Aurora 8B/10B Bus Functional Model

User Guide

UG058 March 31, 2011

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/17/03	1.0	Initial Xilinx release.
06/08/04	1.1	• Added new text and figures to Clocking Scheme, page 18; • Added new arguments to Table 2-2, page 32; • Miscellaneous text and figure edits for clarity.
10/27/04	1.2	• Added Appendix A, Simplex with a Serial Interface; • Miscellaneous non-technical edits for clarity.
05/06/05	1.3	• Added Schedule of Figures (SOF) and Schedule of Tables (SOT); • Added more function calls in Chapter 2, PLI Interface and Test Bench Integration; • Added Appendix B, Aurora 8B/10B BFM: Parallel Interface.
01/18/07	1.4	• Added the SKIP feature and the automatic_compliance_ufc_size parameter in Channel Up Phase in Chapter 1. • Added cup_skip_vector and automatic_compliance_ufc_size to Table 2-2, page 32. • Updated document template.
03/24/08	1.5	• Added symbol replacement feature in Error Injection in Chapter 1.
04/24/09	2.1	Revised to be specifically for Aurora 8B/10B bus functional model. Removed “Protocol Conformance” section and removed “Appendix B: Parallel Interface.”
03/31/11	2.2	Added Table 2-1, page 27.

Table of Contents

Schedule of Figures

Chapter 1: Aurora 8B/10B Bus Functional Model

Chapter 2: PLI Interface and Test Bench Integration

Chapter 3: FLI Interface

Appendix A: Simplex with a Serial Interface

Schedule of Tables

About This User Guide

This document describes an Aurora 8B/10B bus functional model (ABFM 8B/10B). The ABFM 8B/10B models the behavior of the Aurora 8B/10B protocol and can be used to generate stimulus for and to monitor the response of an Aurora 8B/10B interface design, which is referred to as the device under test (DUT).

User Guide Contents

This user guide contains the following chapters:

Chapter 1, Aurora 8B/10B Bus Functional Model is an overview of the ABFM 8B/10B, providing details about the Aurora 8B/10B application model, user application interface, clocking scheme, and error injection.

Chapter 2, PLI Interface and Test Bench Integration describes the details of the programming language interface (PLI) and its integration into the test bench.

Chapter 3, FLI Interface describes the foreign language interface (FLI) that provides the ability to interface with the BFM shared library from a VHDL environment. It can be considered as a VHDL equivalent of the PLI.

Appendix A, Simplex with a Serial Interface describes the specification of the Aurora 8B/10B simplex BFM which supports the verification of Aurora 8B/10B simplex designs.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File → Open**
	Keyboard shortcuts	**Ctrl+C**

Convention	Meaning or Use	Example
Italic font	Variables in a syntax statement for which you must supply values	**ngdbuild** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **bus[7:0]**, they are required.	**ngdbuild** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**lowpwr ={on|off}**
Vertical bar |	Separates items in a list of choices	**lowpwr ={on|off}**
Vertical ellipsis . . .	Repetitive material that has been omitted	IOB #1: Name = QOUT' IOB #2: Name = CLKIN' . . .
Horizontal ellipsis ...	Repetitive material that has been omitted	**allow block** *block_name loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current file or in another file in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Blue, underlined text	Hyperlink to a website (URL)	Go to www.xilinx.com for the latest speed files.

XILINX®

Chapter 1

Aurora 8B/10B Bus Functional Model

Introduction

This document describes an Aurora 8B/10B bus functional model (ABFM 8B/10B). The ABFM 8B/10B models the behavior of the Aurora 8B/10B protocol and can be used to generate stimulus for and to monitor the response of an Aurora 8B/10B interface design, which is referred to as the device under test (DUT).

The ABFM 8B/10B provides parametrization of the protocol parameters (for example, the number of lanes) and that can be used to test any implementation of the Aurora 8B/10B protocol with little overhead. The ABFM 8B/10B provides flexibility, a *clean room* implementation of Aurora 8B/10B, and improved performance over using another Aurora 8B/10B design to verify the DUT.

The ABFM 8B/10B can be easily integrated (described in Test Bench Integration, page 26) into an existing verification environment specifically designed to test the Aurora 8B/10B DUT. Once the ABFM 8B/10B is integrated into the verification environment, it can communicate with the DUT using a programming language interface (PLI) for Verilog environments or a foreign language interface (FLI) for VHDL+ModelSim environment. Both PLI and FLI contain transaction-based calls that are used to establish communication between the Aurora 8B/10B DUT and ABFM 8B/10B.

The ABFM 8B/10B is compatible with multiple simulators and Linux (32/64) platforms, and supports both Verilog and VHDL environments. The combinations available are:

- **ModelSim**: Linux 32/64
- **NCSim**: Linux 32/64

The ABFM 8B/10B package deliverables include:

- A test environment containing test benches (simulates two BFMs communicating with each other configurable for any given design configuration)
- Shared libraries containing PLI routines and interface (for Verilog environments)
- Shared libraries containing FLI routines and interface (for VHDL environments)
- An automated script to configure and run the BFM in the user specified configuration. This facilitates seamless communication between the three components of the package: ABFM 8B/10B, test benches, and the DUT

Figure 1-1 shows a block diagram of the ABFM 8B/10B environment.

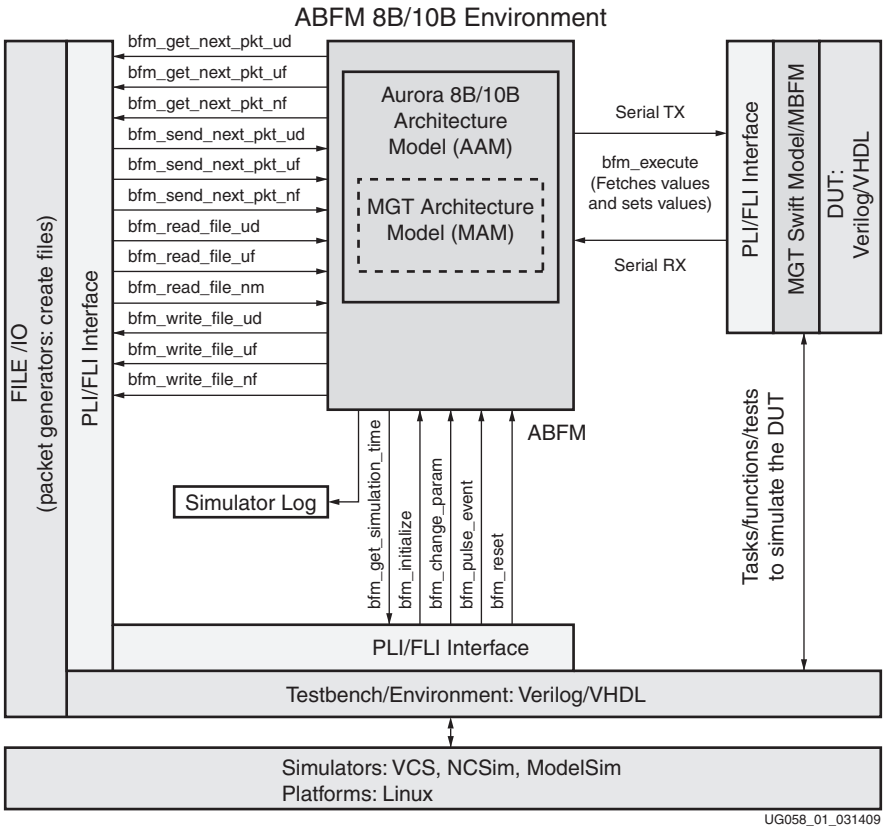

Figure 1-1: **ABFM 8B/10B Environment**

Architecture

The ABFM 8B/10B architecture can be broadly categorized into these sections:

- Aurora 8B/10B Architecture Model, page 13
- User Application Interface, page 16
- Clocking Scheme, page 18
- Error Injection, page 19

The Aurora 8B/10B architecture model (AAM) contains C models of the Aurora 8B/10B protocol and a RocketIO™ multi-gigabit transceiver (MGT). The AAM communicates with the test benches through a PLI interface (for Verilog environments) or an FLI interface (for VHDL environments). The PLI and FLI interfaces contain routines and functions that can be called from the test bench using system calls (refer to Chapter 2, PLI Interface and Test Bench Integration and Chapter 3, FLI Interface for more details).

Table 1-1 shows the PLI and FLI system calls. Note the difference in syntax between the two.

Table 1-1: **PLI and FLI System Calls**

PLI System Call	FLI System Call
$bfm_initialize	bfm_initialize
$bfm_execute	bfm_execute
$bfm_read_file_ud	bfm_read_file_ud
$bfm_read_file_uf	bfm_read_file_uf
$bfm_read_file_nf	bfm_read_file_nf
$bfm_write_file_ud	bfm_write_file_ud
$bfm_write_file_uf	bfm_write_file_uf
$bfm_write_file_nf	bfm_write_file_nf
$bfm_send_next_pkt_ud	bfm_send_next_pkt_ud
$bfm_send_next_pkt_uf	bfm_send_next_pkt_uf
$bfm_send_next_pkt_nf	bfm_send_next_pkt_nf
$bfm_get_next_pkt_ud	bfm_get_next_pkt_ud
$bfm_get_next_pkt_uf	bfm_get_next_pkt_uf
$bfm_get_next_pkt_nf	bfm_get_next_pkt_nf
$bfm_change_param	bfm_change_param
$bfm_reset	bfm_reset
$bfm_get_simulation_time	bfm_get_simulation_time
$bfm_pulse_event	bfm_pulse_event

Aurora 8B/10B Architecture Model

The Aurora 8B/10B architecture model (AAM) emulates the behavior of the Aurora 8B/10B protocol in C. There is a very close relationship between the AAM and the actual implementation of the design. The AAM is derived from the *Aurora 8B/10B Protocol Specification* (SP002) www.xilinx.com/aurora and is used as a *golden model* to verify the protocol conformance of the DUT.

The AAM's goal is to only check the Aurora 8B/10B implementation, not the SERDES operation in the real world or the implementation of the user interface in the DUT. The AAM contains functions that model the individual components of the Aurora 8B/10B protocol.

The AAM is not meant to be cycle-accurate internally. The latencies seen by the DUT are not modeled in the AAM. Any timing-related behavior (for example, pipelining data to meet timing) is not modeled in the AAM. However, the MGT architecture model (MAM) contains hooks to skew lanes to model line characteristics.

Figure 1-2, page 14 describes a test environment example that shows how the ABFM 8B/10B can be used in loopback mode to test the DUT. Packets from a file can be loaded

into the ABFM 8B/10B by making system calls in the wrapper for ABFM 8B/10B blocks that are executed through the PLI interface block. The ABFM 8B/10B then transmits the data across the serial TX port. This data is received by the DUT through the wrapper for ABFM 8B/10B and PLI interface blocks. The DUT processes the data and loops back the received packets through the serial RX port. This data is processed by the ABFM 8B/10B which dumps the packets into a file. The user can verify the results by comparing the files from which the packets were sent and the files into which the packets were received.

