AXI Bus Functional Models v2.1

DS824 April 24, 2012 **Product Specification**

Introduction

The AXI Bus Functional Models (BFMs), developed for Xilinx by Cadence Design Systems, support the simulation of customer-designed AXI-based IP. AXI BFMs support all versions of AXI (AXI3, AXI4, AXI4-Lite and AXI4-Stream). The BFMs are delivered as encrypted Verilog modules. BFM operation is controlled by using a sequence of Verilog tasks contained in a Verilog-syntax text file.

Features

- Supports all protocol data widths and address widths, transfer types and responses
- Transaction-level protocol checking (burst type, length, size, lock type, cache type)
- Behavioral Verilog Syntax
- Verilog Task-based API
- Delivered in ISE® Design Suite, enabled by a Xilinx-generated license

LogiCORE™ IP Facts Table	
Core Specifics	
Supported User Interfaces[1]	AXI4, AXI4-Lite, AXI4-Stream, AXI3
Provided with Core	
Documentation	Product Brief Product Specification
Example Design	Verilog VHDL
Test Bench	Verilog VHDL
Simulation Model	Encrypted Verilog
Supported S/W Driver	N/A
Tested Design Tools	
Design Entry Tools	CORE Generator Xilinx Platform Studio (XPS)
Simulation [2][3]	Aldec Riviera-PRO 2011.10 or later Cadence Incisive Enterprise Simulator (IES) Mentor Graphics ModelSim Synopsys VCS ISim
Support	
Provided by Xilinx @ www.xilinx.com/support	

1. For a complete list of supported derivative devices, see the IDS Embedded Edition Derivative Device Support.
2. Windows XP 64-bit is not supported.
3. For the supported versions of the tools, see the ISE Design Suite 14: Release Notes Guide.

© Copyright 2011–2012 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. ARM is a registered trademark of ARM in the EU and other countries. The AMBA trademark is a registered trademark of ARM Limited. All other trademarks are the property of their respective owners.

Overview

The general AXI BFM architecture is shown in Figure 1.

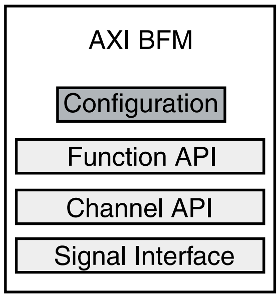


Figure 1: **AXI BFM Architecture**

All of the AXI BFMs consist of three main layers: the signal interface, the channel API and the function API. The signal interface includes the typical Verilog input/output ports and associated signals. The channel API is a set of defined Verilog tasks (see Test Writing API, page 14) that operate at the basic transaction level inherent in the AXI protocol, including:

- Read Address Channel
- Write Address Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

This split enables the tasks associated with each channel to be executed concurrently or sequentially. This allows the test writer to control and implement out of order transfers, interleaved data transfers, and other features.

The next level up in the API hierarchy is the function level API (see Test Writing API, page 14). This level has complete transaction level control; for example, a complete AXI read burst process is encapsulated in a single Verilog task.

One final but important piece of the AXI BFM architecture is the configuration mechanism. This is implemented using Verilog parameters and/or BFM internal variables and is used to set the address bus width, data bus width and other parameters. The reason Verilog parameters are used instead of defines is so that each BFM can be configured separately within a single test bench. For example, it is reasonable to have an AXI master that has a different data bus width than one of the slaves it is attached to (in this case the interconnect needs to handle this). BFM internal variables are used for configuration variables that maybe changed during simulation. For a complete list of configuration options, see Configuration Options, page 3.

The intended use of the AXI BFM is shown in Figure 2.

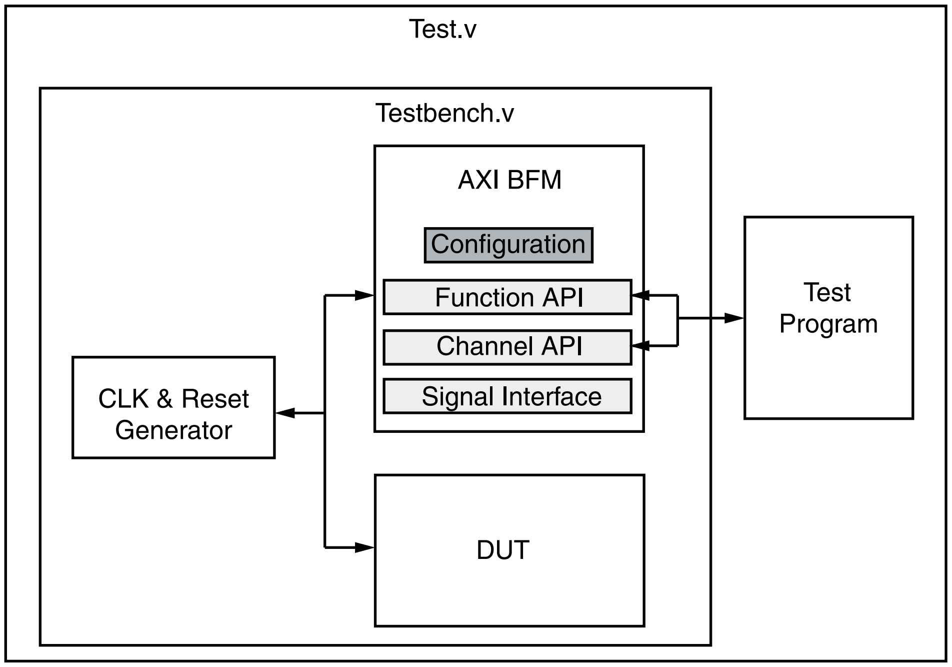


Figure 2: **AXI BFM Use**

Figure 2 shows a single AXI BFM. However, the test bench can contain multiple instances of AXI BFMs. The DUT and the AXI BFMs are instantiated in a test bench that also contains a clock and reset generator. Then, the test writer instantiates the test bench into the test module and creates a test program using the BFM API layers. The test program would call API tasks either sequentially or concurrently using fork and join. See AXI BFM Example Designs, page 35 for practical examples of test programs and test bench setups.

Configuration Options

In most cases, the configuration options are passed to the BFM through Verilog parameters. BFM internal variables are used for options that can be dynamically controlled by the test writer because Verilog parameters do not support run time modifications.

To change the BFM internal variables during simulation, the correct BFM API task should be called. For example, to change the CHANNEL_LEVEL_INFO from 0 to 1, the **`set_channel_level_info(1)`** task call should be made. For more information on the API for changing internal variables, see Test Writing API, page 14.

AXI3 BFMs

The AXI3 BFMs modules and files are named as follows:

- MASTER BFM
  - Module Name: cdn_axi3_master_bfm
  - File Name: `cdn_axi3_master_bfm.v`
- SLAVE BFM
  - Module Name: cdn_axi3_slave_bfm
  - File Name: `cdn_axi3_slave_bfm.v`

AXI3 Master BFM

Table 1 contains a list of parameters and configuration variables supported by the AXI3 Master BFM.

Table 1: **AXI3 Master BFM Parameters**

BFM Parameters	Description
NAME	String name for the master BFM. This is used in the messages coming from the BFMs. The default for the master BFM is “MASTER_0”.
DATA_BUS_WIDTH	Read and write data buses can be 8, 16, 32, 64, 128, 256, 512 or 1024 bits wide. Default is 32.
ADDRESS_BUS_WIDTH	Default is 32.
ID_BUS_WIDTH	Default is 4.
MAX_OUTSTANDING_TRANSACTIONS	This defines the maximum number of outstanding transactions. Any attempt to generate more traffic while this limit has been reached is handled by stalling until at least one of the outstanding transactions has finished. Default is 8.
EXCLUSIVE_ACCESS_SUPPORTED	This parameter informs the master that exclusive access is supported by the slave. A value of 1 means it is supported so the response check expects an EXOKAY, or else give a warning, in response to an exclusive access. A value of 0 means the slave does not support this so a response of OKAY is expected in response to an exclusive access. Default is 1.
WRITE_BURST_DATA_TRANSFER_GAP	The configuration variable can be set dynamically during the run of a test. It controls the gap between the write data transfers that comprise a write data burst. This value is an integer number and is measured in clock cycles. Default is 0. NOTE: If this is set to a value greater than zero *and* concurrent write bursts are called. Then write data interleaving occurs. The depth of this data interleaving depends on the number of parallel writes being performed.
RESPONSE_TIMEOUT	This value, measured in clock cycles, is the value used to determine if a task that is waiting for a response should timeout. Default is 500 clock cycles. A value of zero means that the timeout feature is disabled.
STOP_ON_ERROR	This configuration variable is used to enable/disable the stopping of the simulation on an error condition. The default (1) means stop on error. This configuration variable can be changed during simulation for error testing. NOTE: This is *not* used for timeout errors; such errors always stop simulation.
CHANNEL_LEVEL_INFO	This configuration variable controls the printing of channel level information messages. When set to 1 info messages are printed, when set to zero no channel level information is printed. Default (0) means channel level info messages are disabled.
FUNCTION_LEVEL_INFO	This configuration variable controls the printing of function level information messages. When set to 1 info messages are printed, when set to zero no function level information is printed. Default (1) means function level info messages are enabled.

AXI3 Slave BFM

Table 2 contains a list of parameters and configuration variables supported by the AXI3 Slave BFM:

Table 2: **AXI3 Slave BFM Parameters**

BFM Parameters	Description
NAME	String name for the slave BFM. This is used in the messages coming from the BFMs. The default for the slave BFM is “SLAVE_0”.
DATA_BUS_WIDTH	Read and write data buses can be 8, 16, 32, 64, 128, 256, 512 or 1024 bits wide. Default is 32.
ADDRESS_BUS_WIDTH	Default is 32.
ID_BUS_WIDTH	Slaves can have different ID bus widths compared to the master. The default is 4.
SLAVE_ADDRESS	This is the start address of the slave’s memory range.
SLAVE_MEM_SIZE	This is the size of the memory that the slave models. Starting from address = SLAVE_ADDRESS. This is measured in bytes therefore a value of 4096 = 4K. The default value is 4 bytes, meaning, one 32-bit entry.
MAX_OUTSTANDING_TRANSACTIONS	This defines the maximum number of outstanding transactions. Any attempt to generate more traffic while this limit has been reached is handled by stalling until at least one of the outstanding transactions has finished. Default is 8.
MEMORY_MODEL_MODE	The parameter puts the slave BFM into a simple memory model mode. This means that the slave BFM automatically responds to all transfers and does not require any of the API functions to be called by the test. The memory mode is very simple and only supports aligned and normal INCR transfers. Narrow transfers are not supported, and WRAP and FIXED bursts are also not supported. The size and address range of the memory are controlled by the parameters SLAVE_ADDRESS and SLAVE_MEM_SIZE. The value 1 enables this memory model mode. A value of 0 disables it. Default is 0. The slave channel level API and function level API should not be used while this mode is active.
EXCLUSIVE_ACCESS_SUPPORTED	This parameter informs the slave that exclusive access is supported. A value of 1 means it is supported so the automatic generated response is an EXOKAY to exclusive accesses. A value of 0 means the slave does not support this so a response of OKAY is automatically generated in response to exclusive accesses. Default is 1.
READ_BURST_DATA_TRANSFER_GAP	The configuration variable controls the gap between the read data transfers that comprise a read data burst. This value is an integer number and is measured in clock cycles. Default is 0. NOTE: If this is set to a value greater than zero and concurrent read bursts are called, read data interleaving occurs. The depth of this data interleaving depends on the number of parallel writes being performed. This configuration variable can be changed during simulation.
WRITE_RESPONSE_GAP	This configuration variable controls the gap, measured in clock cycles, between the reception of the last write transfer and the write response. Default is 0. NOTE: This configuration variable can be changed during simulation.

Table 2: **AXI3 Slave BFM Parameters *(Cont'd)***

BFM Parameters	Description
READ_RESPONSE_GAP	This configuration variable controls the gap, measured in clock cycles, between the reception of the read address transfer and the start of the first read data transfer. Default is 0. NOTE: This configuration variable can be changed during simulation.
RESPONSE_TIMEOUT	This configuration variable, measured in clock cycles, is the value used to determine if a task that is waiting for a response should timeout. Default = 500 clock cycles. A value of zero means that the timeout feature is disabled. This configuration variable can be changed during simulation.
STOP_ON_ERROR	This configuration variable is used to enable/disable the stopping of the simulation on an error condition. The default value of one stops the simulation on an error. This configuration variable can be changed during simulation for error testing. NOTE: This is *not* used for timeout errors; such errors always stop simulation.
CHANNEL_LEVEL_INFO	This configuration variable controls the printing of channel level information messages. When set to 1 info messages are printed; when set to zero no channel level information is printed. The default (0) disables the channel level info messages.
FUNCTION_LEVEL_INFO	This configuration variable controls the printing of function level information messages. When set to 1 info messages are printed; when set to zero no function level information is printed. The default (1) enables the function level info messages.

AXI4 BFMs

The AXI4 BFMs modules and files are named as follows:

- Full Master BFM
  - Module Name: cdn_axi4_master_bfm
  - File Name: `cdn_axi4_master_bfm.v`
- Full Slave BFM
  - Module Name: cdn_axi4_slave_bfm
  - File Name: `cdn_axi4_slave_bfm.v`
- Lite Master BFM
  - Module Name: cdn_axi4_lite_master_bfm
  - File Name: `cdn_axi4_lite_master_bfm.v`
- Lite Slave BFM
  - Module Name: cdn_axi4_lite_slave_bfm
  - File Name: `cdn_axi4_lite_slave_bfm.v`
- Streaming Master BFM
  - Module Name: cdn_axi4_streaming_master_bfm
  - File Name: `cdn_axi4_streaming_master_bfm.v`
- Streaming Slave BFM
  - Module Name: cdn_axi4_streaming_slave_bfm
  - File Name: `cdn_axi4_streaming_slave_bfm.v`

AXI4 Master BFM

Table 3 contains a list of parameters and configuration variables supported by the AXI4 Master BFM.

Table 3: **AXI4 Master BFM Parameters**

BFM Parameters	Description
NAME	String name for the master BFM. This is used in the messages coming from the BFMs. The default for the master BFM is "MASTER_0".
DATA_BUS_WIDTH	Read and write data buses can be 8, 16, 32, 64, 128, 256, 512 or 1024 bits wide. Default is 32.
ADDRESS_BUS_WIDTH	Default is 32.
ID_BUS_WIDTH	Default is 4.
AWUSER_BUS_WIDTH	Default is 1.
ARUSER_BUS_WIDTH	Default is 1.
RUSER_BUS_WIDTH	Default is 1.
WUSER_BUS_WIDTH	Default is 1.
BUSER_BUS_WIDTH	Default is 1.
MAX_OUTSTANDING_TRANSACTIONS	This defines the maximum number of outstanding transactions. Any attempt to generate more traffic while this limit has been reached is handled by stalling until at least one of the outstanding transactions has finished. Default is 8.
EXCLUSIVE_ACCESS_SUPPORTED	This parameter informs the master that exclusive access is supported by the slave. A value of 1 means it is supported so the response check expects an EXOKAY, or else give a warning, in response to an exclusive access. A value of 0 means the slave does not support this so a response of OKAY is expected in response to an exclusive access. Default is 1.
WRITE_BURST_DATA_TRANSFER_GAP	The configuration variable can be set dynamically during the run of a test. It controls the gap between the write data transfers that comprise a write data burst. This value is an integer number and is measured in clock cycles. Default is 0. NOTE: If this is set to a value greater than zero and concurrent read bursts are called, then the BFM attempts to perform read data interleaving.
WRITE_BURST_ADDRESS_DATA_ PHASE_GAP	This configuration variable can be set dynamically during the run of a test. It controls the gap between the write address phase and the write data burst inside the WRITE_BURST task. This value is an integer number and is measured in clock cycles. Default is 0.
WRITE_BURST_DATA_ADDRESS_ PHASE_GAP	This configuration variable can be set dynamically during the run of a test. It controls the gap between the write data burst and the write address phase inside the WRITE_BURST_CONCURRENT. This enables the user to start the address phase at anytime during the data burst. This value is an integer number and is measured in clock cycles. Default is 0.
RESPONSE_TIMEOUT	This value, measured in clock cycles, is the value used to determine if a task that is waiting for a response should timeout. Default is 500 clock cycles. A value of zero means that the timeout feature is disabled.

