

## GIGE BFM User Guide

### Table of Contents

|     |   |   |
|-----|---|---|
| 1.0 | Overview  | 2 |
| 2.0 | TX Path Functional Description                    | 3 |
| 3.0 | RX Path Functional + Auto-negotiation Description | 6 |
| 4.0 | Deep Loopback                                     | 8 |
| 5.0 | Top Level Ports                                   | 9 |

### List of Tables

|     |  |    |
|-----|--|----|
| 2.1 | TX Path Source File Descriptions                           | 3  |
| 3.1 | RX Path + Auto-negotiation Source File Description         | 6  |
| 3.2 | RX Path Protocol Checks                                    | 7  |
| 3.3 | Auto-negotiation Protocol Checks                           | 7  |
| 5.1 | TX Path Packet interface to testbench                      | 9  |
| 5.2 | RX Path Packet interface from testbench                    | 9  |
| 5.3 | TX Path line interface to DUT (serial, 10bit TBI, 20bit)   | 10 |
| 5.4 | RX Path line interface from DUT (serial, 10bit TBI, 20bit) | 10 |
| 5.5 | Control Inputs for TX Path                                 | 11 |
| 5.6 | Control Inputs for RX Path                                 | 13 |
| 5.7 | Control Inputs for RX and TX Paths                         | 13 |

## 1.0 Overview

The GIGE BFM is written in Verilog and is designed to be portable to any verification environment that is compatible with Verilog. The interfaces to the BFM are intended to also maximize portability. The testbench interface to the GIGE BFM is a simple clocked data interface and configuration signals. The DUT interface is a data lane that is either serial, 10 bits wide (TBI) or twenty bits wide. This allows the GIGE BFM to be used in application including the SERDES model as well as those that do not.

The GIGE BFM consists of a transmit path (TX Path) that drives the line interface of the DUT and a receive path (RX Path) that monitors the line interface from the DUT. In cases where the DUT has more than one GIGE interface, the GIGE BFM may be instantiated multiple times.

The TX path receives packet payload data from the testbench over a clocked data interface and drives these packets into the DUT over the line interface. Additionally, the TX path performs idle mapping to properly hold the bus in an idle state between packets. During both idle periods as well as during packet transmission, errors can be inserted. The presence, frequency, and type of errors are controlled through testbench parameters. Finally, the TX Path has a DUT state machine monitor for the DUT sync state machine, the DUT receive state machine, and the DUT GMII. In this way, the exact response of the DUT to the presence of errors is checked in a cycle by cycle manner. Functional coverage of these state machines is collected to ensure the testing is exhaustive.

The RX path monitors the line interface from the DUT and passes packet data on to the testbench and terminates the idle codes. Extensive checking is performed to ensure no violations of the protocol occur. Both the packet payload encapsulation and the idle codes in the inter-packet gap are checked.

Auto-negotiation spans both the RX and TX paths. The GIGE BFM has an auto-negotiation checker that follows the auto-negotiation process by monitoring the control and idle codes going into the DUT and coming from the DUT. If the sequence of codes coming from the DUT are not as expected, an error is flagged.

Any errors detected by the RX, TX, or auto-negotiation paths are reported through a single error reporting task that can be customized to best suit the target verification environment.

## 2.0 TX Path Functional Description

The functionality of the TX path is partitioned into several parts. The Verilog source file name corresponds to the module name (see Table 1.1 TX Path Source File Description). The master control of the GIGE link is performed by *gige\_txsm*. The monitoring and checking of the DUT synchronization, receive, and GMII state machines is done by *check\_369* and *check\_367*. The 8b10b serdes *encoder* is used to interface to the DUT lineport.