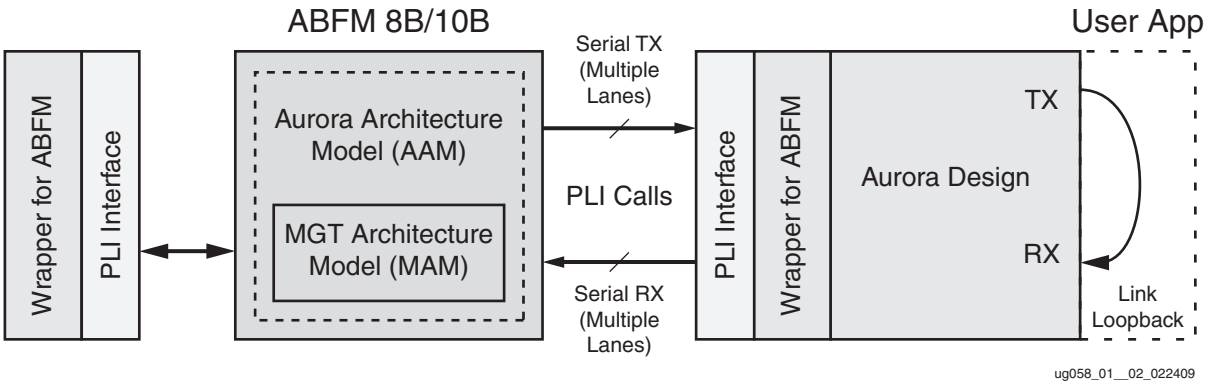

Figure 1-2: **ABFM 8B/10B Test Environment Example**

Figure 1-3, page 15 shows the functional blocks in the ABFM 8B/10B. The functions modeled in the AAM are listed as follows:

- **Link Encap**: Encapsulates each packet with delimiters and adds padding if necessary to create an even-sized encapsulated packet.
- **Src Wait En**: Forces the transmit scheduler to generate idle sequences. This emulates breaks in the data stream received by an Aurora 8B/10B receiver from the source device.
- **TX Scheduler**: Evaluates the next command in the pipeline based on commands currently pending and the priority order of command types. It causes the appropriate symbols and control words to be sent to the lane striper.
- **Lane Striper**: Receives data from the transmit scheduler and stripes it across multiple lanes.
- **Lane Distributor**: Receives the two or four bytes for each lane from lane striper and sends one byte after another to the 8B/10B encode function.
- **8B/10B Encode**: Encodes the received data and generates/monitors the running disparity.
- **Serializer**: Sends out the encoded data one bit at a time.
- **Deserializer**: Accumulates serial data received on the line, one bit at a time.
- **Comma Align**: Performs comma alignment and locks to the code boundary of the incoming data.
- **8B/10B Decode**: Decodes the received data, performs running disparity checks, and sends the data/control symbols to lane destriper.
- **Lane Assembler**: Receives one decoded byte after another from 8B/10B decoder, assembles enough to prepare two or four bytes for the lane and sends it to the lane destriper.
- **Lane Destriper**: Assembles data across all lanes and sends data to the receiver distributor.

- **RX Distributor**: Segregates incoming data into *user protocol data unit* (user PDU), *user flow control* (UFC) PDU, *native flow control* (NFC) PDU; drops idles and *clock compensation* (CC) sequences; sends command to transmit scheduler if channel partner NFC PDU is received.
- **Link Decap**: Drops encapsulation delimiters and padding.
- **Ch Init Verif**: Performs channel initialization, bonding, and verification for all lanes; overrides data being sent from transmit scheduler until successful completion of channel initialization.

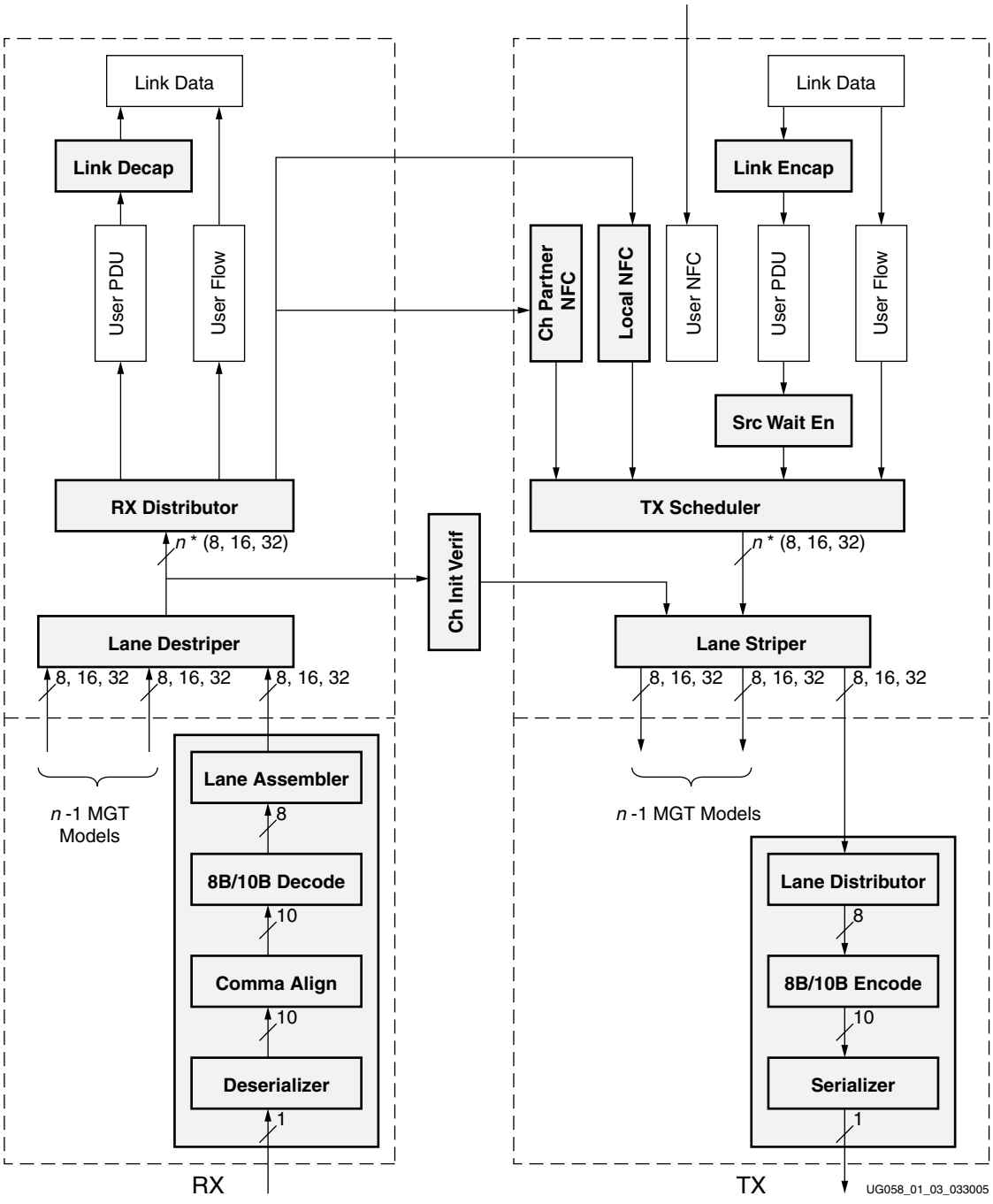

Figure 1-3: **ABFM 8B/10B Functions**

User Application Interface

The ABFM 8B/10B uses file transfer techniques to model the user application interface. This interface enables the user to send and receive user PDU, UFC PDU, and NFC PDU packets to and from the ABFM 8B/10B. The packets are generated by a pre-processor (described in Packet Generator, page 37). The packet format uses simple delimiters such as #SOF and #EOF to describe packets. By default, each line has eight bytes (emulating a 64-bit interface).

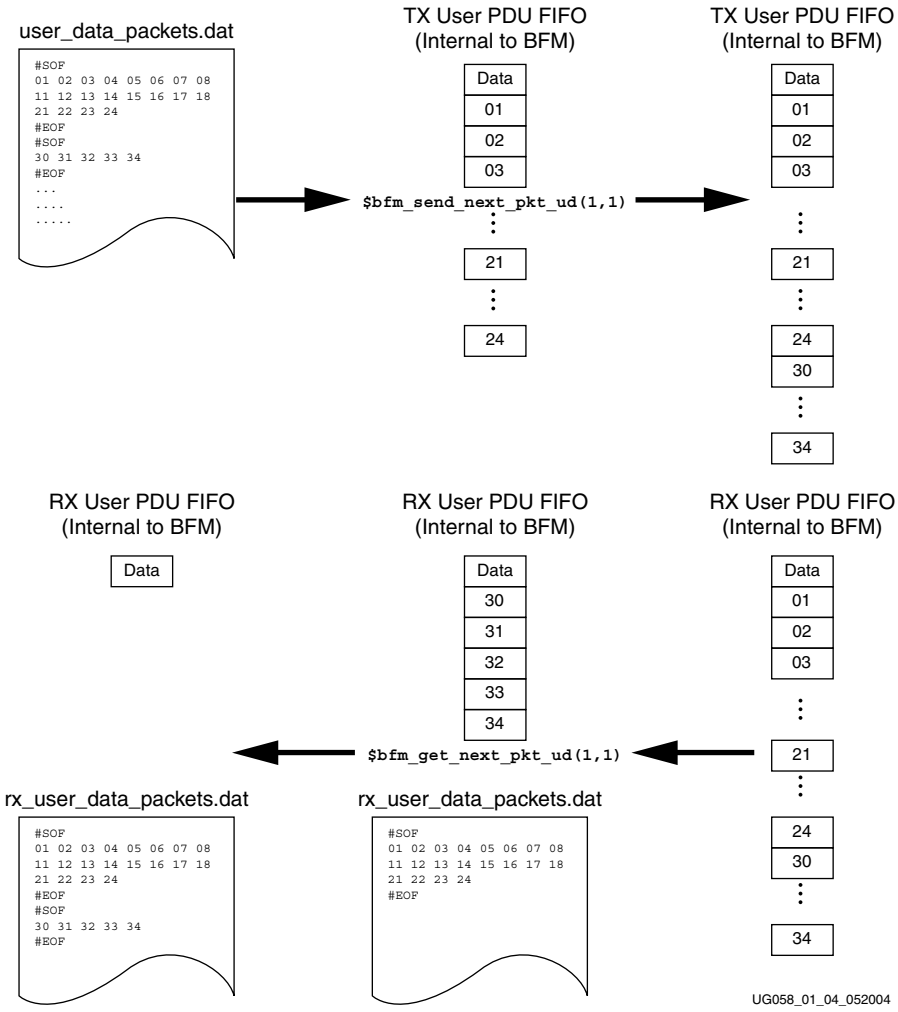

Figure 1-4: **Sending/Receiving User PDU Packets to/from the ABFM 8B/10B**

Several options are provided to modify the packet formats. These are described in detail in Packet Generator, page 37. Internal FIFOs (three types: user PDU, UFC, and NFC) in the ABFM 8B/10B are created for both TX path (TX FIFOs) and RX path (RX FIFOs) of the ABFM 8B/10B. Figure 1-4 is an example showing how user PDU packets are sent to the ABFM 8B/10B from a file and how they are retrieved from the ABFM 8B/10B into a file.

In this particular example, the file that stores packets to be transmitted is called user_data_packets.dat and the file that receives packets is called rx_user_data_packets.dat. First, the user needs to open these files and associate them to the FIFOs in the BFM. This is done using the following system calls in the test bench (refer to Chapter 2, PLI Interface and Test Bench Integration):

- $bfm_read_file_ud(id,"user_data_packets.dat"), which associates the file user_data_packets.dat to TX user PDU FIFO
- $bfm_write_file_ud(id,"rx_user_data_packets.dat"), which associates the file rx_user_data_packets.dat to RX user PDU FIFO

Once the files are opened and associated with the FIFOs, the user can send or receive any number of packets from the internal FIFOs using the following system calls:

- $bfm_send_next_pkt_ud(id,1);
- $bfm_get_next_pkt_ud(id,1);

The example shows how two packets are loaded one after the other into the FIFO showing the state of the FIFO after each system call. It also shows how two packets are drained from the FIFO into a file showing the status of the FIFO and the file after each system call. The transmit scheduler in the ABFM 8B/10B monitors all the TX FIFOs to determine what needs to be sent on the channel. The RX distributor in the ABFM 8B/10B distributes the received data from the channel into the appropriate RX FIFOs. This provides a simple mechanism for emulating the user application.