Table 3: **AXI4 Master BFM Parameters *(Cont'd)***

BFM Parameters	Description
STOP_ON_ERROR	This configuration variable is used to enable/disable the stopping of the simulation on an error condition. The default value of one stops the simulation on an error. This configuration variable can be changed during simulation for error testing. NOTE: This is *not* used for timeout errors; such errors always stop simulation.
CHANNEL_LEVEL_INFO	This configuration variable controls the printing of channel level information messages. When set to 1 info messages are printed, when set to zero no channel level information is printed. The default (0) disables the channel level info messages.
FUNCTION_LEVEL_INFO	This configuration variable controls the printing of function level information messages. When set to 1 info messages are printed, when set to zero no function level information is printed. The default (1) enables the function level info messages.

AXI4 Slave BFM

Table 4 contains a list of parameters and configuration variables supported by the AXI4 Slave BFM.

Table 4: **AXI4 Slave BFM Parameters**

BFM Parameters	Description
NAME	String name for the slave BFM. This is used in the messages coming from the BFMs. The default for the slave BFM is "SLAVE_0".
DATA_BUS_WIDTH	Read and write data buses can be 8, 16, 32, 64, 128, 256, 512 or 1024 bits wide. Default is 32.
ADDRESS_BUS_WIDTH	Default is 32.
ID_BUS_WIDTH	Slaves can have different ID bus widths compared to the master. The default is 4.
AWUSER_BUS_WIDTH	Default is 1.
ARUSER_BUS_WIDTH	Default is 1.
RUSER_BUS_WIDTH	Default is 1.
WUSER_BUS_WIDTH	Default is 1.
BUSER_BUS_WIDTH	Default is 1.
SLAVE_ADDRESS	This is the start address of the slave's memory range.
SLAVE_MEM_SIZE	This is the size of the memory that the slave models. Starting from address = SLAVE_ADDRESS. This is measured in bytes therefore a value of 4096 = 4K. The default value is 4 bytes (one 32 bit entry).
MAX_OUTSTANDING_TRANSACTIONS	This defines the maximum number of outstanding transactions. Any attempt to generate more traffic while this limit has been reached is handled by stalling until at least one of the outstanding transactions has finished. Default is 8.

Table 4: **AXI4 Slave BFM Parameters *(Cont'd)***

BFM Parameters	Description
MEMORY_MODEL_MODE	The parameter puts the slave BFM into a simple memory model mode. This means that the slave BFM automatically responds to all transfers and does not require any of the API functions to be called by the test. The memory mode is very simple and only supports, aligned and normal INCR transfers. Narrow transfers are not supported, and WRAP and FIXED bursts are also not supported. The size and address range of the memory are controlled by the parameters SLAVE_ADDRESS and SLAVE_MEM_SIZE. The value 1 enables this memory model mode. A value of 0 disables it. Default is 0. The slave channel level API and function level API should not be used while this mode is active.
EXCLUSIVE_ACCESS_SUPPORTED	This parameter informs the slave that exclusive access is supported. A value of 1 means it is supported so the automatic generated response is an EXOKAY to exclusive accesses. A value of 0 means the slave does not support this so a response of OKAY is automatically generated in response to exclusive accesses. Default is 1.
READ_BURST_DATA_TRANSFER_GAP	The configuration variable controls the gap between the read data transfers that comprise a read data burst. This value is an integer number and is measured in clock cycles. Default is 0. NOTE: If this is set to a value greater than zero *and* concurrent read bursts are called, then AXI4 protocol is violated as the BFM attempts to perform data interleaving.
WRITE_RESPONSE_GAP	This configuration variable controls the gap, measured in clock cycles, between the reception of the last write transfer and the write response. Default is 0. This configuration variable can be changed during simulation.
READ_RESPONSE_GAP	This configuration variable controls the gap, measured in clock cycles, between the reception of the read address transfer and the start of the first read data transfer. Default is 0. This configuration variable can be changed during simulation.
RESPONSE_TIMEOUT	This configuration variable, measured in clock cycles, is the value used to determine if a task that is waiting for a response should timeout. Default = 500 clock cycles. A value of zero means that the timeout feature is disabled. This configuration variable can be changed during simulation.
STOP_ON_ERROR	This configuration variable is used to enable/disable the stopping of the simulation on an error condition. The default value of 1 stops the simulation on an error. This configuration variable can be changed during simulation for error testing. This is *not* used for timeout errors; such errors always stop simulation.
CHANNEL_LEVEL_INFO	This configuration variable controls the printing of channel level information messages. When set to 1 info messages are printed; when set to zero no channel level information is printed. The default (0) disables the channel level info messages.
FUNCTION_LEVEL_INFO	This configuration variable controls the printing of function level information messages. When set to 1 info messages are printed; when set to zero no function level information is printed. The default (1) enables the function level info messages.

AXI4-Lite Master BFM

Table 5 contains a list of parameters and configuration variables which are supported by the AXI4-Lite Master BFM.

Table 5: **AXI4-Lite Master BFM Parameters**

BFM Parameters	Description
NAME	String name for the master BFM. This is used in the messages coming from the BFMs. The default for the master BFM is "MASTER_0".
DATA_BUS_WIDTH	Read and write data buses can 32 or 64 bits wide only. Default is 32.
ADDRESS_BUS_WIDTH	Default is 32.
MAX_OUTSTANDING_TRANSACTIONS	This defines the maximum number of outstanding transactions. Any attempt to generate more traffic while this limit has been reached is handled by stalling until at least one of the outstanding transactions has finished. Default is 8.
RESPONSE_TIMEOUT	This value, measured in clock cycles, is the value used to determine if a task that is waiting for a response should timeout. Default is 500 clock cycles. A value of zero means that the timeout feature is disabled.
STOP_ON_ERROR	This configuration variable is used to enable/disable the stopping of the simulation on an error condition. The default value of one stops the simulation on an error. This configuration variable can be changed during simulation for error testing. NOTE: This is *not* used for timeout errors; such errors always stop simulation.
CHANNEL_LEVEL_INFO	This configuration variable controls the printing of channel level information messages. When set to 1 info messages are printed, when set to zero no channel level information is printed. The default (0) disables the channel level info messages.
FUNCTION_LEVEL_INFO	This configuration variable controls the printing of function level information messages. When set to 1 info messages are printed, when set to zero no function level information is printed. The default (1) enables the function level info messages.

AXI4-Lite Slave BFM

Table 6 contains a list of parameters and configuration variables which are supported by the AXI4-Lite Slave BFM.

Table 6: **AXI4-Lite Slave BFM Parameters**

BFM Parameters	Description
NAME	String name for the slave BFM. This is used in the messages coming from the BFMs. The default for the slave BFM is "SLAVE_0".
DATA_BUS_WIDTH	Read and write data buses can be 32 or 64 bits wide only. Default is 32.
ADDRESS_BUS_WIDTH	Default is 32.
SLAVE_ADDRESS	This is the start address of the slave's memory range
SLAVE_MEM_SIZE	This is the size of the memory that the slave models. Starting from address = SLAVE_ADDRESS. This is measured in bytes therefore a value of 4096 = 4K. The default value is 4 bytes, that is, one 32 bit entry.

Table 6: **AXI4-Lite Slave BFM Parameters *(Cont'd)***

BFM Parameters	Description
MAX_OUTSTANDING_TRANSACTIONS	This defines the maximum number of outstanding transactions. Any attempt to generate more traffic while this limit has been reached is handled by stalling until at least one of the outstanding transactions has finished. Default is 8.
MEMORY_MODEL_MODE	The parameter puts the slave BFM into a simple memory model mode. This means that the slave BFM automatically responds to all transfers and does not require any of the API functions to be called by the test. The memory mode is very simple and only supports, aligned and normal INCR transfers. Narrow transfers are not supported, and WRAP and FIXED bursts are also not supported. The size and address range of the memory are controlled by the parameters SLAVE_ADDRESS and SLAVE_MEM_SIZE. The value 1 enables this memory model mode. A value of 0 disables it. Default is 0. The slave channel level API and function level API should not be used while this mode is active.
WRITE_RESPONSE_GAP	This configuration variable controls the gap, measured in clock cycles, between the reception of the last write transfer and the write response. Default is 0. This configuration variable can be changed during simulation.
READ_RESPONSE_GAP	This configuration variable controls the gap, measured in clock cycles, between the reception of the read address transfer and the start of the first read data transfer. Default is 0. This configuration variable can be changed during simulation.
RESPONSE_TIMEOUT	This configuration variable, measured in clock cycles, is the value used to determine if a task that is waiting for a response should timeout. Default = 500 clock cycles. A value of zero means that the timeout feature is disabled. This configuration variable can be changed during simulation.
STOP_ON_ERROR	This configuration variable is used to enable/disable the stopping of the simulation on an error condition. The default value of one stops the simulation on an error. This configuration variable can be changed during simulation for error testing. This is *not* used for timeout errors; such errors always stop simulation.
CHANNEL_LEVEL_INFO	This configuration variable controls the printing of channel level information messages. When set to 1 info messages are printed, when set to zero no channel level information is printed. The default (0) disables the channel level info messages.
FUNCTION_LEVEL_INFO	This configuration variable controls the printing of function level information messages. When set to 1 info messages are printed, when set to zero no function level information is printed. The default (1) enables the function level info messages.

AXI4-Stream Master BFM

Table 7 contains a list of parameters and configuration variables which are supported by the AXI4-Stream Master BFM.

Table 7: **AXI4-Stream BFM Parameters**

BFM Parameters	Description
NAME	String name for the master BFM. This is used in the messages coming from the BFMs. The default for the master BFM is "MASTER_0".
DATA_BUS_WIDTH	Read and write data buses can 32 or 64 bits wide. Default is 32.
ID_BUS_WIDTH	Default is 8.
DEST_BUS_WIDTH	Default is 4.
USER_BUS_WIDTH	Default is 8
MAX_PACKET_SIZE	This parameter is an integer value that controls the maximum size of a packet. It is used to size the packet data vector. The value must be specified as an integer multiple of the DATA_BUS_WIDTH. For example, if DATA_BUS_WIDTH = 32 bits and MAX_PACKET_SIZE = 2, then the maximum packet size is 64 bits. The default value is 10.
MAX_OUTSTANDING_TRANSACTIONS	This defines the maximum number of outstanding transactions. Any attempt to generate more traffic while this limit has been reached is handled by stalling until at least one of the outstanding transactions has finished. Default is 8.
STROBE_NOT_USED	Enables and disables the strobe signals. • 0 = Strobe signals used. • 1 = Strobe signals not used. The default is 0. A value of 1 disables the associated checks.
KEEP_NOT_USED	Enables and disables keeping unused signals. • 0 = Keep signals used. • 1 = Keep signals not used. The default is 0. Changing the value to 1 disables the associated checks.
PACKET_TRANSFER_GAP	The configuration variable controls the gap between the transfers in a packet. This value is an integer number and is measured in clock cycles. The default is 0. NOTE: If this is set to a value greater than zero and concurrent SEND_PACKET tasks are called, then the BFM attempts to perform write data interleaving.
RESPONSE_TIMEOUT	This value, measured in clock cycles, is the value used to determine if a task that is waiting for a response should timeout. Default is 500 clock cycles. A value of zero means that the timeout feature is disabled.
STOP_ON_ERROR	This configuration variable is used to enable/disable the stopping of the simulation on an error condition. The default value of 1 stops the simulation on an error. This configuration variable can be changed during simulation for error testing. NOTE: This is NOT used for timeout errors; such errors always stop simulation.
CHANNEL_LEVEL_INFO	This configuration variable controls the printing of channel level information messages. When set to 1, info messages are printed, when set to zero no channel level information is printed. The default (1) enables channel level info messages.

AXI4-Stream Slave BFM

Table 8 contains a list of parameters and configuration variables which are supported by the AXI4-Stream Slave BFM.

Table 8: **AXI4-Stream Slave BFM Parameters**

BFM Parameters	Description
NAME	String name for the slave BFM. This is used in the messages coming from the BFMs. The default for the slave BFM is "SLAVE_0".
DATA_BUS_WIDTH	Read and write data buses can be 32 or 64 bits wide only. Default is 32.
ID_BUS_WIDTH	Default is 8.
DEST_BUS_WIDTH	Default is 4.
USER_BUS_WIDTH	Default is 8
MAX_PACKET_SIZE	This parameter is an integer value that controls the maximum size of a packet. It is used to size the packet data vector. The value must be specified as an integer multiple of the DATA_BUS_WIDTH. For example, if DATA_BUS_WIDTH = 32 bits and MAX_PACKET_SIZE = 2, then the maximum packet size is 64 bits. The default value is 10.
MAX_OUTSTANDING_TRANSACTIONS	This defines the maximum number of outstanding transactions. Any attempt to generate more traffic while this limit has been reached is handled by stalling until at least one of the outstanding transactions has finished. Default is 8.
STROBE_NOT_USED	Enables and disables the strobe signals. • 0 = Strobe signals used. • 1 = Strobe signals not used. The default is 0. A value of 1 only disables the associated checks.
KEEP_NOT_USED	Enables and disables keeping unused signals. • 0 = Keep signals used. • 1 = Keep signals not used. The default is 0. Changing the value to 1 only disables the associated checks.
RESPONSE_TIMEOUT	This configuration variable, measured in clock cycles, is the value used to determine if a task that is waiting for a response should timeout. Default = 500 clock cycles. A value of zero means that the timeout feature is disabled. This configuration variable can be changed during simulation.
STOP_ON_ERROR	This configuration variable is used to enable/disable the stopping of the simulation on an error condition. The default value of 1 stops the simulation on an error. This configuration variable can be changed during simulation for error testing. NOTE: This is *not* used for timeout errors; such errors always stop simulation.
CHANNEL_LEVEL_INFO	This configuration variable controls the printing of channel level information messages. When set to 1, info messages are printed, when set to zero no channel level information is printed. The default (1) enables the channel level info messages.

Test Writing API

The test writing API starts simple and is layered to implement more complex protocol features. This approach enables very complex test cases to be written. For a complete overview of the general AXI BFM architecture, see Overview, page 2.

For all functions in the API, the input and output values used for burst length and burst size are encoded as specified in the AMBA® AXI Specifications [Ref 1]. For example, LEN = 0 as an input means a burst of length 1.

Tasks and functions common to all BFMs are described in Table 9.