| TX Path Source File | Description  |
|---------------------|--|
| <i>gige_txsm.v</i>  | Master control of BFM TX path                      |
| <i>check_369.v</i>  | Checker the DUT alignment state machine            |
| <i>gige_369.v</i>   | Used by <i>check_4889</i> to model state machines. |
| <i>check_367.v</i>  | Checker the DUT receive state machine and GMII     |
| <i>gige_367.v</i>   | Used by <i>check_487</i> to model state machine    |
| <i>encoder.v</i>    | 8b10b encoder (see Serdes8b10b documentation)      |
| <i>stimd.v</i>      | Queue to provide delay to model DUT ingress path.  |

Table 2.1 TX Path Source File Descriptions

The higher level testbench will interface to *gige\_txsm*. One feature the higher level testbench will typically use is the different modes of operation controlled with the *state\_garbage*, *state\_innout*, *state\_idle*, *state\_autoneg*, and *state\_normal* inputs. When *state\_garbage* is asserted, garbage is driven to the DUT to mimic periods of bus instability at startup or cable pull. The *garbage\_type* and *garbage\_vector* inputs allow more precise control of the bad line state. The BFM does a check before each code is selected to ensure that no sync characters could be detected by the DUT during *state\_garbage*.

In *state\_innout* a mix of possibly erred idle codes and configuration codes are sent to the DUT. Synchronization will be achieved and then if enabled in the DUT, auto-negotiation will start and restart as the random mix of idle codes and configuration codes is not a valid auto-negotiation sequence. Auto-negotiation should not complete in this state. Additionally, the erred sequence may drop the DUT in and out of sync.

In *state\_idle* the idle code is sent to the DUT to allow sync to occur. If auto-negotiation is not enabled, the sequence of state indications from the testbench could go directly from *state\_garbage* to *state\_idle* to *state\_normal*.

In *state\_autoneg* a valid stream of configuration codes are sent to the DUT allowing auto-negotiation to complete successfully. The value of the transmitted configuration register is controlled by the top level testbench.

In *state\_normal* the BFM will nominally transmit a packet when indicated by the testbench and fill the gap between packets with idles. There are many input parameters that control how the BFM will select each idle code. The input parameter *illegal* will override other input parameters and cause a legal sequence of idles to be generated. If *illegal* is deasserted the other parameters control the idle sequence to be sent. In any state with *illegal* deasserted the idle mapping is done differently with the intention of more robustly exercising the DUT without causing packet errors.

*irandom* is the next highest priority parameter controlling the idle sequence. *irandom* causes a randomly selected data code as part of the idle code. The IEEE spec indicates the transmitter should only use one of two data codes as part of the idle code; however, it also indicates the receive side should not check which data code is used. Therefore, when testing the receive side of the DUT a randomly selected data code in the idle should be fine.

If neither *illegal* nor *irandom* are asserted, then *il\_weight* randomly selects the type of idle to send by indicating the probability that the idle be /I1/ vs. /I2/.

A carrier extend code may be sent instead of an idle. When *c\_carrier\_extend\_en* is asserted, at each end of packet there is a probability that a sequence of carrier extend codes will follow the packet. This probability is controlled with *c\_carrier\_extend\_percent*. After the first carrier extend is transmitted, the same parameter determines the likelihood that there be an additional carrier extend or an idle is transmitted.

There are three additional sequence of codes that may be used to fill the inter-packet gap. The probability given by *badsop\_probability\_num* indicates if a bad start of packet is to be sent instead of an idle or carrier extend. A bad sop is an sop followed immediately by a K28.5. The first case is when the prior code was carrier extend. In this case, the sequence /R/, /S/, /K/ is sent where the last /K/ is either an idle or configuration code. If *irandomconfig* is asserted, then there is a 40% chance the configuration code is sent instead of the idle code.

The other case where the prior code was not carrier extend is /S/, /R/, /R/, /R/. This bad sop is then followed by a series of idles if the mode is SGMII 10 or 100 speed as opposed to 1000BaseX or SGMII 1000 speed.

These two bad sop cases are provided as a means of achieving full coverage of the receive state machine without the need for any directed testing.