An initialization monitor signal init_monitor[0:no_of_lanes+3] is provided in the ABFM 8B/10B to monitor the status of the ABFM 8B/10B state machines.

The mapping of the bits in the signal are described below:

- 0:no_of_lanes-1: Lane up indicators for each lane
- no_of_lanes: Successful completion of channel bonding
- no_of_lanes+1: Successful completion of channel verification
- no_of_lanes+2: Channel up

Clocking Scheme

The ABFM 8B/10B communicates with the DUT through a serial interface. The ABFM 8B/10B requires a clock input that matches the serial data rate of the DUT. The serial clock, however, is internally generated and inaccessible outside the DUT. Therefore, a serial clock must be generated to match the serial data rate in the test bench. This serial clock should be derived from the clock input to the DUT.

The RX path of the DUT recovers the clock from the incoming data for its internal clocking. The ABFM 8B/10B cannot perform clock recovery, so it expects data and a clock (synchronous to the data) to be provided by the test bench. This clock is used to clock the serial RX path of ABFM 8B/10B. Data from the serial TX path of the ABFM 8B/10B is transmitted synchronous to the same clock. Hence, the ABFM 8B/10B has both TX and RX paths running on the same clock.

The internal functions of the ABFM 8B/10B run on emulated clocks derived from the serial RX clock. The different clock zones in the ABFM 8B/10B are shown in Figure 1-5 for reference. Clock emulation is performed by controlling how often a function is called in relation to the number of clock cycles in the simulation. A function called every 10 clocks runs on a divide-by-10 clock. Similarly a divide-by-20 clock can be generated. A divide-by-*x* clock is emulated by calling a function every *x* clock cycles. The clock zoning of ABFM 8B/10B is for reference only and should not be confused with the internal clocking of the DUT.

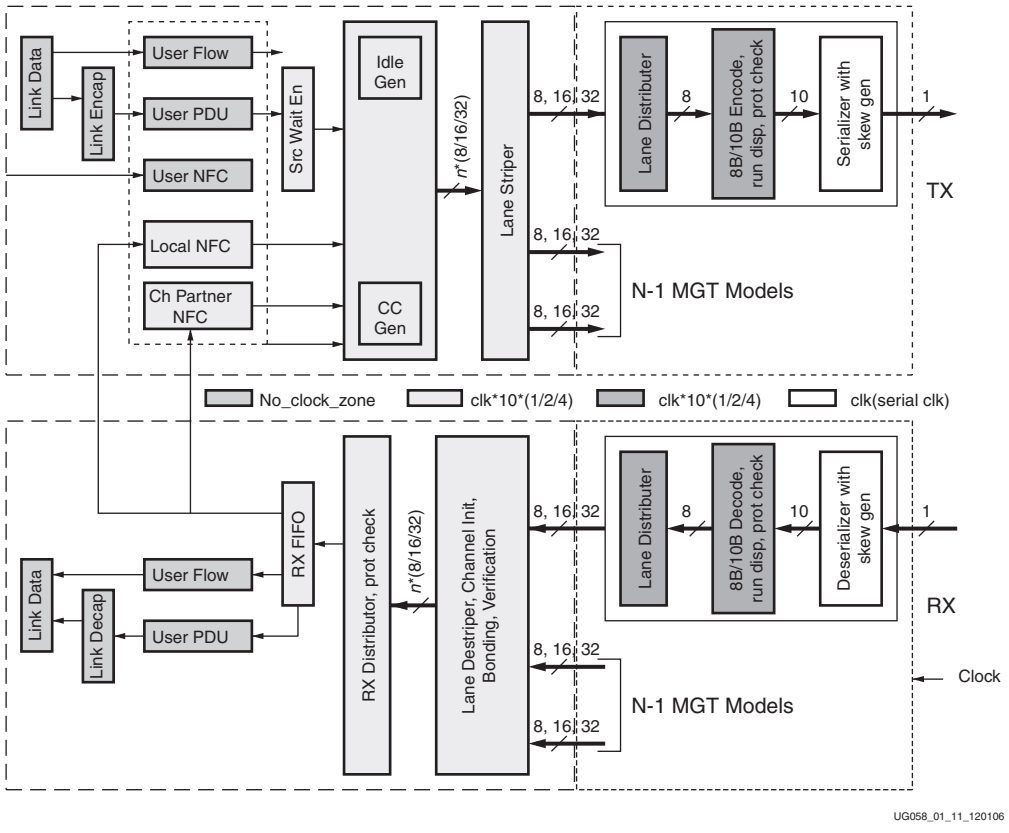

Figure 1-5: **Clocking Scheme**

Error Injection

Error injection capabilities are provided in the ABFM 8B/10B to manipulate the data on the TX path of ABFM 8B/10B.

The error injection capabilities are classified into two categories:

- Symbol Corruption, page 20
- Symbol Deletion, page 22

The user can use this capability to inject errors on the TX path of ABFM 8B/10B and force the RX path of the DUT to process errors and stress corner cases. The $bfm_change_param call (described in $bfm_change_param(id,param1,val,param2..,paramn,valn), page 31) is used to inject errors. The specific call and the parameters are described in BNF format as shown in Figure 1-6. A tabular format is also provided in Table 2-2, page 32.

```
$bfm_change_param("error_inject",error_descriptor)
error_descriptor :=     "error_type error_pattern" |
                        "error_type error_pattern error_size" |
                        "error_type error_pattern error_size symbol_select"
error_type := scp | ecp | suf | snf | idle_a | idle_k | idle_r | sp | spa | v | cc | data
error_pattern := on | off | single | pulse
error_size := 0- 28
symbol_select := bit bit bit bit
bit := 0 | 1
```

Figure 1-6: **BNF Description of the $bfm_change_param Call**

The parameters shown in Figure 1-6 are described in detail below:

- **error_type:** Takes the following values: scp, ecp, suf, snf, idle_a, idle_k, idle_r, sp, spa, v, cc, and data. Details of error_type are provided in Symbol Corruption, page 20.
- **error_pattern:** Turns error injection on or off on a continuous basis, one shot or periodically, and can take the following values:
  - **on:** Turn on error injection. Errors are inserted for all occurrences of error_type until turned off.
  - **off:** Turn off error injection. Errors are not inserted for any occurrences of error_type until turned on.
  - **single:** Apply error insertion to the next occurrence of error type only. Overrides *off* condition.
  - **pulse:** Apply the parameters in the $bfm_pulse_event call to the same error_type.
- **error_size:** Describes the number of bits to corrupt and whether the control bit is to be corrupted or not. Takes the following values 0-8, 10-18, 20-28.
  - **0-8:** Corrupt the defined number of bits. Locations are chosen randomly. Do not corrupt the control bit.
  - **10-18:** Corrupt the defined number of bits (between 0-8). Locations are chosen randomly. Corrupt control bit.
  - **20-28:** Delete 0-8 symbols of specified error_type.

  Examples:
  - 10 = Corrupt the control bit only.
  - 5 = Corrupt five bits of the byte, but do not corrupt the control bit.

- **symbol_select**: A 4-bit mask which turns on/off error injection of a symbol inside an ordered set. Note that for ordered sets which have two symbols, only the first two bits are used as symbol selects. For ordered sets which have one symbol, this parameter is ignored.

  Each bit in the 4-bit mask indicates whether error injection needs to be applied to the symbol. A 0 indicates no error injection and a 1 indicates error injection.

  Default: `1111`. Indicates error injection needs to be applied to all symbols in an ordered set.

  Example: `1000`. Indicates error injection is applied only to the first symbol of the ordered set.

Symbol Corruption

The user can corrupt any symbol at any given instant in the simulation. The user does not have visibility into the internal state of the TX path of ABFM 8B/10B. The ABFM 8B/10B could be sending symbols from any of the following strings at a given instant: user PDU, UFC message, NFC message, idles, CCs. In addition, within a certain type of PDU/message, the ABFM 8B/10B could be sending the header (/SCP/, /SUF/, /SNF/), trailer (/ECP/) or data. Therefore, when a call is made to corrupt a certain type of symbol, it is applied to the next valid occurrence of the symbol.

If consecutive calls are made before the occurrence of a certain type of symbol, then only the most recent call is processed. For symbol corruption, the *error_size* parameter needs to be programmed to values between 0-18.

The following types of symbol corruption are provided:

- **/SCP/:** Corrupts one or both symbols in /SCP/. The symbols within /SCP/are chosen randomly at every call. The next User PDU is targeted for corruption.

  ```
  error_type = scp
  ```

- **/ECP/:** Corrupts one or both symbols in /ECP/. The symbols within /ECP/ are chosen randomly at every call. The next occurrence of /ECP/ is targeted for corruption.

  ```
  error_type = ecp
  ```

- **/PAD/:** Corrupts the next occurring /PAD/ symbol.

  ```
  error_type = pad
  ```

- **/SUF/**: Corrupts the /SUF/ symbol of the next UFC message.

  ```
  error_type = suf
  ```

- **/SNF/:** Corrupts the /SNF/ symbol of the next NFC message.

  ```
  error_type = snf
  ```

- **/A/,/K/,/R/:** Corrupts the next occurrence of the specified idle character.

  ```
  error_type = idle_a/idle_k/idle_r
  ```

- **/SP/:** Corrupts one or more symbols in the next occurrence of /SP/ sequence. The symbols corrupted are chosen randomly.

  ```
  error_type = sp
  ```

- **/SPA/:** Corrupts one or more symbols in the next occurrence of /SP/ sequence. The symbols corrupted are chosen randomly.

  ```
  error_type = spa
  ```

- **/V/:** Corrupts one or more symbols in the next occurrence of /V/ sequence. The symbols corrupted are chosen randomly.

  ```
  error_type = v
  ```

- **/CC/:** Corrupts one or more symbols in the next occurrence of the /CC/ sequence.

  ```
  error_type = cc
  ```

- **Data:** Corrupts the next data symbol.

  ```
  error_type = data
  ```

Examples of symbol corruption are shown in Figure 1-7.