Table 9: **Utility API Tasks/Functions**

API Task/Function Name and Description	Inputs	Outputs
report_status This function can be called at the end of a test to report the final status of the associated BFM.	dummy_bit: The value of this input can be 1 or 0 and does not matter. It is only required because a Verilog function needs at least 1 input.	report_status: This is an integer number which is calculated as: report_status = error_count + warning_count + pending_transactions_count
report_config This task prints out the current configuration as set by the configuration parameters and variables. This task can be called at any time.	None	None
set_channel_level_info This function sets the CHANNEL_LEVEL_INFO internal control variable to the specified input value.	LEVEL: A bit input for the info level.	None
set_function_level_info This function sets the FUNCTION_LEVEL_INFO internal control variable to the specified input value.	LEVEL: A bit input for the info level.	None
set_stop_on_error This function sets the STOP_ON_ERROR internal control variable to the specified input value:	LEVEL: A bit input for the info level.	None
set_read_burst_data_transfer_gap This function sets the SLAVE READ_BURST_DATA_TRANSFER_GAP internal control variable to the specified input value.	TIMEOUT: An integer value measured in clock cycles.	None
set_write_response_gap This function sets the SLAVE WRITE_RESPONSE_GAP internal control variable to the specified input value.	TIMEOUT: An integer value measured in clock cycles.	None
set_read_response_gap This function sets the SLAVE READ_RESPONSE_GAP internal control variable to the specified input value.	TIMEOUT: An integer value measured in clock cycles.	None
set_write_burst_data_transfer_gap This function sets the MASTER WRITE_BURST_DATA_TRANSFER_GAP internal control variable to the specified input value:	TIMEOUT: An integer value measured in clock cycles.	None
set_wrtie_burst_address_data_phase_gap This function sets the AXI 4 FULL MASTER WRITE_BURST_ADDRESS_DATA_PHASE_GAP internal control variable to the specified input value:	GAP_LENGTH: An integer value measured in clock cycles.	None

Table 9: **Utility API Tasks/Functions *(Cont'd)***

API Task/Function Name and Description	Inputs	Outputs
set_write_burst_data_address_phase_gap This function sets the AXI 4 FULL MASTER WRITE_BURST_DATA_ADDRESS_PHASE_GAP internal control variable to the specified input value:	GAP_LENGTH: An integer value measured in clock cycles.	None
set_packet_transfer_gap This function sets the AXI 4 Streaming MASTER PACKET_TRANSFER_GAP internal control variable to the specified input value:	GAP_LENGTH: An integer value measured in clock cycles.	None
set_bfm_clk_delay This task sets the internal variable BFM_CLK_DELAY to the specified input value. This is used to move the BFM internal clock off the simulation clock edge if needed. The default value is zero. If used it must be applied to each BFM separately.	CLK_DELAY: An integer value used for the #BFM_CLK_DELAY on BFM internal clocking.	None
set_task_call_and_reset_handling This task sets the TASK_RESET_HANDLING internal variable to the specified input value: 0 - Ignore reset and continue to process task (default) 1 - Stall task execution until out of reset and print info message 2 - Issue an error and stop (depending on STOP_ON_ERROR value) 3 - Issue a warning and continue	task_reset_handling: An integer value used to define BFM behavior during reset when a channel level API task is called.	None
remove_pending_transaction This task is only required if the test writer is using the channel level API task RECEIVE_READ_DATA instead of RECEIVE_READ_BURST. The RECEIVE_READ_DATA does not decrement the pending transaction counter so this task must be called manually once the full read data transfer is complete.	None	None

AXI3 Master BFM Test Writing API

The channel level API for the AXI3 Master BFM is detailed in Table 10.

Table 10: **Channel Level API for AXI3 Master BFM**

API Task Name and Description	Inputs	Outputs
SEND_WRITE_ADDRESS Creates a write address channel transaction. This task returns after the write address has been acknowledged by the slave. This task emits a "write_address_transfer_complete" event upon completion.	ID: Write Address ID tag ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type	None
SEND_WRITE_DATA Creates a single write data channel transaction. The ID tag should be the same as the write address ID tag it is associated with. The data should be the same size as the width of the data bus. This task returns after is has been acknowledged by the slave. The data input is used as raw bus data, that is, no realignment for narrow or unaligned data. This task emits a "write_data_transfer_complete" event upon completion. NOTE: Should be called multiple times for a burst with correct control of the LAST flag	ID: Write ID tag STOBE: Strobe signals DATA: Data for transfer LAST: Last transfer flag	None
SEND_READ_ADDRESS Creates a read address channel transaction. This task returns after the read address has been acknowledged by the slave. This task emits a "read_address_transfer_complete" event upon completion.	ID: Read Address ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type	None
RECEIVE_READ_DATA This task drives the RREADY signal and monitors the read data bus for read transfers coming from the slave that have the specified ID tag. It then returns the data associated with the transaction and the status of the last flag. The data output here is raw bus data, that is, no realignment for narrow or unaligned data. This task emits a "read_data_transfer_complete" event upon completion. NOTE: This would need to be called multiple times for a burst > 1. Also, the user must call the 'remove_pending_transaction' task when all data is received to ensure that the pending transaction counter is decremented. This is done automatically by the RECEIVE_READ_BURST and RECEIVE_WRITE_RESPONSE channel level API tasks.	ID: Read ID tag	DATA: Data transferred by the slave RESPONSE: The slave read response from the following: [OKAY, EXOKAY, SLVERR, DECERR] LAST: Last transfer flag

Table 10: **Channel Level API for AXI3 Master BFM *(Cont'd)***

API Task Name and Description	Inputs	Outputs
RECEIVE_WRITE_RESPONSE This task drives the BREADY signal and monitors the write response bus for write responses coming from the slave that have the specified ID tag. It then returns the response associated with the transaction. This task emits a "write_response_transfer_complete" event upon completion.	ID: Write ID tag	RESPONSE: The slave write response from the following: [OKAY, EXOKAY, SLVERR, DECERR]
RECEIVE_READ_BURST This task receives a read channel burst based on the ID input. The RECEIVE_READ_DATA from the channel level API is used. This task returns when the read transaction is complete. The data returned by the task is the valid only data, that is, re-aligned data. This task also checks each response and issues a warning if it is not as expected. This task emits a "read_data_burst_complete" event upon completion.	ID: Read ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type	DATA: Valid Data transferred by the slave RESPONSE: This is a vector that is created by concatenating all slave read responses together
SEND_WRITE_BURST This task does a write burst on the write data lines. It does not execute the write address transfer. This task uses the SEND_WRITE_DATA task from the channel level API. This task returns when the complete write burst is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required. This task emits a "write_data_burst_complete" event upon completion.	ID: Write ID tag ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector	None

The function level API for the AXI3 Master BFM is detailed in Table 11.

Table 11: **Function Level API for AXI3 Master BFM**

API Task Name and Description	Inputs	Outputs
READ_BURST This task does a full read process. It is composed of the tasks SEND_READ_ADDRESS and RECEIVE_READ_BURST from the channel level API. This task returns when the read transaction is complete.	ID: Read ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type	DATA: Valid data transferred by the slave RESPONSE: This is a vector that is created by concatenating all slave read responses together
WRITE_BURST This task does a full write process. It is composed of the tasks SEND_WRITE_ADDRESS, SEND_WRITE_BURST and RECEIVE_WRITE_RESPONSE from the channel level API. This task returns when the complete write transaction is complete. This task automatically supports the generation of narrow transfers and unaligned transfers.	ID: Write ID tag ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector	RESPONSE: The slave write response from the following: [OKAY, EXOKAY, SLVERR, DECERR]
WRITE_BURST_CONCURRENT This task does the same function as the WRITE_BURST task; however, it performs the write address and write data phases concurrently.	ID: Write ID tag ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector	RESPONSE: The slave write response from the following: [OKAY, EXOKAY, SLVERR, DECERR]
WRITE_BURST_DATA_FIRST This task does the same function as the WRITE_BURST task; however, it sends the write data burst before sending the associated write address transfer on the write address channel.	ID: Write ID tag ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector	RESPONSE: The slave write response from the following: [OKAY, EXOKAY, SLVERR, DECERR]

AXI3 Slave BFM Test Writing API

The channel level API for the AXI3 Slave BFM is detailed in Table 12.

Table 12: **Channel Level API for AXI3 Slave BFM**

API Task Name and Description	Inputs	Outputs
SEND_WRITE_RESPONSE Creates a write response channel transaction. The ID tag must match the associated write transaction. This task returns after it has been acknowledged by the master. This task emits a "write_response_transfer_complete" event upon completion.	ID: Write ID tag RESPONSE: The chosen write response from the following [OKAY, EXOKAY, SLVERR, DECERR]	None
SEND_READ_DATA Creates a read channel transaction. The ID tag must match the associated read transaction. This task returns after it has been acknowledged by the master. This task emits a "read_data_transfer_complete" event upon completion. NOTE: This would need to be called multiple times for a burst > 1.	ID: Read ID tag DATA: Data to send to the master RESPONSE: The read response to send to the master from the following: [OKAY, EXOKAY, SLVERR, DECERR] LAST: Last transfer flag	None
RECEIVE_WRITE_ADDRESS This task drives the AWREADY signal and monitors the write address bus for write address transfers coming from the master that have the specified ID tag (unless the IDValid bit =0). It then returns the data associated with the write address transaction. If the IDValid bit is 0 then the input ID tag is not used and the next available write address transfer is sampled. This task uses the SLAVE_ADDRESS and SLAVE_MEM_SIZE parameters to determine if the address is valid. This task emits a "write_address_transfer_complete" event upon completion.	ID: Write Address ID tag IDValid: Bit to indicate if the ID input parameter is to be used. When set to 1 the ID is valid and used, when set to 0 it is ignored.	ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type IDTAG: Sampled ID tag
RECEIVE_READ_ADDRESS This task drives the ARREADY signal and monitors the read address bus for read address transfers coming from the master that have the specified ID tag (unless the IDValid bit =0). It then returns the data associated with the read address transaction. If the IDValid bit is 0 then the input ID tag is not used and the next available read address transfer is sampled. This task uses the SLAVE_ADDRESS and SLAVE_MEM_SIZE parameters to determine if the address is valid. This task emits a "read_address_transfer_complete" event upon completion.	ID: Read Address ID tag IDValid: Bit to indicate if the ID input parameter is to be used. When set to 1 the ID is valid and used, when set to 0 it is ignored.	ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type IDTAG: Sampled ID tag
RECEIVE_WRITE_DATA This task drives the WREADY signal and monitors the write data bus for write transfers coming from the master that have the specified ID tag (unless the IDValid bit =0). It then returns the data associated with the transaction and the status of the last flag. NOTE: This would need to be called multiple times for a burst > 1. If the IDValid bit is 0 then the input ID tag is not used and the next available write data transfer is sampled. This task emits a "write_data_transfer_complete" event upon completion.	ID: Write ID tag IDValid: Bit to indicate if the ID input parameter is to be used. When set to 1 the ID is valid and used, when set to 0 it is ignored.	DATA: Data transferred from the master STRB: Strobe signals used to validate the data LAST: Last transfer flag IDTAG: Sampled ID tag

Table 12: **Channel Level API for AXI3 Slave BFM *(Cont'd)***

API Task Name and Description	Inputs	Outputs
RECEIVE_WRITE_BURST This task receives and processes a write burst on the write data channel with the specified ID (unless the IDValid bit =0). It does not wait for the write address transfer to be received. This task uses the RECEIVE_WRITE_DATA task from the channel level API. If the IDValid bit is 0 then the input ID tag is not used and the next available write burst is sampled. This task returns when the complete write burst is complete. This task automatically supports narrow transfers and unaligned transfers; that is, this task aligns the output data with the burst so the final output data should only contain valid data (up to the size of the burst data, shown by the output datasize). This task emits a "write_data_burst_complete" event upon completion.	ID: Write ID tag IDValid: Bit to indicate if the ID input parameter is to be used. When set to 1 the ID is valid and used, when set to 0 it is ignored. ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type	DATA: Data received from the write burst DATASIZE: The size in bytes of the valid data contained in the output data vector IDTAG: Sampled ID tag
RECEIVE_WRITE_BURST_NO_CHECKS This task receives and processes a write burst on the write data channel blindly, that is, with no checking of length, size or anything else. This task uses the RECEIVE_WRITE_DATA task from the channel level API. This task returns when the complete write burst is complete. This task automatically supports narrow transfers and unaligned transfers; that is, this task aligns the output data with the burst so the final output data should only contain valid data (up to the size of the burst data, shown by the output datasize).	ID: Write ID tag	DATA: Data received from the write burst DATASIZE: The size in bytes of the valid data contained in the output data vector
SEND_READ_BURST This task does a read burst on the read data lines. It does not wait for the read address transfer to be received. This task uses the SEND_READ_DATA task from the channel level API. This task returns when the complete read burst is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required. This task emits a "read_data_burst_complete" event upon completion.	ID: Read ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type DATA: Data to be sent over the burst	None
SEND_READ_BURST_RESP_CTRL This task is the same as SEND_READ_BURST except that the response sent to the master can be specified.	ID: Read ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type DATA: Data to be sent over the burst RESPONSE: This is a vector that should contain all of the desired responses for each read data transfer	None

The function level API for the AXI3 Slave BFM is detailed in Table 13.

Table 13: **Function Level API for AXI3 Slave BFM**

API Task Name and Description	Inputs	Outputs
READ_BURST_RESPOND Creates a semi-automatic response to a read request from the master. It checks if the ID tag for the read request is as expected and then provides a read response using the data provided. It is composed of the tasks RECEIVE_READ_ADDRESS and SEND_READ_BURST from the channel level API. This task returns when the complete write transaction is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required.	ID: Read ID tag DATA: Data to send in response to the master read	None
WRITE_BURST_RESPOND This is a semi-automatic task which waits for a write burst with the specified ID tag and responds appropriately. The data received in the write burst is delivered as an output data vector. This task is composed of the tasks RECEIVE_WRITE_ADDRESS, RECEIVE_WRITE_BURST and SEND_WRITE_RESPONSE from the channel level API. This task returns when the complete write transaction is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required.	ID: Write ID tag	DATA: Data received by slave DATASIZE: The size in bytes of the valid data contained in the output data vector
WRITE_BURST_RESPOND_DATA_FIRST This is a semi-automatic task which waits for a write burst with the specified ID tag and responds appropriately. It expects the write data to start arriving before the write address phase. It returns the data received in the write as a data vector. It is composed of the tasks RECEIVE_WRITE_BURST_NO_CHECKS, RECEIVE_WRITE_ADDRESS and SEND_WRITE_RESPONSE from the channel level API. This task returns when the complete write transaction is complete.	ID: Write ID tag	DATA: Data received by slave DATASIZE: The size in bytes of the valid data contained in the output data vector
READ_BURST_RESP_CTRL This task is the same as READ_BURST_RESPONSE except that the responses sent to the master can be specified.	ID: Read ID tag DATA: Data to send in response to the master read. RESPONSE: This is a vector that should contain all of the desired responses for each read data transfer.	None
WRITE_BURST_RESP_CTRL This task is the same as WRITE_BURST_RESPONSE except that the response sent to the master can be specified.	ID: Write ID tag RESPONSE: The chosen write response from the following [OKAY, EXOKAY, SLVERR, DECERR]	DATA: Data received by slave DATASIZE: The size in bytes of the valid data contained in the output data vector

AXI4 Master BFM Test Writing API

The channel level API for the AXI4 Master BFM is detailed in Table 14.