The final possibility for transmission is to send a configuration code in the middle of the idle sequence or carrier extend sequence. The parameter *irandomconfig* enables this anomalous behavior. If set, the probability of the anomalous configuration code is 20%.

Packet transmission follows the IEEE specified sequence of codes with a minimum gap specified by the input parameter *min\_ipg*. The packet starts with a valid idle followed by the /S/ code. A preamble of random length between *preamble\_min* and *preamble\_max* is then sent. Nominally, a single SFD is then transmitted, but if *no\_sfd* is asserted, this code is suppressed. The payload is popped from the testbench and transmitted one Byte at a time until the end of packet is indicated by the testbench. If *no\_sfd* is asserted, then any payload Byte that happens to be a SFD is replaced by another code to ensure that no SFD is transmitted. The *no\_sfd* mode is a convenient way to ensure the MAC will not recognize and not transmit the frame. During data transmission, if the input *in\_error* is asserted, a /E/ is transmitted instead of the indicated payload data. When the end of packet is indicated, a valid terminate sequence is sent.

One method of error insertion is controlled with the input parameter *epn\_constant\_error*. When nonzero, this parameter has the percent likelihood that the lane will contain an error. The error is randomly selected from disparity error, code error, or unexpected k-code. These errors can occur mid-packet and therefore can cause packet loss. NOTE: *epn\_constant\_error* will randomly cause any code to be replaced with an error including the packet delimiters /S/ and /T/. This is useful to fully exercise the receive state machine.

The other method of error injection only affects idles and therefore at an infrequent rate will not cause packet loss. *error\_probability\_num* indicates the frequency of the random errors and typically would be set to a very low frequency. Disparity errors and code errors are chosen by *error\_probability\_num*.

In this way, *gige\_txsm* selects the code to be transmitted. Each code is passed along to the 8b10b encoder in *encoder* (see *serdes\_8b10b* documentation).

The BFM generates the sequence of codes for the line in the manner just described. Additionally for the DUT receive side the testbench has monitors to check that the DUT response to these codes is correct.

*check\_369* uses *stimd* to delay the output of the *gige\_txsm* so it coincides in time with the processing of the DUT sync state machine. To calculate this needed delay, the testbench inserted lane skew is added to the latency added by the DUT to the sync state machine. *check\_369* provides this delayed stimulus to *gige\_369* which implements a testbench version of the IEEE sync state machine. *check\_369* probes into the DUT to observe its sync state machine and compares the two. Depending on the DUT implementation there may be periods of uncertainty where checking cannot be reliably performed, but

nominally, checking is performed. Any mismatch is an error. *gige\_369* tracks state changes and thereby provides functional coverage ensuring all the state transitions are exercised.

Similarly, *check\_367* uses *gige\_367* to model and check the receive state machine and GMII bus behavior. *gige\_367* implements functional coverage and together they check cycle by cycle the proper activity on the GMII bus.

### 3.0 RX Path + Auto-negotiation Functional Description

The GIGE RX monitors the 4 lane output of the testbench to extract packets and check protocol violations. The Verilog source file name corresponds to the module name (see Table 1.2 RX Path Source File Description). The auto-negotiation process checking is included here as well.

| RX Path Source File   | Description   |
|-----------------------|---|
| <i>gige_rxsm.v</i>    | Extract payload, check protocol, forward packets to testbench |
| <i>gige_rxcodes.v</i> | Map and check ordered sets                                    |
| <i>decoder.v</i>      | 8b10b decoder (see Serdes8b10b documentation)                 |
| <i>gige_ancheck.v</i> | models the auto-negotiation process for <i>gige_autoneg</i>   |
| <i>gige_autoneg.v</i> | monitors lanes into and out of DUT to check auto-negotiation  |
| <i>stimd.v</i>        | used by <i>gige_autoneg</i> as fixed delay FIFO               |
| <i>stimshift.v</i>    | used by <i>gige_autoneg</i> as 2 port delay FIFO              |

Table 3.1 RX Path Source File Description

The dataflow starts with the 8b10b *decoder* (see *serdes\_8b10b* documentation) which passes the character aligned 8b10b codes to *gige\_rxcodes*. This module indicates the ordered set to *gige\_rxsm*. *gige\_rxsm* performs further protocol checking and packet extraction.

