read, which is 3 cycles. This is the time between the read command acceptance and the read response provided by the slave. The program gets this time using the `get_response_latency` command. If an Avalon-MM slave component defines the `readLatency` interface property, the `readdatavalid` signal is not used. The `readdatavalid` signal is not necessary because the slave component has a fixed read latency. For more information refer to the Avalon Interface Specifications.
T_{rl_2}	The response latency for the second read, which is 3 cycles. The program gets this time using the `get_response_latency` command.
T_{wrl_1}	The write response latency for the first write, which is 3 cycles. This is the time between when the write command acceptance and the write response is provided by the slave. The program gets this time using the `get_response_latency` command.
S_{ci_1}–S_{ci_4}	Signals when write or read commands are presented on the interface. The event name is `signal_command_issued`.
S_{rc_1},S_{rc_3}	Signals write responses. The event name is `signal_response_complete`.
S_{rc_2},S_{rc_4}	Signals read responses. The event name is `signal_response_complete`.
S_{atc}	Signals the end of the test. The event name is `signal_all_transactions_complete`
T_{ID_1}–T_{ID_4}	Reference number to identify each read or write transaction.
ID_1, ID_3	Reference number to identify each write transaction.
ID_2, ID_4	Reference number to identify each read transaction.

Figure 5. Avalon-MM Master Driving Write and Read Transactions with No readdatavalid Signal

The timing in the following figure shows the sequence of events for an Avalon-MM Master BFM. The Avalon-MM Master BFM drives a write followed by a read when the `readdatavalid` signal is not present.

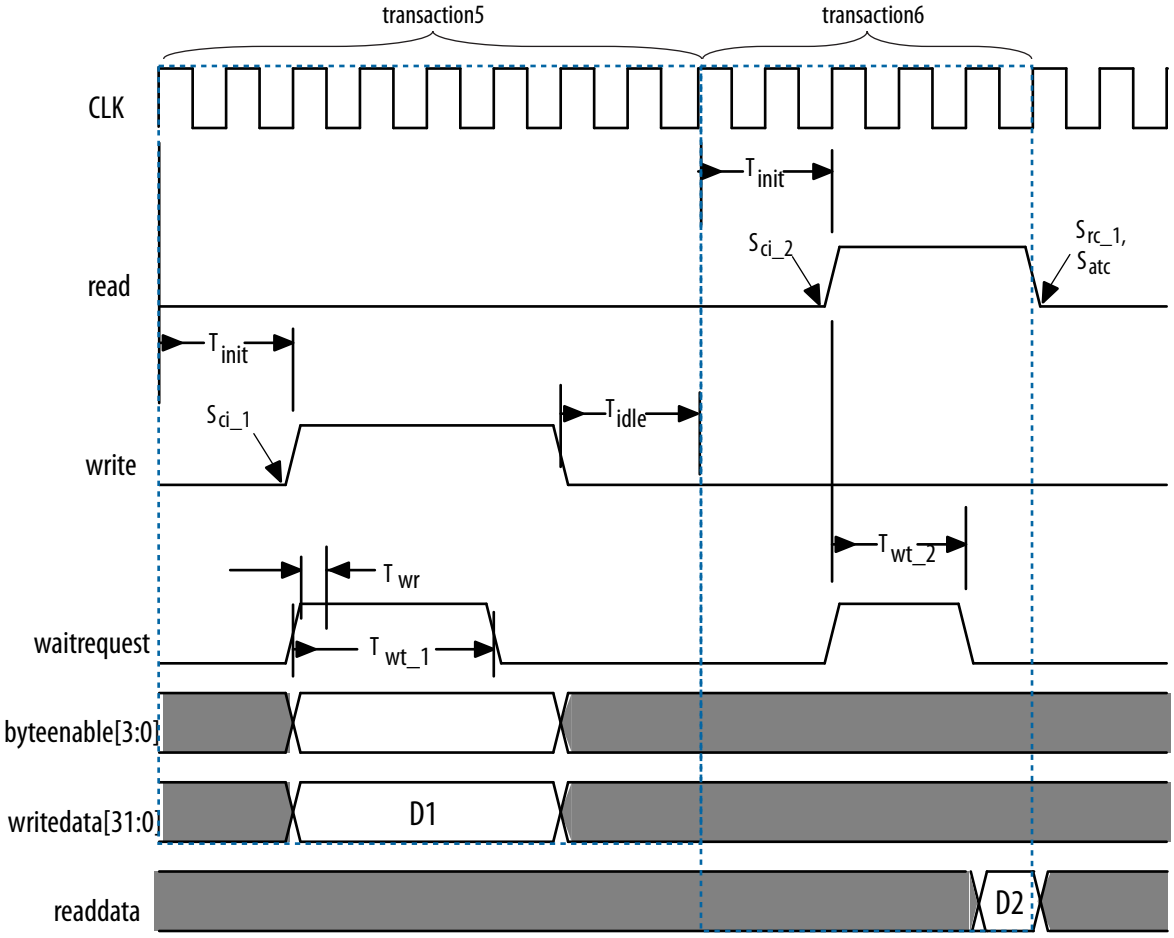


Table 6. Key to the Annotations

The following table lists the annotations used in this figure.

Symbol	Description
T_{init}	The initial command latency, which is 2 cycles for transactions 1 and 2. This time is set by the API command `set_command_init_latency`.
T_{wt_1}	The response wait time, which is 3 cycles. This time is determined by the number of cycles that the `waitrequest` signal is asserted by the slave.The program gets this value using the `get_response_wait_time` command.
T_{wt_2}	The response wait time for the first read, which is 2 cycles. This time is determined by the number of cycles that the `waitrequest` signal is asserted by the slave.The program gets this value using the `get_response_wait_time` command.
T_{wr}	`waitrequest` is always sampled #1 after the falling edge of `clk`.
T_{idle}	The idle time after a transaction. This time is set by the command `set_command_idle`.

continued...

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

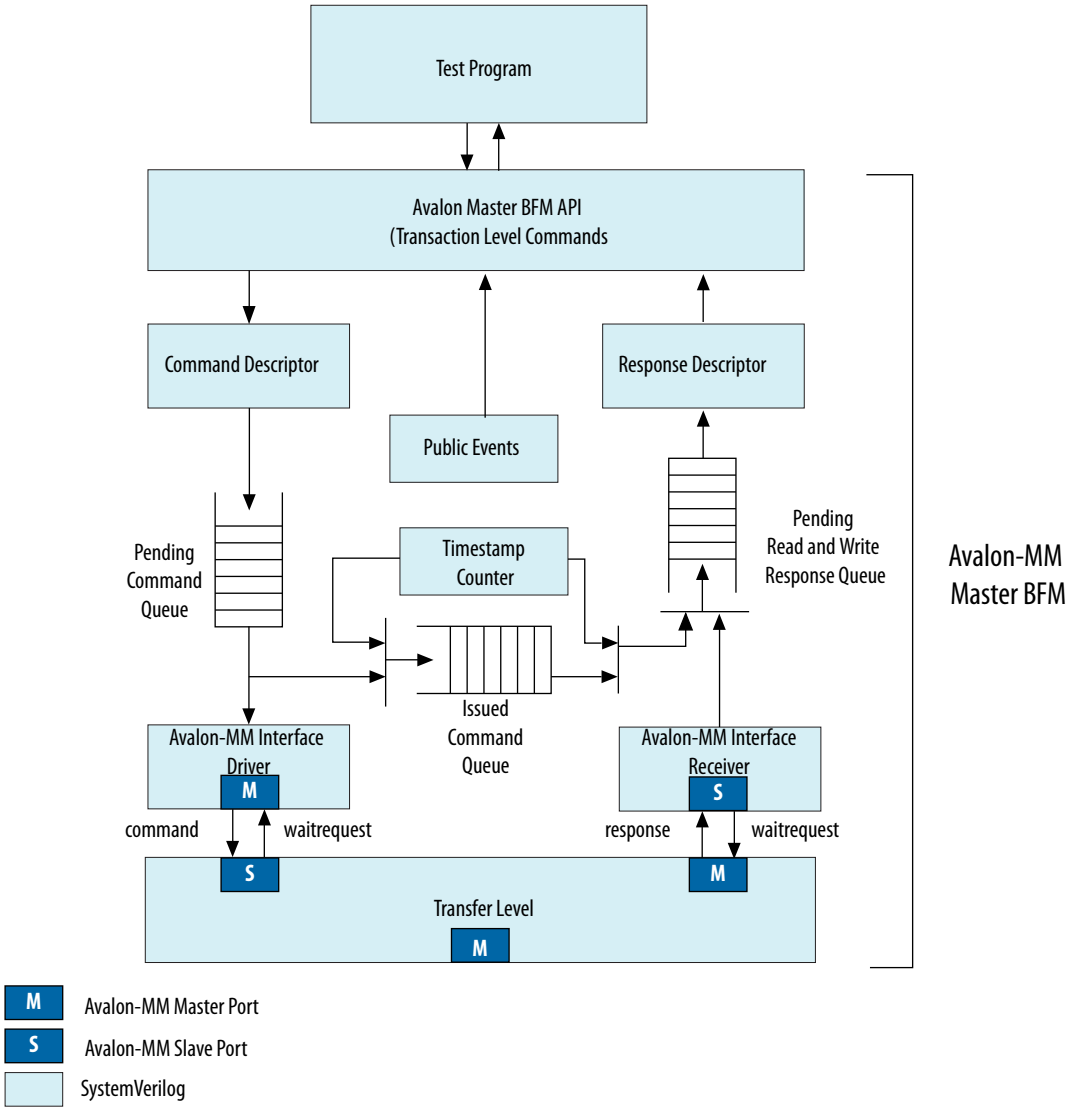
5.2. Block Diagram

The following figure provides 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 call. 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 6. Block Diagram of the Avalon-MM Master BFM



5.3. 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 the following table.

Table 7. Parameters for the Avalon-MM Master BFM

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

continued...

Parameter	Default Value	Legal Values	Description
Port Polarity			
Assert reset high	**On**	**On/Off**	When **On**, `reset` is asserted high.
Assert waitrequest high	**On**	**On/Off**	When **On**, `waitrequest` is asserted high.
Assert read high	**On**	**On/Off**	When **On**, `read` is asserted high.
Assert write high	**On**	**On/Off**	When **On**, `write` is asserted high.
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 incrementing. With a wrapping burst, when the address reaches a burst boundary, it wraps back to the previous burst boundary. Consequently, only the low order bits are 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**	N/A	The maximum number of pending reads that can be queued by the slave.
Fixed read latency (cycles)	**1**	N/A	Sets the read latency for fixed-latency slaves. Not used on interfaces that include the `readdatavalid` signal.
VHDL BFM ID	**0**	0–1023	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.
Timing			
Fixed read wait time (cycles)	**1**	N/A	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**	N/A	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.

5.4. Avalon-MM Master BFM API

all_transactions_complete()

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

5.4.1. event_all_transactions_complete()

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

5.4.2. event_command_issued()

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

5.4.3. event_max_command_queue_size()

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

5.4.4. event_min_command_queue_size()

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

5.4.5. event_read_response_complete()

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

5.4.6. event_response_complete()

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

5.4.7. event_write_response_complete()

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

5.4.8. get_command_issued_queue_size()

Prototype:	`int get_command_issued_queue_size()`
Arguments:	Verilog HDL: None VHDL: `command_issued_queue_size`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`int`
Description:	Queries the issued command queue to determine the number of commands that have been driven to the system interconnect fabric, but not completed.
Language support:	Verilog HDL, VHDL

5.4.9. get_command_pending_queue_size()

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

5.4.10. get_read_response_queue_size()

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

5.4.11. get_response_address()

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

5.4.12. get_response_byte_enable()

Prototype:	`bit [AV_NUMSYMBOLS-1:0] get_response_byte_enable(int index)`
Arguments:	Verilog HDL: `index` VHDL: `response_byte_enable`, `index`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`bit`
Description:	Returns the value of the byte enables in the response descriptor that has been removed from the response queue. Each cycle of a burst response is addressed individually by the specified index.
Language support:	Verilog HDL, VHDL

5.4.13. get_response_burst_size()

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

5.4.14. get_response_data()

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

5.4.15. get_response_latency()

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

5.4.16. get_response_queue_size()

Prototype:	`int get_response_queue_size()`
Arguments:	Verilog HDL: None VHDL: `response_queue_size`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`int`
Description:	Queries the response queue to determine number of response descriptors currently stored in the BFM. This is the number of responses the test program can immediately remove from the response queue for further processing.
Language support:	Verilog HDL, VHDL

5.4.17. get_response_read_id()

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

5.4.18. get_response_read_response()

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

5.4.19. get_response_request()

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

5.4.20. get_response_wait_time()

Prototype:	`int get_response_wait_time(int index)`
Arguments:	Verilog HDL: `index` VHDL: `response_wait_time`, `index`, `bfm_id`, `req_if(bfm_id)`

continued...

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

5.4.21. get_response_write_id()

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

5.4.22. get_write_response_status()

Prototype:	`get_write_response_status()`
Arguments:	Verilog HDL: None VHDL: `write_response_status`, `bfm_id`, `req_if(bfm_id)`
Returns:	`AvalonResponseStatus_t`
Description:	Returns the transaction response status in the write response descriptor that has been popped from the response queue. If the API is called when write response is not enabled, or when it is enabled but not requested, the API returns the default value, i.e. OKAY.
Language support:	Verilog HDL, VHDL

5.4.23. get_write_response_queue_size()

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

5.4.24. get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.

continued...

Returns:	`String`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 14.1 sp1 is encoded as "14.1.1".
Language support:	Verilog HDL

5.4.25. init()

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

5.4.26. pop_response()

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

5.4.27. push_command()

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

5.4.28. set_clken()

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

5.4.29. set_command_address()

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

5.4.30. set_command_arbiterlock()

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

5.4.31. set_command_byte_enable()

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

5.4.32. set_command_burst_count()

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

5.4.33. set_command_burst_size()

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

5.4.34. set_command_data()

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

5.4.35. set_command_debugaccess()

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

5.4.36. set_command_idle()

Prototype:	`void set_command_idle(int idle, int index)`
Arguments:	Verilog HDL: int idle, int index VHDL: `int idle`, `int index`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	Sets idle cycles at the end of each transaction cycle. For read commands, idle cycles are inserted at the end of the command cycle. For burst write commands, idle cycles are inserted at the end of each write data cycle within the burst.
Language support:	Verilog HDL, VHDL

5.4.37. set_command_init_latency()

Prototype:	`void set_command_init_latency(int cycles)`
Arguments:	Verilog HDL: `cycles` VHDL: `cycles`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	Sets the number of cycles to postpone the start of a command.
Language support:	Verilog HDL, VHDL

5.4.38. set_command_lock()

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

5.4.39. set_command_request()

Prototype:	`void set_command_request(Request_t request)`
Arguments:	Verilog HDL: Request_t request VHDL: `Request_t request`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	Sets the transaction type to read or write in the command descriptor. The enumeration type defines `REQ_READ = 0` and `REQ_WRITE = 1`.
Language support:	Verilog HDL, VHDL

5.4.40. set_command_timeout()

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

5.4.41. set_command_transaction_id()

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

5.4.42. set_command_write_response_request()

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

5.4.43. set_max_command_queue_size()

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

5.4.44. set_min_command_queue_size()

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

5.4.45. set_response_timeout()

Prototype:	`void set_response_timeout(int cycles)`
Arguments:	Verilog HDL: int cycles

continued...

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

5.4.46. signal_all_transactions_complete

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

5.4.47. signal_command_issued

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

5.4.48. signal_fatal_error

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

5.4.49. signal_max_command_queue_size

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

5.4.50. signal_min_command_queue_size

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

5.4.51. signal_read_response_complete

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

5.4.52. signal_response_complete

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

5.4.53. signal_write_response_complete

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

6. Avalon-MM Slave BFM

The Avalon-MM Slave BFM implements the slave side of the Avalon-MM interface protocol. The Avalon-MM protocol is a standard memory-mapped protocol. It includes the following functionality:

- Reads and writes typical of simple peripherals
- Reads, writes, burst reads, and burst writes for typical memory devices

This BFM also includes a procedural interface to implement the following functions:

- Monitoring of incoming commands
- Passing incoming commands to the test program
- Accepting response transactions from the test program
- Driving responses

The following figure shows the top-level modules for a testbench. This testbench uses the Avalon-MM Slave BFM to verify an Avalon-MM Master device. In addition to the The example testbench includes the following components:

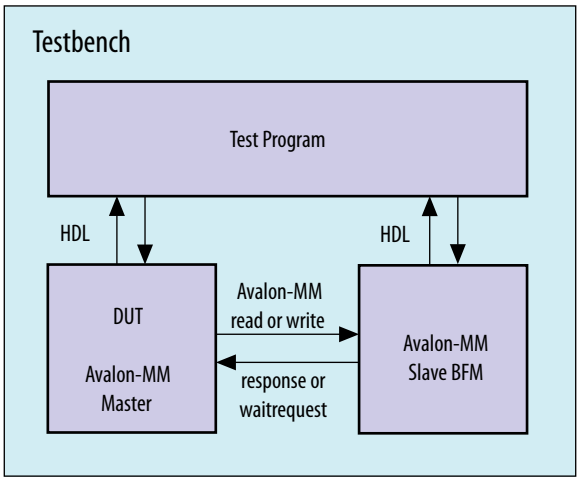
- Avalon-MM Slave BFM
- A test program
- The DUT

The test program is written in HDL. It implements the following functions:

- Programs the Avalon-MM master to issue Avalon-MM transactions
- Programs the Avalon-MM Slave BFM to respond
- Analyzes the results

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

Figure 7. Top-Level Module to Verify an Avalon-MM Master



Related Information

Avalon Interface Specifications

6.1. Timing

The following timing diagram illustrates the sequence of events for an Avalon-MM Slave BFM. It shows the slave BFM responding to interleaved writes and reads when the `readdatavalid` signal is present.

Figure 8. Avalon-MM Slave Responding to Interleaved Write and Read Transactions

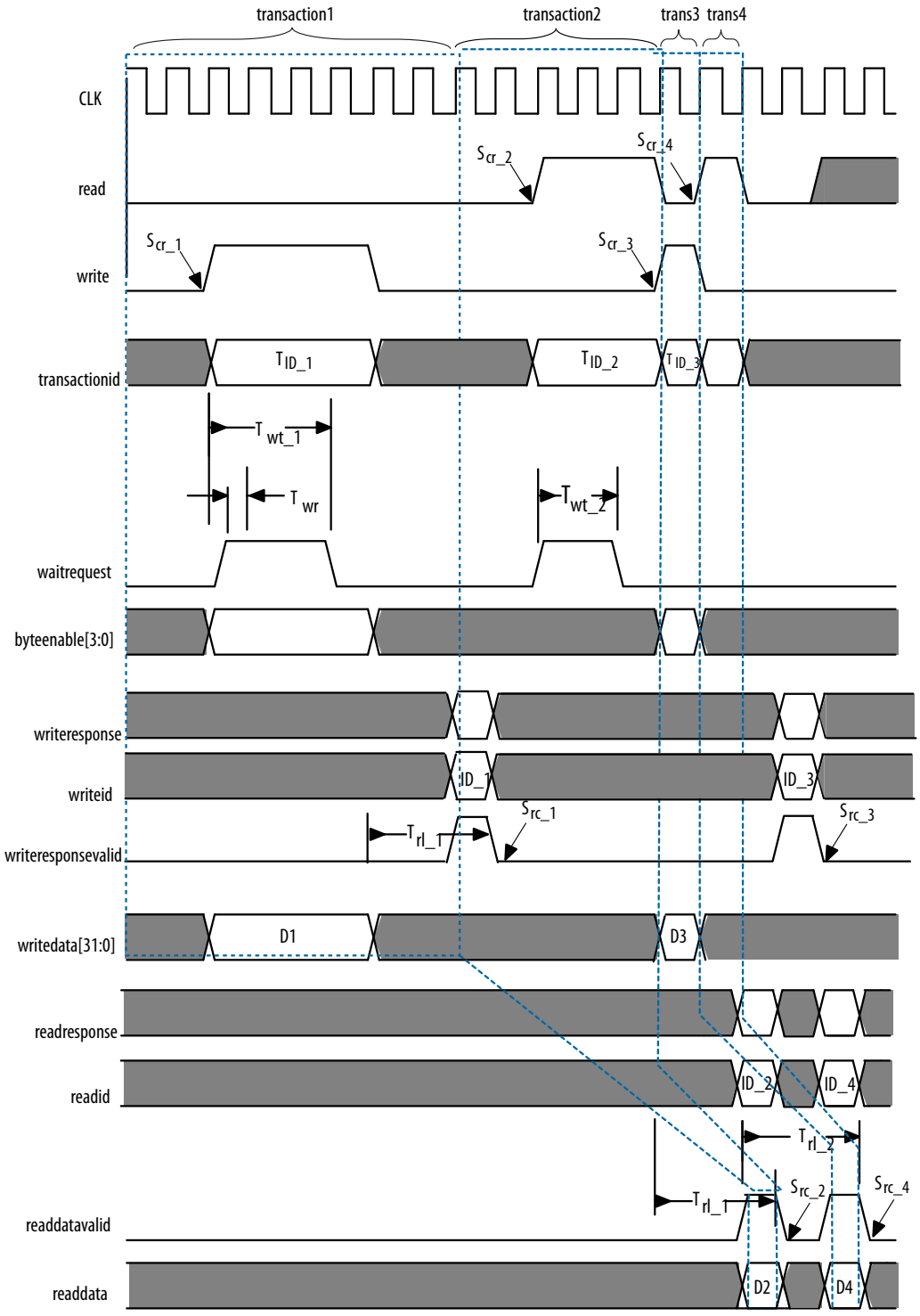


Table 8. Key to Annotations

The following table lists the annotations used in this figure.