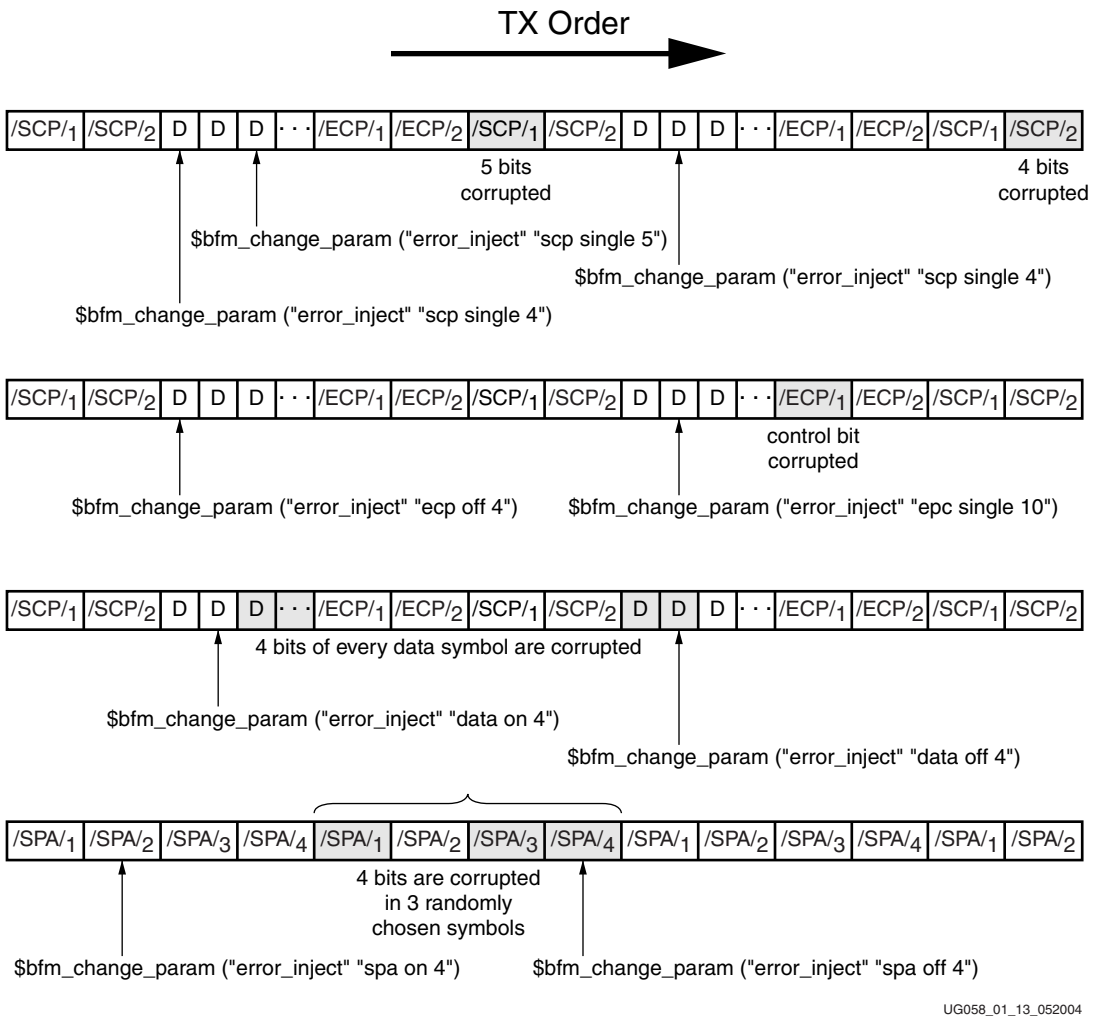

Figure 1-7: **Symbol Corruption Examples**

Symbol Deletion

The user can delete any symbol at any given instant in the simulation. When a call is made to delete a certain type of symbol, it is applied to the next valid occurrence of the symbol. If consecutive calls are made before the occurrence of a certain type of symbol, then only the most recent call is processed. For symbol deletion, the *error_size* parameter needs to be programmed to values between 20-28. The second digit indicates the number of consecutive symbols to delete. The following types of symbol deletion are provided:

- **/SCP/:** Deletes one or both symbols in /SCP/. Valid error_size values are 20-22.

  ```
  error_type = scp
  ```

- **/ECP/:** Deletes one or both symbols in /ECP/. Valid error_size values are 20-22.

  ```
  error_type = ecp
  ```

- **/PAD/:** Deletes the next occurring /PAD/ symbol. error_size value is ignored.

  ```
  error_type = pad.
  ```

- **/SUF/:** Deletes the /SUF/ symbol of the next UFC message. error_size value is ignored.

  ```
  error_type = suf
  ```

- **/SNF/:** Deletes the /SNF/ symbol of the next NFC message. error_size value is ignored.

  ```
  error_type = snf
  ```

- **/A/,/K/,/R/:** Deletes one or more idle characters in the next occurrence of the specified idle character. Valid error_size values are 20-28.

  ```
  error_type = idle_a,idle_k,idle_r
  ```

- **/SP/:** Deletes one or more symbols in the next occurrence of /SP/ sequence. Valid error_size values are 20-24.

  ```
  error_type = sp
  ```

- **/SPA/:** Deletes one or more symbols in the next occurrence of /SPA/ sequence. Valid error_size values are 20-24.

  ```
  error_type = spa
  ```

- **/V/:** Deletes one or more symbols in the next occurrence of /V/ sequence. Valid error_size values are 20-24.

  ```
  error_type = v
  ```

- **/CC/:** Deletes one or more symbols in the next occurrence of the /CC/ sequence. Valid error_size values are 20-28.

  ```
  error_type = cc
  ```

- **Data:** Deletes one or more data symbols in the next occurrence of data symbols. Valid error_size values are 20-28.

  ```
  error_type = data
  ```

Examples of symbol deletion are shown in Figure 1-8, page 23.

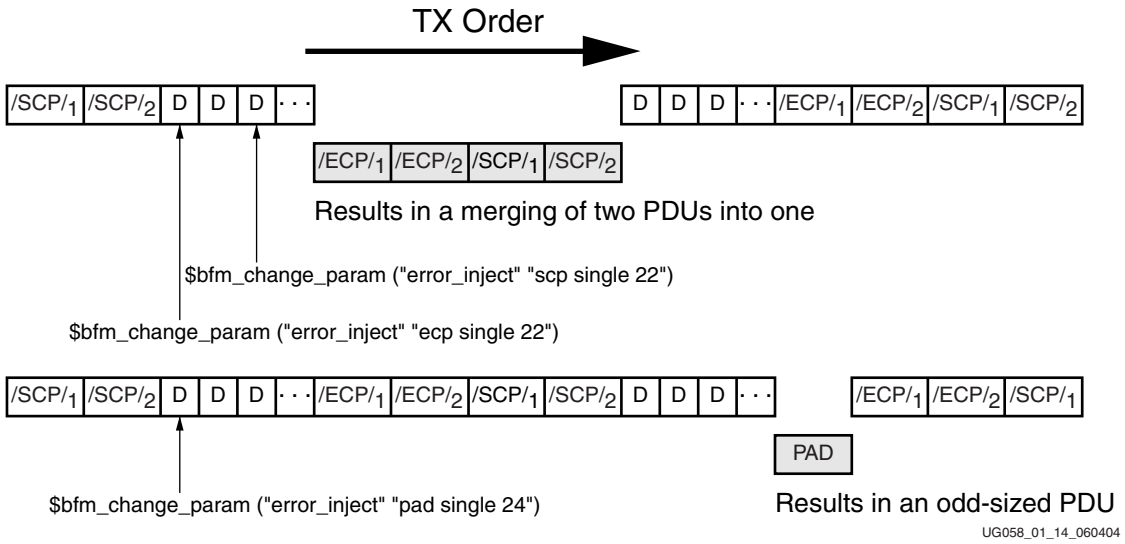

Figure 1-8: **Symbol Deletion Examples**

Periodic injection of errors can be accomplished using the $bfm_change_parameter call by setting error_pattern:= pulse (refer to Error Injection, page 19). Whenever this setting is used, the $bfm_pulse_event call (refer to $bfm_pulse_event(id,"sig name","width_str","freq_str","val_str",...);, page 35) can be used in conjunction with the $bfm_change_param call. Another requirement is that the *error_type* value in the $bfm_change_param call and the *sig_name* value in the $bfm_pulse_event call should be identical.

The $bfm_pulse_event call is described in BNF format as shown in Figure 1-9. A tabular format is also provided in Table 2-3, page 36.

```
$bfm_pulse_event(id, pulse_event_descriptor)
pulse_event_descriptor := "sig_name width_str freq_str val_str"
sig_name := error_type | src_wait_en
error_type := scp | ecp | suf | snf | idle_a | idle_k | idle_r | sp | spa | v | cc | data
width_str := w inc min max
freq_str := f inc min max
val_str := v pattern val
inc := integer
min := integer
max := integer
pattern := fixed | toggle | inc | rand
val := bit string (any number of bits)
```

Figure 1-9: **BNF Description of the $bfm_pulse_event Call**

Three examples of error injections are shown in Figure 1-10. In the first example, two calls are made. The first call, $bfm_change_param("error_inject", "scp pulse 4"), enables the pulse mode for the error injection and defines symbol corruption by corrupting four bits. The second call, $bfm_pulse_event ("scp" "w 1 1 1 0" "f 3 3 3 0","val fixed 1"), applies this error at a periodic rate. Errors are injected for one /SCP/ occurrence followed by two occurrences when there is no injection. This repeats every three occurrences. The remaining two examples show the variation of the error injection scenarios.

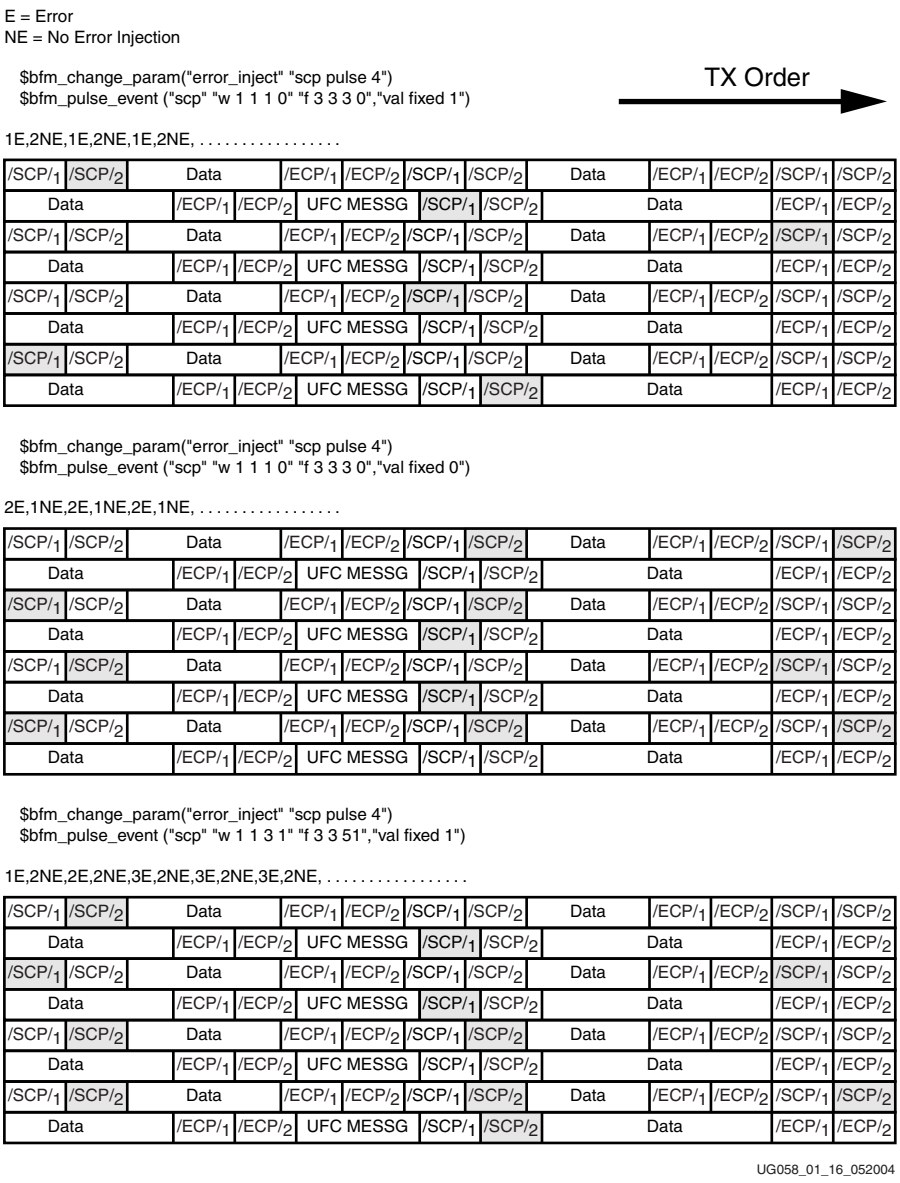

Figure 1-10: **Error Injection with Pulse Event**

Note: Limited testing done on Error Injection feature.