Table 14: **Channel Level API for AXI4 Master BFM**

API Task Name	Inputs	Outputs
SEND_WRITE_ADDRESS Creates a write address channel transaction. This task returns after the write address has been acknowledged by the slave. This task emits a "write_address_transfer_complete" event upon completion.	ID: Write Address ID tag ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type REGION: Region Identifier QOS: Quality of Service Signals AWUSER: Address Write User Defined Signals	None
SEND_WRITE_DATA Creates a single write data channel transaction. The data should be the same size as the width of the data bus. This task returns after is has been acknowledged by the slave. The data input is used as raw bus data; that is, no realignment for narrow or unaligned data. This task emits a "write_data_transfer_complete" event upon completion. NOTE: Should be called multiple times for a burst with correct control of the LAST flag	STOBE: Strobe signals DATA: Data for transfer LAST: Last transfer flag WUSER: Write User Defined Signals	None
SEND_READ_ADDRESS Creates a read address channel transaction. This task returns after the read address has been acknowledged by the slave. This task emits a "read_address_transfer_complete" event upon completion.	ID: Read Address ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type REGION: Region Identifier QOS: Quality of Service Signals ARUSER: Address Read User Defined Signals	None

Table 14: **Channel Level API for AXI4 Master BFM *(Cont'd)***

API Task Name	Inputs	Outputs
RECEIVE_READ_DATA This task drives the RREADY signal and monitors the read data bus for read transfers coming from the slave that have the specified ID tag. It then returns the data associated with the transaction and the status of the last flag. The data output here is raw bus data; that is, no realignment for narrow or unaligned data. This task emits a "read_data_transfer_complete" event upon completion. NOTE: This would need to be called multiple times for a burst > 1. Also, the user must call the 'remove_pending_transaction' task when all data is received to ensure that the pending transaction counter is decremented. This is done automatically by the RECEIVE_READ_BURST and RECEIVE_WRITE_RESPONSE channel level API tasks.	ID: Read ID tag	DATA: Data transferred by the slave RESPONSE: The slave read response from the following: [OKAY, EXOKAY, SLVERR, DECERR] LAST: Last transfer flag RUSER: Read User Defined Signals
RECEIVE_WRITE_RESPONSE This task drives the BREADY signal and monitors the write response bus for write responses coming from the slave that have the specified ID tag. It then returns the response associated with the transaction. This task emits a "write_response_transfer_complete" event upon completion.	ID: Write ID tag	RESPONSE: The slave write response from the following: [OKAY, EXOKAY, SLVERR, DECERR] BUSER: Write Response User Defined Signals
RECEIVE_READ_BURST This task receives a read channel burst based on the ID input. The RECEIVE_READ_DATA from the channel level API is used. This task returns when the read transaction is complete. The data returned by the task is the valid only data, that is, re-aligned data. This task also checks each response and issues a warning if it is not as expected. This task emits a "read_data_burst_complete" event upon completion.	ID: Read ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type	DATA: Valid Data transferred by the slave RESPONSE: This is a vector that is created by concatenating all slave read responses together RUSER: This is a vector that is created by concatenating all slave read user signal data together
SEND_WRITE_BURST This task does a write burst on the write data lines. It does not execute the write address transfer. This task uses the SEND_WRITE_DATA task from the channel level API. This task returns when the complete write burst is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required. This task emits a "write_data_burst_complete" event upon completion.	ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector WUSER: This is a vector that is created by concatenating all write transfer user signal data together	None

The function level API for the AXI4 Master BFM is detailed in Table 15.

Table 15: **Function Level API for AXI4 Master BFM**

API Task Name and Description	Inputs	Outputs
READ_BURST This task does a full read process. It is composed of the tasks SEND_READ_ADDRESS and RECEIVE_READ_BURST from the channel level API. This task returns when the read transaction is complete.	ID: Read ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type REGION: Region Identifier QOS: Quality of Service Signals ARUSER: Address Read User Defined Signals	DATA: Valid data transferred by the slave RESPONSE: This is a vector that is created by concatenating all slave read responses together RUSER: This is a vector that is created by concatenating all slave read user signal data together
WRITE_BURST This task does a full write process. It is composed of the tasks SEND_WRITE_ADDRESS, SEND_WRITE_BURST and RECEIVE_WRITE_RESPONSE from the channel level API. This task returns when the complete write transaction is complete. This task automatically supports the generation of narrow transfers and unaligned transfers.	ID: Write ID tag ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector REGION: Region Identifier QOS: Quality of Service Signals AWUSER: Address Write User Defined Signals WUSER: This is a vector that is created by concatenating all write transfer user signal data together	RESPONSE: The slave write response from the following: [OKAY, EXOKAY, SLVERR, DECERR] BUSER: Write Response Channel User Defined Signals
WRITE_BURST_CONCURRENT This task does the same function as the WRITE_BURST task; however, it performs the write address and write data phases concurrently.	ID: Write ID tag ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector REGION: Region Identifier QOS: Quality of Service Signals AWUSER: Address Write User Defined Signals WUSER: This is a vector that is created by concatenating all write transfer user signal data together	RESPONSE: The slave write response from the following: [OKAY, EXOKAY, SLVERR, DECERR] BUSER: Write Response Channel User Defined Signals

AXI4 Slave BFM Test Writing API

The channel level API for the AXI4 Slave BFM is detailed in Table 16.

Table 16: **Channel Level API for AXI4 Slave BFM**

API Task Name	Inputs	Outputs
SEND_WRITE_RESPONSE Creates a write response channel transaction. The ID tag must match the associated write transaction. This task returns after it has been acknowledged by the master. This task emits a “write_response_transfer_complete” event upon completion.	ID: Write ID tag RESPONSE: The chosen write response from the following [OKAY, EXOKAY, SLVERR, DECERR] BUSER: Write Response User Defined Signals	None
SEND_READ_DATA Creates a read channel transaction. The ID tag must match the associated read transaction. This task returns after it has been acknowledged by the master. This task emits a “read_data_transfer_complete” event upon completion. NOTE: This would need to be called multiple times for a burst > 1.	ID: Read ID tag DATA: Data to send to the master RESPONSE: The read response to send to the master from the following: [OKAY, EXOKAY, SLVERR, DECERR] LAST: Last transfer flag RUSER: Read User Defined Signals	None
RECEIVE_WRITE_ADDRESS This task drives the AWREADY signal and monitors the write address bus for write address transfers coming from the master that have the specified ID tag (unless the IDValid bit =0). It then returns the data associated with the write address transaction. If the IDValid bit is 0 then the input ID tag is not used and the next available write address transfer is sampled. This task uses the SLAVE_ADDRESS and SLAVE_MEM_SIZE parameters to determine if the address is valid. This task emits a “write_address_transfer_complete” event upon completion.	ID: Write Address ID tag IDValid: Bit to indicate if the ID input parameter is to be used. When set to 1 the ID is valid and used, when set to 0 it is ignored.	ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type REGION: Region Identifier QOS: Quality of Service Signals AWUSER: Address Write User Defined Signals IDTAG: Sampled ID tag
RECEIVE_READ_ADDRESS This task drives the ARREADY signal and monitors the read address bus for read address transfers coming from the master that have the specified ID tag (unless the IDValid bit =0). It then returns the data associated with the read address transaction. If the IDValid bit is 0 then the input ID tag is not used and the next available read address transfer is sampled. This task uses the SLAVE_ADDRESS and SLAVE_MEM_SIZE parameters to determine if the address is valid. This task emits a “read_address_transfer_complete” event upon completion.	ID: Read Address ID tag IDValid: Bit to indicate if the ID input parameter is to be used. When set to 1 the ID is valid and used, when set to 0 it is ignored.	ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type CACHE: Cache Type PROT: Protection Type REGION: Region Identifier QOS: Quality of Service Signals ARUSER: Address Read User Defined Signals IDTAG: Sampled ID tag

Table 16: **Channel Level API for AXI4 Slave BFM *(Cont'd)***

API Task Name	Inputs	Outputs
RECEIVE_WRITE_DATA This task drives the WREADY signal and monitors the write data bus for write transfers coming from the master. It then returns the data associated with the transaction and the status of the last flag. NOTE: This would need to be called multiple times for a burst > 1. This task emits a "write_data_transfer_complete" event upon completion.	None	DATA: Data transferred from the master STRB: Strobe signals used to validate the data LAST: Last transfer flag WUSER: Write User Defined Signals
RECEIVE_WRITE_BURST This task receives and processes a write burst on the write data channel. It does not wait for the write address transfer to be received. This task uses the RECEIVE_WRITE_DATA task from the channel level API. This task returns when the complete write burst is complete. This task automatically supports narrow transfers and unaligned transfers; that is, this task aligns the output data with the burst so the final output data should only contain valid data (up to the size of the burst data). This task emits a "write_data_burst_complete" event upon completion.	ADDR: Write Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type	DATA: Data received from the write burst DATASIZE: The size in bytes of the valid data contained in the output data vector WUSER: This is a vector that is created by concatenating all master write user signal data together
SEND_READ_BURST This task does a read burst on the read data lines. It does not wait for the read address transfer to be received. This task uses the SEND_READ_DATA task from the channel level API. This task returns when the complete read burst is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required. This task emits a "read_data_burst_complete" event upon completion.	ID: Read ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type LOCK: Lock Type DATA: Data to be sent over the burst RUSER: This is a vector that is created by concatenating all required slave read user signal data together	None
SEND_READ_BURST_RESP_CTRL This task does a read burst on the read data lines. It does not wait for the read address transfer to be received. This task uses the SEND_READ_DATA task from the channel level API. This task returns when the complete read burst is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required. This task emits a "read_data_burst_complete" event upon completion.	ID: Read ID tag ADDR: Read Address LEN: Burst Length SIZE: Burst Size BURST: Burst Type DATA: Data to be sent over the burst RESPONSE: This is a vector that should contain all of the desired responses for each read data transfer RUSER: This is a vector that is created by concatenating all required slave read user signal data together	None

The function level API for the AXI4 Slave BFM is detailed in Table 17.

Table 17: **Function Level API for AXI4 Slave BFM**

API Task Name and Description	Inputs	Outputs
READ_BURST_RESPOND Creates a semi-automatic response to a read request from the master. It checks if the ID tag for the read request is as expected and then provides a read response using the data provided. It is composed of the tasks RECEIVE_READ_ADDRESS and SEND_READ_BURST from the channel level API. This task returns when the complete write transaction is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required.	ID: Read ID tag DATA: Data to send in response to the master read RUSER: This is a vector that is created by concatenating all required read user signal data together	None
WRITE_BURST_RESPOND This is a semi-automatic task which waits for a write burst with the specified ID tag and responds appropriately. The data received in the write burst is delivered as an output data vector. This task is composed of the tasks RECEIVE_WRITE_ADDRESS, RECEIVE_WRITE_BURST and SEND_WRITE_RESPONSE from the channel level API. This task returns when the complete write transaction is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required.	ID: Write ID tag BUSER: Write Response Channel User Defined Signals	DATA: Data received by slave DATASIZE: The size in bytes of the valid data contained in the output data vector WUSER: This is a vector that is created by concatenating all master write transfer user signal data together
READ_BURST_RESP_CTRL Creates a semi-automatic response to a read request from the master. It checks if the ID tag for the read request is as expected and then provides a read response using the data and response vector provided. It is composed of the tasks RECEIVE_READ_ADDRESS and SEND_READ_BURST_RESP_CTRL from the channel level API. This task returns when the complete write transaction is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required.	ID: Read ID tag DATA: Data to send in response to the master read RESPONSE: This is a vector that should contain all of the desired responses for each read data transfer RUSER: This is a vector that is created by concatenating all required read user signal data together	None
WRITE_BURST_RESP_CTRL This is a semi-automatic task which waits for a write burst with the specified ID tag and responds appropriately using the specified response. The data received in the write burst is delivered as an output data vector. This task is composed of the tasks RECEIVE_WRITE_ADDRESS, RECEIVE_WRITE_BURST and SEND_WRITE_RESPONSE from the channel level API. This task returns when the complete write transaction is complete. This task automatically supports the generation of narrow transfers and unaligned transfers; that is, this task aligns the input data with the burst so data padding is not required.	ID: Write ID tag RESPONSE: The chosen write response from the following [OKAY, EXOKAY, SLVERR, DECERR] BUSER: Write Response Channel User Defined Signals	DATA: Data received by slave DATASIZE: The size in bytes of the valid data contained in the output data vector WUSER: This is a vector that is created by concatenating all master write transfer user signal data together

AXI4-Lite Master BFM Test Writing API

The channel level API for the AXI4-Lite Master BFM is detailed in Table 18.

Table 18: **Channel Level API for AXI4-Lite Master BFM**

API Task Name and Description	Inputs	Outputs
SEND_WRITE_ADDRESS Creates a write address channel transaction. This task returns after the write address has been acknowledged by the slave. This task emits a "write_address_transfer_complete" event upon completion.	ADDR: Write Address PROT: Protection Type	None
SEND_WRITE_DATA Creates a single write data channel transaction. The data should be the same size as the width of the data bus. This task returns after is has been acknowledged by the slave. This task emits a "write_data_transfer_complete" event upon completion.	STOBE: Strobe signals DATA: Data for transfer	None
SEND_READ_ADDRESS Creates a read address channel transaction. This task returns after the read address has been acknowledged by the slave. This task emits a "read_address_transfer_complete" event upon completion.	ADDR: Read Address PROT: Protection Type	None
RECEIVE_READ_DATA This task drives the RREADY signal and monitors the read data bus for read transfers coming from the slave. It returns the data associated with the transaction and the response from the slave. This task emits a "read_data_transfer_complete" event upon completion.	None	DATA: Data transferred by the slave RESPONSE: The slave read response from the following: [OKAY, SLVERR, DECERR]
RECEIVE_WRITE_RESPONSE This task drives the BREADY signal and monitors the write response bus for write responses coming from the slave. It returns the response associated with the transaction. This task emits a "write_response_transfer_complete" event upon completion.	None	RESPONSE: The slave write response from the following: [OKAY, SLVERR, DECERR]

The function level API for the AXI4-Lite Master BFM is detailed inTable 19.

Table 19: **Function Level API for AXI4-Lite Master BFM**

API Task Name and Description	Inputs	Outputs
READ_BURST This task does a full read process. It is composed of the tasks SEND_READ_ADDRESS and RECEIVE_READ_DATA from the channel level API. This task returns when the read transaction is complete.	ADDR: Read Address PROT: Protection Type	DATA: Valid data transferred by the slave RESPONSE: The slave write response from the following: [OKAY, SLVERR, DECERR]
WRITE_BURST This task does a full write process. It is composed of the tasks SEND_WRITE_ADDRESS, SEND_WRITE_DATA and RECEIVE_WRITE_RESPONSE from the channel level API. This task returns when the complete write transaction is complete.	ADDR: Write Address PROT: Protection Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector	RESPONSE: The slave write response from the following: [OKAY, SLVERR, DECERR]
WRITE_BURST_CONCURRENT This task does the same function as the WRITE_BURST task; however, it performs the write address and data phases concurrently.	ADDR: Write Address PROT: Protection Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector	RESPONSE: The slave write response from the following: [OKAY, SLVERR, DECERR]
WRITE_BURST_DATA_FIRST This task does the same function as the WRITE_BURST task; however, it sends the write data burst before sending the associated write address transfer on the write address channel.	ADDR: Write Address PROT: Protection Type DATA: Data to send DATASIZE: The size in bytes of the valid data contained in the input data vector	RESPONSE: The slave write response from the following: [OKAY, SLVERR, DECERR]

AXI4-Lite Slave BFM Test Writing API

The channel level API for the AXI4-Lite Slave BFM is detailed in Table 20.

Table 20: **Channel Level API for AXI4-Lite Slave BFM**

API Task Name and Description	Inputs	Outputs
SEND_WRITE_RESPONSE Creates a write response channel transaction. This task returns after it has been acknowledged by the master. This task emits a “write_response_transfer_complete” event upon completion.	RESPONSE: The chosen write response from the following [OKAY, SLVERR, DECERR]	None
SEND_READ_DATA Creates a read channel transaction. This task returns after it has been acknowledged by the master. This task emits a “read_data_transfer_complete” event upon completion.	DATA: Data to send to the master RESPONSE: The read response to send to the master from the following: [OKAY, SLVERR, DECERR]	None
RECEIVE_WRITE_ADDRESS This task drives the AWREADY signal and monitors the write address bus for write address transfers coming from the master. It returns the data associated with the write address transaction. This task uses the SLAVE_ADDRESS and SLAVE_MEM_SIZE parameters to determine if the address is valid. This task emits a “write_address_transfer_complete” event upon completion.	ADDR: Write Address ADDRValid: Bit to indicate if the address input parameter is to be used. When set to 1 the ADDR is valid and used, when set to 0 it is ignored.	PROT: Protection Type SADDR: Sampled Write Address
RECEIVE_READ_ADDRESS This task drives the ARREADY signal and monitors the read address bus for read address transfers coming from the master. It returns the data associated with the read address transaction. This task uses the SLAVE_ADDRESS and SLAVE_MEM_SIZE parameters to determine if the address is valid. This task emits a “read_address_transfer_complete” event upon completion.	ADDR: Read Address ADDRValid: Bit to indicate if the address input parameter is to be used. When set to 1 the ADDR is valid and used, when set to 0 it is ignored.	PROT: Protection Type SADDR: Sampled Read Address
RECEIVE_WRITE_DATA This task drives the WREADY signal and monitors the write data bus for write transfers coming from the master. It returns the data associated with the transaction. This task emits a “write_data_transfer_complete” event upon completion.	None	DATA: Data transferred from the master STRB: Strobe signals used to validate the data

The function level API for the AXI4-Lite Slave BFM is detailed in Table 21.