Symbol	Description
T_{wt_1}	The response wait time, which is three cycles. The slave sets this value using the `set_interface_wait_time` command.
T_{wr}	`waitrequest` is sampled #1 after the falling edge of `clk`.
T_{wt_2}	The response wait time for the first read, which is 2 cycles. The slave sets this value using the `set_interface_wait_time` command.
S_{cr_1}–S_{cr_2}	Signals when read commands were received. The event name is `signal_command_received`.
T_{rl_1},T_{rl_2}	The response latency for the reads, which is 3 cycles. The slave sets this time using the `set_response_latency` command.
T_{wrl_1}	The write response latency for the first write, which is 3 cycles. This is the time between when the write command is accepted, and the write response is provided by the slave. T
S_{rc_1},S_{rc_3}	Signals write responses. The event name is `signal_response_issued`.
S_{rc_2},S_{rc_4}	Signals read responses. The event name is `signal_response_issued`.
T_{ID_1}–T_{ID_4}	Reference number to identify each read or write transaction.
ID_1, ID_3	Reference number to identify write transactions.
ID_2, ID_4	Reference number to identify read transactions.

Figure 9. Avalon-MM Slave Receiving Write and Read Commands with No readdatavalid Signal

The following timing diagram illustrates the sequence of events for an Avalon-MM Slave BFM. The slave BFM receives a write followed by a read when the `readdatavalid` signal is not present.

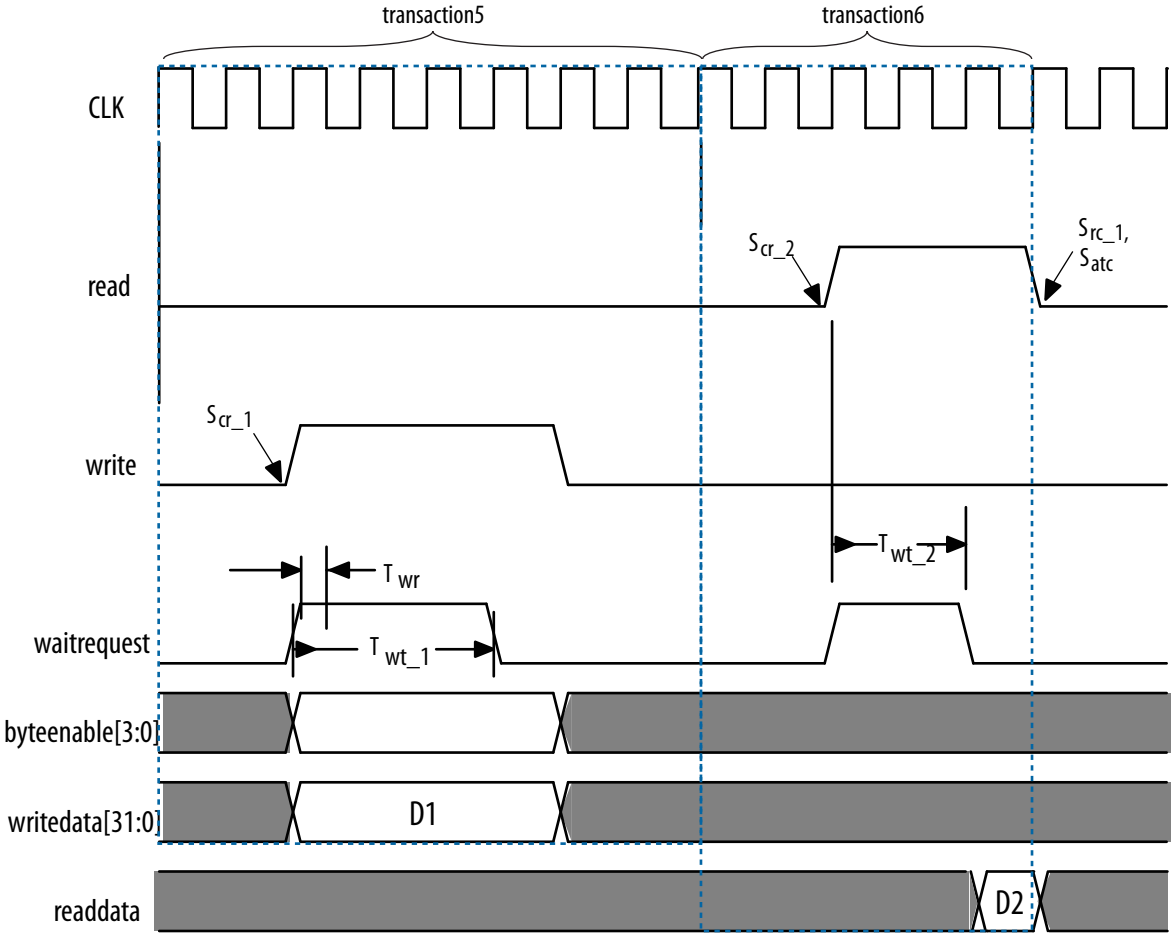


Table 9. Key to Annotations

The following table lists the annotations used in this figure.

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 3 cycles. The master gets this value using the `get_response_wait_time` command.
T_{wt_2}	The response wait time for the first read, which is 2 cycles. The slave sets this value using the `set_interface_wait_time` command.
T_{wr}	`waitrequest` is sampled #1 after the falling edge of `clk`.
T_{rl_1}	The response latency for the first read, which is 0 cycles. The master gets this time using the `get_response_latency` command.

continued...

Symbol	Description
S_{cr_1}, S_{cr_2}	Signals write and read commands. The event name is `signal_command_issued`.
S_{rc_1}	Signals the first read response. The event name is `signal_response_complete`.
S_{atc}	Signals the end of the test. The event name is `signal_all_transactions_complete`

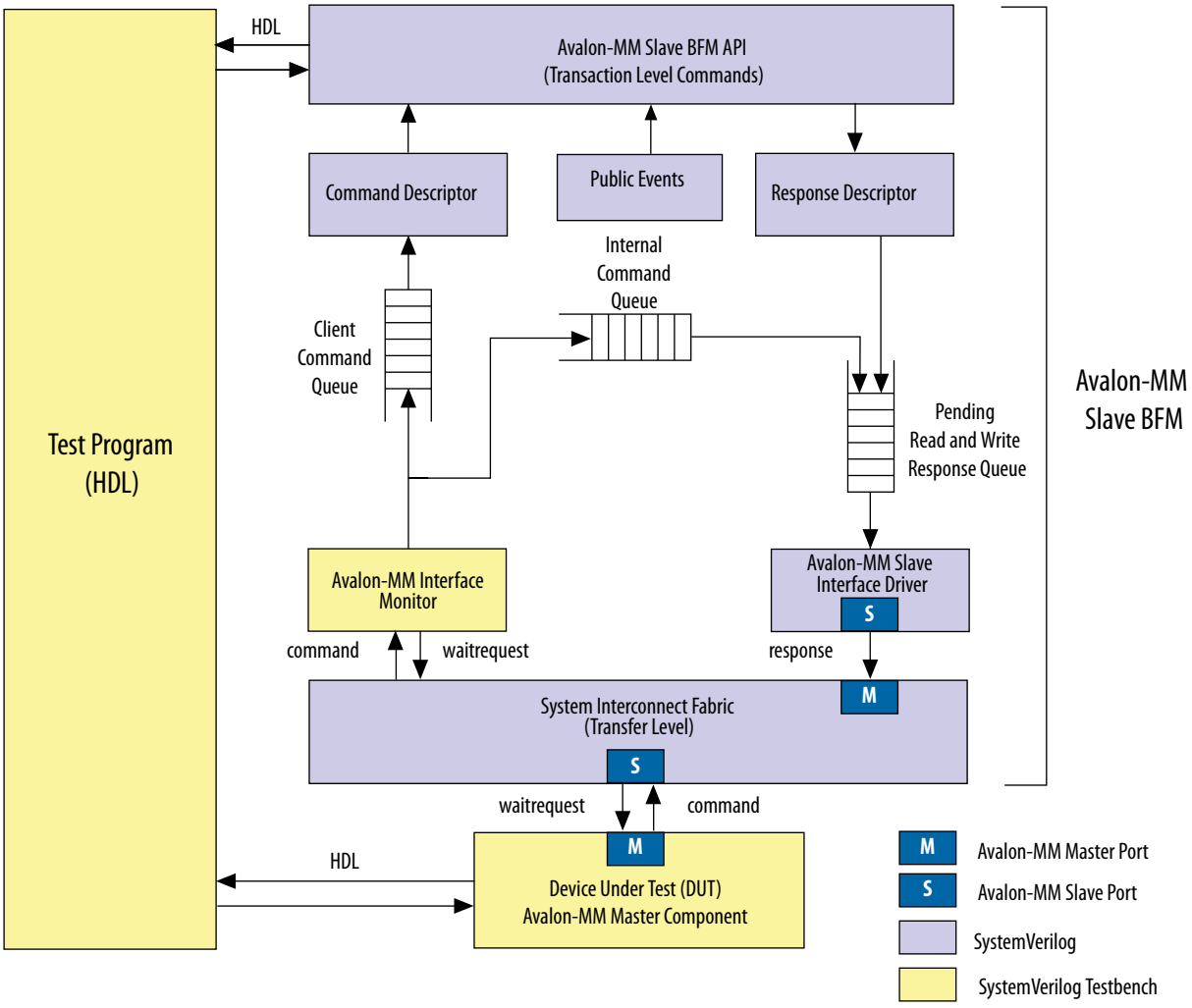
6.2. Block Diagram

The following figure provides 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. Sends completed commands to the Avalon-MM Slave BFM when requested.
- Avalon-MM Interface Monitor—Monitors activity coming from the Avalon-MM Master (DUT). 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 10. Avalon-MM Slave BFM Block Diagram



6.3. Parameters

The Avalon-MM Slave BFM supports the full range of signals defined for the Avalon-MM slave interface. The following table describes parameters you can customize the Avalon-MM slave interface.

Table 10. Parameters for the Avalon-MM Slave BFM

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

continued...

Parameter	Default Value	Legal Values	Description
Assert waitrequest high	**On**	**On/Off**	When **On**, `waitrequest` is asserted high.
Assert read high	**On**	**On/Off**	When **On**, `read` is asserted high.
Assert write high	**On**	**On/Off**	When **On**, `write` is asserted high.
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. Consequently, 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**	N/A	The maximum number of pending reads which can be queued up by the slave.
VHDL BFM ID	**0**	0–1023	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.
Timing			
Fixed read latency (cycles)	**0**	N/A	Sets the read latency for fixed-latency slaves. Not used on interfaces that include the `readdatavalid` signal.
Fixed read wait time (cycles)	**1**	N/A	For slave interfaces that do not use the `waitrequest` signal. The read wait time indicates the number of cycles before the slave responds to a read. The timing is as if the slave asserted `waitrequest` for this number of cycles.
Fixed write wait time (cycles)	**0**	N/A	For slave interfaces that do not use the `waitrequest` signal. The write wait time indicates the number of cycles before the slave accepts a write.
Registered waitrequest	**On**	**On/Off**	Specifies whether to turn on the register stage.
Registered Incoming Signals	**On**	**On/Off**	Specifies whether to register incoming signals.
Interface Address Type			
Set slave interface address type to symbols or words	**WORDS**	**WORDS/ SYMBOLS**	Sets slave interface address type to symbols or words.

6.4. Avalon-MM Slave BFM API

event_error_exceed_max_pending_reads()

Prototype:	`event_error_exceed_max_pending_reads()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`, `req_if`

continued...

Returns:	`void`
Description:	Notifies the testbench that the BFM has more than the maximum pending reads in the pipelined read commands queue waiting to be processed.
Language support:	VHDL

6.4.1. event_command_received()

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

6.4.2. event_response_issued()

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

6.4.3. event_max_response_queue_size()

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

6.4.4. event_min_response_queue_size()

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

6.4.5. get_clken()

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

6.4.6. get_command_address()

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

6.4.7. get_command_arbiterlock()

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

6.4.8. get_command_burst_count()

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

6.4.9. get_command_burst_cycle()

Prototype:	`int get_command_burst_cycle()`
Arguments:	Verilog HDL: None VHDL: `command_burst_cycle`, `bfm_id`, `req_if(bfm_id)`

continued...

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. Each command corresponds to a partially completed burst. This method returns a burst cycle field telling the testbench which burst cycle was active when this descriptor was constructed. This facility enables the testbench to query partially completed write burst operations.The testbench can query the write data word on each burst cycle as it arrives. Consequently, the testbench can begin to process it immediately rather than waiting until the entire burst has been received. This facility means you can implement pipelined write burst processing in the testbench.
Language support:	Verilog HDL, VHDL

6.4.10. get_command_byte_enable()

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

6.4.11. get_command_data()

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

6.4.12. get_command_debugaccess()

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

6.4.13. get_command_queue_size()

Prototype:	`int get_command_queue_size()`
Arguments:	Verilog HDL: None VHDL: `command_queue_size`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`int`
Description:	Queries the command queue to determine number of pending commands.
Language support:	Verilog HDL, VHDL

6.4.14. get_command_lock()

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

6.4.15. get_command_request()

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

6.4.16. get_command_transaction_id()

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

6.4.17. get_command_write_response_request()

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

6.4.18. get_pending_read_latency_cycle()

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

6.4.19. get_pending_write_latency_cycle()

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

6.4.20. get_response_queue_size()

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

6.4.21. vget_slave_bfm_status

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

6.4.22. get_version()

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

6.4.23. init()

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

6.4.24. pop_command()

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

6.4.25. push_response()

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

6.4.26. set_command_transaction_mode()

Prototype:	`void set_command_transaction_mode (int mode);`
Arguments:	Verilog HDL: `mode` VHDL: `mode`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	By default, write burst commands are consolidated into a single command transaction. The single command transaction contains 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 write burst commands yield one command transaction per burst cycle.
Language support:	Verilog HDL, VHDL

6.4.27. set_interface_wait_time()

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

6.4.28. vset_max_response_queue_size()

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

6.4.29. set_min_response_queue_size()

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

6.4.30. set_read_response_id()

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

6.4.31. set_read_response_status()

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

6.4.32. set_response_burst_size()

Prototype:	`void set_response_burst_size(bit [AV_BURSTCOUNT_W-1:0] burst_size).`
Arguments:	Verilog HDL: `burst_size` VHDL: `burst_size`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	Sets the transaction burst count in the response descriptor.
Language support:	Verilog HDL, VHDL

6.4.33. set_response_data()

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

6.4.34. set_response_latency()

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

continued...

	VHDL: `latency`, `index`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	Sets the response latency for read commands. The response is driven `latency` number of cycles after receiving the read command. Designs that set `USE_READDATAVALID` to 1, cannot set the response latency to 0. For read burst commands the following algorithm determines the read latency: • If there are no pending read responses for prior read commands, the response latency is counted from the cycle that the read command is accepted. The read is accepted when the read command is asserted and `waitrequest` is deasserted. • If there are pending responses for prior read commands, the response latency is counted from the cycle in which the read command is presented. The read command is presented when the read command is asserted even if `waitrequest` is asserted.
Language support:	Verilog HDL, VHDL

6.4.35. set_response_request()

Prototype:	`void set_response_request(Request_t request)`
Arguments:	Verilog HDL: `Request_t request` VHDL: `Request_t request`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	Sets the transaction type to read or write in the response descriptor. The enumeration type defines `REQ_READ = 0` and `REQ_WRITE = 1`.
Language support:	Verilog HDL, VHDL

6.4.36. set_response_timeout()

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

6.4.37. set_write_response_id()

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

6.4.38. set_write_response_status()

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

6.4.39. signal_command_received()

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

6.4.40. signal_error_exceed_max_pending_reads

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

6.4.41. signal_max_response_queue_size

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

6.4.42. signal_min_command_queue_size

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

6.4.43. signal_fatal_error

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

6.4.44. signal_response_issued

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

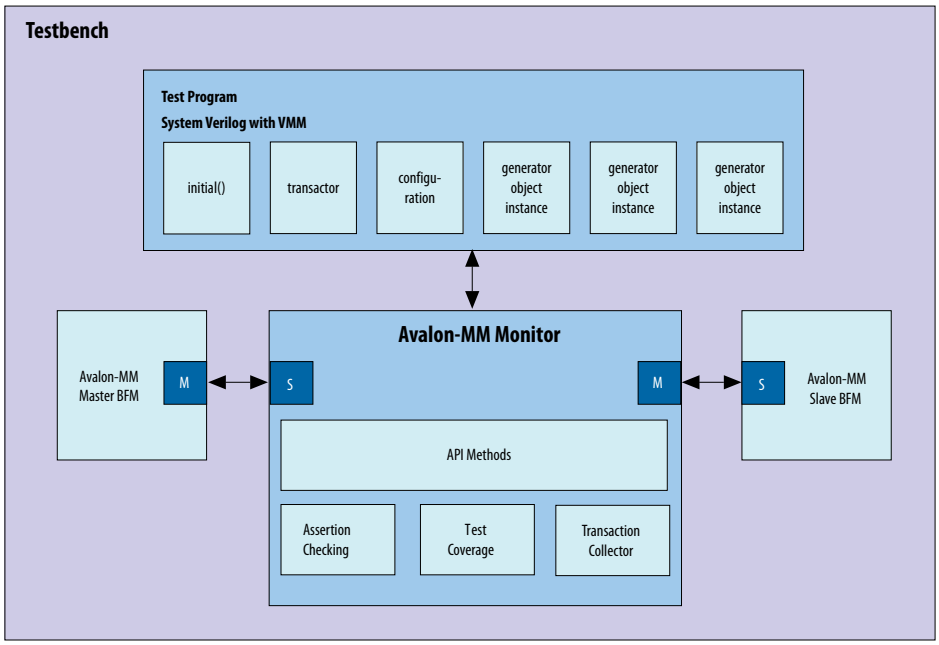
7. Avalon-MM Monitor

The Avalon-MM Monitor verifies Avalon-MM interfaces using SystemVerilog assertions. In addition, it provides test coverage reports. The coverage reports provide the information necessary to determine when your test vectors provide sufficient test coverage of the DUT.

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. However, the assertion checking is disabled.

The following figure 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. 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 11. Testbench Using an Avalon-MM Monitor with Avalon-MM Interfaces



7.1. 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 the following table.

Table 11. Parameters for the Avalon-MM Monitor

Parameter	Default Value	Legal Values	Description
Port Widths			
Address width	**32**	N/A	Address width in bits.
Symbol width	**8**	N/A	Data symbol width in bits. The symbol width should be 8 for byte-oriented interfaces.
Number of symbols	**4**	N/A	Numbers of symbols per word.
Burstcount width	**3**	N/A	The width of the burst count in bits.
Readresponse width	**8**	N/A	Read response signal width in bits.
Writeresponse width	**8**	N/A	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.
Burst Attributes			

continued...

Parameter	Default Value	Legal Values	Description
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. Consequently, only the low order bits are 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**	N/A	Specifies when a timeout occurs if `readdatavalid` is not asserted.
Avalon write timeout (cycles)	**100**	N/A	Specifies when a timeout occurs if a burst write transfer has not completed.
Waitrequest timeout (cycles)	**1024**	N/A	Timeout period for the continuous assertion of `waitrequest`.
Maximum pending reads	**1**	N/A	Specifies the maximum number of pipelined reads that can be pending.
Fixed read latency (cycles)	**0**	N/A	Sets the read latency for fixed-latency slaves. Not used on interfaces that include the `readdatavalid` signal.
Maximum read latency (cycles)	**100**	N/A	Specifies the maximum read latency in cycle for test coverage function
Maximum waitrequest read cycles (for coverage)	**100**	N/A	Specifies the maximum wait time allowed for read cycle for coverage.
Maximum waitrequest write cycles (for coverage)	**100**	N/A	Maximum wait time allowed for write cycle for coverage.
Maximum continuous read (cycles)	**5**	N/A	Maximum continuous read time allowed for coverage.
Maximum continuous write (cycles)	**5**	N/A	Maximum continuous write time allowed for coverage.
Maximum continuous waitrequest (cycles)	**5**	N/A	Maximum continuous wait request time allowed for coverage.
Maximum continuous readdatavalid (cycles)	**5**	N/A	Maximum continuous readdatavalid time allowed for coverage.
VHDL BFM ID	**0**	0–1023	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.
Timing			
Fixed read wait time (cycles)	**1**	N/A	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**	N/A	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.

7.2. Avalon-MM Monitor Assertion Checking API

Assertion checking uses the `enable_waitrequest_timeout` method to verify that `waitrequest` is asserted for fewer cycles than the `waitrequest` timeout period. If the timeout period is violated, an error message displays on the simulation console. 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. Protocol errors are typically injected 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.

7.2.1. set_enable_a_address_align_with_data_width()

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

7.2.2. set_enable_a_beginbursttransfer_exist()

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

7.2.3. set_enable_a_beginbursttransfer_legal()

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

7.2.4. set_enable_a_beginbursttransfer_single_cycle()

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

7.2.5. set_enable_a_begintransfer_exist()

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

7.2.6. set_enable_a_begintransfer_legal()

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

7.2.7. set_enable_a_begintransfer_single_cycle()

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

7.2.8. set_enable_a_burst_legal()

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

7.2.9. set_enable_a_byteenable_legal()

Prototype:	`set_enable_a_byteenable_legal()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables an assertion that ensures the `byteenable` value is legal value. Disabled when `byteenable` is not supported. For more information about legal byte enables, refer to the *Avalon Interface Specifications*.
Language support:	Verilog HDL

Related Information

Avalon Interface Specifications

7.2.10. set_enable_a_constant_during_burst()

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

7.2.11. set_enable_a_constant_during_clk_disabled()

Prototype:	`set_enable_a_constant_during_clk_disabled()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.

continued...

Returns:	`void`
Description:	Enables an assertion that ensures that all signals are held constant if `clken` is deasserted.
Language support:	Verilog HDL

7.2.12. set_enable_a_constant_during_waitrequest()

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

7.2.13. set_enable_a_exclusive_read_write()

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

7.2.14. set_enable_a_half_cycle_reset_legal()

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

7.2.15. set_enable_a_less_than_burstcount_max_size()

Prototype:	`set_enable_a_less_than_burstcount_max_size()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.

continued...

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

7.2.16. set_enable_a_less_than_maximumpendingreadtransactions()

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

7.2.17. set_enable_a_no_readdatavalid_during_reset()

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

7.2.18. set_enable_a_no_read_during_reset()

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

7.2.19. set_enable_a_no_write_during_reset()

Prototype:	`set_enable_a_no_write_during_reset()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.

continued...