XILINX®

Chapter 2

PLI Interface and Test Bench Integration

Scope

This chapter describes the details of the programming language interface (PLI) and its integration into the test bench. The PLI is a collection of functions written in C and is classified in two sections:

- Library Functions
- User Functions

Library Functions

All simulators provide a PLI Library as part of the package. The PLI Library is based on the *IEEE 1364 Verilog PLI Standard*. It contains a large collection of predefined functions, which provide a powerful interface from C to Verilog. There are two categories of functions: TF and ACC. Both types of functions were used to develop the ABFM 8B/10B. These functions are similar to the system functions in C. Any further discussion on the Verilog PLI is beyond the scope of this document.

User Functions

The user functions are written in C and use the library functions to build powerful routines to communicate with the Aurora 8B/10B architecture model (AAM) and the Verilog test bench. The user functions act as a master to the AAM and as a slave to the Verilog test bench. The user makes calls to these functions using system calls in the Verilog test bench. All system calls in Verilog start with the $ symbol. For example, $display and $finish are Verilog system calls that are built into the Verilog language. The ABFM 8B/10B system calls are prefixed with $bfm_ to differentiate them from the Verilog calls. The names of the user functions very closely match the names of the system calls. Because two different system calls can call the same user function, there cannot be an exact 1:1 match between system calls and user functions. A mapping structure describes the mapping between system calls and user functions, which is described in the *Verilog PLI Standard*.

Note: The PLI is compiled into a shared library in binary format.

Test Bench Integration

This section is the most relevant from the user's perspective. The user can make system calls in the test bench to communicate with the ABFM 8B/10B. It is recommended to instantiate the BFM calls in a separate Verilog wrapper whose input/output ports match that of the ABFM 8B/10B. The system calls provided in the ABFM 8B/10B are described in System Calls, page 26.

System Calls

All system calls support multiple instantiations of the ABFM 8B/10B. The very first argument to all system calls is *id*. To initialize two separate ABFM 8B/10Bs, the following calls can be made:

- $bfm_initialize(1,…);
- $bfm_initialize(2,…);

$bfm_initialize(id,in_port1,..,in_portn,out_port1,..,out_portm,arg1,..,argx)

This system call initializes the ABFM 8B/10B. The PLI directly probes the module (the Verilog wrapper) and identifies the input/output/inout ports and their widths based on the declarations in the module. This allows the ABFM 8B/10B to do a check to ensure that all input/output/inout ports are indeed passed as arguments to this system call. The system call expects all the ports and any additional arguments to be passed as arguments. This allows the PLI to store handles for all the arguments, which is mandatory. The first three arguments of this call are reserved for id, number of inputs, and number of outputs. The system call requires that the arguments occur in the following order:

1. Inputs (in the same order as declared in the module)
2. Outputs (in the same order as declared in the module)
3. Inouts (in the same order as declared in the module)
4. Other arguments (in an order chosen by the user)

While it is mandatory to declare all ports, additional arguments can be chosen based on necessity.

The Aurora 8B/10B tester module, ABFM_8B_10B_tester_pli.v, has the following declarations for a four-lane single channel design:

```
input         clk;
input         rst;
input  [0:3]  rx_p;
input  [0:3]  rx_n;

output [0:3]  tx_p;
output [0:3]  tx_n;

parameter     GLOBALDLY = 1;
parameter     MODE      = 0; // Serial Duplex
parameter     INPUTS    = 4;
parameter     OUTPUTS   = 2;

reg         tx_fifo_full_ud,tx_fifo_full_uf,tx_fifo_full_nf;
reg         rx_fifo_full_ud,rx_fifo_full_uf,rx_fifo_full_nf;
reg         rx_fifo_empty_ud,rx_fifo_empty_uf,rx_fifo_empty_nf;
integer     rx_pkts_in_fifo_ud,rx_pkts_in_fifo_uf,rx_pkts_in_fifo_nf;
```

Therefore, the system call in the test bench is specified as follows:

```
initial begin
$bfm_initialize(1,MODE,INPUTS,OUTPUTS,clk,rst,rx_p,rx_n,tx_p,tx_n,tx_f
ifo_full_ud,tx_fifo_full_uf,tx_fifo_full_nf,
            rx_fifo_full_ud,rx_fifo_full_uf,rx_fifo_full_nf,
            rx_fifo_empty_ud,rx_fifo_empty_uf,rx_fifo_empty_nf,
            rx_pkts_in_fifo_ud,rx_pkts_in_fifo_uf,rx_pkts_in_fifo_nf,
            conf_monitor,result_monitor,init_monitor);
end
```

Table 2-1 shows the valid values for the MODE parameter used in the `bfm_initialize` task.

Table 2-1: **MODE Parameter Values**

MODE	Value
0	Duplex
1	Simplex TX
2	Simplex RX
3	Both (Simplex TX and RX)
Other	Invalid

The above call is executed once at the start of the simulation. The FIFO full and empty signals are used to monitor the status of the internal memory of the ABFM 8B/10B. The user can monitor these signals to control sending packets to the TX FIFOs or draining packets from the RX FIFOs. The exact position index of the arguments in this call needs to be maintained to map these signals to the internal structures in the ABFM 8B/10B.

Usage examples are provided under sections $bfm_send_next_pkt_ud(id,no_of_pkts), page 29 and $bfm_get_next_pkt_ud(id,no_of_pkts), page 30.

If additional ports and arguments are added to the module, then the AAM code and the system call arguments list change accordingly and alter the specification. The new ports/arguments add functionality to the ABFM 8B/10B. To track these values, additional handles need to be created in the AAM code. (If they do not affect the ABFM 8B/10B, they should not be declared in this module in the first place.)

$bfm_execute(id)

This system call is the easiest to understand and to use. There are no arguments. All the handles are picked up by the $bfm_initialize call. Every call to this function does the following tasks in order:

1. Fetch all input values (from RX serial ports)
2. Execute RX path
3. Execute TX path
4. Drive all output values (to TX serial port outputs with respect to the ABFM 8B/10B)

The system call is used in the test bench as follows:

```
always @(posedge clk)
 begin
    $bfm_execute(id);
 end
```

The above usage calls the $bfm_execute every clock cycle. At every clock cycle, the state machines in the ABFM 8B/10B transition to the next state and update all internal signals.

Note: The clock used here is the serial clock for the RX path. This is used as the primary clock for the ABFM 8B/10B.

$bfm_read_file_ud(id,"*filename*")

This system call opens file *"filename"* for reading and stores away the file pointer in a user_data file pointer. It expects *"filename"* to contain user data packets. A usage example is shown as follows:

```
initial
begin
$bfm_read_file_ud (1,"user_data_packets.dat");
end
```

The above function needs to be called only once.

$bfm_read_file_uf(id,"*filename*")

This system call opens file *"filename"* for reading and stores away the file pointer in a user_flow file pointer. It expects *"filename"* to contain user flow packets. A usage example is shown as follows:

```
initial
begin
$bfm_read_file_uf (1,"user_flow_packets.dat");
end
```

The above function needs to be called only once.

$bfm_read_file_nf(id,"*filename*")

This system call opens file *"filename"* for reading and stores away the file pointer in a native_flow file pointer. It expects *"filename"* to contain native flow packets. A usage example is shown as follows:

```
initial
begin
$bfm_read_file_nf (1,"native_flow_packets.dat");
end
```

The above function needs to be called only once.

$bfm_write_file_ud(id,"*filename*")

This system call opens file *"filename"* for writing and stores away the file pointer in a user_data file pointer. The BFM dumps the packets received on the RX interface into this file. A usage example is shown as follows:

```
initial
begin
$bfm_write_file_ud (1,"rx_user_data_packets.dat");
end
```

The above function needs to be called only once.

$bfm_write_file_uf(id,"*filename*")

This system call opens file *"filename"* for writing and stores away the file pointer in a user_flow file pointer. The BFM dumps user flow packets received on the RX interface into this file. A usage example is shown as follows:

```
initial
begin
$bfm_write_file_uf (1,"rx_user_flow_packets.dat");
end
```

The above function needs to be called only once.

$bfm_write_file_nf(id,"*filename*")

This system call opens file *"filename"* for writing and stores away the file pointer in a native_flow file pointer. The BFM dumps native flow packets received on the RX interface into this file. A usage example is shown as follows:

```
initial
begin
$bfm_write_file_nf (1,"rx_native_flow_packets.dat");
end
```

The above function needs to be called only once.

$bfm_send_next_pkt_ud(id,no_of_pkts)

This system call picks the next *no_of_pkts* packets from the file opened using $bfm_read_file_ud call. The packet delineation is based on *start of file* (SOF) and *end of file* (EOF) tags. It stores the packets in a local, fixed-size array. An example usage is shown as follows:

```
initial
begin
#1000;
$bfm_send_next_pkt_ud(1,4);
#5000;
$bfm_send_next_pkt_ud(1,6);
end
```

This function can be called as many times as the user pleases. When *end of file* is reached, the BFM rewinds to *start of file* and continues fetching packets.

Note: If the last byte of the fourth packet (in the above example) has not yet been transmitted when the next call is executed, though there is a gap of 5000 ns between the two calls, the BFM will treat it like a continuous stream of ten packets. The gaps between calls ensure there is no overflow of array memory, but do not always mean breaks in the data stream.

To ensure that the call does not send more packets than the array can accommodate, it is recommended that the user poll the tx_fifo_full signal before sending more packets. Another example of using the above system call to ensure that the ABFM 8B/10B sees a continuous stream of packets is as follows:

```
always @(posedge clk)
begin
if (tx_fifo_full_ud != 1)
   $bfm_send_next_pkt(1,1);
end
```

The above code ensures that as soon as there is room in the internal array to store another packet, the next packet is picked up by the BFM.

Note: User flow messages can be embedded inside the user packets that are picked up with the user PDU.

$bfm_send_next_pkt_uf(id,no_of_pkts)

The system call's description is the same as $bfm_send_next_pkt_ud(id,no_of_pkts), page 29 except that the packets in this file will be user flow packets.

This function can be used to send sideband user flow packets.

$bfm_send_next_pkt_nf(id,no_of_pkts)

This system call can be used to send native mode messages. Although unlikely, this function can be used to send multiple packets back to back.

$bfm_get_next_pkt_ud(id,no_of_pkts)

This system call drains the next *no_of_pkts* packets from the internal array and writes into a file opened using the $bfm_write_file_ud call. The packet delineation format used is based on SOF and EOF tags.

An example usage is shown as follows:

```
initial
begin
#5000;
$bfm_get_next_pkt_ud(1,4);
#7000;
$bfm_get_next_pkt_ud(1,6);
end
```

This function can be called as many times as the user pleases.

The internal RX FIFOs store the data received from the RX distributor function in the ABFM 8B/10B. As the FIFOs are of fixed size, it can fill quickly if $bfm_get_next_pkt_* calls are not made frequently. The status of the FIFOs can be monitored from the test bench through signals rx_fifo_empty_* and rx_fifo_full_*. It is essential for the rx_fifo_empty_* and rx_fifo_full_* signals to be declared in the argument list of the $bfm_initialize call. This establishes a connection to the FIFO signals inside the ABFM 8B/10B.

The user should poll these bits and make calls to keep the FIFO in stable condition.