Table 21: **Function Level API for AXI4-Lite Slave BFM**

API Task Name and Description	Inputs	Outputs
READ_BURST_RESPOND Creates a semi-automatic response to a read request from the master. It is composed of the tasks RECEIVE_READ_ADDRESS and SEND_READ_DATA from the channel level API. This task returns when the complete write transaction is complete. If ADDRVALID = 0 the input ADDR is ignored and the first read request is used and responded to. If the ADDRVALID = 1 then the ADDR input is used and the DATA input is used to respond to the read burst with the specified address.	ADDR: Read Address ADDRValid: Bit to indicate if the address input parameter is to be used. When set to 1 the ADDR is valid and used, when set to 0 it is ignored. DATA: Data to send in response to the master read	None
WRITE_BURST_RESPOND This is a semi-automatic task which waits for a write burst from the master and responds appropriately. The data received in the write burst is delivered as an output data vector. This task is composed of the tasks RECEIVE_WRITE_ADDRESS, RECEIVE_WRITE_DATA and SEND_WRITE_RESPONSE from the channel level API. This task returns when the complete write transaction is complete. If ADDRVALID = 0 the input ADDR is ignored and the first write request is used for the DATA output. If the ADDRVALID = 1 then the ADDR input is used and the DATA associated with that transfer is output using the DATA output.	ADDR: Write Address ADDRValid: Bit to indicate if the address input parameter is to be used. When set to 1 the ADDR is valid and used, when set to 0 it is ignored.	DATA: Data received by slave DATASIZE: The size in bytes of the valid data contained in the output data vector
READ_BURST_RESP_CTRL This task is the same as READ_BURST_RESPOND except that the response sent to the master can be specified.	ADDR: Read Address ADDRValid: Bit to indicate if the address input parameter is to be used. When set to 1 the ADDR is valid and used, when set to 0 it is ignored. DATA: Data to send in response to the master read RESPONSE: The chosen write response from the following [OKAY, SLVERR, DECERR]	None
WRITE_BURST_RESP_CTRL This task is the same as WRITE_BURST_RESPOND except that the response sent to the master can be specified.	ADDR: Write Address ADDRValid: Bit to indicate if the address input parameter is to be used. When set to 1 the ADDR is valid and used, when set to 0 it is ignored. RESPONSE: The chosen write response from the following [OKAY, SLVERR, DECERR]	DATA: Data received by slave DATASIZE: The size in bytes of the valid data contained in the output data vector

AXI4-Stream Master BFM Test Writing API

The channel level API for the AXI4-Stream Master BFM is detailed in Table 22.

Table 22: **Channel Level API for AXI4-Stream Master BFM**

API Task Name and Description	Inputs	Outputs
SEND_TRANSFER Creates a single AXI4-Stream transfer. This task emits a "transfer_complete" event upon completion.	ID: Transfer ID Tag DEST: Transfer Destination DATA: Transfer Data STRB: Transfer Strobe Signals KEEP: Transfer Keep Signals LAST: Transfer Last Signal USER: Transfer User Signals	None
SEND_PACKET This task sends a complete packet over the streaming interface. It uses the SEND_TRANSFER task from the channel level API. This task returns when the whole packet has been sent, and emits a "packet_complete" event upon completion.	ID: Transfer ID Tag DEST: Transfer Destination DATA: Vector of Transfer data to send DATASIZE: The size in bytes of the valid data contained in the input data vector (This must be aligned to the multiples of the data bus width) USER: This is a vector that is created by concatenating all transfer user signal data together	None

AXI4-Stream Slave BFM Test Writing API

The channel level API for the AXI4-Stream Slave BFM is detailed in Table 23.

Table 23: **Channel Level API for AXI4-Stream Slave BFM**

API Task Name and Description	Inputs	Outputs
RECEIVE_TRANSFER Receives a single AXI4-Stream transfer. This task emits a "transfer_complete" event upon completion.	None	ID: Transfer ID Tag DEST: Transfer Destination DATA: Transfer Data STRB: Transfer Strobe Signals KEEP: Transfer Keep Signals LAST: Transfer Last Signal USER: Transfer User Signals
RECEIVE_PACKET This task receives and processes a packet from the transfer channel. It returns when the complete packet has been sampled, and emits a "packet_complete" event upon completion. This task uses the RECEIVE_TRANSFER task from the channel level API. If the IDValid or DESTValid bits are 0, the input ID tag and the DEST values are not used. In this case, the next values from the first valid transfer are sampled and used for the full packet irrespective of the ID tag or DEST input values.	ID: Packet ID Tag IDValid: Bit to indicate if the ID input parameter is to be used. When set to 1, the ID is valid and used; when set to 0, it is ignored DEST: Packet Destination DESTValid: Bit to indicate if the DEST input parameter is to be used	PID: Packet ID Tag PDEST: Packet Destination DATA: Packet data vector DATASIZE: The size in bytes of the valid data contained in the output packet data vector USER: This is a vector that is created by concatenating all master user signal data together

Protocol Checking

The purpose of the AXI BFMs is to verify connectivity and basic functionality of AXI masters and AXI slaves. A basic level of protocol checking is included with the AXI BFMs. For comprehensive protocol checking, the Cadence AXI UVC should be deployed [Ref 2].

The following aspects of the AXI3 and AXI4 protocol are checked by the AXI BFMs:

- Reset conditions are checked:
  - Reset values of signals
  - Synchronous release of reset
- Inputs into the test writing API are checked to ensure they are valid to prevent protocol violations.
- Signal inputs into master and slave BFMs, respectively, are checked to ensure they are valid to prevent protocol violations.
- Address ranges are checked in the Slave BFMs.

This section describes the checkers that are implemented as Verilog tasks.

Common BFM Checkers

The AXI checkers that are implemented as Verilog tasks and are common to both the master and slave BFMs are described in Table 24.

Table 24: **Common BFM Checker Tasks**

Checker Task Name	Inputs	Description
check_burst_type	BURST_TYPE	Checks to see if the burst type value is valid.
check_burst_length	BURST_TYPE LENGTH LOCK_TYPE	Checks to see if the burst length value is valid given the burst type. NOTE: LOCK_TYPE input added for AXI4 only. In AXI4, exclusive accesses must be 16 beats or less in length. Also only INCR bursts can be greater than 16 beats in length.
check_burst_size	SIZE	Checks that the burst size is not greater than the data bus size.
check_lock_type	LOCK_TYPE	Checks if the lock type value is valid. NOTE: AXI4 reduces this to a single bit: Normal access = 0, Exclusive access =1.
check_cache_type	CACHE_TYPE	Checks if the cache type value is valid. NOTE: Different valid ranges for AXI4.
check_address	ADDRESS BURST_TYPE SIZE	Checks to see if the address is valid given the burst type and the transfer size. For example, a WRAP burst with an address which is not aligned to the transfer size is illegal.
check_byte_qualifications	STROBE KEEP	This is an AXI 4 Streaming check only. It checks that only valid combinations of strobe and keep signals are used.

The AXI3 and AXI4 checkers that are implemented as tasks and are common to the master and slave BFMs are located in the following Verilog files:

- `cdn_axi3_bfm_checkers.v` - AXI3 common checking tasks
- `cdn_axi4_bfm_checkers.v` - AXI4 common checking tasks
- `cdn_axi4_lite_bfm_checkers.v` - AXI4-Lite common checking tasks
- `cdn_axi4_streaming_bfm_checkers.v` - AXI4-Stream common checking tasks

BFM Specific Checkers

Table 25 details the Verilog checking tasks added to each BFM for a specific check. These checkers are only required for the BFM that they are located in; so, they are not included in a common file.

Table 25: **BFM Specific Checker Tasks**

Checker Task Name	Inputs	Checker Location/s	Description
check_address_range	ADDRESS BURST_TYPE LENGTH	SLAVE BFM	Checks to see if address is valid with respect to the SLAVE configuration, the burst_type and length.
check_strobe	STROBE TRANSFER_NUMBER ADDRESS LENGTH SIZE BURST_TYPE	SLAVE BFM	Checks to see if the input strobe is correct. This check handles normal, narrow and unaligned transfers.

Using AXI BFM for Standalone RTL design

The AXI BFM can be used to verify connectivity and basic functionality of AXI masters and AXI slaves with the custom RTL design flow. The AXI BFM provides example test benches and tests that demonstrate the abilities of AXI3, AXI4, AXI4-Lite and AXI4-Stream Master/Slave BFM pair. These examples can be used as a starting point to create tests for custom RTL design with AXI3, AXI4, AXI4-Lite and AXI4-Stream interface.

Generating AXI BFM Examples and Test Benches from CORE Generator

The AXI BFM is delivered with ISE Design Suite installation at:

- `<ISE_Version_Number>/ise_ds/ise/secureip/mti/axi_bfm_mti`
- `<ISE_Version_Number>/ise_ds/ise/secureip/ncsim/axi_bfm_ncsim`
- `<ISE_Version_Number>/ise_ds/ise/secureip/vcs/axi_bfm_vcs`
- `<ISE_Version_Number>/ise_ds/ise/secureip/aldec/axi_bfm_aldec`

The examples and test benches can be obtained by generating the AXI BFM IP available in the "AXI Infrastructure" or "Debug & Verification" folder of the CORE Generator™ IP catalog. When generated, the AXI BFM IP delivers the user-specified `<component_name>` directory.

The `<component_name>/simulation/functional` directory contains the shell scripts for different simulators.

Shell Script	Simulator
simulate_isim.sh	ISim
simulate_mti.sh	Mentor Graphics ModelSim
simulate_ncsim.sh	Cadence IES
simulate_vcs.sh	Synopsys VCS
simulate_aldec.sh	Aldec Riviera-PRO

AXI BFM Example Designs

This section describes the example test benches and example tests used to demonstrate the abilities of each AXI BFM pair. Example tests are delivered either in VHDL or Verilog based on the design entry while generating the core. These example designs are available in the AXI_BFM installation area. Each AXI master is connected to a single AXI slave, and then direct tests are used to transfer data from the master to the slave and from the slave to the master.

It is worth remembering that the BFMs are not fully autonomous. For example, the AXI Master BFM is only a user-driven verification component that enables the user to generate valid AXI protocol scenarios. Furthermore, if tests are written using the channel level API it is possible that the AXI protocol can be accidentally violated. For this reason, Xilinx recommends using the function level API for each BFM. The AMBA AXI protocol specification [Ref 1], Section 3.3, Dependencies between Channel Handshake Signals, states that:

- the slave can wait for AWVALID or WVALID, or both, before asserting AWREADY
- the slave can wait for AWVALID or WVALID, or both, before asserting WREADY

This implies that the slave does not need to support all three possible scenarios. However, if the AXI Master BFM operates in such a way that is not supported by the slave, then the simulation will stall. Each scenario is handled by the function level API:

Scenario 1

Before the slave asserts AWREADY and/or WREADY, the slave can wait for AWVALID. This is modeled using the function level API, WRITE_BURST.

Scenario 2

Before the slave asserts AWREADY and/or WREADY, the slave can wait for WVALID. This is modeled using the function level API, WRITE_BURST_DATA_FIRST.

Scenario 3

Before the slave asserts AWREADY and/or WREADY, the slave can wait for both AWVALID and WVALID. This is modeled using the function level API, WRITE_BURST_CONCURRENT.

AXI3 BFM Example Test Bench and Tests

The Verilog example test bench and example test for the AXI3 BFMs is shown in Figure 3.

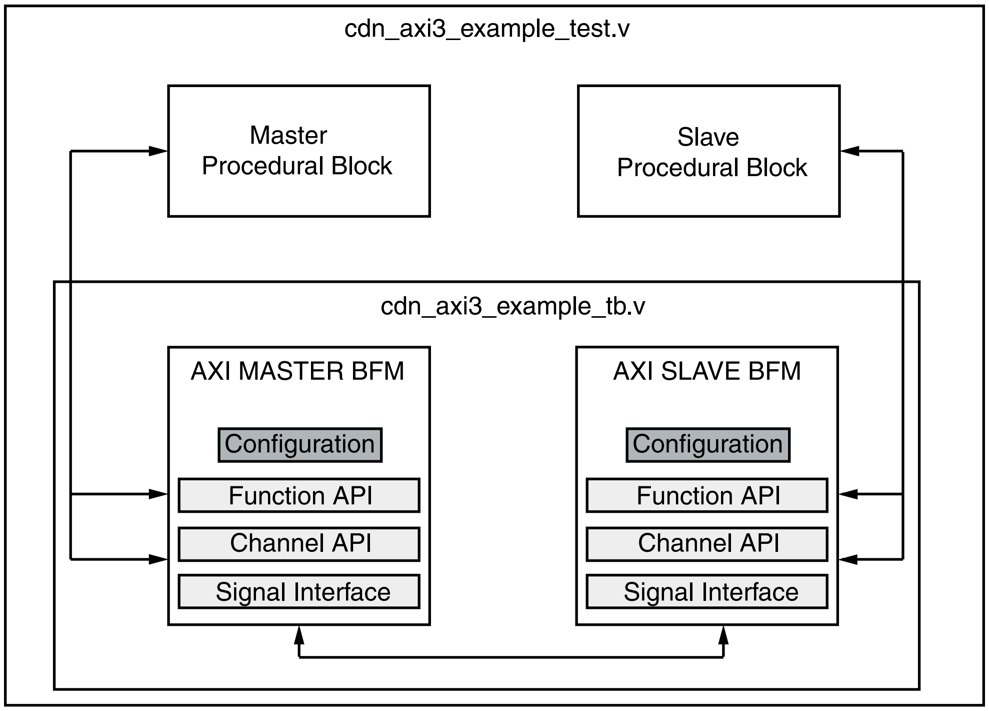


Figure 3: **Verilog Example Test Bench and Test Case Structure**

The example test bench has the master and slave BFMs connected directly to each other. This gives visibility into both sides of the code (master code and slave code) required to hit the scenarios detailed in the example tests.

cdn_axi3_example_test.v

The example test (`simulation/cdn_axi3_example_test.v`) contains the master and slave test code to simulate the following scenarios:

1. Simple sequential write and read burst transfers example
2. Looped sequential write and read transfers example
3. Parallel write and read burst transfers example
4. Narrow write and read transfers example
5. Unaligned write and read transfers example
6. Narrow and unaligned write and read transfers example
7. Out of order write and read burst example
8. Write Bursts performed in two different ways; Data before address, and data with address concurrently
9. Write data interleaving example
10. Read data interleaving example
11. Outstanding transactions example
12. Slave read and write bursts error response example
13. Write and read bursts with different length gaps between data transfers example
14. Write and Read bursts with different length gaps between channel transfers example
15. Write burst that is longer than the data it is sending example

cdn_axi3_example_memory_mode_test.v

The example test (`simulation/cdn_axi3_example_memory_mode_test.v`) contains the slave code to ensure that the slave BFM is configured as a 4 KB memory model. The master code in this test writes maximum length bursts into the memory and reads them back. It does this with two different sets of test values.

The VHDL example test bench and example test for the AXI3 BFMs is shown in Figure 4.