Returns:	`void`
Description:	Enables an assertion that ensures `write` is deasserted if `reset` is asserted. Disabled when `write` is not supported.
Language support:	Verilog HDL

7.2.20. set_enable_a_readid_sequence()

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

7.2.21. set_enable_a_read_response_sequence()

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

7.2.22. set_enable_a_read_response_timeout()

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

7.2.23. set_enable_a_register_incoming_signals()

Prototype:	`set_enable_a_register_incoming_signals()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.

continued...

Returns:	`void`
Description:	Enables an assertion that ensures `waitrequest` is asserted at all times and deasserts a single clock cycle after a read or write transaction.
Language support:	Verilog HDL

7.2.24. set_enable_a_waitrequest_during_reset()

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

7.2.25. set_enable_a_waitrequest_timeout()

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

7.2.26. set_enable_a_write_burst_timeout()

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

7.2.27. set_enable_a_writeid_sequence()

Prototype:	`set_enable_a_writeid_sequence()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.

continued...

Returns:	`void`
Description:	Enables an assertion that verifies if the `writeid` sequence follows the sequence of the `transactionid`.
Language support:	Verilog HDL

7.2.28. 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
```

7.2.28.1. set_enable_c_b2b_read_read()

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

7.2.28.2. set_enable_c_b2b_read_write()

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

7.2.28.3. set_enable_c_b2b_write_read()

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

7.2.28.4. set_enable_c_b2b_write_write()

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

7.2.28.5. set_enable_c_continuous_read()

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

7.2.28.6. set_enable_c_continuous_readdatavalid()

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

7.2.28.7. set_enable_c_continuous_waitrequest()

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

7.2.28.8. set_enable_c_continuous_waitrequest_from_idle_to_read()

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

7.2.28.9. set_enable_c_continuous_waitrequest_from_idle_to_write()

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

7.2.28.10. set_enable_c_continuous_write()

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

7.2.28.11. set_enable_c_idle_before_transaction()

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

7.2.28.12. set_enable_c_idle_in_read_response()

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

7.2.28.13. set_enable_c_idle_in_write_burst()

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

7.2.28.14. set_enable_c_pending_read()

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

7.2.28.15. set_enable_c_read()

Prototype:	`set_enable_c_read()`
Arguments:	Verilog HDL: `Boolean`

continued...

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

7.2.28.16. set_enable_c_read_after_reset()

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

7.2.28.17. set_enable_c_read_burstcount()

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

7.2.28.18. set_enable_c_read_byteenable()

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

7.2.28.19. set_enable_c_read_latency()

Prototype:	`set_enable_c_read_latency()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.

continued...

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

7.2.28.20. set_enable_c_read_response()

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

7.2.28.21. set_enable_c_waitrequest_in_write_burst()

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

7.2.28.22. set_enable_c_waitrequested_read()

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

7.2.28.23. set_enable_c_waitrequest_without_command()

Prototype:	`set_enable_c_waitrequest_without_command()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.

continued...

Returns:	`void`
Description:	Enables a coverage group to verify that no command is asserted between the time when waitrequest is asserted until waitrequest is deasserted.
Language support:	Verilog HDL

7.2.28.24. set_enable_c_waitrequested_write()

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

7.2.28.25. set_enable_c_write()

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

7.2.28.26. set_enable_c_write_with_and_without_writeresponserequest()

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

7.2.28.27. set_enable_c_write_after_reset()

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

7.2.28.28. set_enable_c_write_burstcount()

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

7.2.28.29. set_enable_c_write_byteenable()

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

7.2.28.30. set_enable_c_write_response()

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

7.2.29. Transaction Monitoring

The transaction collector module monitors transactions. The transaction collector performs the following functions:

- Collects the transactions
- Encapsulates transactions into descriptors
- Inserts the transactions into a queue.

The API provides functions to query the transactions in the 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.

7.2.29.1. event_transaction_fifo_threshold()

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

7.2.29.2. event_transaction_fifo_overflow()

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

7.2.29.3. event_command_received()

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

7.2.29.4. event_read_response_complete()

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

7.2.29.5. event_write_response_complete()

Prototype:	`event_write_response_complete()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`

continued...

Returns:	`void`
Description:	Notifies the testbench that a write response was received.
Language support:	VHDL

7.2.29.6. event_response_complete()

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

7.2.29.7. get_clken()

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

7.2.29.8. get_version()

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

7.2.29.9. get_command_address()

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

7.2.29.10. get_command_arbiterlock()

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

7.2.29.11. get_command_burst_count()

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

7.2.29.12. get_command_burst_cycle()

Prototype:	`int get_command_burst_cycle()`
Arguments:	Verilog HDL: None VHDL: `command_burst_cycle`, `bfm_id`, `req_if(bfm_id)`
Returns:	`Int`
Description:	The slave BFM receives and processes write burst commands as a discrete sequence. 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 specifying the burst cycle that was active when this descriptor was constructed. This facility enables the testbench to query partially completed write burst operations. The testbench can query the write data word on each burst cycle as it arrives. The testbench can begin to process it immediately. The testbench does not have to wait until the entire burst has been received. Consequently, it is possible to perform pipelined write burst processing in the testbench.
Language support:	Verilog HDL, VHDL

7.2.29.13. get_command_byte_enable()

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

7.2.29.14. get_command_data()

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

7.2.29.15. get_command_debugaccess()

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

7.2.29.16. get_command_issued_queue_size()

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

7.2.29.17. get_command_queue_size()

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

7.2.29.18. get_command_lock()

Prototype:	`bit get_command_lock()`
Arguments:	Verilog HDL: None VHDL: `command_lock`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`bit`
Description:	Queries the received command descriptor for the transaction lock.
Language support:	Verilog HDL, VHDL

7.2.29.19. get_command_request()

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

7.2.29.20. get_command_transaction_id()

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

7.2.29.21. get_command_write_response_request()

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

7.2.29.22. get_read_response_queue_size()

Prototype:	`int get_read_response_queue_size()`
Arguments:	Verilog HDL: None VHDL: `read_response_queue_size`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`int`
Description:	Queries the read response queue to determine number of response descriptors currently stored in the BFM. This is the number of responses the test program can immediately remove from the response queue for further processing.
Language support:	Verilog HDL, VHDL

7.2.29.23. get_response_address()

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

7.2.29.24. get_response_byte_enable()

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

7.2.29.25. get_response_burst_size()

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

7.2.29.26. get_response_data()

Prototype:	`bit [AV_DATA_W-1:0] get_response_data(int index)`
Arguments:	Verilog HDL: `index` VHDL: `response_data`, `index`, `bfm_id`, `req_if(bfm_id)`

continued...

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

7.2.29.27. get_response_latency()

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

7.2.29.28. get_response_queue_size()

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

7.2.29.29. get_response_read_id()

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

7.2.29.30. get_response_read_response()

Prototype:	`AvalonReadResponse_t get_response_read_response(int index)`
Arguments:	Verilog HDL: `int index` VHDL: `response_read_response`, `int index`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`AvalonReadResponse_t`
Description:	Returns the transaction read status in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

7.2.29.31. get_response_request()

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

7.2.29.32. get_response_wait_time()

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

7.2.29.33. get_response_write_id()

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

7.2.29.34. get_response_write_response()

Prototype:	`AvalonWriteResponse_t get_response_write_response(int index)`
Arguments:	Verilog HDL: `index` VHDL: `response_write_response, index, bfm_id, req_if(bfm_id)`

continued...

Returns:	`AvalonWriteResponse_t`
Description:	Returns the transaction write status in the response descriptor that has been removed from the response queue.
Language support:	Verilog HDL, VHDL

7.2.29.35. get_transaction_fifo_max()

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

7.2.29.36. get_transaction_fifo_threshold()

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

7.2.29.37. get_write_response_queue_size()

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

7.2.29.38. init()

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

7.2.29.39. pop_command()

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

7.2.29.40. pop_response()

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

7.2.29.41. set_command_transaction_mode()

Prototype:	`set_command_transaction_mode()`
Arguments:	Verilog HDL: `int mode` VHDL: `int mode`, `bfm_id`, `req_if(bfm_id)`
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. Write burst commands yield one command transaction per burst cycle.
Language support:	Verilog HDL, VHDL

7.2.29.42. set_transaction_fifo_max()

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

7.2.29.43. set_transaction_fifo_threshold()

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

7.2.29.44. signal_command_received

Prototype:	`signal_command_received`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Notifies the testbench that a command was detected on the Avalon port. When this event is received, the testbench responds with a `set_interface_wait_time` call. This call dynamically backpressures the driving Avalon master.
Language support:	Verilog HDL

7.2.29.45. signal_fatal_error

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

7.2.29.46. signal_read_response_complete

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

7.2.29.47. signal_response_complete

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

7.2.29.48. signal_transaction_fifo_overflow

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

7.2.29.49. signal_transaction_fifo_threshold

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

7.2.29.50. signal_write_response_complete

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

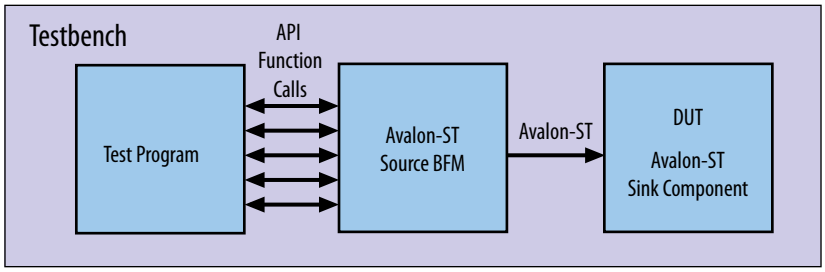
8. Avalon-ST Source BFM

The Avalon-ST Source BFM implements the Avalon-ST interface protocol. The Avalon-ST protocol 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.

The following figure shows the top-level modules for a testbench. This testbench uses an Avalon-ST Source BFM to verify an Avalon-ST sink component. In addition to the Avalon-ST Source BFM component, the testbench typically includes a test program and the DUT.

Note: The BFMs allow illegal transactions so that you can test the error-handling functionality of your DUT. Consequently, you cannot rely on the BFMs to guarantee protocol compliance. The Avalon Monitor components verify protocol compliance.

Figure 12. Top-Level Module to Verify an Avalon-ST Sink Device



Related Information

Avalon Interface Specifications

8.1. Timing

The following figure 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.

Note: 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 13. Avalon-ST Source Sending Data to a Sink

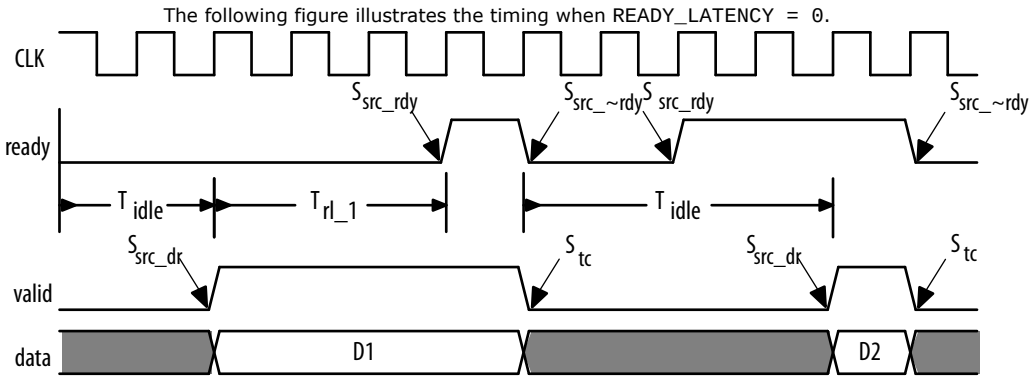


Table 12. Key to Annotations

The following table explains the annotations used in the figure.

Symbol	Description
T_{idle}	The idle time before a transactions. This time is set by the command `set_transaction_idles`.
T_{rl_1}	The response latency for the first source to sink transaction, which is 3 cycles. The source gets this time using the `get_response_latency` command.
S_{src_dr}	Signals that the source is driving valid data. The event name is `signal_src_driving_transaction`.
S_{src_rdy}	Signals the source has received the assertion of `ready` from the sink. The event name is `signal_src_ready`.
S_{tc}	Signals the first transaction is complete. The event name is `signal_src_transaction_complete`.
$S_{src\_\sim rdy}$	Signals the source has received the deassertion of `ready` from the sink. The event name is `signal_src_not_ready`.

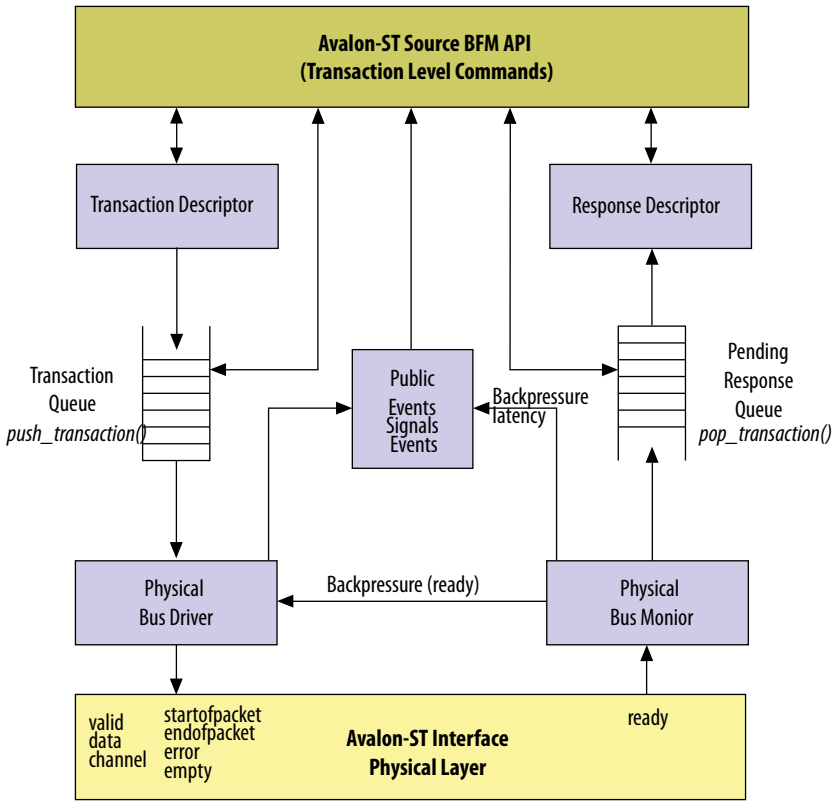
8.2. Block Diagram

The following figure provides a block diagram of the Avalon-ST Source BFM. 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 14. Block Diagram of the Avalon-ST Source BFM



8.3. Parameters

The Avalon-ST Source BFM supports all the of the signals defined for the Avalon-ST source interface. The following table lists the parameters to customize the Avalon-ST Source interface.

Table 13. Parameters for the Avalon-ST Source BFM

Parameter	Default Value	Legal Values	Description
Port Enables			
Include the signals to support packets	**Off**	**On/Off**	When On, the interface includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use the channel port	**Off**	**On/Off**	When **On**, the interface includes `channel` pin or pins.
Use the error port	**Off**	**On/Off**	When **On**, the interface includes `error` pin or pins.
Use the ready port	**On**	**On/Off**	When **On**, the interface includes a `ready` pin.

continued...

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

Related Information

Avalon Interface Specification

8.4. Avalon-ST Source API

event_max_transaction_queue_size()

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

8.4.1. event_min_transaction_queue_size()

Prototype:	`event_min_transaction_queue_size()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`

continued...

Returns:	`void`
Description:	Notifies the testbench that the transaction queue is below the minimum level.
Language support:	VHDL

8.4.2. event_response_done()

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

8.4.3. event_src_driving_transaction()

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

8.4.4. event_src_not_ready()

Prototype:	`event_src_not_ready()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`
Returns:	`void`
Description:	Notifies the testbench that the ready signal was deasserted.
Language support:	VHDL

8.4.5. event_src_ready()

Prototype:	`event_src_ready()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`
Returns:	`void`
Description:	Notifies the testbench that the ready signal was asserted.
Language support:	VHDL

8.4.6. event_src_transaction_complete()

Prototype:	`event_src_transaction_complete()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`
Returns:	`void`
Description:	Notifies the testbench that all transactions were accepted.
Language support:	VHDL

8.4.7. get_response_latency()

Prototype:	`get_response_latency()`
Arguments:	Verilog HDL: None VHDL: `response_latency`, `bfm_id`, `req_if(bfm_id)`
Returns:	`int`
Description:	Returns the response latency in cycles due to back pressure for the most recently removed transaction.
Language support:	Verilog HDL, VHDL

8.4.8. get_response_queue_size()

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

8.4.9. get_src_ready()

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

8.4.10. get_src_transaction_complete()

Prototype:	`get_src_transaction_complete()`
Arguments:	Verilog HDL: None VHDL: `src_transaction_complete`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`bit`
Description:	Returns the transaction complete status.
Language support:	Verilog HDL, VHDL

8.4.11. get_transaction_queue_size()

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

8.4.12. get_version()

Prototype:	`get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`String`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 13.1 SP1 is encoded as "13.1.1".
Language support:	Verilog HDL

8.4.13. init()

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

8.4.14. pop_response()

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

8.4.15. push_transaction()

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

8.4.16. set_max_transaction_queue_size()

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

8.4.17. set_min_transaction_queue_size()

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

8.4.18. set_response_timeout()

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

8.4.19. set_transaction_channel()

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

8.4.20. set_transaction_data()

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

8.4.21. set_transaction_idles()

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

8.4.22. set_transaction_eop()

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

8.4.23. set_transaction_empty()

Prototype:	`set_transaction_empty(STEmpty_t empty)`
Arguments:	Verilog HDL: `empty` VHDL: `empty`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`void`
Description:	Sets the out-going transaction empty value.
Language support:	Verilog HDL, VHDL

8.4.24. set_transaction_error()

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

8.4.25. set_transaction_sop()

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

8.4.26. signal_fatal_error

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

8.4.27. signal_max_transaction_queue_size

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

8.4.28. signal_min_transaction_queue_size

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

8.4.29. signal_response_done

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

8.4.30. signal_src_driving_transaction

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

8.4.31. signal_src_not_ready

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

8.4.32. signal_src_ready

Prototype:	`signal_src_ready`
Arguments:	Verilog HDL: None VHDL: N.A.

continued...

Returns:	`void`
Description:	Signals that the `ready` signal is asserted.
Language support:	Verilog HDL

8.4.33. signal_src_transaction_complete

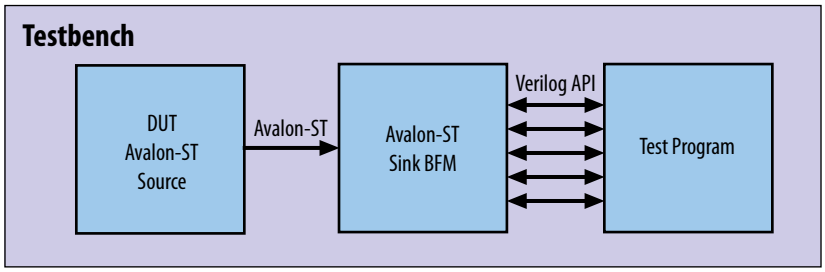
Prototype:	`signal_src_transaction_complete`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that all pending transactions have completed.
Language support:	Verilog HDL

9. Avalon-ST Sink BFM

The Avalon-ST Sink BFM implements the Avalon-ST interface protocol. The Avalon-ST protocol 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. The following figure 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 Avalon-ST Sink BFM component, the testbench includes a test program and the DUT.

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

Figure 15. Top-Level Module to Verify an Avalon-ST Source Device



Related Information

Avalon Interface Specifications

9.1. Timing

The following figure illustrates the timing for an Avalon-ST Sink BFM 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 16. Avalon-ST Source and Sink Timing

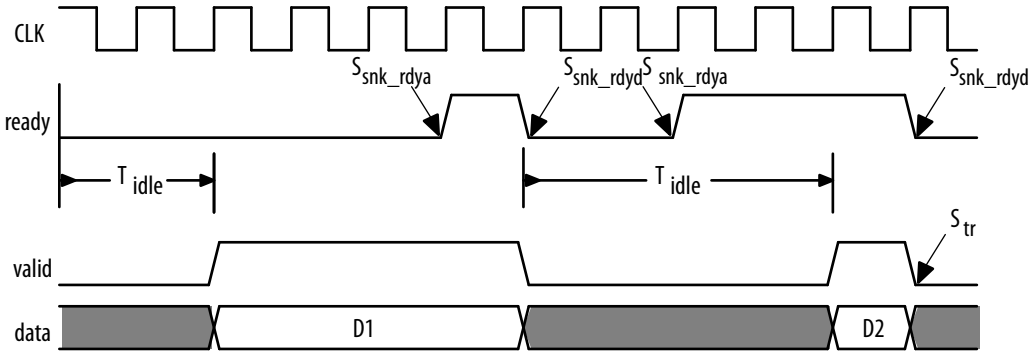


Table 14. Key to Annotations

The following table describes the annotations used in the figure above.

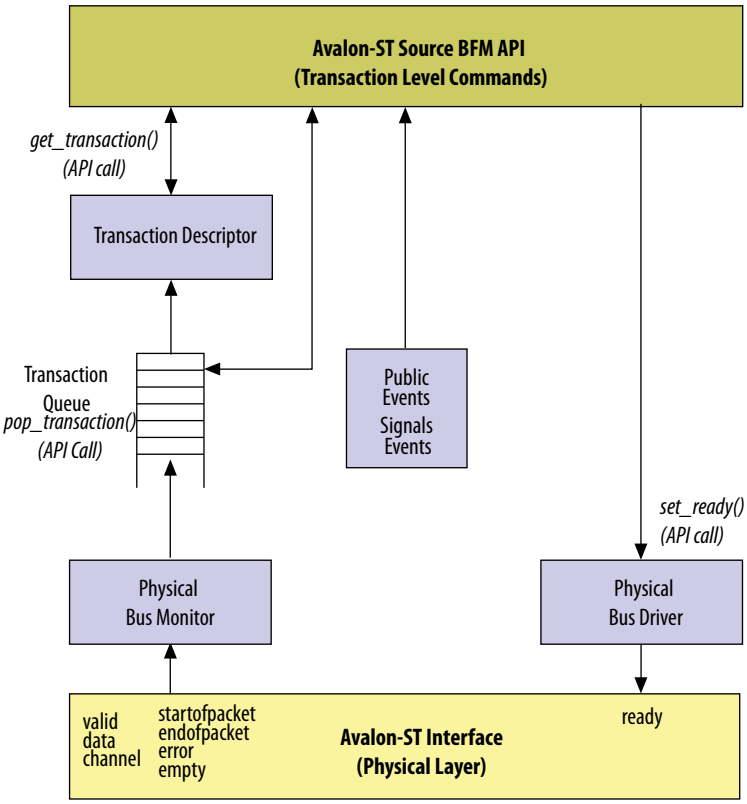
Symbol	Description
T_{idle}	The idle time between transactions. This time is reported by the command `get_transaction_idles`.
S_{snk_rdya}	Signals the sink has asserted `ready`. The event name is `signal_snk_ready_assert`.
S_{tr}	Signals the transaction has been received and queued. The event name is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

9.2. Block Diagram

The following figure 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 17. Block Diagram of the Avalon-ST Sink BFM



9.3. Parameters

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

Table 15. Parameters for the Avalon-ST Sink BFM

Parameter	Default Value	Legal Values	Description
Port Enables			
Include the signals to support packets	**Off**	**On/Off**	When On, the interface includes the `startofpacket`, `endofpacket`, and `empty` signals.

continued...

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

9.4. Application Program Interface

event_transaction_received()

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

9.4.1. event_sink_ready_assert()

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

9.4.2. event_sink_ready_deassert()

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

9.4.3. get_transaction_channel()

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

9.4.4. get_transaction_data()

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

9.4.5. get_transaction_idles()

Prototype:	`get_transaction_idles()`
Arguments:	Verilog HDL: None VHDL: `transaction_idles`,`bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`bit[31:0]`
Description:	Returns the number of idle cycles in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

9.4.6. get_transaction_eop()

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

9.4.7. get_transaction_empty()

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

9.4.8. get_transaction_error()

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

9.4.9. get_transaction_queue_size()

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

9.4.10. get_transaction_sop()

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

9.4.11. get_version()

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

9.4.12. init()

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

9.4.13. pop_transaction()

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

9.4.14. set_ready()

Prototype:	`set_ready()`
Arguments:	Verilog HDL: `read_bit` VHDL: `read_bit`, `bfm_id`, `req_if(bfm_id)`

continued...

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

9.4.15. signal_fatal_error

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

9.4.16. signal_sink_ready_assert

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

9.4.17. signal_sink_ready_deassert

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

9.4.18. signal_transaction_received

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

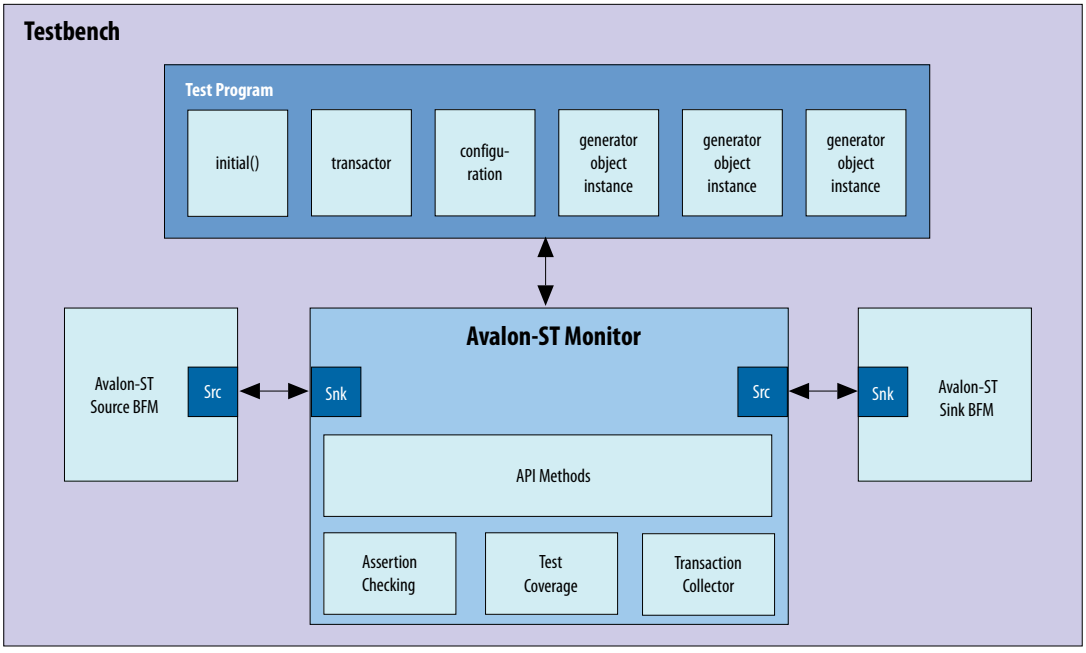
10. Avalon-ST Monitor

The Avalon-ST Monitor verifies Avalon-ST interfaces using SystemVerilog assertions. In addition, it provides test coverage reports. The coverage reports provide the information necessary to determine when your test vectors provide sufficient test coverage of the DUT.

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.

The following figure 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. 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 18. Testbench Using an Avalon-ST Monitor with Avalon-ST Interfaces



10.1. 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 the following table.

Table 16. Parameters for the Avalon-ST Monitor BFM

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

10.2. Avalon-ST Monitor Assertion Checking API

Assertion checking methods enable and disable protocol assertions 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. In some circumstances, you may want to disable assertion checking. For example, when injecting protocol errors to test error handling, you may want to disable assertion checking.

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.

10.2.1. set_enable_a_empty_legal()

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

10.2.2. set_enable_a_less_than_max_channel()

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

10.2.3. set_enable_a_no_data_outside_packet()

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

10.2.4. set_enable_a_non_missing_endofpacket()

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

10.2.5. set_enable_a_non_missing_startofpacket()

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

10.2.6. set_enable_a_valid_legal()

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

10.2.7. 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 - Intel FPGA Edition software, use the following command:

```
run –all
coverage report –details –file report.rpt
```

10.2.7.1. set_enable_c_all_idle_beats()

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

10.2.7.2. set_enable_c_all_valid_beats()

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

10.2.7.3. set_enable_c_b2b_data_different_channel()

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

10.2.7.4. set_enable_c_b2b_data_same_channel()

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

10.2.7.5. set_enable_c_b2b_packet_different_channel()

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

10.2.7.6. set_enable_c_b2b_packet_in_different_transaction()

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

10.2.7.7. set_enable_c_b2b_packet_same_channel()

Prototype:	`set_enable_c_b2b_packet_same_channel()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for back-to-back packet transmission for the same channel. It is disabled when packet transmission or channels are not supported.
Language support:	Verilog HDL

10.2.7.8. set_enable_c_b2b_packet_within_single_cycle()

Prototype:	`set_enable_c_b2b_packet_within_single_cycle()`
Arguments:	Verilog HDL: `Boolean` VHDL: N.A.
Returns:	`void`
Description:	Enables a coverage point that ensures test coverage for back-to-back packet transmission within a single cycle. It is disabled when packet transmission or channels are not supported.
Language support:	Verilog HDL

10.2.7.9. set_enable_c_channel_change_in_packet()

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

10.2.7.10. set_enable_c_empty()

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

10.2.7.11. set_enable_c_error()

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

10.2.7.12. set_enable_c_error_in_middle_of_packet()

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

10.2.7.13. set_enable_c_idle_beat_between_packet()

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

10.2.7.14. set_enable_c_multiple_packet_per_cycle()

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

10.2.7.15. set_enable_c_non_valid_ready()

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

10.2.7.16. set_enable_c_non_valid_non_ready()

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

10.2.7.17. set_enable_c_packet()

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

10.2.7.18. set_enable_c_packet_no_idles_no_back_pressure()

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

10.2.7.19. set_enable_c_packet_size()

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

10.2.7.20. set_enable_c_packet_with_back_pressure()

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

10.2.7.21. set_enable_c_packet_with_idles()

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

10.2.7.22. set_enable_c_partial_valid_beats()

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

10.2.7.23. set_enable_c_single_packet_per_cycle()

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

10.2.7.24. set_enable_c_transfer()

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

10.2.7.25. set_enable_c_transaction_after_reset()

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

10.2.7.26. set_enable_c_valid_non_ready()

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

10.2.8. Transaction Monitoring

The transaction collector module monitors transactions. The transaction collector collects the transactions, encapsulates them into descriptors, and inserts the transactions into the queue.

The API provides functions to query the transactions in the queue and disposes them as they are processed. By default, the transaction collector module is disabled. You must define the `ENABLE_ALTERA_AVALON_TRANSACTION_RECORDING` Verilog macro to enable this feature. This macro is required to ensure backward compatibility and to avoid breaking existing test cases.

event_transaction_received()

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

10.2.8.1. event_transaction_fifo_threshold()

Prototype:	`event_transaction_fifo_threshold()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`
	continued...

Returns:	`void`
Description:	Notifies the testbench that the transaction FIFO exceeds the threshold level.
Language support:	VHDL

10.2.8.2. event_transaction_fifo_overflow()

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

10.2.8.3. get_transaction_channel()

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

10.2.8.4. get_transaction_data()

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

10.2.8.5. get_transaction_empty()

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

10.2.8.6. get_transaction_eop()

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

10.2.8.7. get_transaction_error()

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

10.2.8.8. get_transaction_fifo_max()

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

10.2.8.9. get_transaction_fifo_threshold()

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

10.2.8.10. get_transaction_idles()

Prototype:	`get_transaction_idles()`
Arguments:	Verilog HDL: None VHDL: `transaction_idles`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`bit[31:0]`.
Description:	Returns the number of idle cycles in the most recently removed transaction.
Language support:	Verilog HDL, VHDL

10.2.8.11. get_transaction_queue_size()

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

10.2.8.12. get_transaction_sop()

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

10.2.8.13. get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`String`.
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 13.1 sp1 is encoded as "13.1.1".
Language support:	Verilog HDL

10.2.8.14. pop_transaction()

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

10.2.8.15. set_transaction_fifo_max()

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

10.2.8.16. set_transaction_fifo_threshold()

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

10.2.8.17. signal_fatal_error

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

10.2.8.18. signal_transaction_fifo_overflow

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

10.2.8.19. signal_transaction_fifo_threshold

Prototype:	`signal_transaction_fifo_threshold`
Arguments:	Verilog HDL: None VHDL: N.A.

continued...

Returns:	`void`
Description:	Notifies the testbench that the transaction FIFO threshold level has exceeded.
Language support:	Verilog HDL

10.2.8.20. signal_transaction_received

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

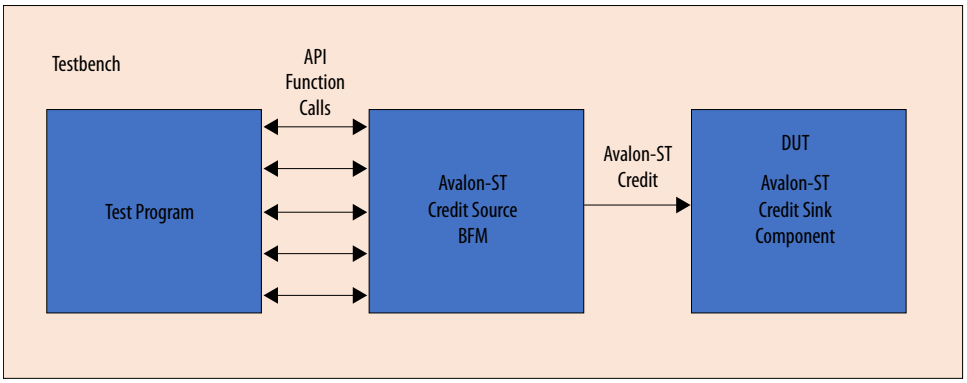
11. Avalon Streaming Credit Source BFM

The Avalon Streaming Credit Source BFM implements the Avalon Streaming Credit Interface protocol. The Avalon Streaming Credit Interface protocol is point-to-point, packet-oriented and drives unidirectional data. This BFM component includes a procedural interface to control signals on the Avalon Streaming Credit Interface including `update`, `credit`, `start of packet`, `empty`, `end of packet` and other signals.

The following figure shows a top-level module for a testbench. This testbench uses an Avalon Streaming Credit Source BFM to verify an Avalon Streaming Credit Sink component. In addition to the Avalon Streaming Credit Source component, the testbench typically includes a test program and the DUT.

The BFMs may allow certain illegal transactions so that you can test the error-handling functionality of the DUT.

Figure 19. Top-level module to verify an Avalon Streaming Credit Sink Interface



11.1. Parameters

The Avalon Streaming Credit Source BFM supports all of the signals defined for the Avalon Streaming Credit Source Interface. The following table lists the parameters to customize the Avalon Streaming Credit Source Interface.

Table 17. Avalon Streaming Credit Source BFM parameters

Parameter	Default Value	Legal Values	Description
USE_CHANNEL	0	0 / 1	When true, interface includes `channel` port.
USE_ERROR	0	0 / 1	When true, interface includes `error` port.
USE_PACKETS	0	0 / 1	When true, interface includes packet-related ports.

continued...

Parameter	Default Value	Legal Values	Description
USE_EMPTY	0	0 / 1	Can be true only when USE_PACKETS is true. When true, interface includes `empty` port.
USER_SIGNALS	0	0 / 1	When true, user signals can be added.
SRC_MAX_CREDIT	1	1,2,4,8,16,32,64,128,256	Specifies Maximum Credit Allowed.
SRC_SYMBOLS_PER_BEAT	4	1 – 1024	Specifies number of Symbols per Beat.
SRC_DATA_BITS_PER_SYMBOL	8	1 – 1024	Specifies Data bits per symbol.
SRC_ERROR_W	1	1 – 1024	Specifies the width of the `error` signal.
SRC_EMPTY_W	1	1 - 1024	Specifies the width of the `empty` signal.
SRC_CHANNEL_W	1	1 – 128	Specifies the width of the `channel` signal.

11.2. Implementation

The underlying transaction handling logic for the Avalon® Streaming Credit Source BFM uses the core of the Avalon Streaming Source BFM. The main difference is there is no `ready` signal and there is an addition of `credit` and `update` signals. API function calls cannot be done from the BFM instance directly. All the APIs sit two levels below inside a module which will be instantiated with name '`c_logic_src`'. This module has all the API definitions that are mentioned below and the credit management logic for the Avalon Streaming Credit Source BFM.

APIs related to user-signals will only be generated if the USER_SIGNAL parameter is set to true. The API will be generated based on the name of user-signals provided.

11.3. Avalon Streaming Interface Credit Source APIs

11.3.1. initialize()

Prototype:	initialize()
Arguments:	Verilog HDL: None
Returns:	void
Description:	Drives the interface to the idle state.
Language Support:	Verilog HDL

11.3.2. rst()

Prototype:	rst()
Arguments:	Verilog HDL: None
Returns:	void
Description:	Drives BFM's internal signals to reset values.
Language Support:	Verilog HDL

11.3.3. get_src_transaction_complete()

Prototype:	get_src_transaction_complete()
Arguments:	Verilog HDL: None
Returns:	bit
Description:	Returns the transaction complete status.
Language Support:	Verilog HDL

11.3.4. get_outstanding_credit()

Prototype:	get_outstanding_credit()
Arguments:	Verilog HDL: None
Returns:	integer
Description:	Returns the value of outstanding credits held by the Avalon-ST Credit Source BFM.
Language Support:	Verilog HDL

11.3.5. push_transaction()

Prototype:	push_transaction()
Arguments:	Verilog HDL: None
Returns:	void
Description:	Inserts the out-going transaction into the local transaction queue. The BFM drives the appropriate signals to the Avalon streaming interface Credit interface based on the transactions in its local queue.
Language Support:	Verilog HDL

11.3.6. set_max_transaction_queue_size()

Prototype:	void set_max_transaction_queue_size (int size)
Arguments:	Verilog HDL: int size
Returns:	void
Description:	Sets the pending transaction queue size maximum threshold. The public event *signal_max_transaction_queue_size* triggers when the threshold is exceeded.
Language Support:	Verilog HDL

11.3.7. set_min_transaction_queue_size()

Prototype:	Void set_min_transaction_queue_size (int size)
Arguments:	Verilog HDL: int size
Returns:	void
Description:	Sets the pending transaction minimum queue size threshold. The public event *signal_min_transaction_queue_size* triggers when the queue size level is below the minimum threshold.
Language Support:	Verilog HDL

11.3.8. set_transaction_data()

Prototype:	set_transaction_data (bit [SRC_DATA_W-1:0] data)
Arguments:	Verilog HDL: data SRC_DATA_W = SRC_SYMBOLS_PER_BEAT*SRC_DATA_BITS_PER_SYMBOL
Returns:	void
Description:	Sets the value of data in the out-going transaction.
Language Support:	Verilog HDL

11.3.9. set_transaction_channel()

Prototype:	set_transaction_channel (bit [SRC_CHANNEL_W-1:0] channel)
Arguments:	Verilog HDL: channel
Returns:	void
Description:	Sets the channel identifier in the out-going transaction.
Language Support:	Verilog HDL

11.3.10. set_transaction_error()

Prototype:	set_transaction_error (bit [SRC_ERROR_W-1:0] error)
Arguments:	Verilog HDL: error
Returns:	void
Description:	Sets the out-going transaction error value.
Language Support:	Verilog HDL

11.3.11. set_transaction_idles()

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

11.3.12. set_transaction_sop()

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

11.3.13. set_transaction_eop()

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

11.3.14. set_transaction_empty()

Prototype:	set_transaction_empty (bit [SRC_EMPTY_W-1:0] empty)
Arguments:	Verilog HDL: empty
Returns:	void
Description:	Sets the out-going transaction's `empty` value.
Language Support:	Verilog HDL

11.3.15. return_credit()

Prototype:	return_credit()
Arguments:	Verilog HDL: None
Returns:	void
Description:	Returns back a credit to the sink interface in next cycle.
Language Support:	Verilog HDL

11.3.16. set_user_signal_per_symbol_data()

Prototype:	set_user_signal_per_symbol_data (“parity”, per_sym_data)
Arguments:	Verilog HDL: 1. Name of PER_SYMBOL user signal as a string (example – “parity” in this case) 2. Data value to be driven for the user signal, in this case `per_sym_data` is a bus that drives the data.
Returns:	void
Description:	Sets the value of input PER_SYMBOL user signal in out-going transaction. If there are multiple PER_SYMBOL user signals, the width of `per_sym_data` is the max value from all PER_SYMBOL user signals. One API call can drive data through one selected signal. Need to make multiple API calls for driving multiple signals.
Language Support:	Verilog HDL

11.3.17. push_user_signal_per_symbol_transaction()

Prototype:	push_user_signal_per_symbol_transaction (“parity”)
Arguments:	Verilog HDL: string; Name of a PER_SYMBOL user signal

continued...

Returns:	void
Description:	Inserts the out-going PER_SYMBOL user signal transaction into the local transaction queue. The BFM drives the appropriate user signal to the Avalon streaming interface Credit interface based on the transaction in its local queue.
Language Support:	Verilog HDL

11.3.18. set_user_signal_per_packet_data()

Prototype:	set_user_signal_per_packet_data ("color", per_pkt_data)
Arguments:	Verilog HDL: 1. Name of PER_PACKET user signal as a string (example – "color" in this case) 2. Data value to be driven for the user signal, in this case `per_pkt_data` is a bus that drives the data.
Returns:	void
Description:	Sets the value of input PER_PACKET user signal in out-going transaction. If there are multiple PER_PACKET user signals, the width of `per_pkt_data` is the max value from all PER_PACKET user signals. One API call can drive data through one selected signal. Need to make multiple API calls for driving multiple signals.
Language Support:	Verilog HDL

11.3.19. push_user_signal_per_packet_transaction()

Prototype:	push_user_signal_per_packet_transaction("color")
Arguments:	Verilog HDL: string; Name of a PER_PACKET user signal
Returns:	void
Description:	Inserts the out-going PER_PACKET user signal transaction into the local transaction queue. The BFM drives the appropriate user signal to the Avalon streaming interface Credit interface based on the transaction in its local queue.
Language Support:	Verilog HDL

11.3.20. signal_credit_arrived

Prototype:	signal_credit_arrived
Arguments:	Verilog HDL: None
Returns:	void
Description:	Signals `update` has arrived with new credits.
Language Support:	Verilog HDL

11.3.21. signal_fatal_error

Prototype:	signal_fatal_error
Arguments:	Verilog HDL: None

continued...

Returns:	void
Description:	Signals that a fatal error has occurred. It terminates the simulation.
Language Support:	Verilog HDL

11.3.22. signal_src_transaction_complete

Prototype:	signal_src_transaction_complete
Arguments:	Verilog HDL: None
Returns:	void
Description:	Signals that all pending transactions have completed.
Language Support:	Verilog HDL

11.3.23. signal_src_driving_transaction

Prototype:	signal_src_driving_transaction
Arguments:	Verilog HDL: None
Returns:	void
Description:	Signals when the source begins to drive a transaction to the interface.
Language Support:	Verilog HDL

11.3.24. signal_max_transaction_queue_size

Prototype:	signal_max_transaction_queue_size
Arguments:	Verilog HDL: Non
Returns:	void
Description:	Signals that the pending transaction queue size threshold has been exceeded.
Language Support:	Verilog HDL

11.3.25. signal_min_transaction_queue_size

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

Note: Currently, the BFM does not support APIs for setting the `channel` and `error` signals. This feature will be included in a future release of Intel® Quartus® Prime.

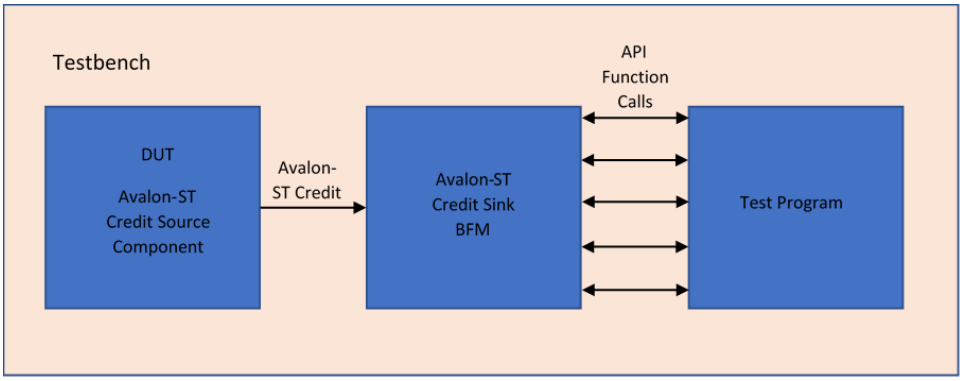
12. Avalon Streaming Credit Sink BFM

The Avalon Streaming Credit Sink BFM implements the Avalon Streaming Credit Interface protocol. The Avalon Streaming Credit Interface protocol is point-to-point, packet oriented and drives unidirectional data. This BFM component includes a procedural interface to respond to the DUT that includes an Avalon Streaming Credit Source Interface.

The following figure shows a top-level module for a testbench. This testbench uses an Avalon Streaming Credit Sink BFM to verify an Avalon Streaming Credit Source component. In addition to the Avalon Streaming Credit Sink BFM component, the testbench typically includes a test program and the DUT.

The BFMs may allow certain illegal transactions so that you can test the error-handling functionality of the DUT.

Figure 20. Top-level module to verify an Avalon Streaming Credit Source Interface



12.1. Parameters

The Avalon Streaming Credit Sink BFM supports all of the signals defined for the Avalon Streaming Credit Sink interface. The following table lists the parameters to customize the Avalon Streaming Credit Sink interface.

Table 18. Avalon Streaming Credit Sink BFM parameters

Parameter	Default Value	Legal Values	Description
USE_CHANNEL	0	0 / 1	When true, interface includes `channel` port.
USE_ERROR	0	0 / 1	When true, interface includes `error` port.
USE_PACKETS	0	0 / 1	When true, interface includes packet-related ports.

continued...

Parameter	Default Value	Legal Values	Description
USE_EMPTY	0	0 / 1	Can be true only when USE_PACKETS is true. When true, interface includes `empty` port.
USER_SIGNALS	0	0 / 1	When true, user signals can be added.
SNK_MAX_CREDIT	1	1,2,4,8,16,32,64,128,256	Specifies Maximum Credit Allowed.
SNK_SYMBOLS_PER_BEAT	4	1 – 1024	Specifies number of Symbols per Beat.
SNK_DATA_BITS_PER_SYMBOL	8	1 – 1024	Specifies Data bits per symbol.
SNK_ERROR_W	1	1 – 1024	Specifies the width of the `error` signal.
SNK_EMPTY_W	1	1 - 1024	Specifies the width of the `empty` signal.
SNK_CHANNEL_W	1	1 – 128	Specifies the width of the `channel` signal.

12.2. Implementation

The underlying transaction handling logic for the Avalon Streaming Credit Sink BFM uses the core of the Avalon Streaming Sink BFM. The main difference is there is no `ready` signal and there is an addition of `credit` and `update` signals. API function calls cannot be done from the BFM instance directly. All the APIs sit two levels below inside a module which will be instantiated with name '`c_logic_snk`'. This module has all the API definitions that are mentioned below and the credit management logic for the Avalon Streaming Credit Sink BFM.

APIs related to user-signals will only be generated if the USER_SIGNAL parameter is set to true. The API will be generated based on the name of user-signals provided.

12.3. Avalon Streaming Interface Credit Sink APIs

12.3.1. initialize()

Prototype:	initialize()
Arguments:	Verilog HDL: None
Returns:	void
Description:	Drives the interface to the idle state.
Language Support:	Verilog HDL

12.3.2. rst()

Prototype:	rst()
Arguments:	Verilog HDL: None
Returns:	void
Description:	Drives BFM's internal signals to reset values.
Language Support:	Verilog HDL

12.3.3. get_outstanding_credit()

Prototype:	get_outstanding_credit ()
Arguments:	Verilog HDL: None
Returns:	Integer
Description:	Returns the value of outstanding credits held by the Avalon Streaming Credit Sink BFM.
Language Support:	Verilog HDL

12.3.4. get_transaction_idles()