*gige\_ancheck* monitors the *gige\_txsm* output to the DUT as well as the *gige\_rxcodes* input from the DUT. Both these streams of codes are then passed along to *gige\_autoneg* which models the auto-negotiation process to detect any errors in the DUT.

A list of the checks that are performed on the RX Path can be found in Table 1.3 RX Path Protocol Checks.

| <b>RX Path Protocol Check</b>                                      | <b>Source File</b>    |
|--|-----------------------|
| All disparity and code errors                                      | <i>decoder.v</i>      |
| Unknown 8b10b characters detected as invalid                       | <i>gige_rxcodes.v</i> |
| Any K28.5 is at the beginning of a properly formed code group      | <i>gige_rxcodes.v</i> |
| All code groups consist of the correct ordered sets                | <i>gige_rxcodes.v</i> |
| /C2/ immediately follows /C1/                                      | <i>gige_rxcodes.v</i> |
| All K28.5 occur on even words                                      | <i>gige_rxsm.v</i>    |
| Disparity after every /I/ is negative                              | <i>gige_rxsm.v</i>    |
| A sequence of /C/ always starts with /C1/                          | <i>gige_rxsm.v</i>    |
| No control codes /T/, /R/, /V/ occur during an /I/ or /C/          | <i>gige_rxsm.v</i>    |
| No extra data codes occur during /I/ or /C/                        | <i>gige_rxsm.v</i>    |
| /S/ only occurs at the start of a packet                           | <i>gige_rxsm.v</i>    |
| Data replication is correct for first Byte of preamble, 1,9,99     | <i>gige_rxsm.v</i>    |
| Other data replication is correct 1, 10, 100                       | <i>gige_rxsm.v</i>    |
| Check for proper number of preamble bytes                          | <i>gige_rxsm.v</i>    |
| Every preamble is followed by SFD                                  | <i>gige_rxsm.v</i>    |
| No special codes mid packet. Packet must terminate with /V/ or /T/ | <i>gige_rxsm.v</i>    |
| Proper termination of packet /T/, /R/, /R/, etc.                   | <i>gige_rxsm.v</i>    |
| Minimum IFG not violated.  | <i>gige_rxsm.v</i>    |

Table 3.2 RX Path Protocol Checks

The Auto-negotiation checks are performed by modeling the DUT auto-negotiation state machine and checking the DUT egress link. The checks are therefore performed in a sequential manner as listed in Table 3.2.

| <b>Auto-negotiation Protocol Check</b>   | <b>Source File</b>    |
|--|-----------------------|
| rx_Config_Reg starts autoneg at 0 and stays 0 til link_timer expires                           | <i>gige_ancheck.v</i> |
| rx_Config_Reg is received contiguously without /I/ or other codes                              | <i>gige_ancheck.v</i> |
| During first link_timer, DUT should ignore tx_Config_Reg                                       | <i>gige_ancheck.v</i> |
| During auto-negotiation, testbench RUDI(invalid) restarts DUT auto-negotiation.                | <i>gige_ancheck.v</i> |
| After the first link_timer expires, DUT should send programmed rx_Config_Reg with ACK bit zero | <i>gige_ancheck.v</i> |

