This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics’ trademarks may be viewed at: www.mentor.com/trademarks.

The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form
# Table of Contents

Preface ........................................................................................................ 17
   About This User Guide ........................................................................... 17
   AMBA AXI Protocol Specification ......................................................... 17
   Protocol Restrictions ............................................................................... 17
      BFM Dependencies Between Handshake Signals ............................... 17
   Mentor VIP AE License Requirements .................................................. 18
   Supported Simulators ............................................................................ 18
   Simulator GCC Requirements ............................................................... 18

Chapter 1
Mentor VIP Altera Edition ....................................................................... 21
   Advantages of Using BFMs and Monitors ............................................. 21
   Implementation of BFMs ........................................................................ 21
   What Is a Transaction? .......................................................................... 22
   AXI4-Lite Transactions .......................................................................... 22
      AXI4-Lite Write Transaction Master and Slave Roles ....................... 23
      AXI Read Transaction Master and Slave Roles ................................... 25

Chapter 2
SystemVerilog API Overview ................................................................. 27
   Configuration ........................................................................................ 28
   Creating Transactions ............................................................................ 28
      Transaction Record ............................................................................ 28
      create*_transaction() ....................................................................... 33
      execute_transaction(), execute*_phase() ......................................... 33
   Waiting Events ...................................................................................... 34
      wait_on() ........................................................................................ 34
      get*_transaction(), get*_phase(), get*_cycle() ............................... 34
   Access Transaction Record .................................................................. 35
      set*() .............................................................................................. 35
      get*() .............................................................................................. 35
   Operational Transaction Fields ............................................................ 35
      Automatic Generation of Byte Lane Strobes ....................................... 36
      Operation Mode ............................................................................... 36
      Channel Handshake Delay ................................................................ 37
      Transaction Done ............................................................................ 38

Chapter 3
SystemVerilog Master BFM .................................................................... 39
   Master BFM Protocol Support .............................................................. 39
   Master Timing and Events .................................................................... 39
Table of Contents

Chapter 5
SystemVerilog Monitor BFM .................................................. 91
Inline Monitor Connection .................................................. 91
Monitor BFM Protocol Support .............................................. 91
Monitor Timing and Events ................................................ 91
Monitor BFM Configuration ................................................. 92
Monitor Assertions .......................................................... 95
  Assertion Configuration .................................................. 95
SystemVerilog Monitor API .................................................. 96
  set_config() ............................................................... 96
  get_config() ............................................................. 97
  create_monitor_transaction() ......................................... 98
  get_rw_transaction() ................................................... 100
  get_write_addr_phase() ............................................... 101
  get_read_addr_phase() ................................................. 102
  get_read_data_phase() ................................................ 103
  get_write_data_phase() .............................................. 104
  get_write_response_phase ........................................... 105
  get_read_addr_ready() ................................................ 106
  get_read_data_ready() ................................................ 107
  get_write_addr_ready() ............................................. 108
  get_write_data_ready() .............................................. 109
  get_write_resp_ready() ............................................... 110
  wait_on() ............................................................... 111
Helper Functions .......................................................... 112
  get_write_addr_data() ................................................ 112
  get_read_addr() ........................................................ 113
  set_read_data() ........................................................ 114

Chapter 6
SystemVerilog Tutorials ..................................................... 115
Verifying a Slave DUT ...................................................... 115
  BFM Master Test Program ............................................ 116
Verifying a Master DUT ................................................... 122
  BFM Slave Test Program ............................................. 122

Chapter 7
VHDL API Overview ......................................................... 137
Configuration .............................................................. 139
Creating Transactions ..................................................... 139
  Transaction Record .................................................... 139
  create*_transaction() ................................................. 144
Executing Transactions .................................................. 144
  execute_transaction(), execute*_phase() ........................ 145
Table of Contents

Chapter 8
VHDL Master BFM ......................................................... 151
  Overloaded Procedure Common Arguments ..................................... 151
  Master BFM Protocol Support .................................................. 152
  Master Timing and Events ...................................................... 152
  Master BFM Configuration ..................................................... 152
  Master Assertions .............................................................. 155
    Assertion Configuration .................................................. 155
  VHDL Master API ............................................................ 156
    set_config() ................................................................ 156
    get_config() ................................................................ 158
    create_write_transaction() ................................................ 159
    create_read_transaction() .................................................. 161
    set_addr() .................................................................. 163
    get_addr() .................................................................. 164
    set_prot() .................................................................. 165
    get_prot() .................................................................. 166
    set_data_words() .............................................................. 167
    get_data_words() .............................................................. 168
    set_write_strobes() .......................................................... 169
    get_write_strobes() .......................................................... 170
    set_resp() .................................................................. 171
    get_resp() .................................................................. 172
    set_read_or_write() ........................................................... 173
    get_read_or_write() ........................................................... 174
    set_gen_write_strobes() ..................................................... 175
    get_gen_write_strobes() ..................................................... 176
    set_operation_mode() ......................................................... 177
    get_operation_mode() ......................................................... 178
    set_write_data_mode() ....................................................... 179
    get_write_data_mode() ....................................................... 181
    set_address_valid_delay() .................................................. 183
    get_address_valid_delay() .................................................. 184
    get_address_ready_delay() .................................................. 185
    set_data_valid_delay() ....................................................... 186
    get_data_valid_delay() ....................................................... 187

Waiting Events ................................................................. 145
  wait_on() ...................................................................... 145
  get*_transaction(), get*_phase(), get*_cycle() .............................. 145
Access Transaction Record ..................................................... 146
  set*() ....................................................................... 146
  get*() ..................................................................... 146
Operational Transaction Fields ................................................... 147
  Automatic Correction of Byte Lane Strobes .................................... 147
  Operation Mode ................................................................. 147
  Channel Handshake Delay ....................................................... 148
  Transaction Done .............................................................. 149

Operation Mode ................................................................. 147

Channel Handshake Delay ....................................................... 148

Assertion Configuration ......................................................... 155

Master Timing and Events ...................................................... 152

Master BFM Protocol Support .................................................. 152

Master Assertions .............................................................. 155

Overloaded Procedure Common Arguments ..................................... 151

Mentor Verification IP AE AXI4-Lite User Guide, V10.4c
January 2016
**Table of Contents**

VHDL Monitor API ................................................................. 287
  set_config() ........................................................................ 288
  get_config() ......................................................................... 289
  create_monitor_transaction() .................................................. 290
  set_addr() ............................................................................ 293
  get_addr() ............................................................................ 294
  set_prot() .............................................................................. 295
  get_prot() ............................................................................... 296
  set_data_words() ................................................................... 297
  get_data_words() ................................................................... 298
  set_write_strobes() ................................................................ 299
  get_write_strobes() ................................................................ 300
  set.resp() ............................................................................... 301
  get.resp() ............................................................................. 302
  set_read_or_write() ................................................................. 303
  get_read_or_write() ................................................................. 304
  set_gen_write_strobes() .......................................................... 305
  get_gen_write_strobes() .......................................................... 306
  set_operation_mode() .............................................................. 307
  get_operation_mode() .............................................................. 308
  set_write_data_mode() ............................................................. 309
  get_write_data_mode() ............................................................. 310
  set_address_valid_delay() ...................................................... 312
  get_address_valid_delay() ....................................................... 313
  get_address_ready_delay() ...................................................... 314
  set_data_valid_delay() ............................................................. 315
  get_data_valid_delay() ............................................................. 316
  get_data_ready_delay() ............................................................ 317
  get_write_response_valid_delay() .......................................... 318
  get_write_response_ready_delay() .......................................... 319
  set_transaction_done() ............................................................ 320
  get_transaction_done() ............................................................ 321
  get_read_data_phase() ............................................................ 322
  get_write_response_phase() .................................................... 323
  get_write_addr_phase() ........................................................... 324
  get_read_addr_phase() ............................................................. 325
  get_write_data_phase() ............................................................ 326
  get_rw_transaction() ............................................................... 327
  get_read_addr_ready() ............................................................. 328
  get_read_data_ready() .............................................................. 329
  get_write_addr_ready() ............................................................ 330
  get_write_data_ready() ............................................................. 331
  get_write_resp_ready() ........................................................... 332
  push_transaction_id() ............................................................... 333
  pop_transaction_id() ............................................................... 335
  print() ................................................................................... 337
  destruct_transaction() ............................................................. 338
  wait_on() .............................................................................. 339
Chapter 11
VHDL Tutorials ................................................................. 341
  Verifying a Slave DUT ...................................................... 341
  BFM Master Test Program ............................................... 342
  Verifying a Master DUT ................................................... 346
  BFM Slave Test Program ................................................. 346