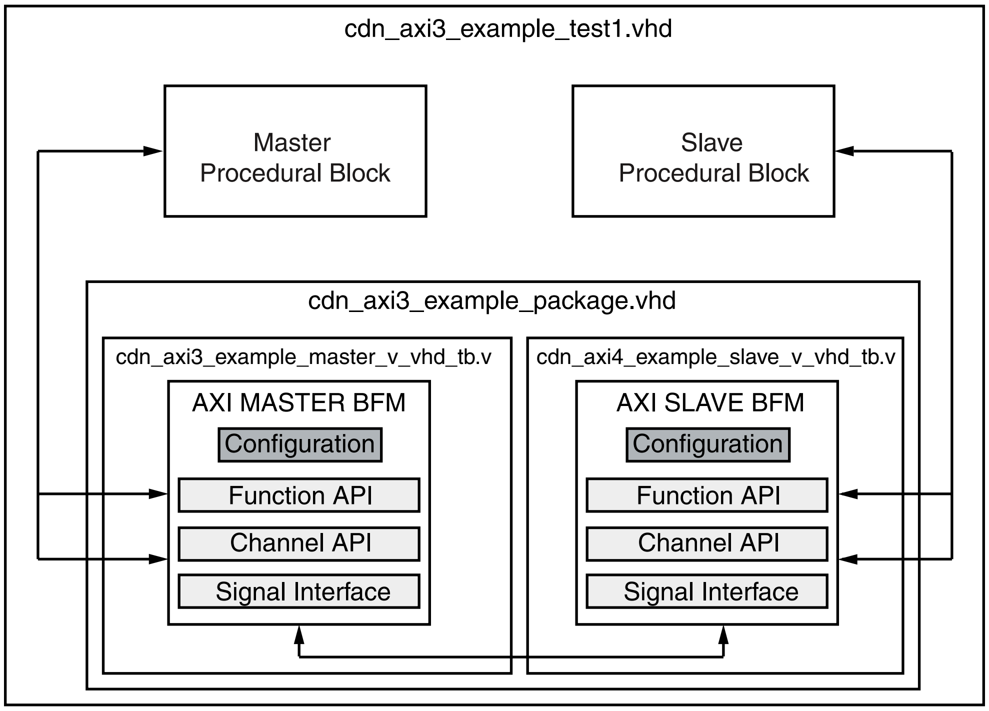


Figure 4: **AXI3 BFM VHDL Example Test Bench and Example Test**

The example test bench has the master and slave BFMs connected directly to each other. This gives visibility into both sides of the code (master code and slave code) required to hit the scenarios detailed in the example tests.

cdn_axi3_example_test1.vhd to cdn_axi3_example_test15.vhd

The example test (`simulation/cdn_axi3_example_test1.vhd` to `cdn_axi3_example_test15.vhd`) contains the master and slave test code to simulate the following scenarios (scenario#1 is covered in Test1, scenario#2 in Test2 and so on):

1. Simple sequential write and read burst transfers example
2. Looped sequential write and read transfers example
3. Parallel write and read burst transfers example
4. Narrow write and read transfers example
5. Unaligned write and read transfers example
6. Narrow and unaligned write and read transfers example
7. Out of order write and read burst example
8. Write Bursts performed in two different ways; Data before address, and data with address concurrently
9. Write data interleaving example
10. Read data interleaving example

11. Outstanding transactions example
12. Slave read and write bursts error response example
13. Write and read bursts with different length gaps between data transfers example
14. Write and Read bursts with different length gaps between channel transfers example
15. Write burst that is longer than the data it is sending example

cdn_axi3_example_memory_model_test.vhd

The example test (`simulation/ cdn_axi3_example_memory_model_test.vhd`) contains the slave code to ensure that the slave BFM is configured as a 4K memory model. The master code in this test writes maximum length bursts into the memory and reads them back. It does this with two different sets of test values.

AXI4 BFM Example Test Bench and Tests

The AXI4 Verilog example test bench structure is identical to the one used for AXI3 shown in Figure 3. The following sections provide details about the example tests available.

cdn_axi4_example_test.v

The example test (`simulation/cdn_axi4_example_test.v`) contains the master and slave test code to simulate the following scenarios:

1. Simple sequential write and read burst transfers example
2. Looped sequential write and read transfers example
3. Parallel write and read burst transfers example
4. Narrow write and read transfers example
5. Unaligned write and read transfers example
6. Narrow and unaligned write and read transfers example
7. Write Bursts performed with address and data channel transfers concurrently
8. Outstanding transactions example
9. Slave read and write bursts error response example
10. Write and read bursts with different length gaps between data transfers example
11. Write and Read bursts with different length gaps between channel transfers example
12. Write burst that is longer than the data it is sending example
13. Read data interleaving example

cdn_axi4_example_memory_mode_test.v

The example test (`simulation/cdn_axi4_example_memory_mode_test.v`) contains the slave code to ensure that the slave BFM is configured as a 4 KB memory model. The master code in this test writes maximum length bursts into the memory and reads them back. It does this with two different sets of test values.

The AXI4 VHDL example test bench structure is identical to the one used for AXI3 shown in Figure 4. The following sections provide details about the example tests available.

cdn_axi4_example_test1.vhd to cdn_axi4_example_test13.vhd

The example test (`simulation/cdn_axi4_example_test1.vhd` to `cdn_axi4_example_test13.vhd`) contains the master and slave test code to simulate the following scenarios (scenario#1 is covered in Test1, scenario#2 in Test2 and so on):

1. Simple sequential write and read burst transfers example
2. Looped sequential write and read transfers example
3. Parallel write and read burst transfers example
4. Narrow write and read transfers example
5. Unaligned write and read transfers example
6. Narrow and unaligned write and read transfers example
7. Write Bursts performed with address and data channel transfers concurrently
8. Outstanding transactions example
9. Slave read and write bursts error response example
10. Write and read bursts with different length gaps between data transfers example
11. Write and Read bursts with different length gaps between channel transfers example
12. Write burst that is longer than the data it is sending example
13. Read data interleaving example

cdn_axi4_example_memory_model_test.vhd

The example test (`simulation/cdn_axi4_example_memory_model_test.vhd`) contains the slave code to ensure that the slave BFM is configured as a 4K memory model. The master code in this test writes maximum length bursts into the memory and reads them back. It does this with two different sets of test values.

AXI4-Lite BFM Example Test Bench and Tests

The AXI4-Lite Verilog example test bench structure is identical to the one used for AXI3 shown in Figure 3. The following sections provide details about the example tests available.

cdn_axi4_lite_example_test.v

The example test (`simulation/cdn_axi4_lite_example_test.v`) contains the master and slave test code to simulate the following scenarios:

1. Simple sequential write and read burst transfers example
2. Looped sequential write and read transfers example
3. Parallel write and read burst transfers example
4. Write Bursts performed in two different ways; Data before address, and data with address concurrently
5. Outstanding transactions example
6. Slave read and write bursts error response example
7. Write and Read bursts with different length gaps between channel transfers example
8. Unaligned write and read transfers example
9. Write burst that has valid data size less than the data bus width

cdn_axi4_lite_example_memory_mode_test.v

The example test (`simulation/cdn_axi4_lite_example_memory_mode_test.v`) contains the slave code to ensure that the slave BFM is configured as a 4 KB memory model. The master code in this test writes data transfers into the memory and reads them back. It does this with two different sets of test values.

The AXI4-Lite VHDL example test bench structure is identical to the one used for AXI3 shown in Figure 4. The following sections provide details about the example tests available.

cdn_axi4_lite_example_test1.vhd to cdn_axi4_lite_example_test9.vhd

The example test (`simulation/ cdn_axi4_lite_example_test1.vhd` to `cdn_axi4_lite_example_test9.vhd`) contains the master and slave test code to simulate the following scenarios (scenario#1 is covered in Test1, scenario#2 in Test2 and so on):

1. Simple sequential write and read burst transfers example
2. Looped sequential write and read transfers example
3. Parallel write and read burst transfers example
4. Write Bursts performed in two different ways; Data before address, and data with address concurrently
5. Outstanding transactions example
6. Slave read and write bursts error response example
7. Write and Read bursts with different length gaps between channel transfers example
8. Unaligned write and read transfers example
9. Write burst that has valid data size less than the data bus width

cdn_axi4_lite_example_memory_model_test.vhd

The example test (`simulation/cdn_axi4_lite_example_memory_model_test.vhd`) contains the slave code to ensure that the slave BFM is configured as a 4K memory model. The master code in this test writes data transfers into the memory and reads them back. It does this with two different sets of test values.

AXI4-Stream BFM Example Test Bench and Tests

The AXI4-Stream Verilog example test bench structure is identical to the one used for AXI3 shown in Figure 3. The following sections provide details about the example tests available.

cdn_axi4_streaming_example_test.v

The example test (`simulation/cdn_axi4_streaming_example_test.v`) contains the master and slave test code to simulate the following scenarios:

1. Simple master to slave transfer example
2. Looped master to slave transfers example
3. Simple master to slave packet example
4. Looped master to slave packet example
5. Ragged (less data at the end of the packet than can be supported) master to slave packet example
6. Packet data interleaving example

The AXI4-Stream VHDL example test bench structure is identical to the one used for AXI3 shown in Figure 4. The following sections provide details about the example tests available.

cdn_axi4_streaming_example_test1.vhd to cdn_axi4_streaming_example_test6.vhd

The example test (`simulation/ cdn_axi4_streaming_example_test1.vhd` to `cdn_axi4_streaming_example_test6.vhd`) contains the master and slave test code to simulate the following scenarios (scenario#1 is covered in Test1, scenario#2 in Test2 and so on):

1. Simple master to slave transfer example
2. Looped master to slave transfers example
3. Simple master to slave packet example
4. Looped master to slave packet example
5. Ragged (less data at the end of the packet than can be supported) master to slave packet example
6. Packet data interleaving example

Useful Coding Guidelines and Examples

Loop Construct Simple Example

While coding directed tests, 'for loops' are typically employed frequently to efficiently generate large volumes of stimulus for both the master and/or slave BFMs. For example:

```
for (m=0;m<2;m =m+1) begin  // Burst Type variable
 for (k=0;k<3;k=k+1)  begin  // Burst Size variable
  $display("--------------------------------------------------");
  $display("EXAMPLE TEST LOCKED and NORMAL ");
  $display("--------------------------------------------------");

  for (i=0; i<16;i=i+1) begin // Burst Length variable
   tb.master_0.WRITE_BURST(mtestID+i,   // Master ID
                           mtestAddr,   // Master Address
                           i,           // Master Burst Length
                           k,           // Master Burst Size
                           m,      // Master Access Type FIXED, INCR
                           `LOCKED_TYPE_FIXED, // Use define
                           4'b0000,      // Buffer/Cachable Hardcoded
                           3'b000,       // Protection Type Hardcoded
                     test_data[i],// Write Data from array
                          response,     // response from slave
    end
  end
end
```

This ‘for loop’ cycles through the following stimulus:

- Access Type (m): FIXED, INCR
- Burst Size (k): 1_BYTE, 2_BYTES, 4_BYTES
- Burst Length (i): 1 to 16

Nested for loops can be used to generate numerous combinations of traffic types, but care must be taken to not violate protocol. The AXI BFMs check the input parameters of the API calls, but this does not prevent higher level protocol being violated.

Loop Construct Complex Example

In some cases, a nested for loop can lead to invalid stimulus if not used correctly. A good example of this is WRAP bursts. The AXI Specification requires that WRAP bursts must be 2,4,8 or 16 transfers in length. For this type of burst, the nested for loop from the Loop Construct Simple Example cannot be used because the nested for loop cycles through burst lengths of 1 to 16. For exhaustive WRAP tests, another for loop declaration is widely used to drive legal stimulus:

```
for (i=2; i <= 16; i=i*2) begin
```

thus giving a burst length of 2, 4, 8 and 16 transfers.

DUT Modeling using the AXI BFMs: Memory Model Example

In most cases, the behavior of a master or slave is more complicated than simple transfer generation. For this reason, the AXI BFM API enables the end user to model higher level DUT functionality. A simple example is a slave memory model. Such a memory model is available as a configuration option in most of the AXI slave BFMs. This example shows the code used for the AXI3 Slave BFM memory model mode, starting with the write datapath.

```
//-----------------------------------------------------------------// Write Path
//-----------------------------------------------------------------
always @(posedge ACLK) begin : WRITE_PATH
  //---------------------------------------------------------------
  //- Local Variables
  //---------------------------------------------------------------
  reg [ID_BUS_WIDTH-1:0] id;
  reg [ADDRESS_BUS_WIDTH-1:0] address;
  reg [`LENGTH_BUS_WIDTH-1:0] length;
  reg [`SIZE_BUS_WIDTH-1:0] size;
  reg [`BURST_BUS_WIDTH-1:0] burst_type;
  reg [`LOCK_BUS_WIDTH-1:0] lock_type;
  reg [`CACHE_BUS_WIDTH-1:0] cache_type;
  reg [`PROT_BUS_WIDTH-1:0] protection_type;
  reg [ID_BUS_WIDTH-1:0] idtag;
  reg [(DATA_BUS_WIDTH*(`MAX_BURST_LENGTH+1))-1:0] data;
  reg [ADDRESS_BUS_WIDTH-1:0] internal_address;
  reg [`RESP_BUS_WIDTH-1:0] response;
  integer i;
  integer datasize;
  //---------------------------------------------------------------
  // Implementation Code
  //---------------------------------------------------------------
  if (MEMORY_MODEL_MODE == 1) begin
    // Receive the next available write address
    RECEIVE_WRITE_ADDRESS(id,`IDVALID_FALSE,address,length,size,
    burst_type,lock_type,cache_type,protection_type,idtag);
    // Get the data to send to the memory.
    RECEIVE_WRITE_BURST(idtag,`IDVALID_TRUE,address,length,size,
    burst_type,data,datasize,idtag);
    // Put the data into the memory array
    internal_address = address - SLAVE_ADDRESS;
    for (i=0; i < datasize; i=i+1) begin
      memory_array[internal_address+i] = data[i*8 +: 8];
    end
    // End the complete write burst/transfer with a write response
    // Work out which response type to send based on the lock type.
    response = calculate_response(lock_type);
    repeat(WRITE_RESPONSE_GAP) @(posedge ACLK);
    SEND_WRITE_RESPONSE(idtag,response);
  end
end
```

As shown in the code above, it is possible to create the write datapath for a simple memory model using three of the tasks from the slave channel level API. This is achieved in the following four steps:

1. The first step is to wait for any write address request on the write address bus. This is done by calling RECEIVE_WRITE_ADDRESS with `IDVALID_FALSE. This ensures that the first detected and valid write address handshake is recorded and the details passed back. This task is blocking; so the WRITE_PATH process does not proceed until it has found a write address channel transfer.
2. The second step is to get the write data burst that corresponds to the write address request in the previous step. This is done by calling RECEIVE_WRITE_BURST with the id tag output from the RECEIVE_WRITE_ADDRESS call and with IDVALID_TRUE. This ensures that the entire write data burst that has the specified id tag is captured before execution returns to the WRITE_PATH process.
3. The third step is to take the data from the write data burst and put it into a memory array. In this case, the memory array is an array of bytes.
4. The last step to complete the AXI3 protocol is to send a response. The internal function 'calculate_reponse' is used to work out if the transfer was exclusive or not and to deliver an EXOKAY or OK response (NOTE: More code could be added here to support DECERR or SLVERR response types). When the response has been calculated, the WRITE_PATH process waits for the defined internal control variable WRITE_RESPONSE_GAP in clock cycles before sending the response back to the slave with the same ID tag as the write data transfer.