An example is shown below:

```
always @(posedge clk)
begin
if (rx_fifo_empty_ud == 0)
  if(rx_fifo_full_ud == 1)
    $bfm_get_next_pkt(1,1);
end
```

The above code ensures that as soon as the FIFO is full the next packet is drained by the ABFM 8B/10B. (Threshold values are set to ensure that when a *full* condition is reached, there is room for one more packet of max packet size = 1500 bytes). The FIFO conditions are discussed in $bfm_change_param(id,param1,val,param2..,paramn,valn), page 31, which addresses configuring the ABFM 8B/10B.

$bfm_get_next_pkt_uf(id,no_of_pkts)

This system call drains packets from the user_flow array in the ABFM 8B/10B and written into a file. An option is provided in the ABFM 8B/10B to store embedded user flow messages in the user flow array. By default, embedded user flow messages are stored in user data array. It makes file comparisons easier.

$bfm_get_next_pkt_nf(id,no_of_pkts)

This system call is used to log all the native mode messages received by the ABFM 8B/10B.

$bfm_change_param(id,param1,val,param2..,paramn,valn)

This system call allows the user to modify parameters, configurations, and so forth in real time while the simulation is running. This must be used very judiciously. The arguments to this call must be provided in pairs, with the argument name first and the value second. The user can use this function to configure the number of lanes, the lane width, and whether in immediate or completion mode to suit a specific implementation of an Aurora 8B/10B design.

Table 2-2 shows the names and possible values of arguments incorporated into the ABFM 8B/10B.

Table 2-2: **Arguments and Values for $bfm_change_param Call**

Argument	Value(s)	Default Value	Description
cc_length	> 2	12	Configures the length of a clock sequence per lane.
cc_frequency	> 2	10000	Configures the frequency at which CC sequences need to be sent.
no_of_lanes	1-16	4	Configures the number of lanes.
imm_comp_mode	0,1	0	0 = Immediate mode 1 = Completion mode
lane_skew	0-100 format: "skew0 skew1..."	0	0 = No skew Indicates the transmit delay of serial bits in a lane. Lane skew can be achieved by setting different values for each lane.
codes_per_lane	2,4	2	Configures the length of each code passed through the lane.
src_wait_en	on, off, rand	rand	Injects idle cycles in PDU data to emulate source waits.
tx_fifo_thr_l	1- 20,000 bytes	1500	Indicates the empty condition threshold for all FIFOs used in transmit path (user PDU, UFC, NFC).
tx_fifo_thr_h	1- 20,000 bytes	18500	Indicates the full condition threshold for all FIFOs used in transmit path.
rx_fifo_thr_l	1- 20,000 bytes	1500	Indicates empty condition threshold for all FIFOs used in receive path (user PDU, UFC, NFC).
rx_fifo_thr_h	1- 20,000 bytes	18500	Indicates full condition threshold for all FIFOs used in receive path.
verbose	0-2	off	Indicates verbosity of log messages: 2 = Highest level of verbosity 0 = Lowest level of verbosity

Table 2-2: **Arguments and Values for $bfm_change_param Call *(Cont'd)***

Argument	Value(s)	Default Value	Description
error_inject	error_type	"data"	error_type can take the values: scp, ecp, suf, snf, idle_a, idle_k, idle_r, sp, spa, v, cc, data
	error_pattern	off	error_pattern can take the values: on, off, single, pulse
	error_size	0	error_size can take values: 0-18: For symbol corruption 20-28: For symbol deletion
	symbol_select	1111	Each bit of *symbol_select* can take values 0 or 1: 0 = No symbol corruption 1 = Symbol corruption
	symbol_replace	"data"	error_type can take the values: scp, ecp, suf, snf, idle_a, idle_k, idle_r, sp, spa, v, cc, data
tx_polarity	0,1	0	0 = Normal operation 1 = Invert serial tx polarity
rx_polarity	0,1	0	0 = Normal operation 1 = Invert serial rx polarity
log_error	0-3	0	0: Do not print error log in simulator's output 1: One shot: Print error log once 2: Print error log whenever a new error is encountered 3: Print error log for the next error only
chk_init	0,1	0	0 = Normal mode 1 = Enable initialization conformance
chk_cup_conf	0,1	0	0 = Normal mode 1 = Enable channel up conformance
chk_softerr_conf	0,1	0	0 = Normal mode 1 = Enable soft error conformance
chk_conf	0,1	0	0 = Normal mode 1 = Enable all phases of conformance checks Overrides chk_init, chk_cup_conf and chk_softerr_conf settings.
same_databeat_pkts	0,1	1	0 = Only one packet of same kind can be transmitted in same data beat 1 = Allow more than one packet of same kind (user, UFC, NFC) to be transmitted in same data beat
pdu_lane_start	seq, rand, 0,…, no_of_lanes-1	0	seq = Starting from lane 0, every successive PDU is sent on the next lane in a round robin fashion rand = Lane number is picked randomly 0,..., no_of_lanes -1 = PDU is sent on the indicated lane

Table 2-2: **Arguments and Values for $bfm_change_param Call *(Cont'd)***

Argument	Value(s)	Default Value	Description
echo mode	0,1	1	0 = Normal mode 1 = Sets BFM in *echo* mode. RX looped back to TX
sym_conf_len	0-4	4	Indicates the number of symbols used to match two ordered sets during protocol conformance. For example, 2 indicates that the receive side will only check the first two symbols of /SP/,/SPA/ or /V/ to qualify an *ordered set match.*
system_latency	any integer > 0	20	Worst latency of the DUT (from link interface to serial interface) in units of Aurora 8B/10B cycles. **Note:** A conservative number is recommended while modifying this parameter. This is referenced with serial clock rate /20 clock.
first_sp_thr	any integer > 0	200	Minimum number of Aurora 8B/10B clock cycles after which the DUT is guaranteed to send an /SP/. **Note:** A conservative number is recommended while modifying this parameter.
cup_skip_vector	0	0 - 63	Skips packets according the value set.
automatic_compliance_ufc_size	16	2,4,6…16	Defines UFC size during automatic compliance.

An example usage is:

```
initial
begin
$bfm_change_param (1,no_of_lanes,4,codes_per_lane,2,imm_mode,0);/*804 implementation,
completion mode*/
$bfm_initialize
(1,...,no_of_lanes,codes_per_lane,tx_fifo_thr_h,rx_fifo_thr_l,rx_fifo_thr_h);
end
initial
begin
$bfm_change_param (1,no_of_lanes,4,codes_per_lane,2);/*804 implementation*/
#5000
$bfm_change_param (1,tx_fifo_thr_h,19000);
#7000;
$bfm_change_param (1,rx_fifo_thr_l,1000,rx_fifo_thr_h,18000);
end
```

The above code can be used to configure the full and empty thresholds for the TX FIFOs and RX FIFOs. This provides a mechanism to modify the internal parameters of the ABFM 8B/10B in a convenient way.

$bfm_reset(id)

This system call allows the user to reset the ABFM 8B/10B. This call transfers control of the ABFM 8B/10B to the internal initialization routine. During internal initialization, the FIFOs are flushed, the state machines are reset, and all parameters are set to their initial values. The user can choose to call this function together with the rst signal for the design.

$bfm_get_simulation_time(id)

This system call allows the user to get the simulation time at any given point in the simulation run.

$bfm_pulse_event(id,"sig name","width_str","freq_str","val_str",...);

This call is used to create a periodic or random event on any signal or data bus. The syntax for this call is as follows:

```
$bfm_pulse_event (1,"sig name 1","width_str", "freq_str", "val_str",
"sig name 2",...);
```

Table 2-3 shows the four parameters for the pulse event call.

Table 2-3: **Pulse Event Parameters**

Parameter	String	Description
sig name		Can be a signal name or an internal variable in the ABFM 8B/10B
width_str	w	Qualifies the current parameter as a width parameter
	inc	Indicates the increase in pulse width at the next pulse event
	min	Starting pulse width of the signal
	max	Ending pulse width of the signal **Note:** No more increments of width are performed if the increment results in a value larger than max.
freq_str	f	Qualifies the current parameter as a frequency (period) parameter
	inc	Indicates the increase in period at the next pulse event
	min	Starting period of the signal
	max	Ending period of the signal **Note:** No more increments of period are performed if the increment results in a value larger than max.
val_str	v	Qualifies the current parameter as being the value of the signal pulse
	fixed/toggle /inc/rand	At every period of the signal, the new value of the signal pulse can be one of the following: • fixed - the value determined by the *val* entry • toggle - toggle the value of the previous signal pulse • inc - increment the value • rand - any arbitrary value **Note:** For 1-bit signals, toggle and inc achieve the same result.
	val	The value of the signal pulse when the *fixed* type is used **Note:** *val* fixes the value of the signal during the pulse width phase. During the rest of the period, the value will be the complement.

Examples of the pulse event call and associated parameters are:

- $bfm_pulse_event (1,"clk_div","w 0 1 1","f 0 2 2","v fixed 0") generates a clock signal that is half the frequency of the ABFM 8B/10B clock.
- $bfm_pulse_event (1,"data", "w 1 1 1", "f 2 2 2", "v inc 1") (where data is an 8-bit bus) creates the hexadecimal data 1h, feh, 2h, 2h, fdh, fdh, 3h, 3h, 3h, fch, fch, fch, …

Packet Generator

A packet generator script is used to generate stimulus files used by the test bench. It uses a simple formatting technique that delineates packets using #SOF and #EOF tags. Packets can be configured with the options shown in Table 2-4.

Table 2-4: **Packet Configuration Options**

Option	Default Value	Description
-bw	32	A bus width option that takes integer values as input
-p	10	Determines the number of packets
-l	64 bytes	Packet length If the value is an integer, then the packet length is fixed to that integer. If the value is *var*, then the packet length is variable and is randomly picked between 1 and the max_pkt_length value as determined by the -maxl option.
-maxl	1,500 bytes	Maximum packet length (max_pkt_length) This takes integer values and fixes an upper limit on the size of the packet. This is relevant only when *-l var* option is used.
-ty	ud	Packet type: user data or user flow ud = User data packet uf = User flow packet (not yet implemented)
-seq or -rand	-seq	Determines data byte order within a packet -seq = Sequential -rand = Random numbers between 0 and 255

The user can type:

```
packet_gen.pl -h
```

to display the packet generator options such as:

```
-bw <bus_width>, in bytes. default is 32
-p <no_of_pkts>, default is 10
-l <pkt_len>, takes integer values (default is 100)
     or "var"(randomized packet lengths)
-maxl <max_pkt_len>, takes integer values (default is 1500)
-ty <packet_type>,"ud"(user_data) or "uf"(user flow)
     default is user data
-seq or -rand, data values are generated sequentially or random
              default is sequential
-o <output_file>, name of output file
     default names for "ud" and "uf" are user_data_packets.dat
     and user_flow_packets.dat
```

Simulation Procedure

Simulations can be run using the ABFM 8B/10B with minimal changes to the test environment. ABFM 8B/10B simulations involve the following steps:

1. Instantiate a BFM module

   To instantiate a BFM module, first create a module that contains all of the BFM calls and ports that interface to the rest of the test bench environment. The ABFM_8B_10B_tester_pli.v file described in the System Calls, page 26, can be used as a starting point.

2. Generate packet files

   To generate packet files, use the packet generator script described in Packet Generator, page 37. The packet generator command is executed from the run_sim.pl script, which is used to run the simulation. This causes a new packet data file to be created whenever a simulation is run.

3. Command line options

   A perl script run_sim.pl is provided which is used to run the simulations.

   The script can take the following arguments to run the simulations for a design configuration:

   Simulator, platform, interface, interface width, language, channel partner, number and length of packets for PDU, UFC, and NFC interface, data flow mode, NFC mode, number of lanes and line rate.