Chapter 12
Getting Started with Qsys and the BFMss................................. 359
  Setting Up Simulation from a UNIX Platform ....................... 359
  Setting Up Simulation from the Windows GUI ..................... 360
    Running the Qsys Tool .................................................. 362
    Running a Simulation .................................................... 365

Appendix A ................................................................. 375
  AXI4-Lite Assertions .................................................... 375

Third-Party Software for Mentor Verification IP Altera Edition

End-User License Agreement
## List of Examples

| Example 2-1. AXI4 Transaction Definition | 29 |
| Example 2-2. Slave Test Program Using `get_write_addr_phase()` | 34 |
| Example 6-1. master_ready_delay_mode | 117 |
| Example 6-2. m_wr_resp_phase_ready_delay | 118 |
| Example 6-3. m_rd_data_phase_ready_delay | 118 |
| Example 6-4. Configuration and Initialization | 118 |
| Example 6-5. Create and Execute Write Transactions | 119 |
| Example 6-6. Create and Execute Read Transactions | 120 |
| Example 6-7. handle_write_resp_ready() | 121 |
| Example 6-8. Internal Memory | 123 |
| Example 6-9. do_byte_read() | 124 |
| Example 6-10. do_byte_write() | 124 |
| Example 6-11. m_rd_addr_phase_ready_delay | 124 |
| Example 6-12. m_wr_addr_phase_ready_delay | 125 |
| Example 6-13. m_wr_data_phase_ready_delay | 125 |
| Example 6-14. set_read_data_valid_delay() | 125 |
| Example 6-15. set_wr_resp_valid_delay() | 126 |
| Example 6-16. slave_ready_delay_mode | 127 |
| Example 6-17. Initialization and Transaction Processing | 130 |
| Example 6-18. process_read() | 131 |
| Example 6-19. handle_read | 131 |
| Example 6-20. process_write | 132 |
| Example 6-21. handle_write() | 132 |
| Example 6-22. handle_write_addr_ready() | 133 |
| Example 7-1. AXI4-Lite Transaction Definition | 140 |
| Example 7-2. Slave BFM Test Program Using `get_write_addr_phase()` | 146 |
| Example 11-1. m_wr_resp_phase_ready_delay | 342 |
| Example 11-2. m_rd_data_phase_ready_delay | 343 |
| Example 11-3. Configuration and Initialization | 343 |
| Example 11-4. Create and Execute Write Transactions | 344 |
| Example 11-5. Create and Execute Read Transactions | 344 |
| Example 11-6. Process handle_write_resp_ready | 345 |
| Example 11-7. Internal Memory | 347 |
| Example 11-8. m_wr_addr_phase_ready_delay | 348 |
| Example 11-9. m_rd_addr_phase_ready_delay | 348 |
| Example 11-10. m_wr_data_phase_ready_delay | 349 |
| Example 11-11. set_wr_resp_valid_delay() | 349 |
| Example 11-12. set_read_data_valid_delay() | 349 |
| Example 11-13. process_read | 352 |
| Example 11-14. handle_read | 353 |
## List of Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11-15</td>
<td>process_write</td>
<td>353</td>
</tr>
<tr>
<td>11-16</td>
<td>handle_write</td>
<td>355</td>
</tr>
<tr>
<td>11-17</td>
<td>handle_response</td>
<td>356</td>
</tr>
<tr>
<td>11-18</td>
<td>handle_write_addr_ready</td>
<td>357</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Execute Write Transaction</td>
<td>23</td>
</tr>
<tr>
<td>1-2</td>
<td>Master Write Transaction Phases</td>
<td>24</td>
</tr>
<tr>
<td>1-3</td>
<td>Slave Write Transaction Phases</td>
<td>25</td>
</tr>
<tr>
<td>1-4</td>
<td>Master Read Transaction Phases</td>
<td>26</td>
</tr>
<tr>
<td>1-5</td>
<td>Slave Read Transaction Phases</td>
<td>26</td>
</tr>
<tr>
<td>2-1</td>
<td>SystemVerilog BFM Internal Structure</td>
<td>27</td>
</tr>
<tr>
<td>5-1</td>
<td>Inline Monitor Connection Diagram</td>
<td>91</td>
</tr>
<tr>
<td>6-1</td>
<td>Slave DUT Top-Level Test Bench Environment</td>
<td>115</td>
</tr>
<tr>
<td>6-2</td>
<td>master_ready_delay_mode = AXI4_VALID2READY</td>
<td>117</td>
</tr>
<tr>
<td>6-3</td>
<td>master_ready_delay_mode = AXI4_TRANS2READY</td>
<td>117</td>
</tr>
<tr>
<td>6-4</td>
<td>Master DUT Top-Level Test Bench Environment</td>
<td>122</td>
</tr>
<tr>
<td>6-5</td>
<td>slave_ready_delay_mode = AXI4_VALID2READY</td>
<td>126</td>
</tr>
<tr>
<td>6-6</td>
<td>slave_ready_delay_mode = AXI4_TRANS2READY</td>
<td>127</td>
</tr>
<tr>
<td>6-7</td>
<td>Slave Test Program Advanced API Tasks</td>
<td>129</td>
</tr>
<tr>
<td>7-1</td>
<td>VHDL BFM Internal Structure</td>
<td>138</td>
</tr>
<tr>
<td>10-1</td>
<td>Inline Monitor Connection Diagram</td>
<td>283</td>
</tr>
<tr>
<td>11-1</td>
<td>Slave DUT Top-Level Test Bench Environment</td>
<td>341</td>
</tr>
<tr>
<td>11-2</td>
<td>Master DUT Top-Level Test Bench Environment</td>
<td>346</td>
</tr>
<tr>
<td>11-3</td>
<td>Slave Test Program Advanced API Processes</td>
<td>351</td>
</tr>
<tr>
<td>12-1</td>
<td>Copy the Contents of qsys-examples from the Installation Folder</td>
<td>360</td>
</tr>
<tr>
<td>12-2</td>
<td>Paste qsys-examples from Installation to Work Folder</td>
<td>361</td>
</tr>
<tr>
<td>12-3</td>
<td>Select Qsys from the Quartus II Software Top-Level Menu</td>
<td>362</td>
</tr>
<tr>
<td>12-4</td>
<td>Open the ex1_back_to_back_sv.qsys Example</td>
<td>362</td>
</tr>
<tr>
<td>12-5</td>
<td>Show System With Qsys Interconnect</td>
<td>363</td>
</tr>
<tr>
<td>12-6</td>
<td>System With Qsys Interconnect Parameters Tab</td>
<td>364</td>
</tr>
<tr>
<td>12-7</td>
<td>Qsys Generation Window Options</td>
<td>365</td>
</tr>
<tr>
<td>12-8</td>
<td>Select the Work Directory</td>
<td>367</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Simulator GCC Requirements</td>
<td>18</td>
</tr>
<tr>
<td>2-1</td>
<td>Transaction Fields</td>
<td>30</td>
</tr>
<tr>
<td>2-2</td>
<td>Master and Slave *_valid_delay Configuration Fields</td>
<td>37</td>
</tr>
<tr>
<td>2-3</td>
<td>Master and Slave *_ready_delay Transaction Fields</td>
<td>38</td>
</tr>
<tr>
<td>3-1</td>
<td>Master BFM Signal Width Parameters</td>
<td>40</td>
</tr>
<tr>
<td>3-2</td>
<td>Master BFM Configuration</td>
<td>41</td>
</tr>
<tr>
<td>4-1</td>
<td>Slave BFM Signal Width Parameters</td>
<td>64</td>
</tr>
<tr>
<td>4-2</td>
<td>Slave BFM Configuration</td>
<td>65</td>
</tr>
<tr>
<td>5-1</td>
<td>AXI Monitor BFM Signal Width Parameters</td>
<td>92</td>
</tr>
<tr>
<td>5-2</td>
<td>AXI Monitor BFM Configuration</td>
<td>93</td>
</tr>
<tr>
<td>7-1</td>
<td>Transaction Fields</td>
<td>141</td>
</tr>
<tr>
<td>7-2</td>
<td>Master and Slave *_valid_delay Configuration Fields</td>
<td>149</td>
</tr>
<tr>
<td>7-3</td>
<td>Master and Slave *_ready_delay Fields</td>
<td>149</td>
</tr>
<tr>
<td>8-1</td>
<td>Master BFM Signal Width Parameters</td>
<td>152</td>
</tr>
<tr>
<td>8-2</td>
<td>Master BFM Configuration</td>
<td>153</td>
</tr>
<tr>
<td>9-1</td>
<td>Slave BFM Signal Width Parameters</td>
<td>216</td>
</tr>
<tr>
<td>9-2</td>
<td>Slave BFM Configuration</td>
<td>216</td>
</tr>
<tr>
<td>10-1</td>
<td>Signal Parameters</td>
<td>284</td>
</tr>
<tr>
<td>10-2</td>
<td>Monitor BFM Configuration</td>
<td>285</td>
</tr>
<tr>
<td>12-1</td>
<td>SystemVerilog README Files and Script Names for all Simulators</td>
<td>366</td>
</tr>
<tr>
<td>A-1</td>
<td>AXI4 Assertions</td>
<td>375</td>
</tr>
</tbody>
</table>
Preface