The following code illustrates the steps required to make the read datapath for a simple slave memory model:

```
//-----------------------------------------------------------------
// Read Path
//-----------------------------------------------------------------always @(posedge ACLK)
begin : READ_PATH
  //-----------------------------------------------------------------
  // Local Variables
  //-----------------------------------------------------------------
  reg [ID_BUS_WIDTH-1:0] id;
  reg [ADDRESS_BUS_WIDTH-1:0] address;
  reg [`LENGTH_BUS_WIDTH-1:0] length;
  reg [`SIZE_BUS_WIDTH-1:0] size;
  reg [`BURST_BUS_WIDTH-1:0] burst_type;
  reg [`LOCK_BUS_WIDTH-1:0] lock_type;
  reg [`CACHE_BUS_WIDTH-1:0] cache_type;
  reg [`PROT_BUS_WIDTH-1:0] protection_type;
  reg [ID_BUS_WIDTH-1:0] idtag;
  reg [(DATA_BUS_WIDTH*(`MAX_BURST_LENGTH+1))-1:0] data;
  reg [ADDRESS_BUS_WIDTH-1:0] internal_address;
  integer i;
  integer number_of_valid_bytes;
  //-----------------------------------------------------------------
  // Implementation Code
  //-----------------------------------------------------------------
  if (MEMORY_MODEL_MODE == 1) begin
  // Receive a read address transfer
  RECEIVE_READ_ADDRESS(id,`IDVALID_FALSE,address,length,size,
  burst_type,lock_type,cache_type,protection_type,idtag);
  // Get the data to send from the memory.
  internal_address = address - SLAVE_ADDRESS;
  data = 0;
  number_of_valid_bytes =
(decode_burst_length(length)*transfer_size_in_bytes(size))-(address % (DATA_BUS_WIDTH/8));

  for (i=0; i < number_of_valid_bytes; i=i+1) begin
    data[i*8 +: 8] = memory_array[internal_address+i];
  end
  // Send the read data
  repeat(READ_RESPONSE_GAP) @(posedge ACLK);
  SEND_READ_BURST(idtag,address,length,size,burst_type,
  lock_type,data);
  end
end
```

As shown in the code above, it is possible to create the read datapath for a simple memory model using two of the tasks from the slave channel level API. This is achieved in the following two steps:

1. The first step is to wait for any read address request on the read address bus. This is done by calling RECEIVE_READ_ADDRESS with IDVALID_FALSE. This ensures that the first detected and valid read address handshake is recorded and the details are passed back. This task is blocking; so the READ_PATH process does not proceed until it has found a read address channel transfer.
2. The second step is to take the requested data from the memory array and send it in a read burst. This is done by extracting the data byte by byte into a data vector which is used as an input into the SEND_READ_BURST task. Before sending the read data burst, the READ_PATH process waits for the clock cycles determined in the internal control variable READ_RESPONSE_GAP.

Using AXI BFM for Embedded Designs with XPS

For Xilinx Platform Studio (XPS)-based systems, pcore wrappers around the AXI BFMs are provided under **Verification** in the EDK Install IP catalog. Additionally, the XPS Create IP (CIP) Wizard creates simple example projects. See Getting Started with EDK and AXI BFM, page 46 for detailed steps.

This section only applies to AXI BFM cores that are instantiated inside the EDK XPS project with the following requirements:

- Xilinx EDK and AXI BFM Licenses
- Supported simulator

Adding AXI BFMs to EDK

The AXI BFMs are wrapped into EDK pcores to allow easy integration into an XPS project. The BFMs are added and connected to an EDK system in the same way other Xilinx AXI-based IP cores: add the core to the project, parameterize the core, then connect the 'Bus Interface' of the related AXI interface to the rest of the system.

See Configuration Options, page 3 for more information on the BFM-specific parameters on the pcores. Additional parameters exist in the **XPS Core Config** GUI under the **Interconnect Settings for BUSIF** tab to modify the function of an attached AXI Interconnect block. For more information on these parameters, see [Ref 3].

Providing Stimulus

User control/stimulus for the AXI BFMs is provided by making function calls to a hierarchy-specific AXI BFM core instance. For example, to initiate a write burst transaction with a AXI4 Master BFM, the WRITE_BURST() function-level API command might be issued in the test bench:

```
dut. my_master0.my_master0.cdn_axi4_master_bfm_inst.WRITE_BURST(arguments);
```

This command specifies the hierarchy of the AXI BFM instance, and stimulates the core to perform the write burst with the address, data and other transfer qualifiers specified in the arguments, as documented in Test Writing API, page 14.

Determining Fully-Qualified Instance Name

One design challenge is to determine the fully-qualified instance name for EDK AXI BFM pcores for use in the API commands. In general, the path in a standalone XPS project resembles:

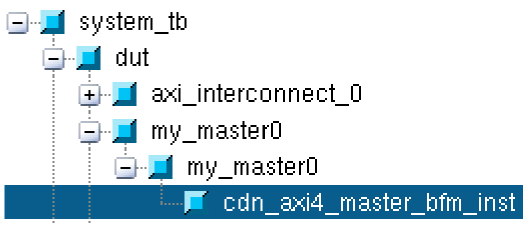
```
    <Project Instance>.<MHS Instance>. <MHS Instance>.<BFM Core Name>
```

where:

- <Project Instance> is the instance name of the EDK system, not including the test bench name.
- <MHS Instance> is the instance name given to the AXI BFM core name in the user MHS file.
- < BFM Core Name > is the core name of the AXI BFM pcore in EDK. Options include:
  - cdn_axi4_master_bfm_wrap
  - cdn_axi4_slave_bfm_wrap
  - cdn_axi4_lite_master_bfm_wrap
  - cdn_axi4_lite_slave_bfm_wrap
  - cdn_axi4_streaming_master_bfm_wrap
  - cdn_axi4_streaming_slave_bfm_wrap
  - cdn_axi3_master_bfm_wrap
  - cdn_axi3_slave_bfm_wrap

Any additional levels of hierarchy, such as when using with ISE Project Navigator, are added to the left of the path.

To determine the fully-qualified name of the AXI BFM core in EDK, elaborate the design in a simulator, before specifying the API function calls. This is shown in Figure 5 for the example above, using ModelSim.



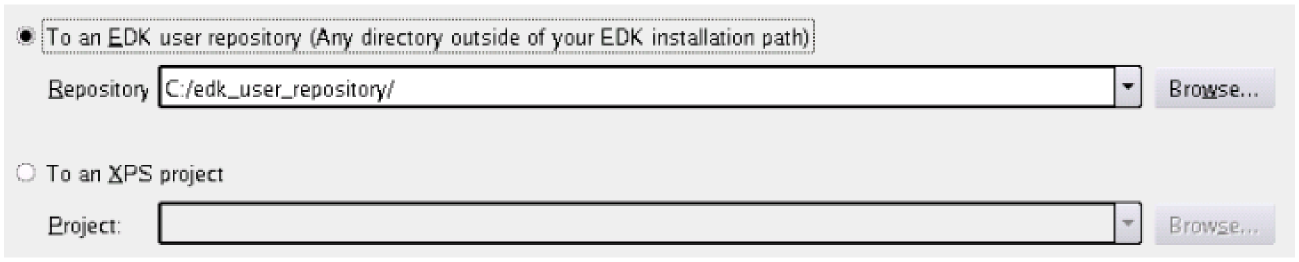
```
system_tb
  dut
    axi_interconnect_0
    my_master0
      my_master0
        cdn_axi4_master_bfm_inst
```

Figure 5: **ModelSim Example Hierarchy**

Getting Started with EDK and AXI BFM

This section describes how to use CIP Wizard in XPS to create an AXI-based IP with AXI BFM simulation. By creating a custom IP core, the CIP wizard provides a matching AXI BFM project to aid in development and verification of the custom IP core.

1. In XPS GUI, select **Hardware** -> **Create or Import Peripherals** to enable CIP wizard.
2. Click **Next** and choose **Create templates for a new peripheral.**
3. Click **Next** and choose the repository for storing the peripheral.



[✓ To an EDK user repository (Any directory outside of your EDK installation path)]

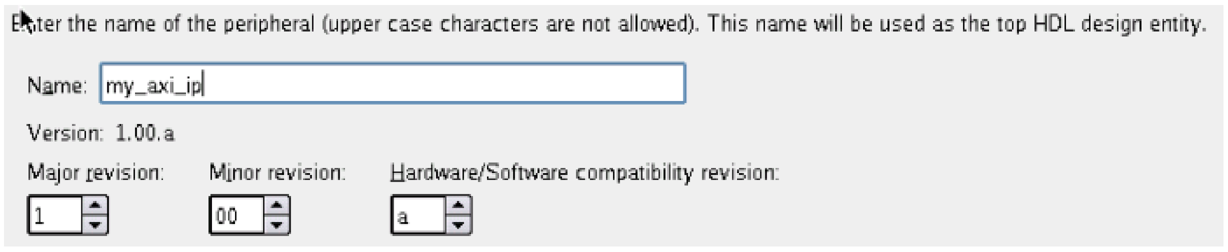
Repository: C:/edk_user_repository/

To an XPS project

Project:

Figure 6: **Select the Repository Path**

4. Click **Next** and name the AXI-based IP **my_axi_ip.**



Enter the name of the peripheral (upper case characters are not allowed). This name will be used as the top HDL design entity.

Name: my_axi_ip

Version: 1.00.a

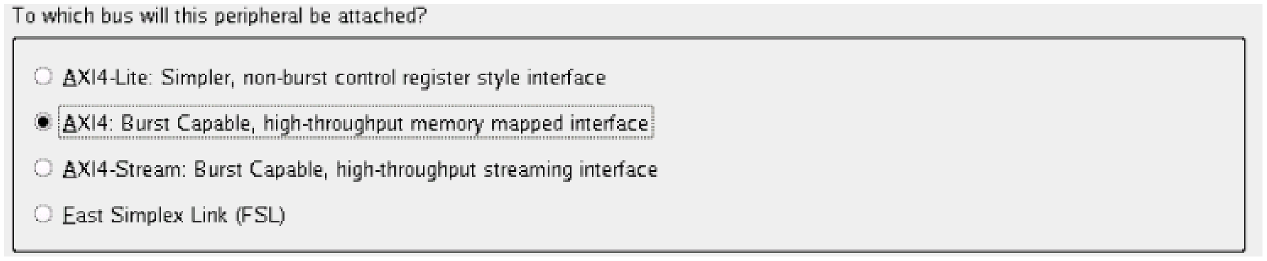
Major revision: 1

Minor revision: 00

Hardware/Software compatibility revision: a

Figure 7: **Name the Peripheral**

5. For this example, choose **AXI4** bus interface and click **Next**.



To which bus will this peripheral be attached?

AXI4-Lite: Simpler, non-burst control register style interface

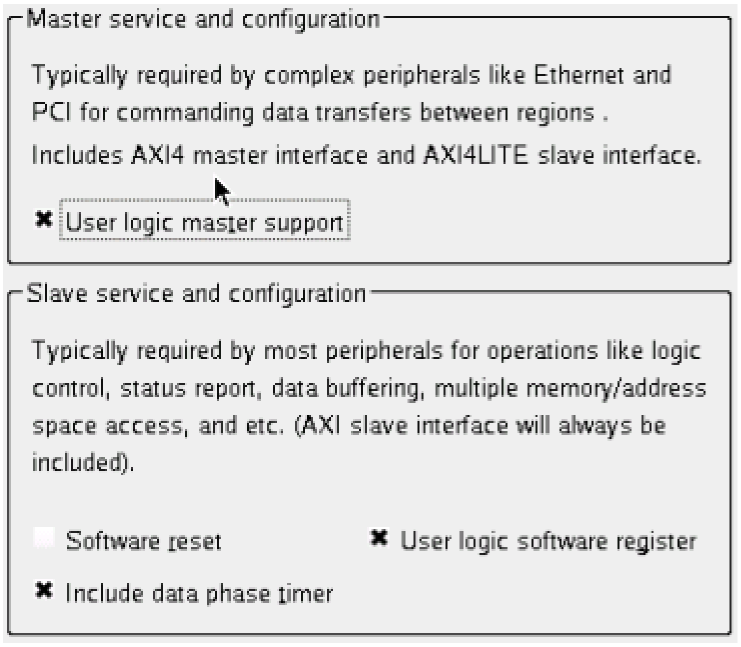
[✓ AXI4: Burst Capable, high-throughput memory mapped interface]

AXI4-Stream: Burst Capable, high-throughput streaming interface

Fast Simplex Link (FSL)

Figure 8: **Choose a Bus**

6. Check **User Logic Master Support** in **IPIF Services** tab if an AXI4-based master IP is needed, and then click **Next** until the **Peripheral Simulation Support** tab.



Master service and configuration

Typically required by complex peripherals like Ethernet and PCI for commanding data transfers between regions .
Includes AXI4 master interface and AXI4LITE slave interface.

✖ User logic master support

Slave service and configuration

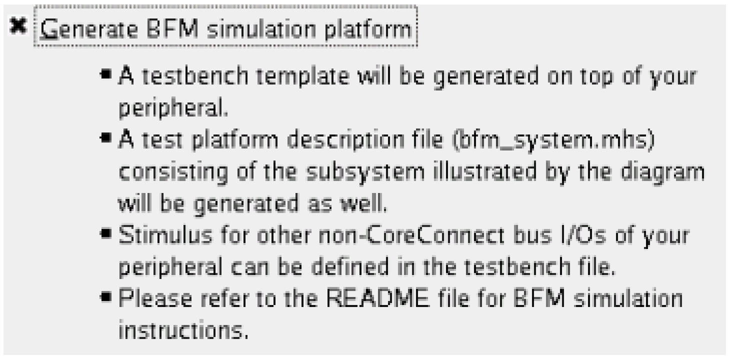
Typically required by most peripherals for operations like logic control, status report, data buffering, multiple memory/address space access, and etc. (AXI slave interface will always be included).

☐ Software reset ✖ User logic software register

✖ Include data phase timer

Figure 9: **Select Master Support**

7. Check **Generate BFM simulation platform** and **Next.**



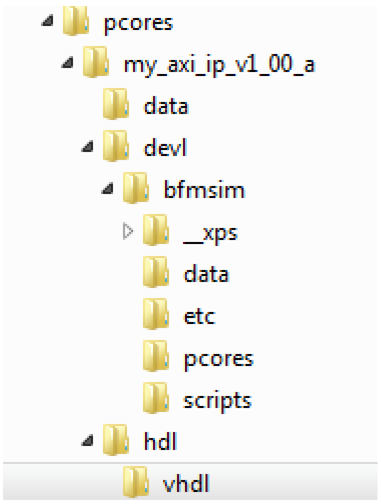
✖ Generate BFM simulation platform

- A testbench template will be generated on top of your peripheral.
- A test platform description file (bfm_system.mhs) consisting of the subsystem illustrated by the diagram will be generated as well.
- Stimulus for other non-CoreConnect bus I/Os of your peripheral can be defined in the testbench file.
- Please refer to the README file for BFM simulation instructions.

Figure 10: **Generate Simulation Platform**

8. Click **Next** and **Finish.**

Figure 11 shows the directory structure of the generated AXI4-based IP (named `my_axi_ip`).



```
pcores
  my_axi_ip_v1_00_a
    data
    devl
      bfmsim
        __xps
        data
        etc
        pcores
        scripts
    hdl
      vhdl
```

Figure 11: **CIP Wizard Output Directory Structure**

The AXI BFM simulation can be run from `\devl\bfmsim`.

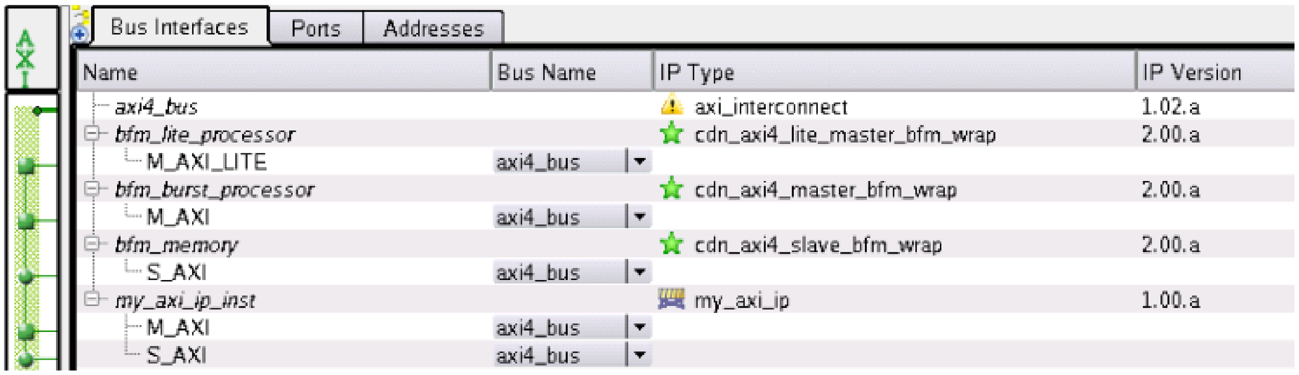
Running AXI BFM simulation with ModelSim

This section describes how to run AXI BFM simulation on the generated AXI-based IP within ModelSim.