   These can be provided to the script either through excel file interface or through command line interface.

   A sample excel file (sim_matrix.xls) is provided to illustrate how they can be given through excel file.

4. Run simulations

   To run simulations with a BFM, an extra option must be declared at run time. For the ModelTech environment, -pli pli_shared_object.sl needs to be added to the runtime command. The current implementation uses the following command:

   ```
   "vsim -c -do vsim.do abfm_tb.v  -pli ABFM_8B_10B.sl "
   ```

5. Check results

   To check results, a simple checktalk routine strips comment lines from the input file and output file and compares them using a *diff* function. It echoes a SIMULATION PASSED or SIMULATION FAILED message based on the result.

   All of the above steps are incorporated into the run_sim.pl script making it relatively easy to use.

XILINX®

Chapter 3

FLI Interface

Scope

The foreign language interface (FLI) provides the ability to interface with the BFM shared library from a VHDL environment. It can be considered as a VHDL equivalent of PLI. FLI is a proprietary language which was developed to provide an interface between VHDL and C/C++ routines only in a ModelSim simulator environment. The FLI and PLI share the Aurora 8B/10B architecture model (AAM). The user functions written for FLI use proprietary structures provided by the ModelSim simulator to communicate with a VHDL test bench.

There are three components required to run a simulation that interfaces with the BFM:

- BFM Shared Library: ABFM_8B_10B.sl
- Foreign Subprograms File: ABFM_8B_10B_fli_subprograms.vhd
- Initialization Function, page 40

Simulation Components

BFM Shared Library

The BFM shared library ABFM_8B_10B.sl is included in the release. This library is compiled using the AAM and user functions written specifically for the FLI.

Foreign Subprograms File

The foreign subprogram file ABFM_8B_10B_fli_subprograms.vhd is included with the release. This file contains the procedure definitions for all the foreign subprograms (equivalent to system calls in the PLI) that are called from the VHDL test bench. This file must be compiled at simulation time.

A foreign subprogram example is shown below for bfm_change_param, the equivalent of a $bfm_change_param (1,"no_of_lanes","4") in a PLI environment. The number of arguments defined in the user function must explicitly map to the number of arguments in the foreign subprogram call. This affects only the bfm_change_param subprogram.

Foreign subprogram definition in ABFM_8B_10B_fli_subprograms.vhd:

```
procedure bfm_change_param(
  id : IN integer;
  param_name   : IN string;
  param_val1   : IN string;
  param_val2   : IN string
  );
```

```
  attribute foreign of bfm_change_param : procedure is
"bfm_change_param ABFM_8B_10B.sl";
procedure bfm_change_param(
      id : IN integer;
      param_name   : IN string;
      param_val1   : IN string;
      param_val2   : IN string
    ) is
    begin
      assert false report "ERROR: foreign subprogram not called"
severity note;
    end;
```

Foreign subprogram call in the test bench:

```
init_cfg: process
  begin
    bfm_change_param(id,"no_of_lanes","4","");
    wait
  end process;
```

Although the fourth parameter is not relevant, this empty parameter is needed. The PLI has the ability to scan the arguments and append empty arguments to satisfy the user function requirements. The FLI, however, cannot do the same, therefore, the empty argument is required. This applies only to the bfm_change_param subprogram. Sample test benches provided in the release further illustrate the usage of the foreign subprograms.

Initialization Function

The initialization function is called and executed during the elaboration phase of the simulator. Only the bfm_initialize function is defined and executed during this phase. It needs to be defined as a separate entity in the test bench. The architecture for this entity must be defined as follows:

```
ENTITY bfm_initialize IS
   port (
        id :  in integer :=0;
        clk  :  in bit := '0';
        rst  :  in bit := '0';
        rx_p  :  in bit_vector(3 downto 0) := "1010";
        rx_n  :  in bit_vector(3 downto 0) := "1010";
        tx_p  :  out bit_vector(3 downto 0) ;
        tx_n  :  out bit_vector(3 downto 0) ;
        tx_fifo_full_ud: out bit;
        tx_fifo_full_uf: out bit;
        tx_fifo_full_nf: out bit;
        rx_fifo_full_ud: out bit;
        rx_fifo_full_uf: out bit;
        rx_fifo_full_nf: out bit;
        rx_fifo_empty_ud: out bit;
        rx_fifo_empty_uf: out bit;
        rx_fifo_empty_nf: out bit;
        rx_pkts_in_fifo_ud: out integer;
        rx_pkts_in_fifo_uf: out integer;
        rx_pkts_in_fifo_nf: out integer;
        conf_monitor   : in bit_vector(7 downto 0);
        conf_result : in bit_vector(NO_OF_LANES+5 downto 0);
        init_monitor   : in bit_vector(NO_OF_LANES+3 downto 0)
        );
```

```
END bfm_initialize;
ARCHITECTURE c_model OF bfm_initialize IS
  attribute foreign : string;
  attribute foreign of c_model : architecture is "bfm_initialize
./ABFM_8B_10B.sl";
  begin
  end;
```

Sample test benches provided in the release further illustrate the usage of the initialization function.

Although the system calls and functions in PLI begin with $, the foreign subprograms and initialization calls in FLI cannot begin with $.

XILINX®

Appendix A

Simplex with a Serial Interface

Introduction

The original architecture model is targeted for duplex designs. This appendix extends the architecture to include simplex designs.

The Aurora 8B/10B simplex BFM supports the verification of Aurora 8B/10B simplex designs. A single BFM is provided to be configured to work as a transmit (TX) BFM, a receive (RX) BFM, or both a TX and RX BFM.

The BFM deliverable contain:

- A test environment containing test benches connecting the serial lines and sideband signals between two BFMs communicating with each other
- Shared libraries containing PLI routines and interface (for Verilog environments)
- Shared libraries containing FLI routines and interface (for VHDL environments).
- An automated script to configure and run the BFM.

The simplex BFM inherits all features in the duplex BFM.

Architecture

The Aurora 8B/10B simplex BFM architecture is shown in Figure A-1, page 44. This architecture is derived by partitioning the duplex architecture into TX/RX and further replacing internal handshake between TX and RX during initialization with external sideband ports. The Aurora 8B/10B simplex BFM can be used in three scenarios:

- **Scenario 1:** A simplex TX BFM talking to a simplex RX DUT (Figure A-2, page 45)
- **Scenario 2:** A a simplex RX BFM talking to a simplex TX DUT (Figure A-3, page 45)
- **Scenario 3:** A a single TX and RX BFM in "both" mode talking to one simplex TX DUT and one simplex RX DUT (Figure A-4, page 46)

Note: For details on the Aurora 8B/10B duplex BFM operation, see the section Aurora 8B/10B Architecture Model, page 13.)

Aurora 8B/10B Simplex BFM Architecture (TX and RX)

Figure A-1 shows the Aurora 8B/10B simplex BFM architecture (TX and RX).

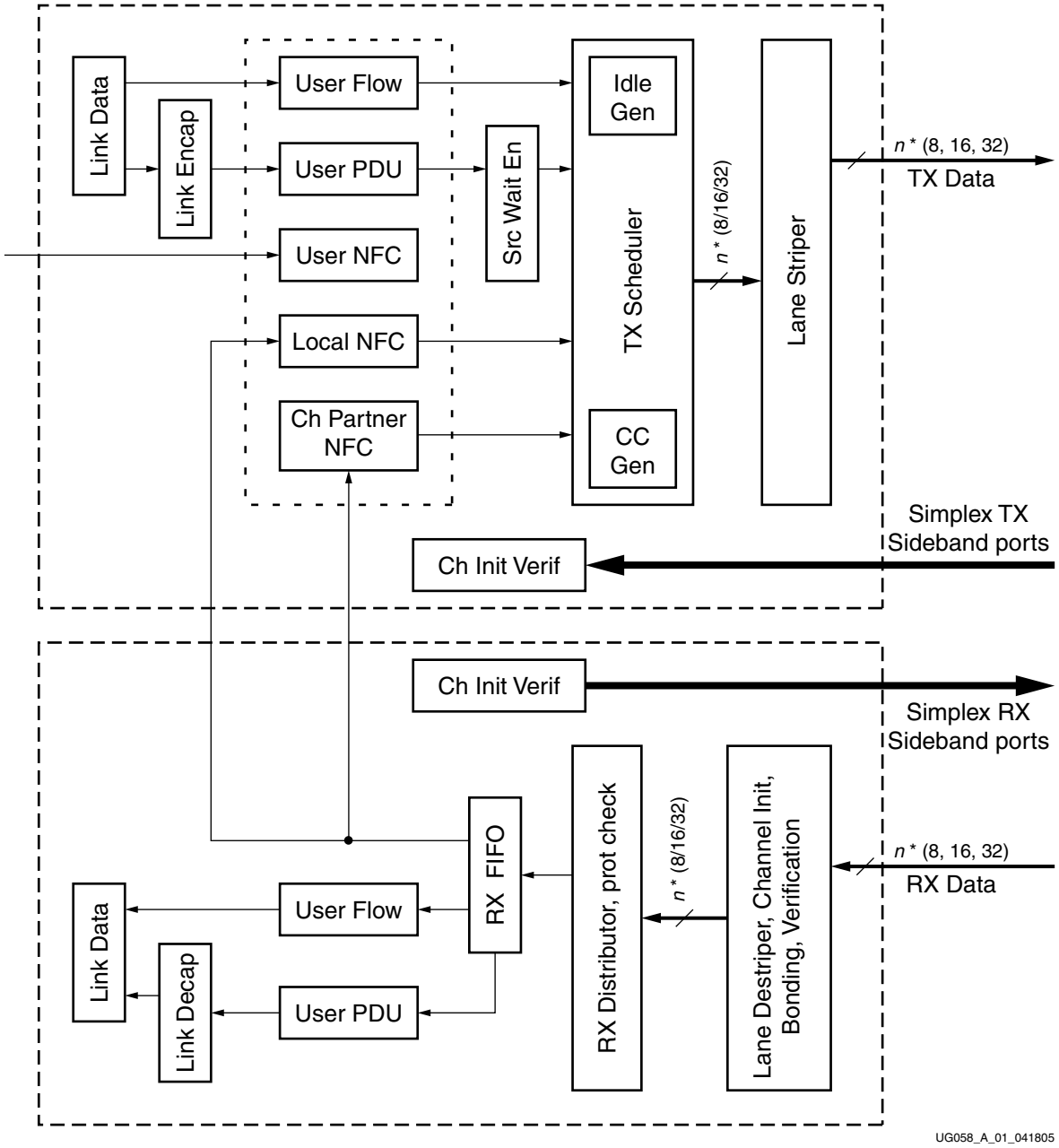

Figure A-1: **Aurora 8B/10B Simplex BFM Architecture (TX and RX)**

Aurora 8B/10B Simplex TX BFM

Figure A-2 shows the Aurora 8B/10B simplex TX BFM module in the test environment with a simplex RX DUT.