About This User Guide

This user guide describes the AXI4-Lite application interface (API) of the Mentor® Verification IP (VIP) Altera® Edition (AE) and how it conforms to the AMBA® AXI™ and ACE™ Protocol Specification, AXI3™, AXI4™, and AXI4-Lite™, ACE, and ACE-Lite™ Issue E (ARM IHI 0022E).

Note

This release supports only the AMBA AXI3, AXI4, AXI4-Lite, and AXI4-Stream™ protocols. The AMBA ACE protocol is not supported in this release.

AMBA AXI Protocol Specification

The Mentor VIP AE conforms to the AMBA® AXI™ and ACE™ Protocol Specification, AXI3™, AXI4™, and AXI4-Lite™, ACE and ACE-Lite™ Issue E (ARM IHI 0022E). For restrictions to this protocol, refer to the section Protocol Restrictions.

This user guide refers to this specification as the “AXI Protocol Specification.”

Protocol Restrictions

The Mentor VIP AE supports all but the following features of this AXI Protocol Specification, which gives you a simplified API to create desired protocol stimulus.

BFM Dependencies Between Handshake Signals

Starting a write data phase before its write address phase in a transaction is not supported. However, starting a write data phase simultaneously with its write address phase is supported.

The above statement disallowing a write data phase to start before its write address phase in a transaction modifies the AXI4-Lite Protocol Specification slave write response handshake dependencies diagram, Figure A3-7 in Section A3.3.1, by effectively adding double-headed arrows between AWVALID to WVALID and AWREADY to WVALID, with the provision that they can be simultaneous.
Mentor VIP AE License Requirements

Note
A license is required to access the Mentor Graphics VIP AE Bus Functional Models (BFMs) and Inline Monitor.

- To access the Mentor Graphics VIP AE and upgrade to the Quartus II Subscription Edition software version 15.1 from a previous version, you must regenerate your license file.
- To access the Mentor VIP AE with the Quartus II Web Edition software, you must upgrade to version 15.1 and purchase a Mentor VIP AE seat license by contacting your Altera sales representative.

You can generate and manage license files for Altera software and IP products by visiting the Self-Service Licensing Center of the Altera website.

Supported Simulators
Mentor VIP AE supports the following simulators:

- Mentor Graphics Questa SIM and ModelSim 10.4d
- Synopsys® VCS® and VCS-MX 2015.09 on Linux
- Cadence® Incisive® Enterprise Simulator (IES) 15.10.010 on Linux

Simulator GCC Requirements
Mentor VIP requires that the installation directory of the simulator includes the GCC libraries shown in Table 1. If the installation of the GCC libraries was an optional part of the simulator’s installation and the Mentor VIP does not find these libraries, an error message displays similar to the following:

```
ModelSim / Questa Sim
# ** Error: (vsim-8388) Could not find the MVC shared library : GCC not found in installation directory (/home/user/altera2/14.0/modelsim_ase) for platform "linux". Please install GCC version "gcc-4.7.4-linux"
```

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Version</th>
<th>GCC Version(s)</th>
<th>Search Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mentor Questa SIM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Table-1. Simulator GCC Requirements (cont.)**

<table>
<thead>
<tr>
<th>Simulator</th>
<th>GCC Version</th>
<th>Platform</th>
<th>Install Path</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mentor ModelSim</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.4d</td>
<td>4.7.4</td>
<td>(Linux 32 bit)</td>
<td>$&lt;install dir&gt;/gcc-4.7.4-linux</td>
</tr>
<tr>
<td></td>
<td>4.7.4</td>
<td>(Linux 64 bit)</td>
<td>$&lt;install dir&gt;/gcc-4.7.4-linux_x86_64</td>
</tr>
<tr>
<td></td>
<td>4.2.1</td>
<td>(Windows 32 bit)</td>
<td>$&lt;install dir&gt;/gcc-4.2.1-mingw32vc9</td>
</tr>
<tr>
<td><strong>Synopsys VCS/VCS-MX</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014.03-SP1 or 2014.12</td>
<td>4.7.2</td>
<td>(Linux 32 bit)</td>
<td>$VCS_HOME/gnu/linux/4.7.2_32-shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Linux 64 bit)</td>
<td>$VCS_HOME/gnu/4.7.2_32-shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Linux 64 bit)</td>
<td>$VCS_HOME/gnu/linux/4.7.2_64-shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$VCS_HOME/gnu/4.7.2_64-shared</td>
</tr>
<tr>
<td><strong>Cadence Incisive Enterprise</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.20.02 or 14.10.014</td>
<td>4.4</td>
<td>(Linux 32/64 bit)</td>
<td>$&lt;install dir&gt;/tools/cdsgcc/gcc/4.4</td>
</tr>
</tbody>
</table>

**Note:** If you set the environment variable VG_GNU_PACKAGE, then it is used instead of the VCS_HOME environment variable.

**Note:** Use the \texttt{cds\_tools.sh} executable to find the Incisive installation. Ensure \texttt{SPATH} includes the installation path and $<install dir>/tools/cdsgcc/gcc/4.4/install/bin. Also, ensure the \texttt{LD\_LIBRARY\_PATH} includes $<install dir>/tools/cdsgcc/gcc/4.4/install/lib.
Chapter 1
Mentor VIP Altera Edition

The Mentor VIP AE provides BFMs to simulate the behavior and to facilitate IP verification. The Mentor VIP AE includes the following interface:

- AXI4-Lite BFM with master, slave, and inline monitor interfaces

**Advantages of Using BFMs and Monitors**

Using the Mentor VIP AE has the following advantages:

- Accelerates the verification process by providing key verification test bench components
- Provides BFM components that implement the AMBA AXI Protocol Specification, which serves as a reference for the protocol
- Provides a full suite of configurable assertion checking in each BFM

**Implementation of BFMs**

The Mentor VIP AE BFMs, master, slave, and inline monitor components are implemented in SystemVerilog. Also included are wrapper components so that you can use the BFMs in VHDL verification environments with simulators that support mixed-language simulation.

The Mentor VIP AE provides a set of APIs for each BFM that you can use to construct, instantiate, control, and query signals in all BFM components. Your test programs must use only these public access methods and events to communicate with each BFM. To ensure support in current and future releases, your test programs must use the standard set of APIs to interface with the BFMs. Nonstandard APIs and user-generated interfaces cannot be supported in future releases.

The test program drives the stimulus to the DUTs and determines whether the behavior of the DUTs is correct by analyzing the responses. The BFMs translate the test program stimuli (transactions), creating the signaling for the AMBA AXI Protocol Specification. The BFMs also check for protocol compliance by firing an assertion when a protocol error is observed.
What Is a Transaction?

A transaction for Mentor VIP AE represents an instance of information that is transferred between a master and a slave peripheral, and that adheres to the protocol used to transfer the information. For example, a write transaction transfers an address phase, a data phase, followed by a response phase. A subsequent instance of transferred information requires a new and unique transaction.