1. Start XPS and open the BFM_SYSTEM project in the directory `\devl\bfmsim`.

Xilinx has provided AXI BFM wrapper files to be used with AXI-based IP BFM simulations. When an AXI-based master/slave IP is generated, a corresponding AXI BFM core is added to assist in developing the custom core.

In this example, "User Logic Master Support" is enabled. Therefore, `my_axi_ip` has an AXI4 Master interface and an AXI4-Lite Slave interface, which is connected through an AXI4 bus and AXI4-Lite bus interface, respectively. In the AXI BFM simulation directory, the simulation uses AXI4_MASTER_BFM_WRAPPER, AXI4_LITE_MASTER_BFM_WRAPPER and AXI4_SLAVE_BFM_WRAPPER for the simulation. Figure 12 shows the XPS GUI.



Name	Bus Name	IP Type	IP Version
axi4_bus		axi_interconnect	1.02.a
bfm_lite_processor		cdn_axi4_lite_master_bfm_wrap	2.00.a
M_AXI_LITE	axi4_bus		
bfm_burst_processor		cdn_axi4_master_bfm_wrap	2.00.a
M_AXI	axi4_bus		
bfm_memory		cdn_axi4_slave_bfm_wrap	2.00.a
S_AXI	axi4_bus		
my_axi_ip_inst		my_axi_ip	1.00.a
M_AXI	axi4_bus		
S_AXI	axi4_bus		

Figure 12: **XPS CIP AXI BFM Project**

2. Click **Simulation** -> **Launch HDL Simulator** to launch ModelSim (assuming the EDK and BFM libraries have been properly compiled within ModelSim simulator).
3. Copy the test bench file (`bfm_system_tb.v`) from `\devl\bfmsim\scripts` to `\devl\bfmsim\simulation\behavioral`.

4. Make `run.do` script and save it in `\devl\bfmsim\simulation\behavioral`.

```
do bfm_system_setup.do
# Compile BFM test modules
c
# Load BFM test platform
s
# Load Wave window
w
# Run test time
run 100 us
```

5. After launching ModelSim, type **`do run.do`** in the ModelSim console.

The AXI BFM simulation starts running and activity is seen on the AXI4 and AXI4-Lite interfaces.

Analyzing AXI BFM Simulation

This section describes how to analyze the simulation results to verify custom IP function. With the default visibility, the following AXI BFM configuration details can be seen from the ModelSim console.

```
# BFM Xilinx: License succeeded for Xilinx_AXI_BFM, version 2010.100000
# **********************************************************
# * Cadence AXI 4 LITE MASTER BFM                          *
# **********************************************************
# * VERSION NUMBER : 1.9
# **********************************************************
# * CONFIGURATION:
# * NAME = MASTER_0
# * DATA_BUS_WIDTH = 32
# * ADDRESS_BUS_WIDTH = 32
# * MAX_OUTSTANDING_TRANSACTIONS = 8
# * RESPONSE_TIMEOUT = 500
# * STOP_ON_ERROR = 1
# * CHANNEL_LEVEL_INFO = 0
# * FUNCTION_LEVEL_INFO = 1
# **********************************************************
# BFM Xilinx: License succeeded for Xilinx_AXI_BFM, version 2010.100000
# **********************************************************
# * Cadence AXI 4 MASTER BFM                               *
# **********************************************************
# * VERSION NUMBER : 1.9
# **********************************************************
# * CONFIGURATION:
# * NAME = MASTER_0
# * DATA_BUS_WIDTH = 32
# * ADDRESS_BUS_WIDTH = 32
# * ID_BUS_WIDTH = 1
# * AWUSER_BUS_WIDTH = 1
# * ARUSER_BUS_WIDTH = 1
# * RUSER_BUS_WIDTH = 1
# * WUSER_BUS_WIDTH = 1
# * BUSER_BUS_WIDTH = 1
# * MAX_OUTSTANDING_TRANSACTIONS = 8
# * EXCLUSIVE_ACCESS_SUPPORTED = 0
# * WRITE_BURST_DATA_TRANSFER_GAP = 0
# * RESPONSE_TIMEOUT = 500
# * STOP_ON_ERROR = 1
# * CHANNEL_LEVEL_INFO = 0
# * FUNCTION_LEVEL_INFO = 1
# **********************************************************
# BFM Xilinx: License succeeded for Xilinx_AXI_BFM, version 2010.100000
# *********************************************************
# * Cadence AXI 4 SLAVE BFM                               *
# *********************************************************
# * VERSION NUMBER : 1.9
# *********************************************************
# * CONFIGURATION:
# * NAME = SLAVE_0
# * DATA_BUS_WIDTH = 32
# * ADDRESS_BUS_WIDTH = 32
# * ID_BUS_WIDTH = 2
# * AWUSER_BUS_WIDTH = 1
# * ARUSER_BUS_WIDTH = 1
# * RUSER_BUS_WIDTH = 1
# * WUSER_BUS_WIDTH = 1
# * BUSER_BUS_WIDTH = 1
# * SLAVE_ADDRESS = 0x40000000
# * SLAVE_MEM_SIZE = 0x10000
# * MAX_OUTSTANDING_TRANSACTIONS = 8
# * MEMORY_MODEL_MODE = 1
# * EXCLUSIVE_ACCESS_SUPPORTED = 0
# * READ_BURST_DATA_TRANSFER_GAP = 0
# * WRITE_RESPONSE_GAP = 0
# * READ_RESPONSE_GAP = 0
# * RESPONSE_TIMEOUT = 500
# * STOP_ON_ERROR = 1
# * CHANNEL_LEVEL_INFO = 0
# * FUNCTION_LEVEL_INFO = 1
# *********************************************************
```

The test bench shows BFM_BURST_PROCESSOR doing two WRITE bursts and one READ burst to BFM_MEMORY peripheral through the AXI4 bus.

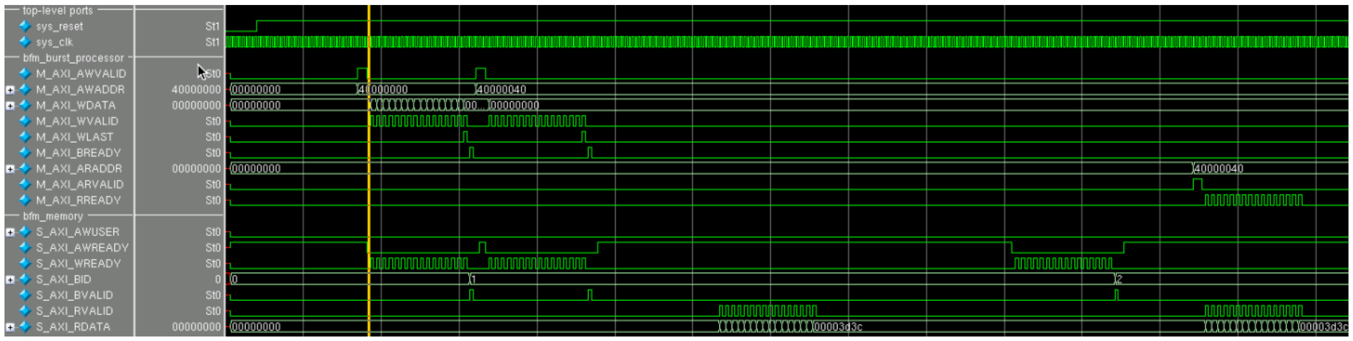


Figure 13: **BFM_BURST_PROCESSOR Simulation Waveform**

In Figure 13, BFM_BURST_PROCESSOR is performing two WRITE bursts to BFM_MEMORY at address 0x4000000 and then to 0x40000040. Next it performs a READ burst from BFM_MEMORY at address 0x40000040. The ModelSim console output is:

```
# ----------------------------------------------------
# Master Verification
# ----------------------------------------------------
# Initializing first          16 locations of AXI Slave BFM memory with value
# [830] : MASTER_0 : *INFO : WRITE_BURST Task Call - id = 0x0, address = 0x40000000, length
= 16, size = 4, burst_type = 0x1, lock_type = 0x0, cache_type = 0x0, protection_type = 0x0,
valid data size (in bytes) = 64, region = 0x0, qos = 0x0, awuser = 0x0
# EXAMPLE TEST 1 : Burst 64,WRITE DATA =
0x03d3c3b3a393837363534333231302f2e2d2c2b2a292827262524232221201f1e1d1c1b1a191817161514131
211100f0e0d0c0b0a09080706050403020100, response = 0x0
# Initializing second          16 locations of AXI Slave BFM memory with value
# [1590] : MASTER_0 : *INFO : WRITE_BURST Task Call - id = 0x0, address = 0x40000040, length
= 16, size = 4, burst_type = 0x1, lock_type = 0x0, cache_type = 0x0, protection_type = 0x0,
valid data size (in bytes) = 64, region = 0x0, qos = 0x0, awuser = 0x0
# EXAMPLE TEST 1 : Burst 64,WRITE DATA =
0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000, response = 0x0
# Requesting master to read the data and write to different location
# [6190] : MASTER_0 : *INFO : READ_BURST Task Call - id = 0x0, address = 0x40000040, length
= 16, size = 4, burst_type = 0x1, lock_type = 0x0, cache_type = 0x0, protection_type = 0x0,
region = 0x0, qos = 0x0, aruser = 0x0
# EXAMPLE TEST 1 : READ DATA =
0x03d3c3b3a393837363534333231302f2e2d2c2b2a292827262524232221201f1e1d1c1b1a191817161514131
211100f0e0d0c0b0a09080706050403020100, vresponse = 0x00000000
# -----------------------------------------------------
# Peripheral Verification Completed Successfully
# -----------------------------------------------------
```

At the AXI4 bus interface, `my_axi_ip` is another master that does a continuous single READ and single WRITE to BFM_MEMORY.

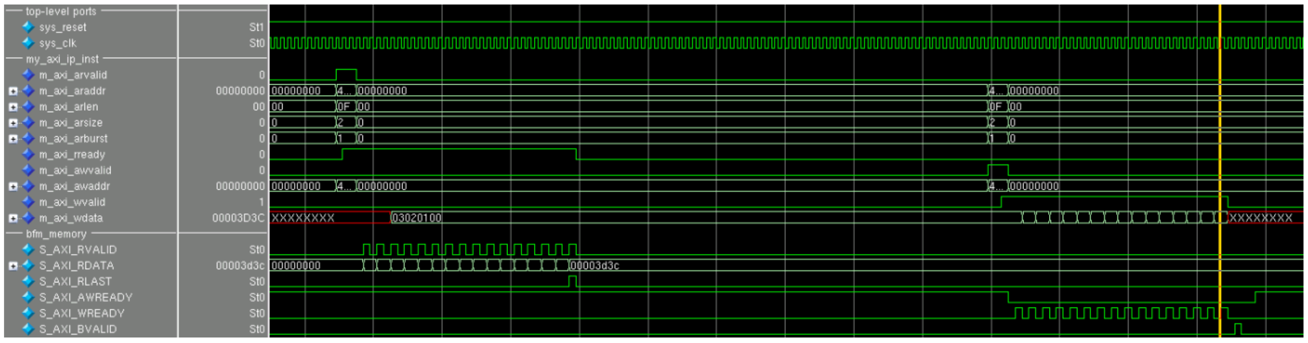


Figure 14: **my_axi_ip Waveform**

In Figure 14, `my_axi_ip` issues Read commands and BFM_MEMORY responds with read data. In addition, `my_axi_ip` issues Write commands with the date and BFM_MEMORY responds `WREADY` to accept the date from the master

At the AXI4-Lite bus interface, the BFM_Lite_Processor is the master that performs single-transfer Write and Read transactions to `my_axi_ip`.

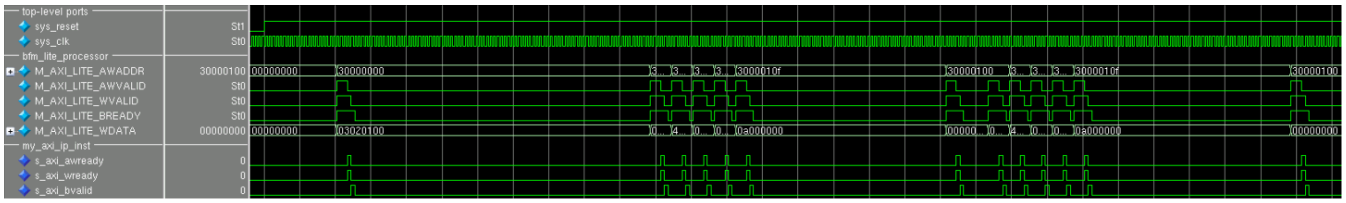


Figure 15: **BFM_Lite_Processor Write Waveform**

In Figure 15, BFM_Lite_Processor is performing a single Write at address 0x30000000 with data = 0x03020100. `my_axi_ip` responds with `WREAD/WVALID` signals for each of the Write transactions from the master.

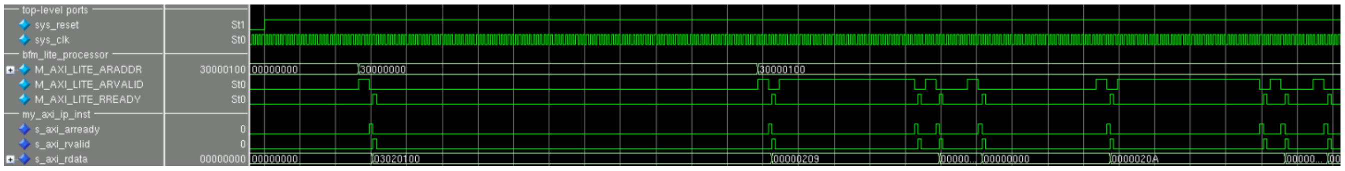


Figure 16: **BFM_Lite_Processor Read Waveform**

In Figure 16, BFM_Lite_Processor issues a single Read command to `my_axi_ip`, and `my_axi_ip` responds with read data.

As a result of these transactions, the ModelSim console outputs:

```
# ---------------------------------------------------
# Full Registers write followed by a full Registers read
# ---------------------------------------------------
# Writing to Slave Register addr=0x30000000 data=0x03020100
# Reading from Slave Register addr=0x30000000 data=0x03020100
```

References

1. ARM® AMBA® AXI Protocol v2.0 Specification (ARM IHI 0022C)
2. *Cadence AXI UVC User Guide* (VIPP 9.2/VIPP 10.2 releases)
3. AXI Interconnect IP Data Sheet (DS768)

Support

Xilinx provides technical support for this product when used as described in the product documentation. Xilinx cannot guarantee functionality or support of this product not defined in the documentation, if modified in a way not described in the product documentation, or if changes are made to any section of the design labeled *DO NOT MODIFY*.

Ordering Information

The AXI Bus Functional Model is provided under the terms of the Xilinx Core License Agreement. A full license for the model must be purchased and obtained from Xilinx. To access the full functionality of the core, visit the AXI Bus Functional Model web page. Contact your local Xilinx sales representative for pricing and availability of additional Xilinx modules and software. Information about additional Xilinx solutions is available on the Xilinx IP Center.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
06/22/2011	1.0	Initial Xilinx release. This document was previously released as UG783. Added additional details in Using AXI BFM for Standalone RTL design, page 34 and Using AXI BFM for Embedded Designs with XPS, page 44.
10/19/11	1.1	Updated for Release 13.3. Added write burst address and data parameters to Table 3 and Table 9. Added clk_delay and call_and_reset handling to Table 9. Added section on possible scenarios in AXI BFM Example Designs, page 35.
04/24/12	1.2	Updated for Release 14.1. Version 2.1 of core. Core now CORE Generator compliant; added VHDL example tests.

Notice of Disclaimer

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications: http://www.xilinx.com/warranty.htm#critapps.