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

12.3.5. get_transaction_data()

Prototype:	get_transaction_data ()
Arguments:	Verilog HDL: None
Returns:	bit [SNK_DATA_W - 1:0] SNK_DATA_W = SNK_SYMBOLS_PER_BEAT*SNK_DATA_BITS_PER_SYMBOL
Description:	Returns the data in the most recently removed transaction.
Language Support:	Verilog HDL

12.3.6. get_transaction_channel()

Prototype:	get_transaction_channel ()
Arguments:	Verilog HDL: None
Returns:	bit [SNK_CHANNEL_W-1:0]
Description:	Returns the channel identifier for the most recently removed transaction.
Language Support:	Verilog HDL

12.3.7. get_transaction_error()

Prototype:	get_transaction_error ()
Arguments:	Verilog HDL: None
Returns:	bit [SNK_ERROR_W-1:0]
Description:	Returns the error in the most recently removed transaction.
Language Support:	Verilog HDL

12.3.8. get_transaction_sop()

Prototype:	get_transaction_sop ()
Arguments:	Verilog HDL: None
Returns:	bit
Description:	Returns the transaction's `start of packet` status in the most recently removed transaction.
Language Support:	Verilog HDL

12.3.9. get_transaction_eop()

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

12.3.10. get_transaction_empty()

Prototype:	get_transaction_empty ()
Arguments:	Verilog HDL: None
Returns:	bit [SNK_EMPTY_W-1:0]
Description:	Returns the number of `empty` symbols in the most recently removed transaction
Language Support:	Verilog HDL

12.3.11. pop_transaction()

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

12.3.12. send_credit()

Prototype:	send_credit (bit [SNK_CRD_W-1 :0] credit)
Arguments:	Verilog HDL: credit, here SNK_CRD_W is clog2 of SNK_MAX_CREDIT
Returns:	void
Description:	Sends credit values to the connected source interface
Language Support:	Verilog HDL

12.3.13. get_user_signal_per_symbol_data()

Prototype:	get_user_signal_per_symbol_data (“parity”)
Arguments:	Verilog HDL: Name of a PER_SYMBOL user signal
Returns:	PER_SYMBOL user signal data field
Description:	Returns the PER_SYMBOL data for the user signal provided as a string input from the most recently removed transaction.
Language Support:	Verilog HDL

12.3.14. pop_user_signal_per_symbol_transaction()

Prototype:	pop_user_signal_per_symbol_transaction(“parity”)
Arguments:	Verilog HDL: Name of a PER_SYMBOL user signal
Returns:	void
Description:	Removes the transaction descriptor of PER_SYMBOL user signal provided as a string input. This removes the transaction descriptor from the queue, so that the testbench can query it using the get_transaction methods. *Note:* The pop_user_signal_per_symbol_transaction API must be called when the pop_transaction API is called. This is to avoid buffer overflow in BFM queues and data synchronization issues.
Language Support:	Verilog HDL

12.3.15. get_user_signal_per_packet_data()

Prototype:	get_user_signal_per_packet_data (“color”)
Arguments:	Verilog HDL: Name of a PER_PACKET user signal
Returns:	PER_PACKET user signal data field
Description:	Returns the PER_PACKET data for the user signal provided as a string input from the most recently removed transaction.
Language Support:	Verilog HDL

12.3.16. pop_user_signal_per_packet_transaction()

Prototype:	pop_user_signal_per_packet_transaction(“color”)
Arguments:	Verilog HDL: Name of a PER_PACKET user signal
Returns:	void
Description:	Removes the transaction descriptor of PER_PACKET user signal provided as a string input. This removes the transaction descriptor from the queue, so that the testbench can query it using the get_transaction methods. *Note:* The pop_user_signal_per_packet_transaction API must be called when the pop_transaction API is called. This is to avoid buffer overflow in BFM queues and data synchronization issues.
Language Support:	Verilog HDL

12.3.17. signal_fatal_error

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

12.3.18. signal_has_max_credits

Prototype:	signal_has_max_credits
Arguments:	Verilog HDL: None
Returns:	void
Description:	Signals that the Sink BFM has credits = SNK_MAX_CREDIT
Language Support:	Verilog HDL

12.3.19. signal_transaction_received

Prototype:	signal_transaction_received
Arguments:	Verilog HDL: None
Returns:	void
Description:	Signals that the transaction has been received and queued.
Language Support:	Verilog HDL

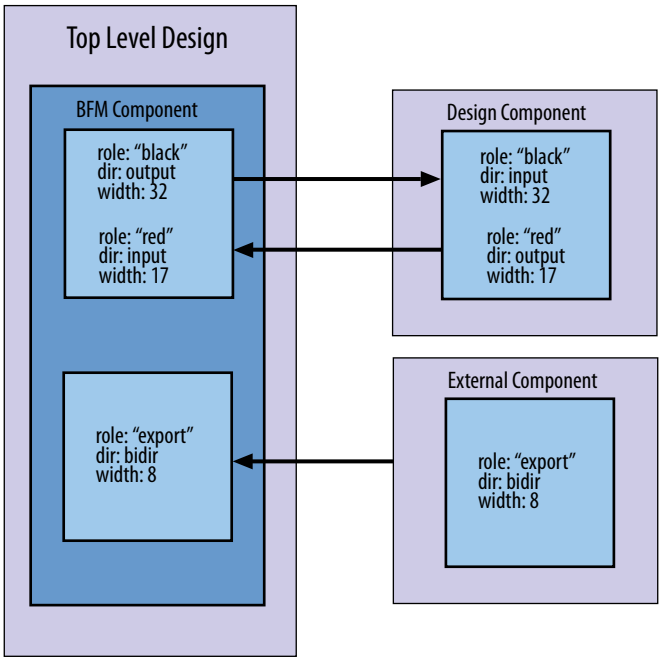
Note: Currently the BFM does not support getting transactions for the `channel` and `error` signals. This feature will be added in a future release of Intel Quartus Prime.

13. Conduit BFM

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

- Port compatibility and polarity
- Legal port widths

Figure 21. Conduit BFM Block Diagram



An Avalon Conduit interface can have an arbitrary number of ports. Conduit ports can have the following characteristics:

- Conduit ports can be an input, output, or bidirectional
- Conduit port widths can range from 1-1024 bits
- Conduit ports have an associated role name. This role name is an arbitrary string. Qsys uses the role name to verify conduit interconnect compatibility between components.
- A conduit port 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 conduit 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.

The conduit API constructs or deconstructs transactions. Transactions are driven out on the physical conduit interface.

To simulating conduit interfaces, you must understand the following points:

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

13.1. Parameters

Table 19. Conduit BFM Parameter Settings

Option	Default Value	Legal Values	Description
Role	N/A	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.

13.2. Conduit BFM API

event_input_<role name>_change

Prototype:	`event_input_<role name>_change`
Arguments:	Verilog HDL: N.A. VHDL: None
Returns:	`void`
Description:	Notifies the testbench when a port changes its value<*role name*>. For a bidirectional port, this event is only triggered if its input value defers from its last input value.
Language support:	VHDL

13.2.1. event_reset_asserted

Prototype:	`event_reset_asserted`
Arguments:	Verilog HDL: N.A. VHDL: None

continued...

Returns:	`void`
Description:	Notifies the testbench that reset has been asserted.
Language support:	VHDL

13.2.2. get_<role name>()

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

13.2.3. get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`string`
Description:	Returns BFM version as a string of three integers separated by periods. For example, version 13.1 sp1 is encoded as "13.1.1".
Language support:	Verilog HDL

13.2.4. set_<role name>()

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

13.2.5. set_<role name>_oe()

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

13.2.6. signal_input_<role name>_change

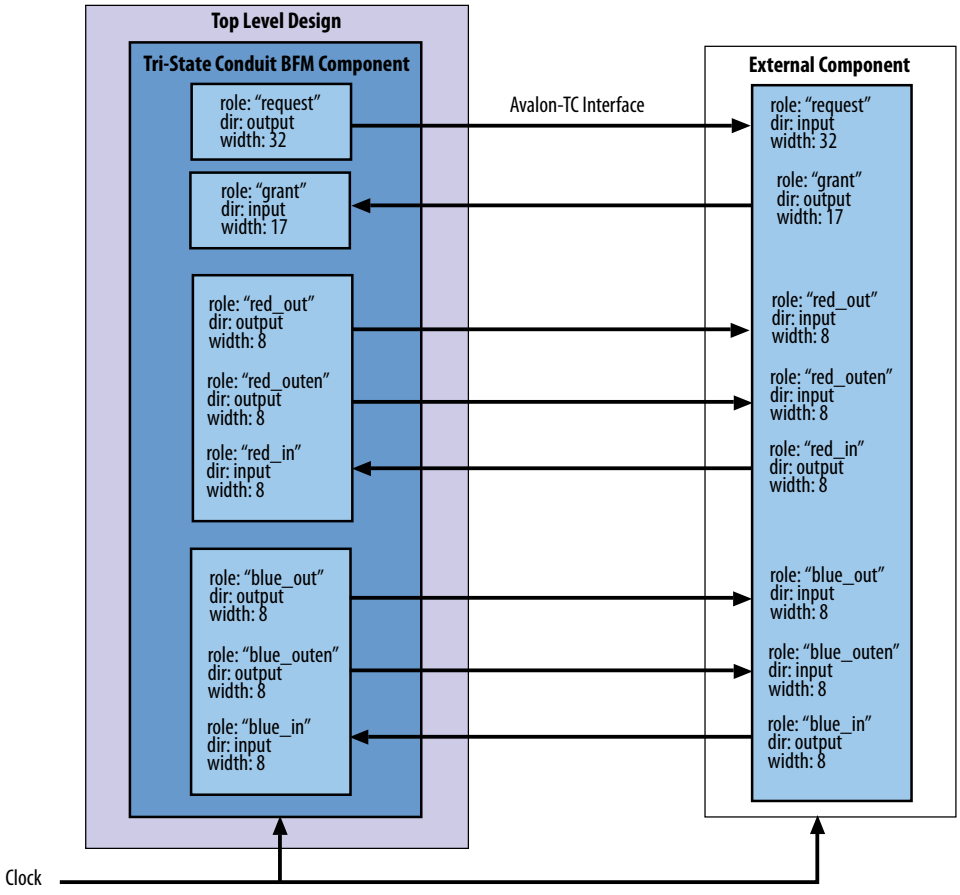
Prototype:	`signal_input_<role name>_change`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Triggers when the input signal for a particular port changes its value. For a bidirectional port, this event is only triggered if its input value defers from its last input value.
Language support:	Verilog HDL

14. Tri-State Conduit BFM

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

- Port compatibility and polarity
- Legal port widths

Figure 22. Conduit BFM Block Diagram



14.1. Parameters

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

Table 20. Tri-State Conduit BFM Parameter Settings

Option	Default Value	Legal Values	Description
Role	N/A	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**	N/A	Specifies the maximum transactions of data while request and grant signals are asserted. The value is constraint by the number of roles.

Related Information

- Avalon Interface Specifications
- Avalon Tri-State Conduit Components User Guide

14.2. Tri-State Conduit BFM API

event_all_transactions_complete()

Prototype:	`event_all_transactions_complete()`
Arguments:	Verilog HDL: N.A. VHDL: None
Returns:	`void`
Description:	Notifies the testbench that all commands have completed.
Language support:	VHDL

14.2.1. event_interface_granted()

Prototype:	`event_interface_granted()`
Arguments:	Verilog HDL: N.A. VHDL: None
Returns:	`void`
Description:	Notifies the testbench that the interface has been granted.
Language support:	VHDL

14.2.2. event_grant_deasserted_while_request_remain_asserted

Prototype:	`event_grant_deasserted_while_request_remain_asserted`
Arguments:	Verilog HDL: N.A. VHDL: None
Arguments:	None.

continued...

Returns:	`void`
Description:	Notifies the testbench when the grant signal changes value from high to low while the request signal remains asserted.
Language support:	VHDL

14.2.3. event_max_transaction_queue_size()

Prototype:	`event_max_transaction_queue_size()`
Arguments:	Verilog HDL: N.A. VHDL: None
Returns:	`void`
Description:	Notifies the testbench that the transaction queue is above the maximum level.
Language support:	VHDL

14.2.4. event_min_transaction_queue_size()

Prototype:	`event_min_transaction_queue_size()`
Arguments:	Verilog HDL: N.A. VHDL: None
Returns:	`void`
Description:	Notifies the testbench that the transaction queue is below the minimum level.
Language support:	VHDL

14.2.5. get_input_transaction_queue_size()

Prototype:	`int get_input_transaction_queue_size()`
Arguments:	Verilog HDL: `queue_size` VHDL: `queue_size`
Returns:	`int`
Description:	Returns the size of the queued input transaction in the BFM.
Language support:	Verilog HDL, HDL

14.2.6. get_output_transaction_queue_size()

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

14.2.7. get_transaction_<role name>_in()

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

14.2.8. get_transaction_latency()

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

14.2.9. get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`string`
Description:	Returns the BFM version as a string of three integers separated by periods. For example, version 13.1 sp1 is encoded as "13.1.1".
Language support:	Verilog HDL

14.2.10. pop_transaction()

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

14.2.11. push_transaction()

Prototype:	`void push_transaction()`
Arguments:	Verilog HDL: None VHDL: None

continued...

Returns:	`void`
Description:	Registers an output transaction into the BFM. All registered output transactions are put into transaction queue. A fatal error triggers if you insert a transaction while the BFM is reset.
Language support:	Verilog HDL, VHDL

14.2.12. set_max_transaction_queue_size()

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

14.2.13. set_min_transaction_queue_size()

Prototype:	`int set_min_transaction_queue_size()`
Arguments:	Verilog HDL: `size` VHDL: `size`
Returns:	`void`
Description:	Sets the minimum size of the queued transactions. The BFM triggers an event when the queued transactions falls below the minimum size.
Language support:	Verilog HDL, VHDL

14.2.14. set_num_of_transactions()

Prototype:	`int set_num_of_transactions()`
Arguments:	Verilog HDL: `multiple_transaction_num` VHDL: `multiple_transaction_num`
Returns:	`void`
Description:	Sets the number of transactions to the DUT.
Language support:	Verilog HDL, VHDL

14.2.15. set_transaction_<role name>_out()

Prototype:	`void set_transaction_<role name>_out()`
Arguments:	Verilog HDL: `index, new_value` VHDL: `index, new_value`
Returns:	`void`
Description:	Sets the value of the transaction to the `<role name>_out` output ports.
Language support:	Verilog HDL, VHDL

14.2.16. set_transaction_<role name>_outen()

Prototype:	`string set_transaction_<role name>_outen()`
Arguments:	Verilog HDL: `index, outen` VHDL: `index, outen`
Returns:	`void`
Description:	Sets the value of the transaction to the `<role name>_outen` output ports.
Language support:	Verilog HDL, VHDL

14.2.17. set_transaction_idles()

Prototype:	`void set_transaction_idles()`
Arguments:	Verilog HDL: `idle_cycles` VHDL: `idle_cycles`
Returns:	`void`
Description:	Sets the number of idle cycles that elapse before driving the out-going transaction.
Language support:	Verilog HDL, VHDL

14.2.18. set_valid_transaction_<role name>_out()

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

14.2.19. signal_all_transactions_complete

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

14.2.20. signal_fatal_error

Prototype:	`signal_fatal_error`
Arguments:	Verilog HDL: None VHDL: N.A.

continued...

Returns:	`void`
Description:	Notifies the testbench that a fatal error has occurred in this module.
Language support:	Verilog HDL

14.2.21. signal_grant_deasserted_while_request_remain_asserted

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

14.2.22. signal_interface_granted

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

14.2.23. signal_max_transaction_queue_size

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

14.2.24. signal_min_transaction_queue_size

Prototype:	`signal_min_transaction_queue_size`
Arguments:	Verilog HDL: None VHDL: N.A.
Arguments:	None.

continued...

Returns:	`void`
Description:	Triggers when the size of the pending queue falls below the minimum size.
Language support:	Verilog HDL

15. External Memory BFM

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

- Read and write operations
- Memory initialization

Figure 23. Usage of External Memory BFM with Tri-State Components

The following figure shows how to use the external memory BFM with tri-state components.

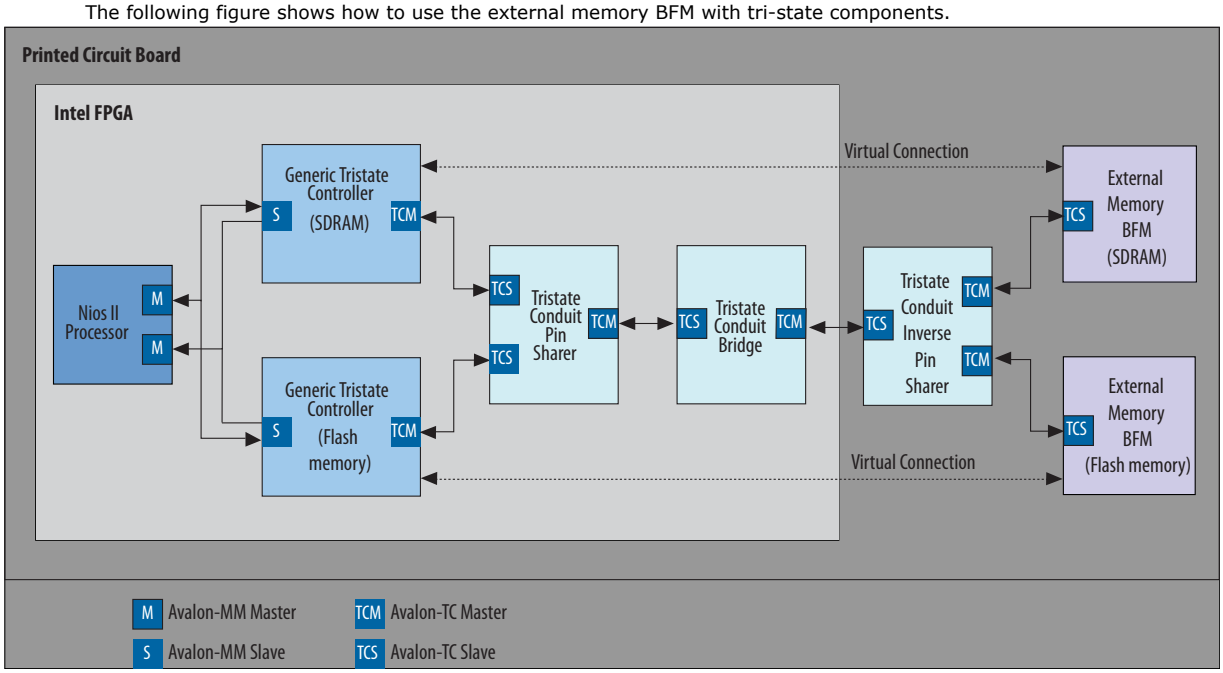


Table 21. External Memory BFM and Related Components

Component	Description
External memory BFM	Represents the external RAM. The external memory BFM is a memory model with an Avalon-TC interface. The BFM also models a set of memories that are supported by the generic tri-state controller component.
Tri-State Conduit Bridge	Converts Avalon-TC signals into conduit signals.

continued...

Component	Description
Tri-State Conduit Pin Sharer	Carries the shared address bus and data.
Tri-State Conduit Inverse Pin Sharer	
Generic Tri_State Controller	Controls the external memory BFM. The generic tri-state controller accepts read and write requests. It converts these requests into necessary SDRAM and bank management commands.

Related Information

Avalon Tri-State Conduit Components User Guide.

15.1. Using the External Memory BFM

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 to the Memory Content

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

The BFM uses `cdt_data_io` as a bidirectional data port. During read transfers, this port acts as an output port. It 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

15.2. Parameters

Table 22. External Memory BFM Parameter Settings

Option	Default Value	Legal Values	Description
Port Enables			
Use the byteenable signal	**On**	**On/Off**	When **On**, the interface includes a `byteenable` pin to enable specific byte lanes during transfer.

continued...

Option	Default Value	Legal Values	Description
Use the chip select signal	**On**	**On/Off**	When **On**, the interface includes a `chipselect` pin. When present, the slave port ignores all Avalon-MM signals unless `chipselect` is asserted. `chipselect` is always present in combination with `read` or `write`.
Use the write signal	**On**	**On/Off**	When **On**, the interface includes a `write` pin that enables the write-request signal.
Use the read signal	**On**	**On/Off**	When **On**, the interface includes a `read` pin that enables the read-request signal.
Use the output enable signal	**On**	**On/Off**	When **On**, the interface includes an `outputenable` pin.
Use the begintransfer signal	**On**	**On/Off**	When **On**, the interface includes a `begintransfer` pin.
Use the reset input signal	**Off**	**On/Off**	When **On**, the interface includes a `reset` pin.
Use the active low byteenable signal	**Off**	**On/Off**	When **On**, the interface includes an active low `byteenable` pin.
Use the active low chipselect signal	**Off**	**On/Off**	When **On**, the interface includes an active low `chipselect_n` pin.
Use the active low write signal	**Off**	**On/Off**	When **On**, the interface includes an active low `write_n` pin.
Use the active low read signal	**Off**	**On/Off**	When **On**, the interface includes an active low `read_n` pin.
Use the active low outputenable signal	**Off**	**On/Off**	When **On**, the interface includes an active low `outputenable_n` pin.
Use the active low begintransfer signal	**Off**	**On/Off**	When **On**, the interface includes an active low `begintransfer_n` pin.
Use the active low reset signal	**Off**	**On/Off**	When **On**, the interface includes an active low `reset_n` pin.
Interface Signals Name			
Address Role	`cdt_address`	N/A	Specifies the conduit interface role name that matches the role name on the external memory device.
Data Role	`cdt_data_io`	N/A	
Write Role	`cdt_write`	N/A	
Read Role	`cdt_read`	N/A	
Byteenable Role	`cdt_byteenable`	N/A	
Chip Select Role	`cdt_chipselect`	N/A	
Outputenable Role	`cdt_outputenable`	N/A	

continued...

Option	Default Value	Legal Values	Description
Begintransfer Role	`cdt_begintransfer`	N/A	
Reset Role	`cdt_reset`	N/A	
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`	N/A	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**	N/A	Specifies the read latency of the interface.
Miscellaneous Properties			
VHDL BFM ID	**0**	0–1023	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.

15.3. External Memory BFM API

fill()

Prototype:	`fill()`	
Arguments:	Verilog HDL: `logic[DATA_W-1:0]` data `bit[DATA_W-1:0]` increment `bit[CDT_ADDRESS_W-1:0]` address low `bit[CDT_ADDRESS_W-1:0]` address high	VHDL: `logic[DATA_W-1:0]` data `bit[DATA_W-1:0]` increment `bit[CDT_ADDRESS_W-1:0]` address low `bit[CDT_ADDRESS_W-1:0]` address high `bfm_id req_if(bfm_id)`
Returns:	`void`	
Description:	Overwrites the memory content at the starting address specified by `address_low` until the ending address specified by `address_high`. The `data` field indicates the data value. The `increment` field indicates the data value increment from one address to the next address. For example, fill (`data[1]`, `increment[2]`, `address_low[10]`, `address_high[12]`) fills the memory as follows: • `memory[address=10]` is filled with data value 1 • `memory[address=11]` is filled with data value 3 • `memory[address=12]` is filled with data value 5	
Language support:	Verilog HDL, VHDL	

15.3.1. read()

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

15.3.2. signal_api_call

Prototype:	`signal_api_call`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Triggers when a client make an API call.
Language support:	Verilog HDL

15.3.3. write()

Prototype:	`write()`	

Arguments:	Verilog HDL:
	`bit[CDT_ADDRESS_W-1:0] address`
	`logic[DATA_W-1:0] data`
		VHDL:
		`bit[CDT_ADDRESS_W-1:0] address`
		`logic[DATA_W-1:0] data`
		`bfm_id`
		`req_if(bfm_id)`

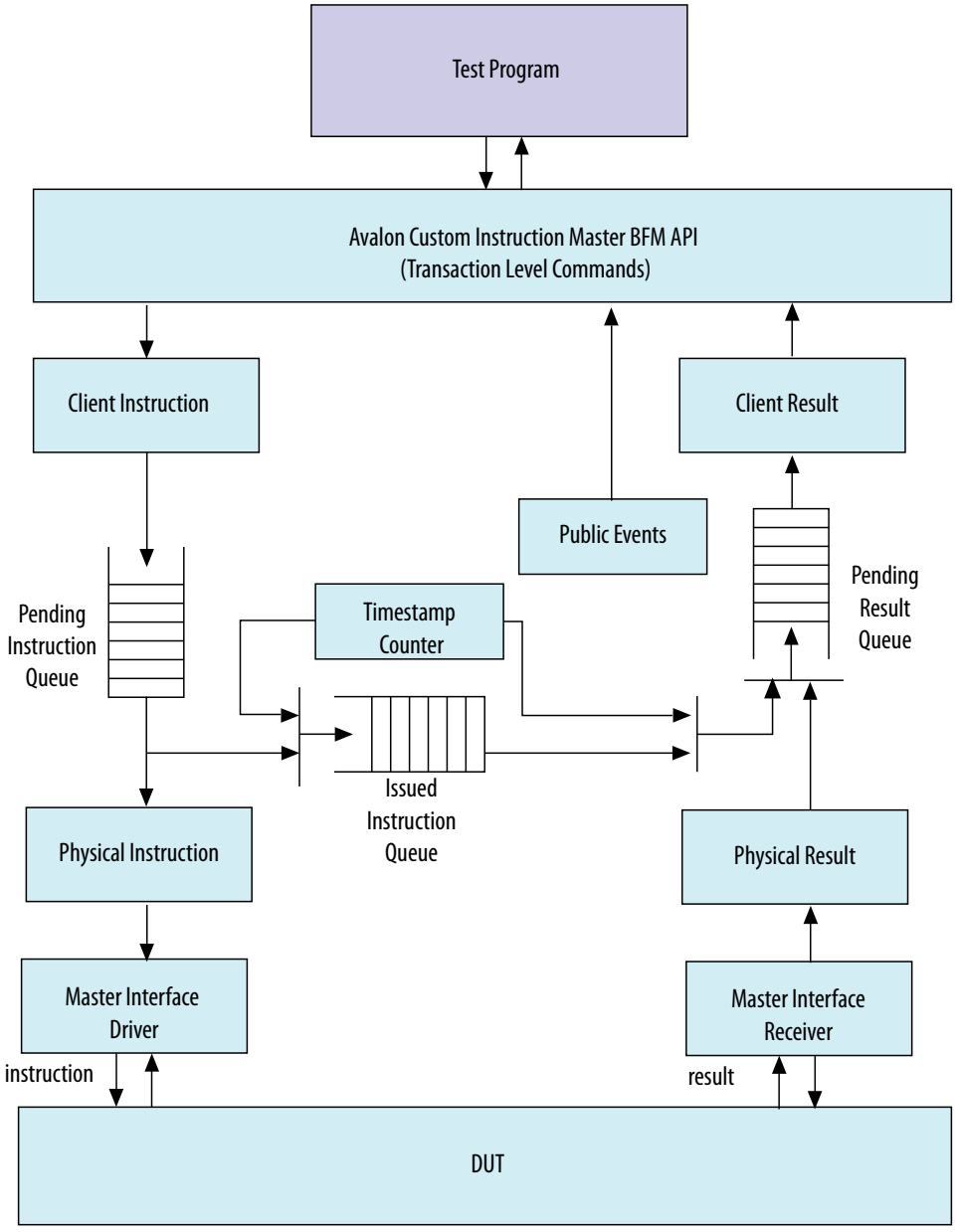
Returns:	`void`	
Description:	Overwrites the memory content at an address you specify.	
Language support:	Verilog HDL, VHDL	

16. Nios II Custom Instruction Master BFM

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

- Combinational and multicycle master custom instructions
- Extended instructions

Figure 24. Custom Instructions Master BFM Block Diagram



The Nios II Custom Instruction Master BFM uses queues to manage instructions. You can use this BFM to manage instructions in the following ways:

- 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 results are inserted into a result queue. You can remove the result on an event basis, or at the end of the simulation.

16.1. Parameters

Table 23. 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**	N/A	Specifies the fixed length for multi-cycle mode.
Port Enables			
Use Result Port	**On**	**On/Off**	When **On**, the interface includes a `result` pin.
Use Multi-cycle Mode	**Off**	**On/Off**	When **On,** the interface can include a `start` pin, a `done` pin , both pins, or neither pins. The result returns in any of the following conditions: • With a `start` signal—Result returns together with an instruction. • Without a `start` signal—Result returns with instruction on the bus at every clock cycle. • With a `done` signal—Result returns at any time. • Without a `done` signal—Result returns at a fixed cycle.