|  |                        |
|--|------------------------|
| DUT continues sending ACK bit zero until it receives 3 consecutive identical tx_Config_Reg from testbench.   | <i>gige_anccheck.v</i> |
| After DUT sends rx_Config_Reg and receives 3 consecutive identical tx_config_Reg from testbench it sets ACK bit to one.  | <i>gige_anccheck.v</i> |
| DUT continues sending rx_Config_Reg with ACK bit one until it first gets 3 contiguous tx_Config_Reg from testbench with ACK bit one and then link_timer expires (second link_timer). | <i>gige_anccheck.v</i> |
| after second link_timer expires DUT begins sending /I/   | <i>gige_anccheck.v</i> |
| DUT continues sending /I/ until third link_timer expires and then it sees 3 back to back /I/ from the testbench.   | <i>gige_anccheck.v</i> |
| After third link_timer expires and DUT sees 3 back to back /I/ from testbench, DUT transitions out of auto-negotiation state.  | <i>gige_anccheck.v</i> |
| During auto-negotiation, when DUT loses synchronization for one link_timer, auto-negotiation restarts.   | <i>gige_anccheck.v</i> |

Table 3.3 Auto-negotiation Protocol Checks

## 4.0 Loopback

The GIGE BFM can be used in various line loopback modes that allow the packet generation and checking to be done elsewhere. In all these loopback modes, the checking features of the BFM can be enabled.

The first loopback possibility is simply to connect the DUT line output to both the BFM receive and the DUT receive. In this mode, the DUT receives exactly what it transmits and the BFM is passively monitoring the DUT egress stream.

The second loopback takes the decoded output of the decoder and passes it back to the input of the encoder as well as the *gige\_rxsm*. This is similar to the prior loopback except now data is only looped back after the decoder achieves sync and the features of the encoder may be used. For example, the encoder can shift the bit stream by adding a number of bits of skew.

The third loopback is at the BFM interface to the testbench. This packet based loopback occurs and only packets are looped back. All idles are terminated by the BFM on the BFM receive side and generated by the BFM on the BFM transmit side. This loopback allows all the features of the BFM to be employed while allowing the packet generation to be performed elsewhere.

## 5.0 Top Level Ports

The GIGE BFM top level is called *gigebfm*. *gigebfm* fully encapsulates the GIGE BFM. *gigebfm* interfaces directly to the DUT line interfaces and it interfaces directly to the testbench. Additionally, it optionally monitors the DUT internal state.

The following tables list the ports of *gigebfm* and provide usage guidance.

| TX Path Packet Interface |           |   |
|--------------------------|-----------|---|
| Port Type                | Port Name | Port Usage Description  |
| input                    | valid     | A zero indicates idle, a one indicates data. A data transfer must be contiguous from start to end without any idle codes. A transition from 0 to 1 infers a SOP and a transition from 1 to 0 infers an EOP. |
| input                    | tx_abort  | Causes an abrupt end of packet without proper /T/ termination. If <i>tx_abort</i> goes high, <i>valid</i> must go low before <i>tx_abort</i> goes low.<br>abort = even && tx_abort && !valid;               |
| input                    | in_error  | While <i>valid</i> is high, <i>in_error</i> causes /V/ character to be transmitted.   |
| input [7:0]              | data      | The transmit data associated with <i>valid</i> .  |
| output                   | pop       | Indicates the BFM consumed <i>data</i> . <i>data</i> should not advance unless <i>pop</i> is one.   |
| output                   | ready     | Indicates the BFM is ready for a data transfer. <i>valid</i> should remain all zero until <i>ready</i> becomes one.   |
| input                    | clk       | The BFM acts on these signals near the posedge, so testbench should act on them near the negedge.   |

Table 5.1 TX Path Packet interface to testbench

| RX Path Packet Interface |           |   |
|--------------------------|-----------|---|
| Port Type                | Port Name | Port Usage Description  |
| output                   | out_valid | A zero indicates idle code, a one indicates a data code. A data transfer must be contiguous from start to end without any idle columns.                                       |
| output                   | out_eop   | Coincident with the /T/ code or first control code not /V/ at the end of the packet. Positive edge of <i>out_eop</i> should coincide with negative edge of <i>out_valid</i> . |

|              |           |   |
|--------------|-----------|---|
| output       | out_error | While <i>out_valid</i> is high, the reception of /V/ will continue <i>out_valid</i> high, but <i>out_error</i> will also be high. If the packet terminates with cod other than /T/ <i>out_error</i> will be high coincident with out eop. |
| output [7:0] | rxoctet   | The received data Byte  |
| output       | rx_clk    | The BFM acts on these signals near the posedge, so testbench should act on them near the negedge.   |