Each transaction has a dynamic Transaction Record that exists for the life of the transaction. The life of a transaction record starts when it is created and ends when the transaction completes. The transaction record is automatically discarded when the transaction ends.

When created, a transaction contains transaction fields that you set to define two transaction aspects:

- **Protocol fields** are transferred over the protocol signals.
- **Operation fields** determine how the information is transferred and when the transfer is complete.

For example, a write transaction record holds the protection information in the prot protocol field; the value of this field is transferred over the AWPROT protocol signals during an address phase. A write transaction also has a transaction_done operation field that indicates when the transaction is complete; this field is not transferred over the protocol signals. These two types of transaction fields, protocol and operation, establish a dynamic record during the life of the transaction.

In addition to transaction fields, you specify arguments to tasks, functions, and procedures that permit you to create, set, and get the dynamic transaction record during the lifetime of a transaction. Each BFM has an API that controls how you access the BFM transaction record. How you access the record also depends on the source code language, whether it is VHDL or SystemVerilog. Methods for accessing transactions based on the language you use are explained in detail in the relevant chapters of this user guide.

AXI4-Lite Transactions

A complete read/write transaction transfers information between a master and a slave peripheral. Transaction fields described in “What Is a Transaction?” on page 22 determine what is transferred and how information is transferred. During the lifetime of a transaction, the roles of the master and slave ensure that a transaction completes successfully and that transferred information adheres to the protocol specification. Information flows in both directions during a transaction with the master initiating the transaction, and the slave reporting back to the master that the transaction has completed.

An AXI4-Lite protocol uses five channels (three write channels and two read channels) to transfer protocol information. Each of these channels has a pair of handshake signals, *VALID
and *READY, that indicates valid information on a channel and the acceptance of the information from the channel.

**AXI4-Lite Write Transaction Master and Slave Roles**

**Note**

The following description of a write transaction references SystemVerilog BFM API tasks. There are equivalent VHDL BFM API procedures that perform the same functionality.

For a write transaction, the master calls the `create_write_transaction()` task to define the information to be transferred and then calls the `execute_transaction()` task to initiate the transfer of information as **Figure 1-1** illustrates.

**Figure 1-1. Execute Write Transaction**

The `execute_transaction()` task results in the master calling the `execute_write_addr_phase()` task followed by the `execute_write_data_phase()` task as illustrated in **Figure 1-2**.
The master then calls the `get_write_response_phase()` task to receive the response from the slave and to complete its role in the write transaction.

The slave also creates a transaction by calling the `create_slave_transaction()` task to accept the transfer of information from the master. The address phase and data phase are received by the slave calling the `get_write_addr_phase()` task, followed by the `get_write_data_phase()` task as illustrated in Figure 1-3.
The slave then executes a write response phase by calling the `execute_write_response_phase()` task and completes its role in the write transaction.

**AXI Read Transaction Master and Slave Roles**

*Note*

The following description of a read transaction references the SystemVerilog BFM API tasks. There are equivalent VHDL BFM API procedures that perform the same functionality.

A read transaction is similar to a write transaction. The master initiates the read by calling the `create_read_transaction()` and `execute_transaction()` tasks. The `execute_transaction()` calls the `execute_read_addr_phase()` task followed by the `get_read_data_phase()` task as illustrated in Figure 1-4.
The slave creates a read transaction by calling the `create_slave_transaction()` task to accept the transfer of read information from the master. The slave accepts the address phase by calling the `get_read_addr_phase()` task, and then executes the data burst phase by calling the `execute_read_data_phase()` task as illustrated in Figure 1-5.
Chapter 2
SystemVerilog API Overview

This chapter provides the functional description of the SystemVerilog (SV) Application Programming Interface (API) for all BFM (master, slave, and monitor) components. For each BFM, you can configure the protocol transaction fields that are executed on the protocol signals, as well as control the operational transaction fields that permit delays to be introduced between the handshake signals for each of the five address, data, and response channels.

In addition, each BFM API has tasks that wait for certain events to occur on the system clock and reset signals, and tasks to get and set information about a particular transaction.

Figure 2-1. SystemVerilog BFM Internal Structure

Test Program SystemVerilog

<table>
<thead>
<tr>
<th>Configuration</th>
<th>set_config/get_config</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating Transaction</td>
<td>create*_transaction1</td>
</tr>
<tr>
<td>Executing Transaction</td>
<td>execute_transaction, execute*_phase2</td>
</tr>
<tr>
<td>Waiting Events</td>
<td>wait_on, get*_phase3</td>
</tr>
<tr>
<td>Access Transaction</td>
<td>get_rw_transaction/get<em>_phase3, get</em>_addr/get*_data3</td>
</tr>
</tbody>
</table>

Notes: 1. Refer to create*_transaction()
2. Refer to execute_transaction(), execute*_phase()
3. Refer to get*()
Configuration

Configuration sets timeout delays, error reporting, and other attributes of the BFM. Each BFM has a `set_config()` function that sets the configuration of the BFM. Refer to the individual BFM APIs for details.

Each BFM also has a `get_config()` function that returns the configuration of the BFM. Refer to the individual BFM APIs for details.

**set_config()**

The following test program code sets the burst timeout factor for a transaction in the master BFM.

```c
// Setting the burst timeout factor to 1000
master_bfm.set_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

**get_config()**

The following test program code gets the protocol signal hold time in the master BFM.

```c
// Getting hold time value
hold_time = master_bfm.get_config(AXI4_CONFIG_HOLD_TIME);
```

Creating Transactions

To transfer information between a master BFM and a slave DUT over the protocol signals, you must create a transaction in the master test program. Similarly, to transfer information between a master DUT and a slave BFM, you must create a transaction in the slave test program. To monitor the transfer of information using a monitor BFM, you must create a transaction in the monitor test program.

When you create a transaction, a **Transaction Record** is created and exists for the life of the transaction. This transaction record can be accessed by the BFM test programs during the life of the transaction as it transfers information between the master and slave.

**Transaction Record**

The transaction record contains two types of transaction fields, *protocol* and *operational*, that either transfer information over the protocol signals or define how and when a transfer occurs.

Protocol fields contain transaction information that is transferred over protocol signals. For example, the `prot` field is transferred over the AWPROT protocol signals during a write transaction.
Operational fields define how and when the transaction is transferred. Their content is not transferred over protocol signals. For example, the `operation_mode` field controls the blocking/nonblocking operation of a transaction, but this information is not transferred over the protocol signals.

**AXI4-Lite Transaction Definition**

The transaction record exists as a SystemVerilog class definition in each BFM. Example 2-1 shows the definition of the `axi4_transaction` class members that form the transaction record.

```
// Global Transaction Class
class axi4_transaction;
  // Protocol
  axi4_rw_e read_or_write;
  bit [((`MAX_AXI4_ADDRESS_WIDTH) - 1):0] addr;
  axi4_prot_e prot;
  bit [3:0] region; // Not supported in AXI4-Lite
  axi4_size_e size; // Not supported in AXI4-Lite
  axi4_burst_e burst; // Not supported in AXI4-Lite
  axi4_lock_e lock; // Not supported in AXI4-Lite
  axi4_cache_e cache; // Not supported in AXI4-Lite
  bit [3:0] qos; // Not supported in AXI4-Lite
  bit [((`MAX_AXI4_ID_WIDTH) - 1):0] id; // Not supported in AXI4-Lite
  bit [7:0] burst_length; // Not supported in AXI4-Lite
  bit [((`MAX_AXI4_USER_WIDTH) - 1):0] addr_user; // Not supported in AXI4-Lite
  bit [(((`MAX_AXI4_RDATA_WIDTH > `MAX_AXI4_WDATA_WIDTH) ?
    `MAX_AXI4_RDATA_WIDTH : `MAX_AXI4_WDATA_WIDTH)) - 1):0] data_words [];
  bit [(((`MAX_AXI4_WDATA_WIDTH / 8)) - 1):0] write_strobes [];
  axi4_response_e resp[];
  int address_valid_delay;
  int data_valid_delay[];
  int write_response_valid_delay;
  int address_ready_delay;
  int data_ready_delay[];
  int write_response_ready_delay;

  // Housekeeping
  bit gen_write_strobes = 1'b1;
  axi4_operation_mode_e operation_mode = AXI4_TRANSACTION_BLOCKING;
  axi4_write_data_mode_e write_data_mode = AXI4_DATA_WITH_ADDRESS;
  bit data_beat_done[]; // Not supported in AXI4-Lite
  bit transaction_done;

  ...
The contents of the transaction record are defined in Table 2-1.