Using start port	**On**	**On/Off**	When **On**, the interface includes a `start` pin.
Using done port	**On**	**On/Off**	When **On**, the interface includes a `done` pin.
Use Extended Port	**Off**	**On/Off**	When **On**, the interface includes a `n` pin.
Extended Port Width	**1**	N/A	Specifies the width of the extended `n` port.
Use Internal Register a	**Off**	**On/Off**	When **On**, the interface includes the `readra` and `a` pins.
Use Internal Register b	**Off**	**On/Off**	When **On**, the interface includes the `readrb` and `b` pins.
Use Internal Register c	**Off**	**On/Off**	When **On**, the interface includes the `readrc` and `c` pins.
Miscellaneous Properties			
VHDL BFM ID	**0**	0–1023	For VHDL BFMs only. Use this option to assign a unique number to each BFM in the testbench design.

16.2. Nios II Custom Instruction API

event_instruction_start()

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

16.2.1. event_result_received()

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

16.2.1.1. event_unexpected_result_received()

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

16.2.1.2. event_instructions_completed()

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

16.2.1.3. event_max_instruction_queue_size()

Prototype:	`event_max_instruction_queue_size()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`

continued...

Returns:	`void`
Description:	Indicates that pending instructions exceed the maximum level.
Language support:	VHDL

16.2.1.4. event_min_instruction_queue_size()

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

16.2.1.5. event_max_result_queue_size()

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

16.2.1.6. event_min_result_queue_size()

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

16.2.1.7. get_instruction_queue_size()

Prototype:	`int get_instruction_queue_size(int size)`
Arguments:	Verilog HDL: None VHDL: `instruction_queue_size`, `bfm_id`, `req_if(bfm_id)`
Returns:	`int size`.
Description:	Returns the number of instructions in the queue.
Language support:	Verilog HDL, VHDL

16.2.1.8. get_result_delay()

Prototype:	`int get_result_delay()`
Arguments:	Verilog HDL: None VHDL: `result_delay`, `bfm_id`, `req_if(bfm_id)`
Returns:	Width of the data `(ci_data_t)`that can contain the following variables: • `[Word_width-1:0]` • `[Ext_width-1:0]` • `[Addr_width-1:0]`
Description:	Returns the result delay.
Language support:	Verilog HDL, VHDL

16.2.1.9. get_result_queue_size()

Prototype:	`int get_result_queue_size(int size)`
Arguments:	Verilog HDL: None VHDL: `result_queue_size`, `bfm_id`, `req_if(bfm_id)`
Returns:	`int size`.
Description:	Returns the number of results in the queue.
Language support:	Verilog HDL, VHDL

16.2.1.10. get_result_value()

Prototype:	`string get_result_value()`
Arguments:	Verilog HDL: None VHDL: `result_value`, `bfm_id`, `req_if(bfm_id)`
Returns:	Width of the data `(ci_data_t)`that can contain the following variables: • `[Word_width-1:0]` • `Ext_width-1:0]` • `[Addr_width-1:0]`
Description:	Returns the instruction result.
Language support:	Verilog HDL, VHDL

16.2.1.11. get_version()

Prototype:	`string get_version()`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`string`
Description:	Returns the BFM version as a string of three integers separated by periods. For example, version 13.1 sp1 is encoded as "13.1.1".
Language support:	Verilog HDL

16.2.1.12. insert_instruction()

Prototype:	`void insert_instruction()`	
Arguments:	Verilog HDL: `ci_data_t dataa`	VHDL: `ci_data_t dataa`
	`ci_data_t datab`	`ci_data_t datab`
	`ci_n_t n`	`ci_n_t n`
	`ci_addr_t a`	`ci_addr_t a`
	`ci_addr_t b`	`ci_addr_t b`
	`ci_addr_t c`	`ci_addr_t c`
	`logic readra`	`logic readra`
	`logic readrb`	`logic readrb`
	`logic writerc`	`logic writerc`
	`ci_data_t idle`	`ci_data_t idle`
	`int err_inj`	`int err_inj`
		`bfm_id`
		`req_if(bfm_id)`
Returns:	`void`	
Description:	A simplified API to set and push instructions.	
Language support:	Verilog HDL, VHDL	

16.2.1.13. pop_result()

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

16.2.1.14. push_instruction()

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

16.2.1.15. retrieve_result()

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

16.2.1.16. set_ci_clk_en()

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

16.2.1.17. set_clock_enable_timeout()

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

16.2.1.18. set_instruction_a()

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

16.2.1.19. set_instruction_b()

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

16.2.1.20. set_instruction_c()

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

16.2.1.21. set_instruction_dataa()

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

16.2.1.22. set_instruction_datab()

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

16.2.1.23. set_instruction_err_inject()

Prototype:	`void set_instruction_err_inject()`
Arguments:	Verilog HDL: `int err_inj` VHDL: `int err_inj, bfm_id, req_if(bfm_id)`

continued...

Returns:	`void`
Description:	Sets the instruction to execute in pre-defined error.
Language support:	Verilog HDL, VHDL

16.2.1.24. set_instruction_idle()

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

16.2.1.25. set_instruction_n()

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

16.2.1.26. set_instruction_readra()

Prototype:	`void set_instruction_readra()`
Arguments:	Verilog HDL: `logic enable` VHDL: `logic enable`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	Sets the instruction register file read a value.
Language support:	Verilog HDL, VHDL

16.2.1.27. set_instruction_readrb()

Prototype:	`void set_instruction_readrb()`
Arguments:	Verilog HDL: `logic enable` VHDL: `logic enable`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	Sets the instruction register file read b value.
Language support:	Verilog HDL, VHDL

16.2.1.28. set_instruction_timeout()

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

16.2.1.29. set_instruction_writerc()

Prototype:	`void set_instruction_writerc()`
Arguments:	Verilog HDL: `logic enable` VHDL: `logic enable`, `bfm_id`, `req_if(bfm_id)`
Returns:	`void`
Description:	Sets the instruction register file write `c` value.
Language support:	Verilog HDL, VHDL

16.2.1.30. set_max_instruction_queue_size()

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

16.2.1.31. set_max_result_queue_size()

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

16.2.1.32. set_min_instruction_queue_size()

Prototype:	`void set_min_instruction_queue_size(int size).`
Arguments:	Verilog HDL: `int size` VHDL: `int size`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`void`
Description:	Sets the pending instruction queue size minimum threshold.
Language support:	Verilog HDL, VHDL

16.2.1.33. set_min_result_queue_size()

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

16.2.1.34. set_result_timeout()

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

16.2.1.35. signal_unexpected_result_received

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

16.2.1.36. signal_fatal_error

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

16.2.1.37. signal_instructions_completed

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

16.2.1.38. signal_instruction_start

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

16.2.1.39. signal_max_instruction_queue_size

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

16.2.1.40. signal_max_result_queue_size

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

16.2.1.41. signal_min_instruction_queue_size

Prototype:	`signal_min_instruction_queue_size`
Arguments:	Verilog HDL: None VHDL: N.A.

continued...

Returns:	`void`
Description:	Signals that the pending instruction queue size is below the minimum threshold.
Language support:	Verilog HDL

16.2.1.42. signal_min_result_queue_size

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

16.2.1.43. signal_result_received

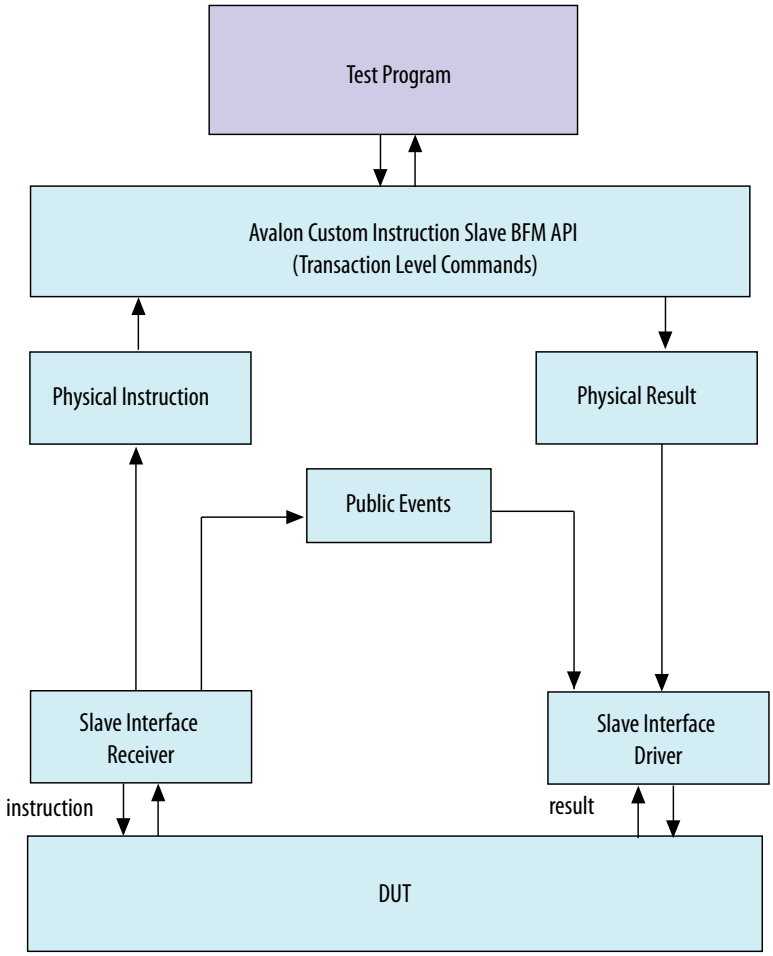
Prototype:	`signal_result_received`
Arguments:	Verilog HDL: None VHDL: N.A.
Returns:	`void`
Description:	Signals that a result has been received.
Language support:	Verilog HDL

17. Nios II Custom Instruction Slave BFM

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

- Combinational and multicycle slave custom instructions
- Extended instructions

Figure 25. 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. The BFM drives the old result onto the interface if you do not provide a new result for an instruction. If there is no instruction, the BFM drives unknown (X) on the interface.

17.1. Parameters

Table 24. Custom Instruction Slave BFM Parameter Settings

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

17.2. Nios II Custom Instruction Slave BFM API

event_known_instruction_received()

Prototype:	`event_known_instruction_received()`
Arguments:	Verilog HDL: N.A. VHDL: `bfm_id`

continued...

Returns:	`void`
Description:	Indicates that a change occurred on the instruction interface and there is no unknown value.
Language support:	VHDL

17.2.1. event_instruction_inconsistent()

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

17.2.2. event_instruction_unchanged()

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

17.2.3. event_result_driven()

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

17.2.4. event_result_done()

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

17.2.5. event_unknown_instruction_received()

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

17.2.6. get_ci_clk_en()

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

17.2.7. get_instruction_a()

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

17.2.8. get_instruction_b()

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

17.2.9. get_instruction_c()

Prototype:	`string get_instruction_c()`
Arguments:	Verilog HDL: None VHDL: `instruction_c`, `bfm_id`, `req_if(bfm_id)`

continued...

Returns:	`ci_addr_t`
Description:	Retrieves the instruction register file address `c` value.
Language support:	Verilog HDL, VHDL

17.2.10. get_instruction_dataa()

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

17.2.11. get_instruction_datab()

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

17.2.12. get_instruction_idle()

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

17.2.13. get_instruction_n()

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

17.2.14. get_instruction_readra()

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

17.2.15. get_instruction_readrb()

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

17.2.16. get_instruction_writerc()

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

17.2.17. get_version()

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

17.2.18. insert_result()

Prototype:	`void insert_result()`	
Arguments:	Verilog HDL: `ci_data_t value`	VHDL: `ci_data_t value`

continued...

	ci_data_t delay	ci_data_t delay
	int err_inj	int err_inj
		bfm_id
		req_if(bfm_id)
Returns:	void	
Description:	A simplified API to set results.	
Language support:	Verilog HDL, VHDL	

17.2.19. retrieve_instruction()

Prototype:	void retrieve_instruction.	
Arguments:	Verilog HDL: output ci_data_t dataa	VHDL: output ci_data_t dataa
	output ci_data_t datab	output ci_data_t datab
	output ci_n_t n	output ci_n_t n
	output ci_addr_t a	output ci_addr_t a
	output ci_addr_t b	output ci_addr_t b
	output ci_addr_t c	output ci_addr_t c
	output logic readra	output logic readra
	output logic readrb	output logic readrb
	output logic writerc	output logic writerc
	output ci_data_t idle	output ci_data_t idle
		bfm_id
		req_if(bfm_id)
Returns:	void	
Description:	A simplified API to retrieve instruction.	
Language support:	Verilog HDL, VHDL	

17.2.20. set_clock_enable_timeout()

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

17.2.21. set_instruction_a()

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

17.2.22. set_instruction_b()

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

17.2.23. set_instruction_c()

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

17.2.24. set_instruction_timeout()

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

17.2.25. set_result_delay()

Prototype:	`void set_result_delay()`
Arguments:	Verilog HDL: `ci_data_t delay` VHDL: `ci_data_t delay, bfm_id, req_if(bfm_id)`

continued...

Returns:	`void`
Description:	Sets the instruction result delay.
Language support:	Verilog HDL, VHDL

17.2.26. set_result_err_inject()

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

17.2.27. set_result_value()

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

17.2.28. signal_fatal_error

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

17.2.29. signal_instructions_inconsistent

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

17.2.30. signal_known_instruction_received

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

17.2.31. signal_result_done

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

17.2.32. signal_result_driven

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

17.2.33. signal_unknown_instruction_received

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

18. Avalon-ST Verilog HDL Testbench

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

Click this link to download the testbench Avalon Verification IP Suite Design Files.

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

- Quartus Prime software
- ModelSim software
- The `ug_avalon_verification.zip` file. This `.zip` file includes files for the both the Avalon-ST and Avalon-MM tutorials for Quartus Prime Standard and Quartus Prime Pro.

This testbench is available for Verilog HDL. Refer to the link below for an example VHDL testbench.

Related Information

Avalon-MM VHDL Testbench Description on page 192

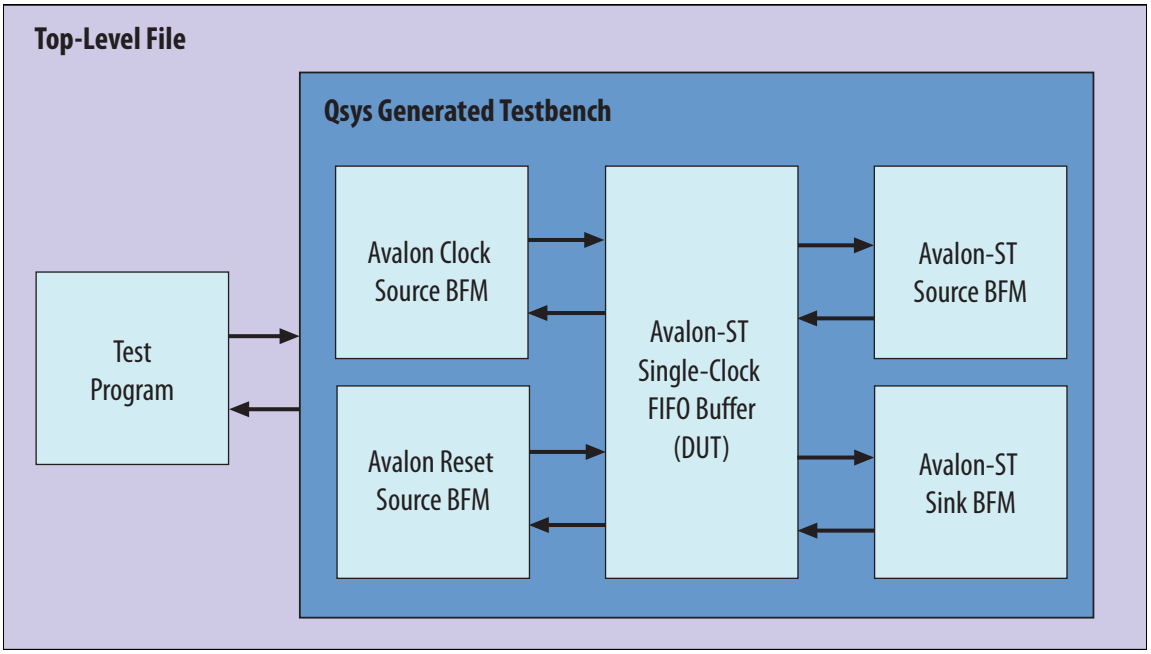
18.1. Verifying Avalon-ST DUT

The following figure shows the testbench to verify the Avalon-ST Single-Clock FIFO buffer using the Avalon-ST Source and Sink BFMs.

The testbench includes the following components:

- The Avalon Clock Source BFM provides the clock to the DUT.
- The Reset Source BFM provides the reset to the DUT.
- The Avalon-ST Source BFM connects to the DUT and drives transactions.
- The Avalon-ST Sink BFM monitors transactions from the Avalon-ST Single-Clock FIFO buffer.
- The test program controls the BFMs using the BFM API to drive and monitor transactions.

Figure 26. Top-Level Testbench for Avalon-ST DUT Component



18.2. About the Test

The test flow includes the following steps:

1. The test program initializes the BFMs.
2. The test program runs the following three parallel processes:
   a. Process 1 creates and sends four test transactions to the source BFM. The transactions consist of the following six Avalon-ST signals: `data`, `channel`, `error`, `empty`, `startofpacket`, `endofpacket`, and `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. The Avalon-ST Source BFM prints the transaction values in the ModelSim transcript console.
   b. Process 2 controls the Avalon-ST Sink BFM. When the Avalon-ST Sink BFM receives a transaction, it completes the following steps:

- It reads the transaction values.
- It prints the transaction values on the ModelSim transcript console.
- It compares the values it receives to the values from the Avalon-ST Source BFM.
- It reports any mismatch in values as failures.
- During this process, the Avalon-ST Sink BFM backpressures the Avalon-ST Single-Clock FIFO buffer.

c. Process 3 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 the following condition is met: All transactions the Source BFM sends to the Single-Clock FIFO buffer match the transactions received by the Sink BFM.

18.2.1. Creating the Qsys Design

In this section you generate a testbench system in Qsys 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. Launch the Quartus Prime software.
2. On the File menu, click **Open**. Select **st_bfm_project.qpf** located in `<working_directory>/ug_avalon_verification/avst/<quatus_version>`.
3. On the Tools menu, click **Qsys** or **Qsys Pro**.
4. On the Qsys File menu, open **st_bfm_qsys_tutorial.qsys**. This is a blank Qsys system.

   For Quartus Prime Pro, you must also specify the **st_bfm_project.qpf** project file.
5. Type `fifo` in the IP Catalog search box. From the search results, select the **Avalon-ST Single Clock FIFO** .
6. In the parameter editor, change the parameter values to match the values listed in the following table.

Table 25. Avalon-ST Single Clock FIFO Parameter Values

Parameters	Value
Symbols per beat	**4**
Bits per symbol	**8**
FIFO depth	**2**
Channel width	**3**

continued...

Parameters	Value
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**
Enable explicit maxChannel	**Off**
Explicit maxChannel	**Off**

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

Table 26. Avalon-ST Single Clock FIFO Exported Interface Names

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

10. Save **st_bfm_qsys_tutorial.qsys**.

18.2.2. Generating a Qsys Testbench System

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

1. On the **Generation** tab, select **Generate Testbench System**.
2. Change the parameter values to match the values listed in the table.

Table 27. Generation Tab Parameter Values

Parameters	Value
Simulation	
Create testbench Qsys system	**Standard, BFMs for standard Qsys Interfaces**
Create simulation model	**Verilog**
Output Directory	
Testbench	Accept the path specified. (This path is not shown for the Quartus Prime Pro Edition software.)

3. Click **Generate**. Save the system if you are prompted to do so. Click **Close**.

18.2.3. Running the Simulation

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

1. Navigate to <*working_directory*> `avst/user_test_program_`<*quartus_version*>. <*quartus_version*> is *classic* for Quartus Prime Standard and *pro* for Quartus Prime Pro.
2. Start the ModelSim software.
3. On the Compile menu, click **Compile Options**.
4. Click the **Verilog & System Verilog** tab.
5. In the **Language Syntax** box, select **Use SystemVerilog** and click **OK**.
6. On the File menu, click **Load > Macro File**.

   Note: Ensure you activate your cursor in the Transcript window, otherwise the **Load** function is disabled.
7. Select **load_sim.tcl,** and click **Open**. The Tcl file creates a new working library, compiles all source files and loads signals into the ModelSim waveform viewer.
8. To run the simulation, type the following command in the ModelSim transcript console:

   ```
   run 1200 ns
   ```

   The Transcript window displays text messages tracking the simulation.

18.2.4. Observing the Results

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

- In the ModelSim transcript console
- In the waveforms window

The transcript ModelSim transcript provides the following information:

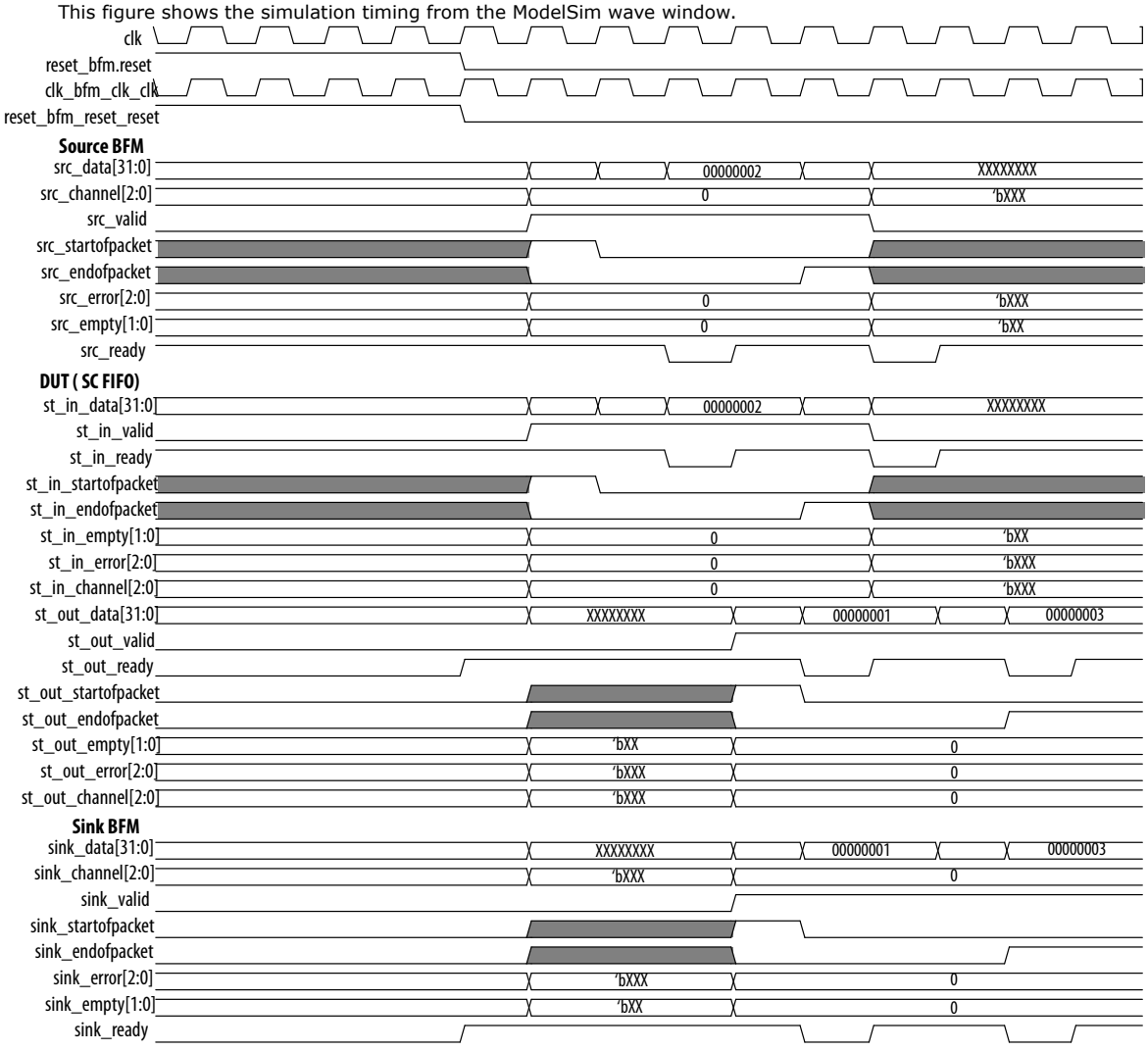
- When the Avalon-ST source BFM drives a transaction, it also prints the transaction to the ModelSim transcript window.
- The Avalon-ST Sink BFM also prints the transactions it receives on the transcript window.
- The Avalon-ST Sink BFM compares the transaction it receives with the one sent by the Avalon-ST Source BFM. The results of the comparison are printed on the transcript window.
- 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. The three-cycle delay is necessary for the transaction to propagate from the input to the output of the sink FIFO buffer.
  - The difference in values for the `idle` field is not an error. The Avalon-ST interface protocol allows source and sink components to have different latencies.

The following example shows the ModelSim transcript for the source response latency. This latency 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. It is not able to receive the transaction. As a result, the Avalon-ST Single-Clock FIFO buffer backpressures the Avalon-ST Source BFM.