Table 5.2 RX Path Packet interface from testbench

| TX Path Line Interface |              |   |
|------------------------|--------------|---|
| Port Type              | Port Name    | Port Usage Description  |
| output                 | txserial_d   | serial data output transitions on posedge txserial_clk  |
| input                  | txserial_clk | txserial_clk must have exactly 10 clock cycles for each cycle of tx_column_clk.   |
| output [19:0]          | tx_tbid20    | little endian twenty bit interface valid only with tbimd20 asserted   |
| output                 | txtbic20     | twenty bit interface clock with data transitions on posedge. Every posedge of tx_tbic coincides with an edge on txtbic20. |
| output [9:0]           | tx_tbid      | little endian TBI data valid only with tbimd asserted   |
| output                 | tx_tbic      | TBI clock with data transitions on posedge.<br>tx_tbic = !tx_column_clk && (tbimd    tbimd20);                            |

Table 5.3 TX Path line interface to DUT (serial, 10bit TBI, 20bit)

| RX Path Line Interface |              |  |
|------------------------|--------------|--|
| Port Type              | Port Name    | Port Usage Description   |
| input                  | rxserial_d   | serial data input must transition on posedge                   |
| input                  | rxserial_clk | serial data clock  |
| input [19:0]           | rx_tbid20    | little endian twenty bit interface must transition on posedge. |
| input [9:0]            | rx_tbid      | little endian TBI data. must transition on posedge.            |
| input                  | rx_tbic      | clock for TBI<br>(must be active in both tbimd and tbimd20)    |

|       |           |   |
|-------|-----------|---|
| input | rx_tbic20 | clock for twenty bit interface. (if tbimd20 active, rx_tbic20 must have all edges coincide with rx_tbic edges.) |
|-------|-----------|---|

Table 5.4 RX Path line interface from DUT (serial, 10bit TBI, 20bit)

| Control Inputs for TX Path |                     |   |
|----------------------------|---------------------|---|
| Port Type                  | Port Name           | Port Usage Description  |
| input                      | state_garbage       | When asserted holds <i>gige_txsm</i> in garbage state.  |
| input                      | state_innout        | When asserted and <i>state_garbage</i> deasserted, holds <i>gige_txsm</i> in innout state.  |
| input                      | state_idle          | When asserted and <i>state_garbage</i> && <i>state_innout</i> deasserted, holds <i>gige_txsm</i> in idle state.   |
| input                      | state_autoneg       | When asserted and <i>state_garbage</i> && <i>state_innout</i> && <i>state_idle</i> deasserted, holds <i>gige_txsm</i> in auto-negotiation state.  |
| input                      | state_normal        | When asserted and <i>state_garbage</i> && <i>state_innout</i> && <i>state_idle</i> && <i>state_autoneg</i> deasserted, holds <i>gige_txsm</i> in normal state.  |
| input [2:0]                | garbage_type        | garbage type controls the <i>gige_txsm</i> transmission during state garbage.<br>3'd1: garbage vector selects error type.<br>3'd2: randomly select between all valid codes except K28.5                       |
| input [1:0]                | garbage_vector      | garbage vector selects invalid codes per lane:<br>1: send random invalid codes.<br>2: hold lane low<br>3:hold lane high   |
| input                      | illegal             | When asserted, the idles are transmitted as indicated in the IEEE spec.   |
| input                      | irandom             | when <i>illegal</i> not asserted, <i>irandom</i> causes random data codes as second code in idles.  |
| input [31:0]               | i1_weight           | when neither <i>illegal</i> nor <i>irandom</i> are asserted, the sequence of  I1  vs.  I2  is indicated by <i>i1_weight</i> . Legal values are 0-1000 where 0 indicates all  I2  and 1000 indicates all  I1 . |
| input                      | c_carrier_extend_en | Enables transmission of carrier extension after each end of packet.   |