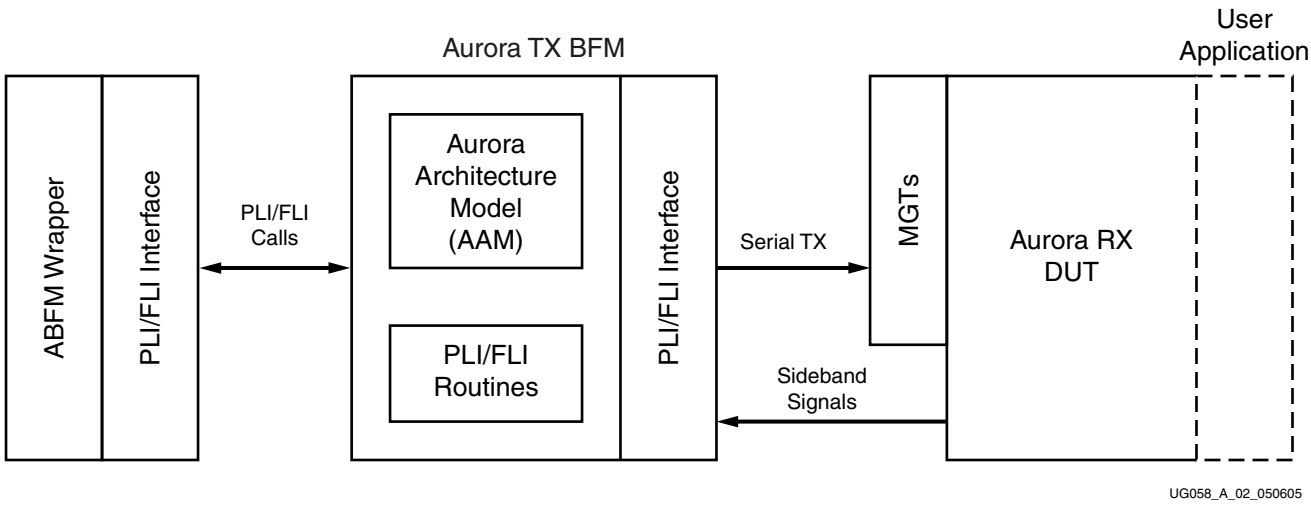

Figure A-2: **Aurora 8B/10B Simplex TX BFM: Test Environment with Simplex RX DUT**

Aurora 8B/10B Simplex RX BFM

Figure A-3 shows the Aurora 8B/10B simplex RX BFM module in the test environment with a simplex TX DUT.

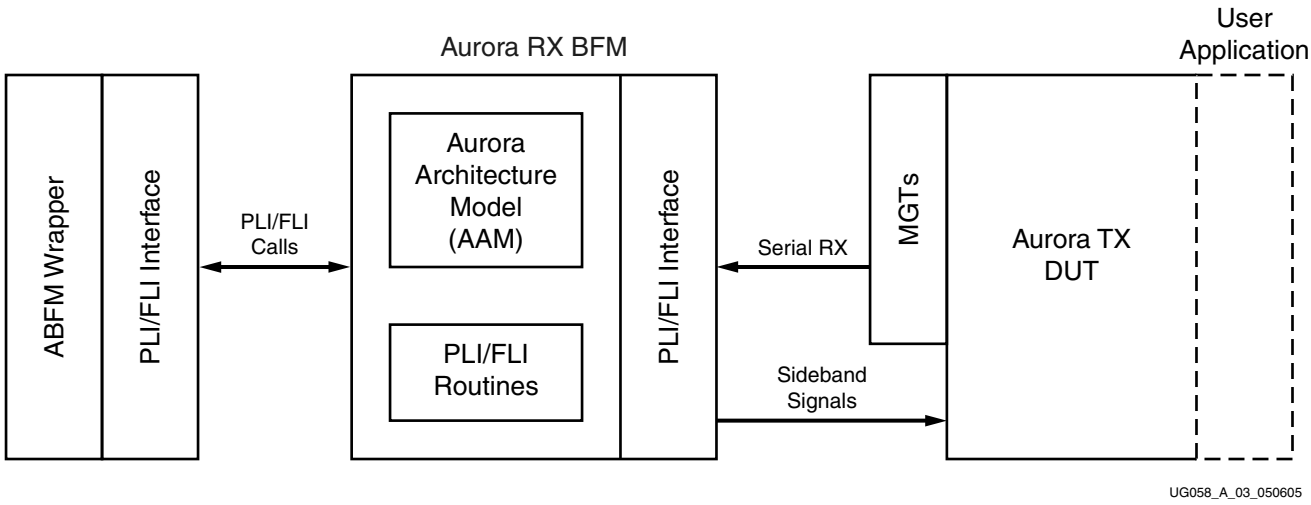

Figure A-3: **Aurora 8B/10B Simplex RX BFM: Test Environment with Simplex TX DUT**

Aurora 8B/10B Simplex "Both" BFM

Figure A-4 shows the Aurora 8B/10B simplex "Both" BFM module in the test environment with a simplex RX DUT and a simplex TX DUT.

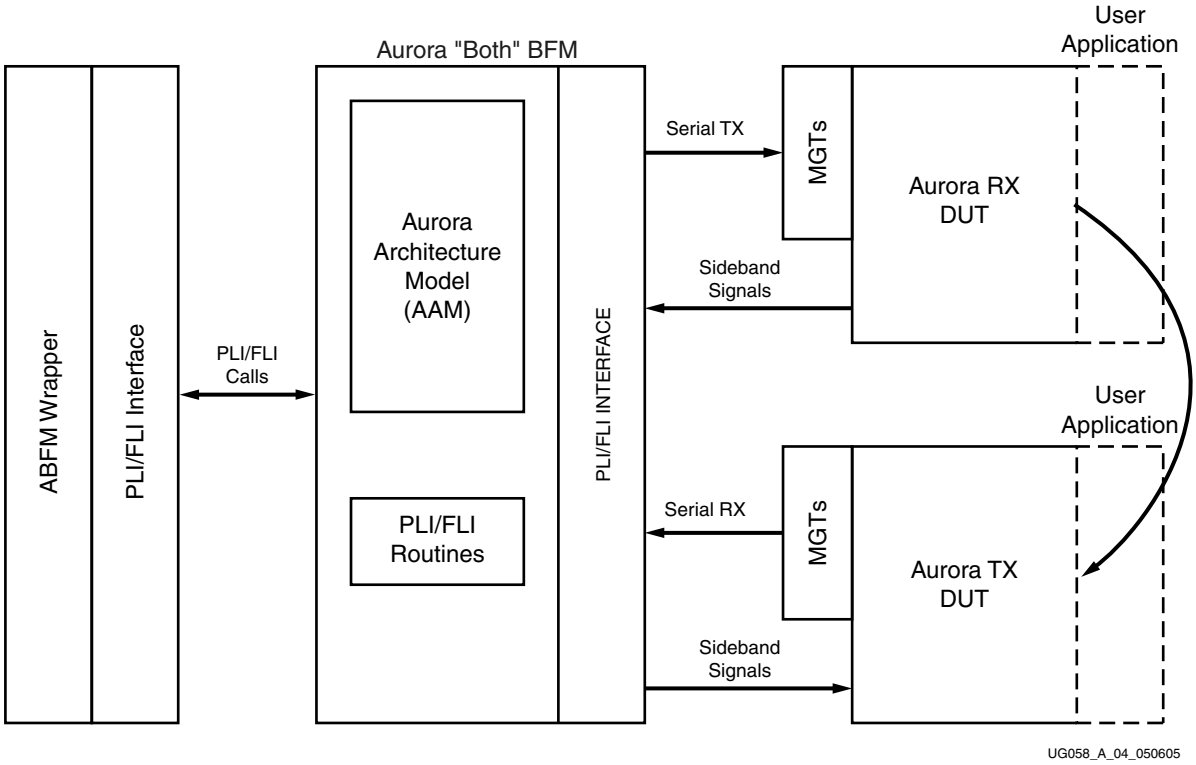

Figure A-4: **Aurora 8B/10B Simplex "Both" BFM: Test Environment with Simplex RX and TX DUTs**

The additional sideband signals for the simplex mode BFMs are described in Table A-1 and Table A-2.

Table A-1: **Sideband Signals for Simplex TX**

Label	Direction	Description
TX_ALIGNED	Input	Connects to RX_ALIGNED signal of partner.
TX_BONDED	Input	Connects to RX_BONDED signal of partner.
TX_VERIFIED	Input	Connects to RX_VERIFIED signal of partner.
TX_RESET	Input	Connects to RX_RESET signal of partner.

Table A-2: **Sideband Signals for Simplex RX**

Label	Direction	Description
RX_ALIGNED	Output	Connects to TX_ALIGNED signal of partner.
RX_BONDED	Output	Connects to TX_BONDED signal of partner.
RX_VERIFIED	Output	Connects to TX_VERIFIED signal of partner.
RX_RESET	Output	Connects to TX_RESET signal of partner.

PLI Interface and Test Bench Integration

The PLI interface for the simplex BFM is very similar to the duplex BFM with a few modifications. The modifications affect only the system call $bfm_initialize. The port descriptions and parameters are defined in $bfm_initialize and $bfm_change_param, page 48.

$bfm_initialize

```
$bfm_initialize(id,mode,no_of_inputs,no_of_outputs,in_port1,..,in_port
n,out_port1,..,out_portm,arg1,..argx);
```

The *mode* parameter in the above system call can take the following values:

- 1 := Simplex TX
- 2 := Simplex RX
- 3 := Both (Simplex TX and RX)

The following statements enable accurate port mapping between the ABFM 8B/10B and the test bench. The BFM package includes examples that shows how the wrapper for the BFM must be written. A specific 804 simplex TX BFM example is shown below:

```
input  clk;
input  rst;
output [0:3] tx_p;
output [0:3] tx_n;
input  tx_aligned;
input  tx_bonded;
input  tx_verified;
input  tx_reset;
reg    tx_fifo_full_ud,tx_fifo_full_uf,tx_fifo_full_nf;
reg    tx_fifo_full_ud,rx_fifo_full_uf,rx_fifo_full_nf;
reg    rx_pkts_in_fifo_ud,rx_pkts_in_fifo_uf,rx_pkts_in_fifo_nf;

reg    [0:7] conf_monitor;
reg    [0:12] conf_result;
reg    [0:6] init_monitor;
```

The system call in simplex TX is:

```
$bfm_initialize(id,1,6,2,clk,rst,tx_aligned,tx_bonded,tx_verified,tx_r
eset,tx_p,tx_n
tx_fifo_full_ud,tx_fifo_full_uf,tx_fifo_full_nf,
rx_fifo_full_ud,rx_fifo_full_uf,rx_fifo_full_nf,
rx_pkts_in_fifo_ud,rx_pkts_in_fifo_uf,rx_pkts_in_fifo_nf,
conf_monitor,conf_result,init_monitor);
```

$bfm_change_param

```
$bfm_change_param(id,param1,val,..,paramn,val)
```

The parameters shown in Table A-3 are added to the BFM parameter list to configure the behavior of the simplex BFM.

The system call usage and the majority of the parameter definitions are the same as described in System Calls, page 26.

Table A-3: **Additional Parameters for Simplex BFM Operation**

Argument	Value(s)	Default Value	Description
simplex_tx_auto_init	000 to 111	000	The three bits correspond to auto completion of each phase of initialization: *Alignment*, *channel bonding*, and *channel verification*. • 0: No auto completion • 1: Auto completion The state machine will exit the current phase indicating success after a fixed number of symbols is received. This is used in conjunction with parameters simplex_tx_align_timer, simplex_tx_bond_timer, and simplex_tx_verify_timer
simplex_tx_align_timer	Any integer value	100	Indicates the number of symbols after which the state machine will exit the phase and set tx_aligned to 1.
simplex_tx_bond_timer	Any integer value	200	Indicates the number of symbols after which the state machine will exit the phase and set tx_bonded to 1.
simplex_tx_verify_timer	Any integer value	300	Indicates the number of symbols after which the state machine will exit the phase and set tx_verified to 1.
rx_reset_pulse_width	Any integer value	1	Indicates the number of Aurora 8B/10B cycles for which the rx_reset signal will remain at 1 when hard errors are encountered causing BFM to reset.