```
# 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 27. Timing from ModelSim Simulation

This figure shows the simulation timing from the ModelSim wave window.



19. Avalon-MM Verilog HDL and VHDL Testbenches

The Avalon-MM testbench is available for both Verilog HDL and VHDL.

This testbench has two versions. The first version tests a single Avalon-MM Master and Slave pair. The second version includes two Avalon-MM Masters. Each Avalon-MM Master connects to two Avalon-MM Slaves.

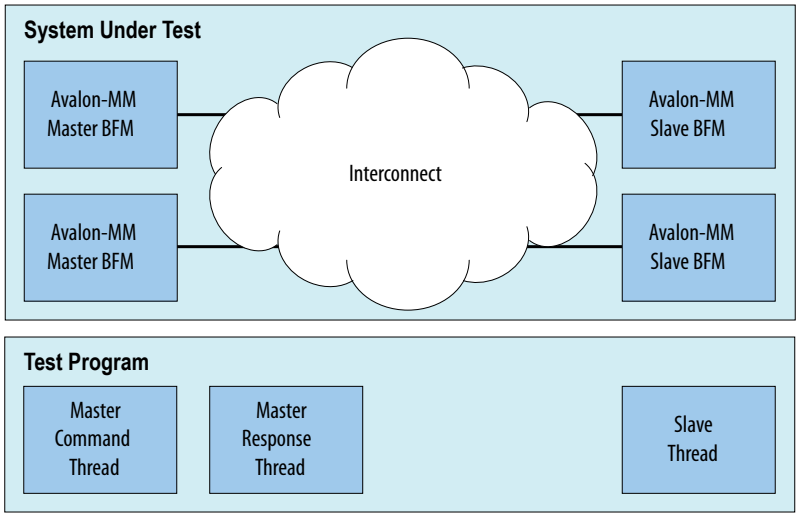
Click this link to download the testbench files from the Intel website Avalon Verification IP Suite Design Files.

19.1. Avalon-MM Verilog HDL Testbench Description

At the top-level, the Verilog HDL version of the Avalon-MM testbench includes two modules:

- The System Under Test: This module includes the Avalon-MM Master and Slaves components, the Avalon-MM Master and Slave BFMs, and the Interconnect.
- The Test Program: The module includes Master Command, Master Response, and Slave Threads.

Figure 28. Verilog HDL Testbench for Two Avalon-MM Masters and Slaves



The Master Command Thread performs the following functions:

- Generates random commands
- Passes the commands to Avalon-MM Master BFM
- Saves the commands in a FIFO for command and response verification

The Slave Thread performs the following functions:

- Randomly sets backpressure cycles to Avalon-MM Slave BFM
- Waits for valid commands
- Retrieves valid commands from the Avalon-MM Slave BFM
- Verifies commands against the expected command
- Sends read data for read commands.
- Saves read data in a FIFO for verification

The Master Response Thread performs the following functions:

- Waits for valid read data responses
- Retrieves read responses from the BFM
- Verifies the read response against the expected data

The test program sends the following transaction types:

- Non-bursting writes
- Non-bursting reads
- Bursting writes
- Bursting reads

19.1.1. Running the Verilog HDL Testbench for a Single Avalon-MM Master and Slave Pair

1. Unzip `ug_avalon_verification.zip` to a working directory.
2. For the Quartus Prime Standard Edition software, open `<working_dir>/avmm/avlmm1x1_verilog/avlm_avls_1x1.qsys`.
3. For the Quaratus Prime Pro Edition software, specify the `<working_dir>/avmm/avlmm1x1_verilog_pro/avlm_avls_1x1.qpf` project file and `<working_dir>/avmm/avlmm1x1_verilog_pro/avlm_avls_1x1.qsys` system in the Open System dialog box.
4. On the Generate menu, select **Generate HDL**.
5. Specify the parameters shown in the following table:

Table 28. Generation Parameters

Parameter	Value
Synthesis	
Create HDL design files for synthesis	**Verilog**
Create timing and resource estimates for third-party EDA synthesis tools	Leave this option off
Create block symbol file (.bsf)	Leave this option on
IP-XACT	Leave this option off. (This parameter is only available in the Quartus Prime Pro Edition software.)
Simulation	

continued...

Parameter	Value
Create simulation model	**Verilog**
Output Directory	
Path	Accept the path specified. (This path is not shown for the Quartus Prime Pro Edition software.)
Clear output directories for selected generation targets	You can leave this parameter off the first time you generate.

6. Close the **Generate** window.
7. Start theModelSim simulator.
8. To run the simulation, type the following command in your working directory:

```
do run_simulation.tcl
```

This command compiles all the required HDL files, elaborates, and runs the simulation.

Example 1. Timing for a Write Burst with a Burst Count of Four

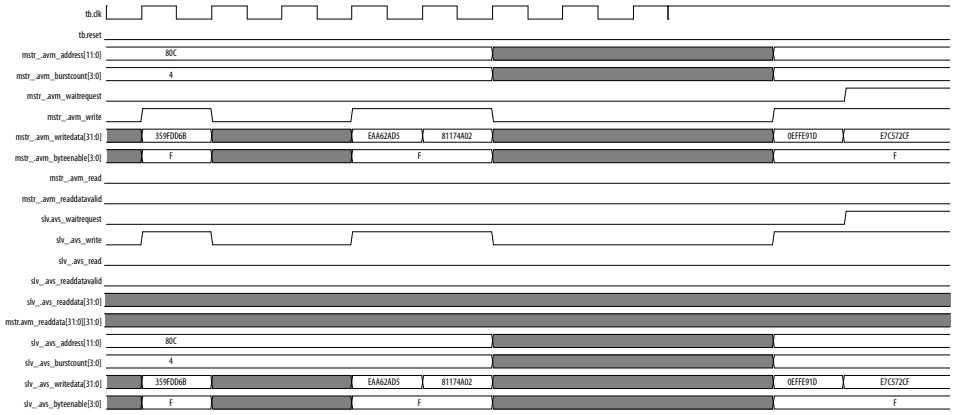
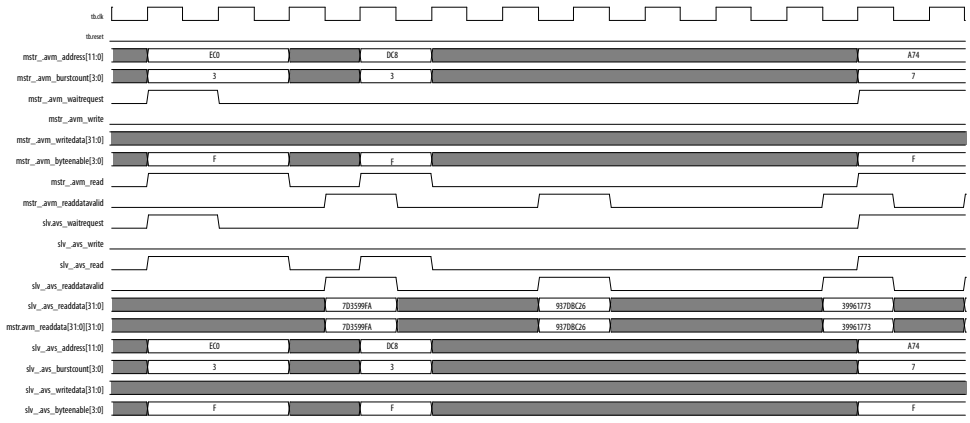


Figure 29. Timing for a Read with Burst Count of Three



19.1.2. Running the Verilog HDL Testbench for the Two Avalon-MM Masters and Slaves

1. Unzip `ug_avalon_verification.zip` to a working directory.
2. For the Quartus Prime Standard Edition, open `<working_dir>/avmm/avlmm_2x2_verilog/avlm_avls_2x2.qsys`.
3. For the Quartus Prime Pro Edition, specify the `<working_dir>/avmm/avlmm2x2_verilog_pro/avlm_avls_2x2.qpf` project file and `<working_dir>/avmm/avlmm2x2_verilog_pro/avlm_avls_2x2.qsys` system in the Open System dialog box.
4. On the Generate menu, select **Generate HDL**.
5. Specify the parameters shown in the following table:

Table 29. Generation Parameters

Parameter	Value
Synthesis	
Create HDL design files for synthesis	**Verilog**
Create timing and resource estimates for third-party EDA synthesis tools	Leave this option off
Create block symbol file (.bsf)	Leave this option on
IP-XACT	Leave this option off. (This parameter is only available in the Quartus Prime Pro Edition software.)
Simulation	
Create simulation model	**Verilog**
Output Directory	
Path	Accept the path specified. (This path is not shown for the Quartus Prime Pro Edition software.)
Clear output directories for selected generation targets	You can leave this parameter off the first time you generate.