### Table 2-1. Transaction Fields

<table>
<thead>
<tr>
<th>Transaction Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Protocol Transaction Fields</strong></td>
<td></td>
</tr>
<tr>
<td>addr</td>
<td>A bit vector (the length is equal to the ARADDR/AWADDR signal bus width) containing the starting address of the first transfer (beat) of a transaction. The addr value is transferred over the ARADDR or AWADDR signals for a read or write transaction, respectively.</td>
</tr>
<tr>
<td>prot</td>
<td>An enumeration containing the protection type of a transaction. The types of protection are:</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_SEC_DATA (default)</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_SEC_DATA</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_NONSEC_DATA</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_NONSEC_DATA</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_SEC_INST</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_SEC_INST</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_NONSEC_INST</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_NONSEC_INST</td>
</tr>
<tr>
<td>data_words</td>
<td>A bit vector (of length equal to the greater of the RDATA/WDATA signal bus widths) to hold the data words of the payload. A data_words is transferred over the RDATA or WDATA signals per beat of the read or write data channel, respectively.</td>
</tr>
<tr>
<td>write_strobes</td>
<td>A bit vector (of length equal to the WDATA signal bus width divided by 8) to hold the write strobes. A write_strobes is transferred over the WSTRB signals per beat of the write data channel.</td>
</tr>
<tr>
<td>resp</td>
<td>An enumeration to hold the response of a transaction. The types of response are:</td>
</tr>
<tr>
<td></td>
<td>AXI4_OKAY;</td>
</tr>
<tr>
<td></td>
<td>AXI4_SLVERR;</td>
</tr>
<tr>
<td></td>
<td>AXI4_DECERR;</td>
</tr>
<tr>
<td></td>
<td>A resp is transferred over the RRESP signals per beat of the read data channel, and over the BRESP signals for a write transaction, respectively.</td>
</tr>
</tbody>
</table>
**Table 2-1. Transaction Fields (cont.)**

<table>
<thead>
<tr>
<th>Transaction Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operational Transaction Fields</strong></td>
<td></td>
</tr>
<tr>
<td>read_or_write</td>
<td>An enumeration to hold the <em>read or write</em> control flag. The types of <code>read_or_write</code> are:</td>
</tr>
<tr>
<td></td>
<td><code>AXI4_TRANS_READ</code></td>
</tr>
<tr>
<td></td>
<td><code>AXI4_TRANS_WRITE</code></td>
</tr>
<tr>
<td>address_valid_delay</td>
<td>An integer to hold the delay value of the address channel <code>AWVALID</code> and <code>ARVALID</code> signals (measured in ACLK cycles) for a read or write transaction, respectively.</td>
</tr>
<tr>
<td>data_valid_delay</td>
<td>An integer to hold the delay value of the data channel <code>WVALID</code> and <code>RVALID</code> signals (measured in ACLK cycles) for a read or write transaction, respectively.</td>
</tr>
<tr>
<td>write_response_valid_delay</td>
<td>An integer to hold the delay value of the write response channel <code>BVALID</code> signal (measured in ACLK cycles) for a write transaction.</td>
</tr>
<tr>
<td>address_ready_delay</td>
<td>An integer to hold the delay value of the address channel <code>AWREADY</code> and <code>ARREADY</code> signals (measured in ACLK cycles) for a read or write transaction, respectively.</td>
</tr>
<tr>
<td>data_ready_delay</td>
<td>An integer to hold the delay value of the data channel <code>WREADY</code> and <code>RREADY</code> signals (measured in ACLK cycles) for a read or write transaction, respectively.</td>
</tr>
<tr>
<td>write_response_ready_delay</td>
<td>An integer to hold the delay value of the write response channel <code>BREADY</code> signal (measured in ACLK cycles) for a write transaction.</td>
</tr>
<tr>
<td>gen_write_strobes</td>
<td>Automatically correct write strobes flag. Refer to <strong>Automatic Generation of Byte Lane Strobes</strong> for details.</td>
</tr>
<tr>
<td>operation_mode</td>
<td>An enumeration to hold the <em>operation mode</em> of the transaction. The two types of <code>operation_mode</code> are:</td>
</tr>
<tr>
<td></td>
<td><code>AXI4_TRANSACTION_NON_BLOCKING</code></td>
</tr>
<tr>
<td></td>
<td><code>AXI4_TRANSACTION_BLOCKING</code></td>
</tr>
</tbody>
</table>
Creating Transactions

The master BFM API allows you to create a master transaction by providing only the address argument for a read or write transaction. All other protocol transaction fields automatically default to legal protocol values to create a complete master transaction record. Refer to the `create_read_transaction()` and `create_write_transaction()` functions for default protocol read and write transaction field values.

The slave BFM API allows you to create a slave transaction without providing any arguments. All protocol transaction fields automatically default to legal protocol values to create a complete slave transaction record. Refer to the `create_slave_transaction()` function for default protocol transaction field values.

The monitor BFM API allows you to create a monitor transaction without providing any arguments. All protocol transaction fields automatically default to legal protocol values to create a complete slave transaction record. Refer to the `create_monitor_transaction()` function for default protocol transaction field values.

Note

If you change the default value of a protocol transaction field, this value is valid for all future transactions until you set a new value.

---

**Table 2-1. Transaction Fields (cont.)**

<table>
<thead>
<tr>
<th>Transaction Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>write_data_mode</td>
<td>An enumeration to hold the write data mode control flag. The types of write_data_mode are:</td>
</tr>
<tr>
<td></td>
<td>AXI4_DATA_AFTER_ADDRESS</td>
</tr>
<tr>
<td></td>
<td>The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a data_valid_delay setting instead to avoid the possible deadlock situation.</td>
</tr>
<tr>
<td></td>
<td>AXI4_DATA_WITH_ADDRESS (default)</td>
</tr>
<tr>
<td></td>
<td>The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay.</td>
</tr>
<tr>
<td>transaction_done</td>
<td>A bit to hold the done flag for a transaction when it has completed.</td>
</tr>
</tbody>
</table>
**create*Transaction()**

There are two master BFM API functions available to create transactions, `create_read_transaction()` and `create_write_transaction()`, a `create_slave_transaction()` for the slave BFM API, and a `create_monitor_transaction()` for the monitor BFM API.

For example, the following master BFM test program creates a simple write transaction with a start address of 1, and a single data phase with a data value of 2, the master BFM test program would contain the following code:

```
// Define a variable trans of type axi4_transaction
axi4_transaction write_trans;

// Create master write transaction
write_trans = bfm.create_write_transaction(1);
write_trans.data_words = 2;
```

For example, to create a simple slave transaction, the slave BFM test program contains the following code:

```
// Define a variable slave_trans of type axi4_transaction
axi4_transaction slave_trans;

// Create slave transaction
slave_trans = bfm.create_slave_transaction();
```

**Executing Transactions**

Executing a transaction in a master/slave BFM test program initiates the transaction onto the protocol signals. Each master/slave BFM API has execution tasks that push transactions into the BFM internal transaction queues. Figure 2-1 on page 27 illustrates the internal BFM structure.

**execute_transaction(), execute*_phase()**

If the DUT is a slave, then the `execute_transaction()` task is called in the master BFM test program. If the DUT is a master, then the `execute*_phase()` task is called in the slave BFM test program.

For example, to execute a master write transaction the master BFM test program contains the following code:

```
// By default the execution of a transaction will block
bfm.execute_transaction(write_trans);
```

For example, to execute a slave write response phase, the slave BFM test program contains the following code:
// By default the execution of a transaction will block
bfm.execute_write_response_phase(slave_trans);

## Waiting Events

Each BFM API has tasks that block the test program code execution until an event has occurred.

The `wait_on()` task blocks the test program until an ACLK or ARESETn signal event has occurred before proceeding.

The `get*_transaction(), get*_phase(), get*_cycle()` tasks block the test program code execution until a complete transaction, phase, or cycle has occurred, respectively.