|              |                          |   |
|--------------|--------------------------|---|
| input [31:0] | c_carrier_extend_percent | When <i>c_carrier_extend_en</i> asserted, this parameter (0-1000) determines the probability that carrier extend will follow and end of packet. After carrier extend starts, this parameter indicates the probability that it will continue for 2 more code words or not. |
| input [31:0] | badstop_probability_num  | (0-1000) probability that any idle or carrier extend will be replaced by a bad start of packet.   |
| input        | irandomconfig            | enables random injection of spurious configuration codes into the idle stream.  |
| input [31:0] | min_ipg                  | Minimum number of idle columns between packets. This parameter does not include the mandatory  T ,  R ,  I  or  T ,  R ,  R ,  I  |
| input [31:0] | preamble_min             | minimum number of preamble Bytes  |
| input [31:0] | preamble_max             | maximum number of preamble Bytes  |
| input        | no_sfd                   | indicates that no SFD code will be sent by <i>gige_txsm</i> during a packet transmission.   |
| input        | in_error                 | when asserted during packet data transfer, a /E/ is transmitted instead of the indicated data.  |
| input [31:0] | epn_constant_error       | This value must be between 0 and 1000. It is the probability that a code will have an error where 0 indicates no errors and 1000 indicates all errors, randomly code, disparity, or unexpected K-code error.  |
| input [31:0] | error_probability_number | Each code of an idle is subject to this check: $\$random\_range(error\_probability\_num==1)$ If the check is met, then that code gets an error, randomly code or disparity.   |
| input [6:0]  | txbit_delay              | Number of bits to delay the bitstream for lane2. For example, when TBI mode is enabled if the bit delay is 3, then the 10 bit codes will be offset by 3 bits. Must be in the range 1-129, but the range 1-20 is advised.  |
| input        | enable369                | When set to one, the check369 module is enabled to report errors.   |
| input        | enable367                | When set to one, the check367 module is enabled to report errors.   |
| input        | clegal                   | send legal sequence of /C/  |
| input [31:0] | c1_weight                | when clegal not asserted this indicates the probability that the next /C/ is /C1/ as opposed to /C2/. (must be in the range 0-1000)   |

|              |               |   |
|--------------|---------------|---|
| input [15:0] | tx_config_reg | config register value to encapsulate in /C/ |
| input        | reset         | Included only for future use. tie to zero.  |

Table 5.5 Control Inputs for TX Path

| <b>Control Inputs RX Path</b> |                  |  |
|-------------------------------|------------------|--|
| <b>Port Type</b>              | <b>Port Name</b> | <b>Port Usage Description</b>  |
| input [3:0]                   | rxresync         | Included only for future use. tie to zero.   |
| input [31:0]                  | rx_minifg_bytes  | minimum IFG to check against in gige_rxsm.v  |
| input                         | gmii_loopback    | Must be asserted only when the DUT loops its RXGMII back to its TXGMII at the PCS. |

Table 5.6 Control Inputs for RX Path

| <b>Control Inputs shared RX and TX Paths</b> |                  |  |
|--|------------------|--|
| <b>Port Type</b>                             | <b>Port Name</b> | <b>Port Usage Description</b>  |
| input  | tbimd            | Asserted puts BFM in TBI mode.<br>Never assert both tbimd and tbimd20.   |
| input  | tbimd20          | Asserted puts BFM in twenty bit interface mode.<br>Never assert both tbimd and tbimd20.  |
| input [31:0]                                 | speed_duplicate  | Number of times data bytes are replicated.<br>Typically a value of 1 for 1000 speed, a value of 10 for 100 speed, and a value of 100 for 10 speed. |

Table 5.7 Control Inputs for RX and TX Paths