6. Close the **Generate** window.
7. Start the ModelSim simulator.
8. To run the simulation, type the following command in your working directory:

```
do run_simulation.tcl
```

This command compiles all the required HDL files, elaborates, and runs the simulation.

Figure 30. Avalon-MM Master0 and Slave0 Writes

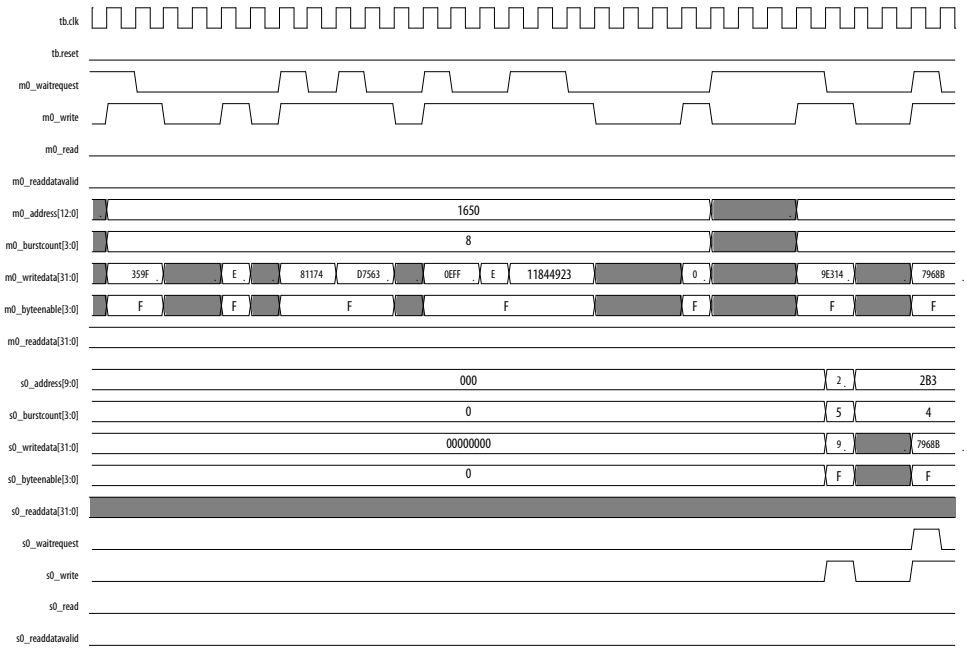
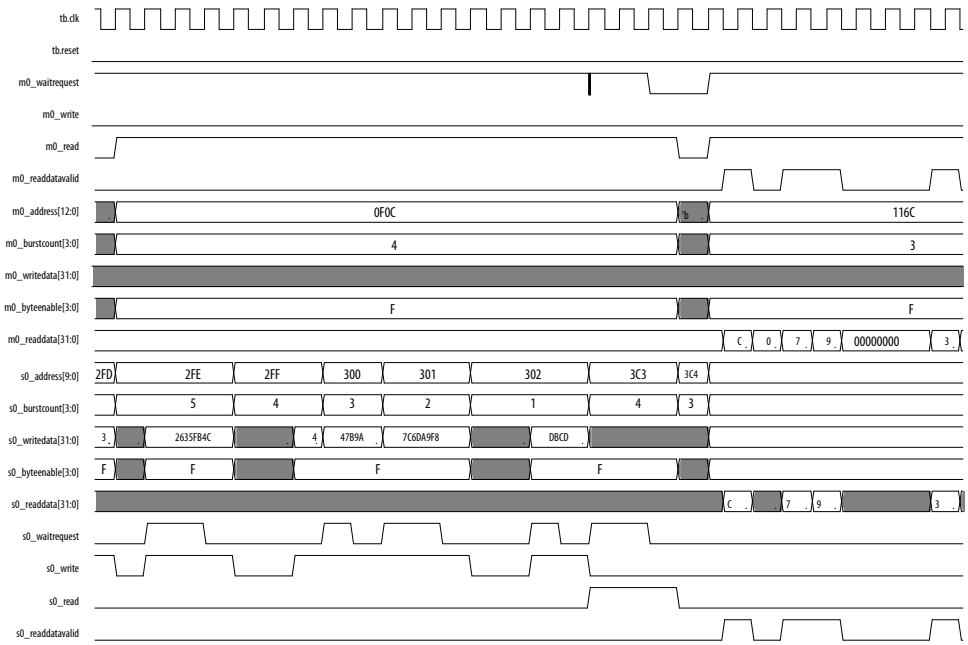


Figure 31. Avalon-MM Master0 and Slave0 Reads

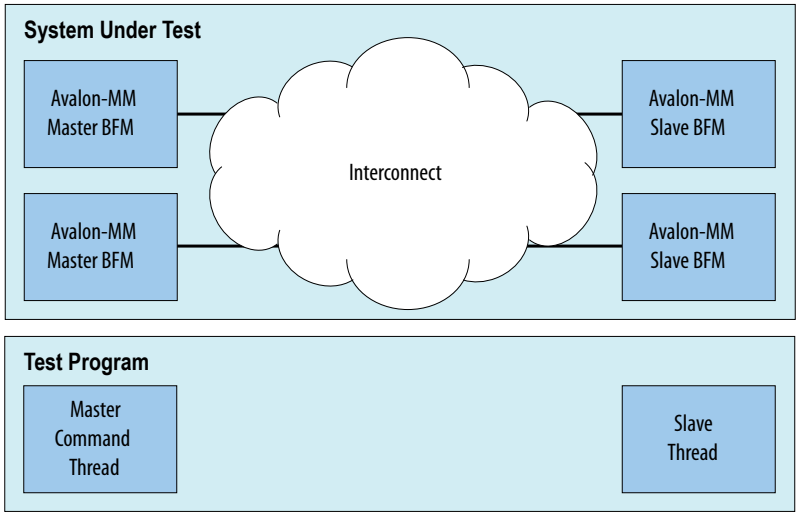


19.2. Avalon-MM VHDL Testbench Description

At the top-level, the VHDL HDL version of the Avalon-MM testbench includes two modules:

- The System Under Test: This module includes the Avalon-MM Master and Slaves components and the Avalon-MM Master and Slave BFMs.
- The Test Program: The module includes Master Command and Slave Threads.

Figure 32. VHDL Testbench for Two Avalon-MM Masters and Slaves



The Master Command Thread performs the following functions:

- Generates random commands
- Passes the commands to Avalon-MM Master BFM
- Saves the commands in a FIFO for command and response verification
- For read commands, the master waits for a valid response and verifies it against the expected read data.

The Slave Thread performs the following functions:

- Randomly sets backpressure cycles to Avalon-MM Slave BFM
- Waits for valid commands
- Retrieves valid commands from the Avalon-MM Slave BFM
- Verifies commands against the expected command
- Sends read data for read commands.
- Saves read data in a FIFO for verification

The test program sends the following transaction types:

- Non-bursting writes
- Non-bursting reads
- Bursting writes
- Bursting reads

19.2.1. Running the Testbench for a Single Avalon-MM Master and Slave Pair

1. Unzip `ug_avalon_verification.zip` to a working directory.
2. For the Quartus Prime Standard Edition software, open *<working_dir>*`/avlmm_1x1_vhdl/avlm_avls_1x1.qsys`.
3. For the Quartus Prime Pro Edition software, specify the *<working_dir>*`/avlmm1x1_vhdl_pro/avlm_avls_1x1.qpf` project and *<working_dir>*`/avlmm1x1_vhdl_pro/avlm_avls_1x1.qsys` system in the Open System dialog box.
4. Complete the following steps to generate the testbench:
   a. On the Generate menu, select **Generate HDL**.
   b. Specify the parameters shown in the following table:

Table 30. Generation Parameters

Parameter	Value
Synthesis	
Create HDL design files for synthesis	**VHDL**
Create timing and resource estimates for third-party EDA synthesis tools	Leave this option off
Create block symbol file (.bsf)	Leave this option on
Simulation	
Create simulation model	**VHDL**
Output Directory	
Path	Accept the path specified. (This path is not shown for the Quartus Prime Pro Edition software.)

   c. Click **Generate**.
   The Qsys **Generate** window displays informational messages as it generates the testbench.
   d. Close the **Generate** window.
5. To provide versioned libraries for the VHDL RTL simulation, run the following command:

   ./update.sh

   You must rerun `./update.sh` each time you regenerate the testbench.
6. Start the ModelSim simulator.
7. To run the simulation, type the following command in your working directory:

```
do run_simulation.tcl
```

   This command compiles all the required HDL files, elaborates, and runs the simulation.

Figure 33. Timing for a Write Burst with a Burst Count of Five

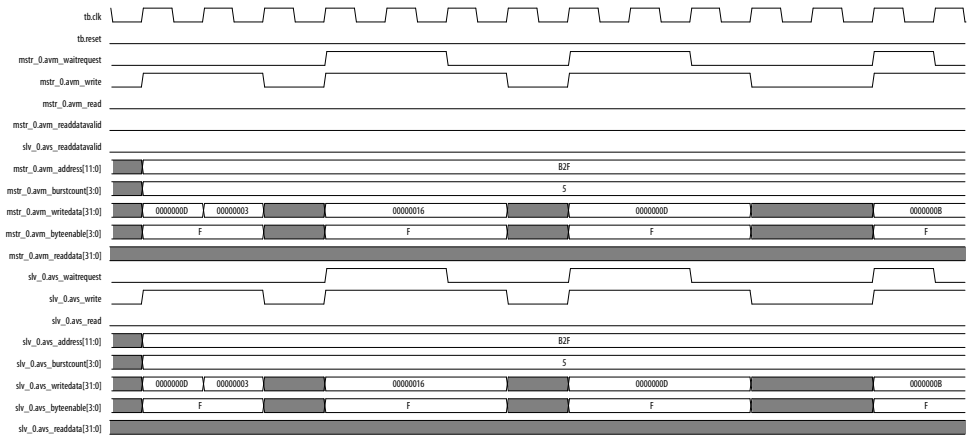
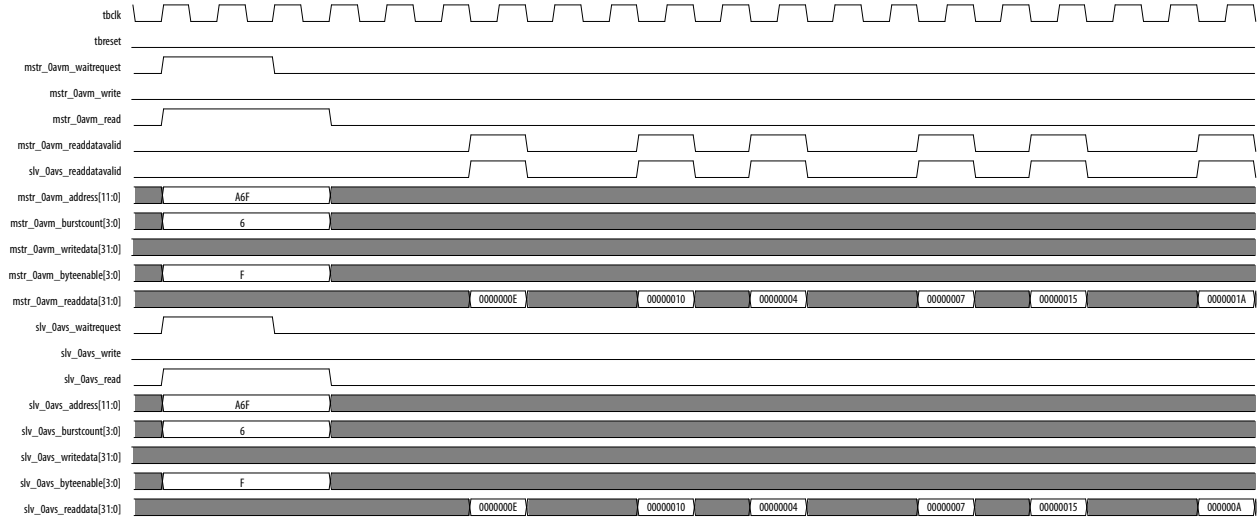


Figure 34. Timing for a Read with a Burst Count of Six



19.2.2. Running the Testbench for Two Avalon-MM Masters Slaves

1. Unzip `ug_avalon_verification.zip` to a working directory.
2. For the Quartus Prime Standard Edition software, open <*working_dir*>/`avlmm_2x2_vhdl/avlm_avls_2x2.qsys`.
3. For the Quartus Prime Pro Edition software, specify the <*working_dir*>/`avlmm12x2_vhdl_pro/avlm_avls_2x2.qpf` project and <*working_dir*>/`avlmm2x2_vhdl_pro/avlm_avls_2x2.qsys` system in the Open System dialog box.
4. Complete the following steps to generate the testbench:
   a. On the Generate menu, select **Generate HDL**.
   b. Specify the parameters shown in the following table:

Table 31. Generation Parameters

Parameter	Value
Synthesis	
Create HDL design files for synthesis	**VHDL**
Create timing and resource estimates for third-party EDA synthesis tools	Leave this option off
Create block symbol file (.bsf)	Leave this option on
IP-XACT	Leave this option off. (This parameter is only available in the Quartus Prime Pro Edition software.)
Simulation	
Create simulation model	**VHDL**
Output Directory	
Path	Accept the path specified. (This path is not shown for the Quartus Prime Pro Edition software.)
Clear output directories for selected generation targets	You can leave this parameter off the first time you generate.

c. Click **Generate**.
The Qsys **Generate** window displays informational messages as it generates the testbench.

d. Close the **Generate** window.

5. To provide versioned libraries for the VHDL RTL simulation, run the following command:

   ./update.sh

   You must rerun `./update.sh` each time you regenerate the testbench.

6. Start the ModelSim simulator.

7. To run the simulation, type the following command in your working directory:

   ```
   do run_simulation.tcl
   ```

   This command compiles all the required HDL files, elaborates, and runs the simulation.

Figure 35. Timing for a Write Burst with a Burst Count of Four

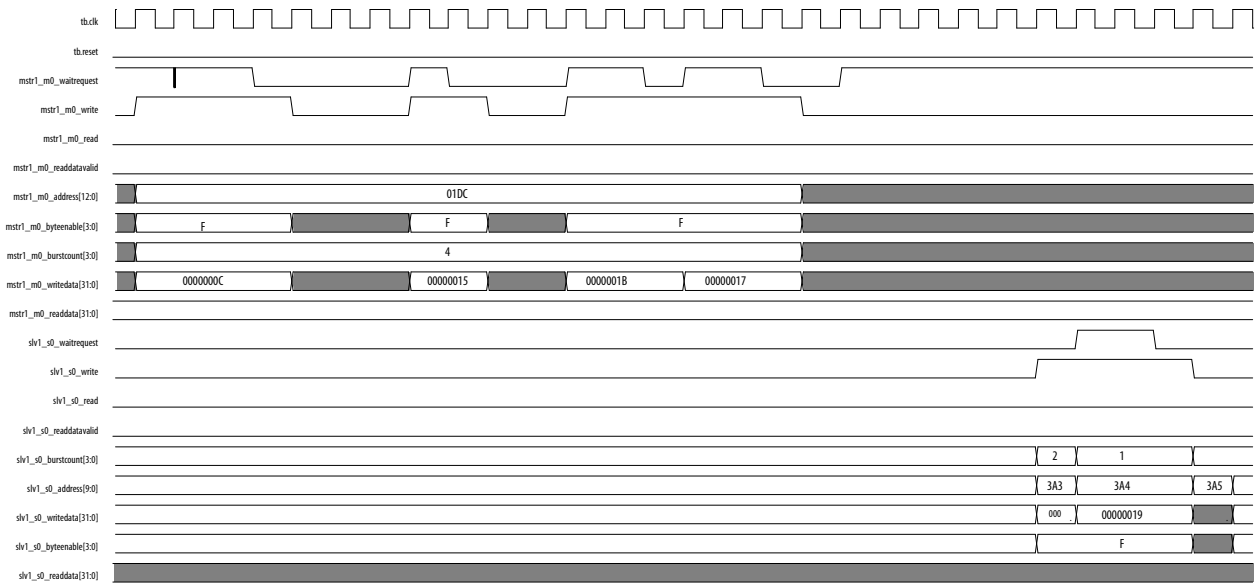
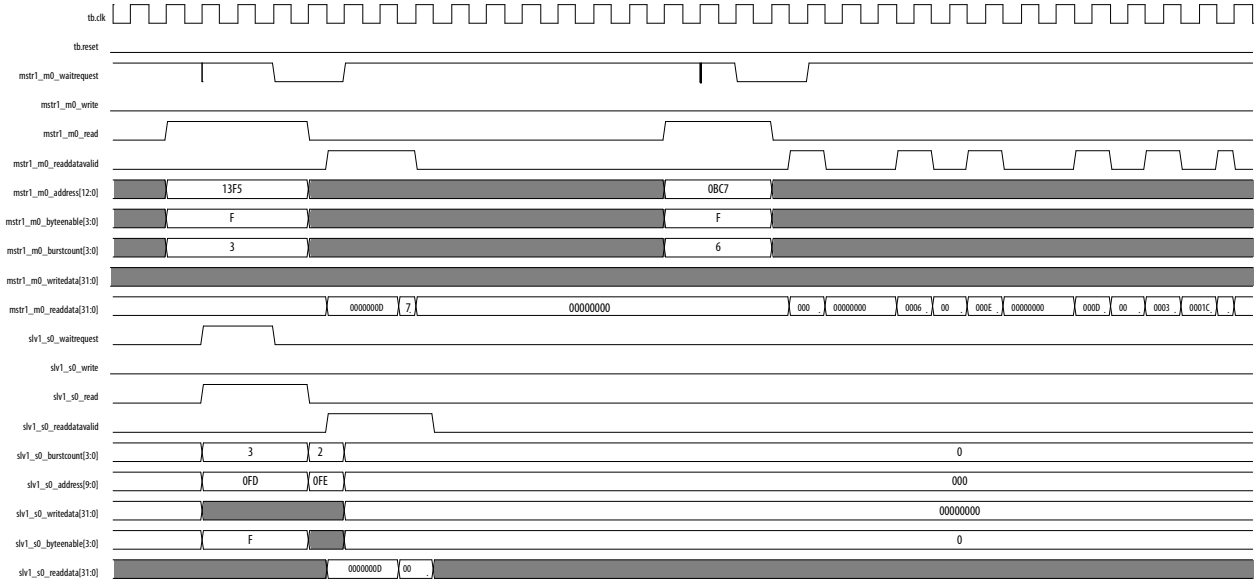


Figure 36. Timing for a Read with a Burst Count of Three



19.3. Using the VHDL BFMs

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

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

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

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

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

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

1. Generate a VHDL simulation model of the testbench design.
2. Create a custom test program.
3. Compile and load the Qsys design and testbench in a simulator.
4. Run the simulation.

Related Information

Avalon-ST Verilog HDL Testbench on page 181

20. Document Revision History

The following table shows the revision history for this document.

Document Version	Intel Quartus Prime Version	Changes
2021.01.29	20.4	Replaced the get_response_write_response() API with get_write_response_status() in the *Avalon-MM Master BFM* chapter. Updated the API set_write_response_status() in the *Avalon-MM Slave BFM* chapter.
2020.12.14	20.4	Added three new APIs to the *Avalon Streaming Credit Source BFM* chapter: get_outstanding_credit(), set_transaction_channel() and set_transaction_error(). Also added three new APIs to the *Avalon Streaming Credit Sink BFM* chapter: get_outstanding_credit(), get_transaction_channel() and get_transaction_error().
2020.04.24	20.1	Added the chapters *Avalon Streaming Credit Source BFM* and *Avalon Streaming Credit Sink BFM*.
2019.04.03	16.1	Removed VHDL from the Language Support field for the APIs clock_stop(), get_run_state() and get_version().
2016.12.13	16.1	Made the following changes: • Restored missing Avalon-ST files to `ug_avalon_verification.zip`. • Added support for Quartus Prime Pro Edition software. • Simplified design creation for both the Avalon-ST and Avalon-MM designs. • Corrected minor errors and typos.
2015.06.04	15.1	Made the following changes: • Updated the definition of `set_response_latency` for the Avalon-MM slave BFM. • Corrected language support for the Clock Source BFM and Reset Source BFM. These BFMs support both Verilog HDL and VHDL. • Updated the name of the `.zip` file of example designs to `ug_avalon_verification.zip`.
June 2014	3.3	Made the following changes: • Revised the VHDL API arguments. The following changes were made for all VHDL procedures: — Removed the `req_if` argument from `event_*` procedures. — Changed the `req_if` argument to `req_if(bfm_id)` for all `set_*`, `get_*`, `push_*`, and `pop_*` procedures. — Changed the first argument to the `get_*` procedures to the return value. • Added support for VHDL for the conduit and tri-state conduit interfaces. • Updated Qsys tutorial to work in with Quartus II 14.0 software. • Added *Avalon-MM Testbenches for Verilog HDL and VHDL*. This chapter provides the following testbenches: — Verilog HDL testbench for single Avalon-MM Master and Slave pair. — Verilog HDL testbench for 2 Avalon-MM Masters that both connect to 2 Avalon-MM Slaves. — VHDL testbench for single Avalon-MM Master and Slave pair. — Verilog HDL testbench for 2 Avalon-MM Masters that both connect to 2 Avalon-MM Slaves. • Reformatted.
May 2013	3.2	Added information on VHDL support for verification IP.

continued...

Document Version	Intel Quartus Prime Version	Changes
		Removed information on the following wrappers. These wrappers are not supported in the Quartus II software version 13.0 and higher. • Avalon-MM master BFM with Avalon-ST API wrapper • Avalon-MM slave BFM with Avalon-ST API wrapper • Avalon-ST source BFM with Avalon-ST API wrapper • Avalon-ST sink BFM with Avalon-ST API wrapper Removed references to SOPC Builder.
June 2012	3.1	• Updated SOPC Tutorial chapter. • Updated Qsys Tutorial chapter.