### `wait_on()`

A BFM test program can wait for the positive edge of the ARESETn signal using the following code:

```plaintext
// Block test program execution until the positive edge of the clock
bfm.wait_on(AXI4_RESET_POSEDGE);
```

### `get*_transaction(), get*_phase(), get*_cycle()`

A slave BFM test program can use a received write address phase to form the response to the write transaction. The test program gets the write address phase for the transaction by calling the `get_write_addr_phase()` task. This task blocks until it has received the address phase, allowing the test program to call the `execute_write_response_phase()` task for the transaction at a later stage, as shown in the slave BFM test program in **Example 2-2**.

**Example 2-2. Slave Test Program Using `get_write_addr_phase()`**

```plaintext
slave_trans = bfm.create_slave_transaction();
bfm.get_write_addr_phase(slave_trans);
...
bfm.execute_write_response_phase(slave_trans);
```

### Note

Not all BFM APIs support the full complement of `get*_transaction(), get*_phase(), get*_cycle()` tasks. Refer to the individual master, slave, or monitor BFM API for details.
Access Transaction Record

Each BFM API has tasks that can access a complete or partially complete Transaction Record. The `set*()` and `get*()` tasks are used in a test program to set and get information from the transaction record.

**Note**

The `set*()` and `get*()` tasks are not explicitly described in each BFM API chapter. The simple rule for the task name is `set_` or `get_` followed by the name of the transaction field accessed. Refer to “Transaction Fields” on page 30 for transaction field name details.

**set*()**

For example, to set the WSTRB write strobes signal in the Transaction Record of a write transaction, the master test program would use the `set_write_strobes()` task, as shown in the following code:

```systemverilog
code
write_trans.set_write_strobes(4'b0010);
```

**get*()**

For example, a slave BFM test program uses a received write address phase to get the AWPROT signal value from the Transaction Record, as shown in the following slave BFM test program code:

```systemverilog
code
// Define a variable prot_value of type axi4_transaction
axi4_prot_e prot_value;
slave_trans = bfm.create_slave_transaction();

// Wait for a write address phase
bfm.get_write_addr_phase(slave_trans);

... ...

// Get the AWPROT signal value of the slave transaction
prot_value = bfm.get_prot(slave_trans);
```

**Operational Transaction Fields**

Operational transaction fields control the way a transaction is executed onto the protocol signals. They also indicate when a data phase (beat) or transaction is complete.
Automatic Generation of Byte Lane Strobes

The master BFM permits unaligned and narrow write transfers by using byte lane strobe (WSTRB) signals to indicate which byte lanes contain valid data per data phase (beat).

When you create a write transaction in your master BFM test program, the write_strobes variable is available to store the write strobe values for the write data phase (beat) in the transaction. To assist you in creating the correct byte lane strobes, automatic correction of any previously set write_strobes is performed by default during execution of the write transaction, or write data phase (beat). You can disable this default behavior by setting the operational transaction field gen_write_strobes = 0, which allows any previously set write_strobes to pass through uncorrected onto the protocol WSTRB signals. In this mode, with the automatic correction disabled, you are responsible for setting the correct write_strobes for the whole transaction.

The automatic correction algorithm performs a bit-wise AND operation on any previously set write_strobes. To do the corrections, the correction algorithm uses the equations described in the AMBA AXI Protocol Specification, Section A3.4.1, that define valid write data byte lanes for legal protocol. Therefore, if you require automatic generation of all write_strobes, before the write transaction executes, you must set all write_strobes to 1, indicating that all bytes lanes initially contain valid write data prior to the execution of the write transaction. Automatic correction then sets the relevant write_strobes to 0 to produce legal protocol WSTRB signals.

Operation Mode

By default, each read or write transaction performs a blocking operation which prevents a following transaction from starting until the current active transaction completes.

You can configure this behavior to be nonblocking by setting the operation_mode transaction field to the enumerate type value AXI4_TRANSACTION_NON_BLOCKING instead of the default AXI4_TRANSACTION_BLOCKING.

For example, in a master BFM test program you create a transaction by calling the create_read_transaction() or create_write_transaction() tasks, which creates a transaction record. Before executing the transaction record, you can change the operation_mode as follows:

```c
// Create a write transaction to create a transaction record
trans = bfm.create_write_transaction(1);

// Change operation_mode to be nonblocking in the transaction record
trans.operation_mode(AXI4_TRANSACTION_NON_BLOCKING);
```
Channel Handshake Delay

Each of the five protocol channels have *VALID and *READY handshake signals that control the rate at which information is transferred between a master and slave. Refer to the Handshake Delay for details of the AXI4-Lite BFM API.

Handshake Delay

The delay between the *VALID and *READY handshake signals for each of the five protocol channels is controlled in a BFM test program using `execute_*_ready()`, `get_*_ready()`, and `get_*_cycle()` tasks. The `execute_*_ready()` tasks place a value onto the *READY signals and the `get_*_ready()` tasks retrieve a value from the *READY signals. The `get_*_cycle()` tasks wait for a *VALID signal to be asserted and are used to insert a delay between the *VALID and *READY signals in the BFM test program.

For example, the master BFM test program code below inserts a specified delay between the read channel RVALID and RREADY handshake signals using the `execute_read_data_ready()` and `get_read_data_cycle()` tasks.

```
// Set the RREADY signal to '0' so that it is nonblocking
fork
  bfm.execute_read_data_ready(1'b0);
join_none

// Wait until the RVALID signal is asserted and then wait_on the specified // number of ACLK cycles
bfm.get_read_data_cycle;
repeat(5) bfm.wait_on(AXI4_CLOCK_POSEDGE);

// Set the RREADY signal to '1' so that it blocks for an ACLK cycle
bfm.execute_read_data_ready(1'b1);
```

VALID Signal Delay Transaction Fields

The transaction record contains a *_valid_delay transaction field for each of the five protocol channels to configure the delay value prior to the assertion of the *VALID signal for the channel. The master BFM holds the delay configuration for the *VALID signals that it asserts, and the slave BFM holds the delay configuration for the *VALID signals that it asserts. Table 2-2 specifies which *valid_delay fields are configured by the master and slave BFMs.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Operational Transaction Field</th>
<th>Configuration BFM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWVALID</td>
<td>address_valid_delay</td>
<td>Master</td>
</tr>
<tr>
<td>WVALID</td>
<td>data_valid_delay</td>
<td>Master</td>
</tr>
<tr>
<td>BVALID</td>
<td>write_response_valid_delay</td>
<td>Slave</td>
</tr>
</tbody>
</table>
**SystemVerilog API Overview**

**Operational Transaction Fields**

---

**Table 2-2. Master and Slave *valid_delay Configuration Fields (cont.)**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Operational Transaction Field</th>
<th>Configuration BFM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARVALID</td>
<td>address_valid_delay</td>
<td>Master</td>
</tr>
<tr>
<td>RVALID</td>
<td>data_valid_delay</td>
<td>Slave</td>
</tr>
</tbody>
</table>

---

**READY Handshake Signal Delay Transaction Fields**

The transaction record contains a *ready_delay transaction field for each of the five protocol channels to store the delay value that occurred between the assertion of the *VALID and *READY handshake signals for the channel. Table 2-3 specifies the *ready_delay field corresponding to the *READY signal delay.

**Table 2-3. Master and Slave *ready_delay Transaction Fields**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Operational Transaction Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWREADY</td>
<td>address_ready_delay</td>
</tr>
<tr>
<td>WREADY</td>
<td>data_ready_delay</td>
</tr>
<tr>
<td>BREADY</td>
<td>write_response_ready_delay</td>
</tr>
<tr>
<td>ARREADY</td>
<td>address_ready_delay</td>
</tr>
<tr>
<td>RREADY</td>
<td>data_ready_delay</td>
</tr>
</tbody>
</table>

---

**Transaction Done**

The transaction_done field in each transaction indicates when the transaction is complete.

In a master BFM test program, you call the get_read_data_phase() task to investigate whether a read transaction is complete, and the get_write_response_phase() to investigate whether a write transaction is complete.
Chapter 3
SystemVerilog Master BFM

This chapter provides information about the SystemVerilog master BFM. Each BFM has an API that contains tasks and functions to configure the BFM and to access the dynamic Transaction Record during the lifetime of the transaction.

Master BFM Protocol Support

The AXI4-Lite master BFM supports the AMBA AXI4-Lite protocol with restrictions described in “Protocol Restrictions” on page 17.

Master Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI Protocol Specification chapter, which you can use to reference details of the following master BFM API timing and events.

The AMBA AXI Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the master BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the test bench and design IP as a result of these directives, declarations, options, or initialization files:

- `timescale directives in design elements
- Timeprecision declarations in design elements
- Compiler command-line options
- Simulation command-line options
- Local or site-wide simulator initialization files

If there is no timescale directive, the default time unit and time precision are tool specific. The recommended practice is to use timeunit and timeprecision declarations. For details, refer to Section 3.14, “System Time Units and Precision,” of the IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800™-2012 , February 21, 2013. This user guide refers to this document as the IEEE Standard for SystemVerilog.
Master BFM Configuration

A master BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. It has parameters that configure the widths of the address and data signals, and transaction fields to specify timeout factors, setup and hold times, and so on.

You can change the address and data signal widths from their default settings by assigning them new values, usually in the top-level module of the test bench. These new values are then passed to the master BFM using a parameter port list of the master BFM module. For example, the code extract below shows the master BFM with the address and data signal widths defined in module top() and passed to the master_test_program parameter port list:

```verilog
module top();
    parameter AXI4_ADDRESS_WIDTH = 24;
    parameter AXI4_RDATA_WIDTH = 16;
    parameter AXI4_WDATA_WIDTH = 16;

    master_test_program #(AXI4_ADDRESS_WIDTH, AXI4_RDATA_WIDTH, AXI4_WDATA_WIDTH) bfm_master(....);
```

Table 3-1 lists parameter names for the address, data signals, etc, and their default values.

<table>
<thead>
<tr>
<th>Signal Width Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_ADDRESS_WIDTH</td>
<td>Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.</td>
</tr>
<tr>
<td>AXI4_RDATA_WIDTH</td>
<td>Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>AXI4_WDATA_WIDTH</td>
<td>Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>index</td>
<td>Ignored for the SystemVerilog master BFM.</td>
</tr>
<tr>
<td>READ_ISSUING_CAPABILITY</td>
<td>The maximum number of outstanding read transactions that can be issued from the master BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
</tbody>
</table>
A master BFM has configuration fields that you can set with the `set_config()` function to configure timeout factors, and setup and hold times, and so on. You can also get the value of a configuration field using the `get_config()` function. Table 3-2 describes the full list of configuration fields.

### Table 3-2. Master BFM Configuration

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE_ISSUING_CAPABILITY</td>
<td>The maximum number of outstanding write transactions that can be issued from the master BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
<tr>
<td>COMBINED_ISSUING_CAPABILITY</td>
<td>The maximum number of outstanding combined read and write transactions that can be issued from the master BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
<tr>
<td>AXI4_CONFIG_SETUP_TIME</td>
<td>The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_HOLD_TIME</td>
<td>The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR</td>
<td>The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_BURST_TIMEOUT_FACTOR</td>
<td>The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY</td>
<td>The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods. Default: 1000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY</td>
<td>The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods. Default: 10000.</td>
</tr>
</tbody>
</table>
### Table 3-2. Master BFM Configuration (cont.)

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY</td>
<td>The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY</td>
<td>The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY</td>
<td>The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods. Default 10000.</td>
</tr>
</tbody>
</table>

### Master Attributes

| AXI4_CONFIG_AXI4LITE_axi4 | Configures the AXI4 master BFM to be AXI4-Lite compatible.  
\[0 = \text{disabled (default)}
\]
\[1 = \text{enabled} \] |

### Slave Attributes

| AXI4_CONFIG_SLAVE_START_ADDR | Configures the start address map for the slave. |
| AXI4_CONFIG_SLAVE_END_ADDR | Configures the end address map for the slave. |

### Error Detection

| AXI4_CONFIG_ENABLE_ALL_ASSERTIONS | Global enable/disable of all assertion checks in the BFM.  
\[0 = \text{disabled}
\]
\[1 = \text{enabled (default)} \] |
| AXI4_CONFIG_ENABLE_ASSERTION | Individual enable/disable of assertion check in the BFM.  
\[0 = \text{disabled}
\]
\[1 = \text{enabled (default)} \] |

1. Refer to [Master Timing and Events](#) for details of simulator time-steps.

### Master Assertions

Each master BFM performs protocol error checking using the built-in assertions.
Note

The built-in BFM assertions are independent of programming language and simulator.

Assertion Configuration

By default, all built-in assertions are enabled in the master AXI4-Lite BFM. To globally disable them in the master BFM, use the `set_config()` command as the following example illustrates:

```systemverilog
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS, 0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of `get_config()` and `set_config()` commands on the respective assertion. For example, to disable assertion checking for the AWADDR signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```systemverilog
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI4_CONFIG_ENABLE_ASSERTION);

// Assign the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI4_AWADDR_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector);
```

Note

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the `AXI4_AWADDR_CHANGED_BEFORE_AWREADY` assertion, follow the above code sequence and assign the assertion within the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI4-Lite assertions, refer to “AXI4-Lite Assertions” on page 375.

SystemVerilog Master API

This section describes the SystemVerilog master API.
**set_config()**

This function sets the configuration of the master BFM.

**Prototype**

```
function void set_config
{
    input axi4_config_e config_name,
    input axi4_max_bits_t config_val
};
```

**Arguments**

- **config_name**
  - Configuration name:
    - AXI4_CONFIG_SETUP_TIME
    - AXI4_CONFIG_HOLD_TIME
    - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
    - AXI4_CONFIG_AXI4LITE_axi4
    - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
    - AXI4_CONFIG_ENABLE_ASSERTION
    - AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
    - AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
    - AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
    - AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
    - AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
    - AXI4_CONFIG_SLAVE_START_ADDR
    - AXI4_CONFIG_SLAVE_END_ADDR

See “Master BFM Configuration” on page 40 for descriptions and valid values.

**Returns**

None

**Example**

```
set_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR, 1000);
```
get_config()

This function gets the configuration of the master BFM.

Prototype

```systemverilog
function void get_config
    (input axi4_config_e config_name,
    );
```

Arguments

- **config_name**
  - Configuration name:
    - AXI4_CONFIG_SETUP_TIME
    - AXI4_CONFIG_HOLD_TIME
    - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
    - AXI4_CONFIG_AXI4LITE_axi4
    - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
    - AXI4_CONFIG_ENABLE_ASSERTION
    - AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
    - AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
    - AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
    - AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
    - AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
    - AXI4_CONFIG_SLAVE_START_ADDR
    - AXI4_CONFIG_SLAVE_END_ADDR

Returns

- **config_val**
  - See “Master BFM Configuration” on page 40 for descriptions and valid values.

Example

```systemverilog
get_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR);
```
create_write_transaction()

This nonblocking function creates a write transaction with a start address $addr$ argument. All other transaction fields default to legal protocol values, unless previously assigned a value. It returns with the $axi4_transaction$ record.

**Prototype**

```vhdl
function automatic axi4_transaction create_write_transaction
(  
   input bit [((AXI4_ADDRESS_WIDTH) - 1):0]  addr);
Arguments

addr   Start address

**Protocol**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prot</td>
<td>Protection:</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_SEC_DATA; (default)</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_SEC_DATA;</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_NONSEC_DATA;</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_NONSEC_DATA;</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_SEC_INST;</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_SEC_INST;</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_NONSEC_INST;</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_NONSEC_INST;</td>
</tr>
</tbody>
</table>

**Transaction Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_words</td>
<td>Data words.</td>
</tr>
<tr>
<td>write_strobes</td>
<td>Write strobes: Each strobe 0 or 1.</td>
</tr>
<tr>
<td>resp</td>
<td>Burst response:</td>
</tr>
<tr>
<td></td>
<td>AXI4_OKAY;</td>
</tr>
<tr>
<td></td>
<td>AXI4_SLVERR;</td>
</tr>
<tr>
<td></td>
<td>AXI4_DECERR;</td>
</tr>
</tbody>
</table>

**Operational Transaction Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_write_strobes</td>
<td>Generate write strobes flag:</td>
</tr>
<tr>
<td></td>
<td>0 = user supplied write strobes.</td>
</tr>
<tr>
<td></td>
<td>1 = auto-generated write strobes (default).</td>
</tr>
<tr>
<td>operation_mode</td>
<td>Operation mode:</td>
</tr>
<tr>
<td></td>
<td>AXI4_TRANSACTION_NON_BLOCKING;</td>
</tr>
<tr>
<td></td>
<td>AXI4_TRANSACTION_BLOCKING; (default)</td>
</tr>
<tr>
<td>write_data_mode</td>
<td>Write data mode:</td>
</tr>
<tr>
<td></td>
<td>AXI4_DATA_AFTER_ADDRESS;</td>
</tr>
<tr>
<td></td>
<td>The master first drives the address phase and,</td>
</tr>
<tr>
<td></td>
<td>after it completes, it drives the corresponding</td>
</tr>
<tr>
<td></td>
<td>data phases. The master waits for AWREADY</td>
</tr>
<tr>
<td></td>
<td>before asserting WVALID. For a slave designed</td>
</tr>
<tr>
<td></td>
<td>to wait for WVALID before asserting AWREADY,</td>
</tr>
<tr>
<td></td>
<td>using this mode may cause a deadlock situation.</td>
</tr>
<tr>
<td></td>
<td>This mode will force the data transfer to</td>
</tr>
<tr>
<td></td>
<td>start after the address transfer completes;</td>
</tr>
<tr>
<td></td>
<td>however, it is recommended that you use the</td>
</tr>
<tr>
<td></td>
<td>**_DATA_WITH_ADDRESS along with a data_valid_delay</td>
</tr>
<tr>
<td></td>
<td>setting instead to avoid the possible deadlock</td>
</tr>
<tr>
<td></td>
<td>situation.</td>
</tr>
<tr>
<td></td>
<td>AXI4_DATA_WITH_ADDRESS; (default)</td>
</tr>
<tr>
<td></td>
<td>The master drives the address and the data</td>
</tr>
<tr>
<td></td>
<td>phase in a nonblocking process; it asserts</td>
</tr>
<tr>
<td></td>
<td>AWVALID and then asserts WVALID depending on</td>
</tr>
<tr>
<td></td>
<td>data_valid_delay. If data_valid_delay is set</td>
</tr>
<tr>
<td></td>
<td>to 0, then AWVALID and WVALID are asserted at</td>
</tr>
<tr>
<td></td>
<td>the same time; otherwise, WVALID is asserted</td>
</tr>
<tr>
<td></td>
<td>after data_valid_delay.</td>
</tr>
</tbody>
</table>
### Operational Transaction Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address_valid_delay</td>
<td>Address channel AWVALID delay measured in ACLK cycles for this transaction (default = 0).</td>
</tr>
<tr>
<td>data_valid_delay</td>
<td>Write data channel WVALID delay measured in ACLK cycles for this transaction (default = 0).</td>
</tr>
<tr>
<td>write_response_ready_delay</td>
<td>Write response channel BREADY delay measured in ACLK cycles for this transaction (default = 0).</td>
</tr>
<tr>
<td>transaction_done</td>
<td>Write transaction done flag for this transaction.</td>
</tr>
</tbody>
</table>

### Returns

The `axi4_transaction` record.

### Example

```verbatim
// Create a write transaction to start address 16.
trans = bfm.create_write_transaction(16);
trans.set_data_words = ('hACE0ACE1, 0); //Note: array element 0.
```
**create_read_transaction()**

This nonblocking function creates a read transaction with a start address `addr`. All other transaction fields default to legal AXI4-Lite protocol values, unless previously assigned a value. It returns the `axi4_transaction` record.

**Prototype**

```verbatim
function automatic axi4_transaction create_read_transaction
    (input bit [((AXI4_ADDRESS_WIDTH) - 1):0]  addr);
```

**Arguments**

- **addr**: Start address

**Protocol Transaction Fields**

- **prot**: Protection:
  - AXI4_NORM_SEC_DATA; (default)
  - AXI4_PRIV_SEC_DATA;
  - AXI4_NORM_NONSEC_DATA;
  - AXI4_PRIV_NONSEC_DATA;
  - AXI4_NORM_SEC_INST;
  - AXI4_PRIV_SEC_INST;
  - AXI4_NORM_NONSEC_INST;
  - AXI4_PRIV_NONSEC_INST;

- **data_words**: Data words.

- **resp**: Burst response:
  - AXI4_OKAY;
  - AXI4_EXOKAY;
  - AXI4_SLVERR;
  - AXI4_DECERR;

**Operational Transaction Fields**

- **operation_mode**: Operation mode:
  - AXI4_TRANSACTION_NON_BLOCKING;
  - AXI4_TRANSACTION_BLOCKING; (default)

- **address_valid_delay**: Address channel ARVALID delay measured in ACLK cycles for this transaction (default = 0).

- **data_ready_delay**: Read data channel RREADY delay array measured in ACLK cycles for this transaction (default = 0).

- **transaction_done**: Read transaction done flag for this transaction.

**Returns**

- **axi4_transaction**: The transaction record:

**Example**

```verbatim
// Read data to start address 16.
trans = bfm.create_read_transaction(16);
```
execute_transaction()

This task executes a master transaction previously created by the `create_write_transaction()`, or `create_read_transaction()`, functions. The transaction can be blocking (default) or nonblocking, defined by the transaction record `operation_mode` field.

The results of `execute_transaction()` for write transactions varies based on how write transaction fields are set. If the `gen_write_strobes` transaction field is set, `execute_transaction()` automatically corrects any previously set `write_strobes`. However, if the `gen_write_strobes` field is not set, then any previously assigned `write_strobes` will be passed through onto the WSTRB protocol signals, which can result in a protocol violation if not correctly set. Refer to “Automatic Correction of Byte Lane Strobes” on page 147 for more details.

If a write transaction `write_data_mode` field is set to AXI4_DATA_WITH_ADDRESS, `execute_transaction()` calls the `execute_write_addr_phase()` and `execute_write_data_phase()` tasks simultaneously; otherwise, `execute_write_data_phase()` will be called after `execute_write_addr_phase()` so that the write data phase will occur after the write address phase (default). It will then call the `get_write_response_phase()` task to complete the write transaction.

For a read transaction, `execute_transaction()` calls the `execute_read_addr_phase()` task followed by the `get_read_data_phase()` task to complete the read transaction.

**Prototype**

```systemverilog
task automatic execute_transaction
    (axi4_transaction trans);
```

**Arguments**

- `trans` The `axi4_transaction` record.

**Returns**

- None

**Example**

```systemverilog
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Execute the read_trans transaction.
bfm.execute_transaction(read_trans);
```
execute_write_addr_phase()

This task executes a master write address phase previously created by the 
create_write_transaction() function. This phase can be blocking (default) or nonblocking, as 
defined by the transaction operation_mode field.

It sets the AWVALID protocol signal at the appropriate time defined by the transaction 
address_valid_delay field.

Prototype

task automatic execute_write_addr_phase

(axi4_transaction trans);

Arguments

trans The axi4_transaction record.

Returns

None

Example

// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Execute the write_trans transaction.
bfm.execute_transaction(write_trans);
execute_read_addr_phase()

This task executes a master read address phase previously created by the 
create_read_transaction() function. This phase can be blocking (default) or nonblocking, as 
defined by the transaction operation_mode field.

It sets the ARVALID protocol signal at the appropriate time, defined by the transaction 
address_valid_delay field.

Prototype

```plaintext
task automatic execute_read_addr_phase
(
    axi4_transaction trans
);
```

Arguments

- **trans**: The axi4_transaction record.

Returns

None

Example

```plaintext
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Execute the write_trans transaction.
bfm.execute_transaction(read_trans);
```
execute_write_data_phase()

This task executes a write data phase (beat) previously created by the
create_write_transaction() task. This phase can be blocking (default) or nonblocking, defined
by the transaction record operation_mode field.

The execute_write_data_phase() sets the WVALID protocol signal at the appropriate time
defined by the transaction record data_valid_delay field.

Prototype

```plaintext
task automatic execute_write_data_phase
    (axi4_transaction trans
    int index = 0, // Optional
    output bit last
    );
```

Arguments

- trans: The axi4_transaction record.
- index: Data phase (beat) number. Note: '0' for AXI4-Lite
- last: Flag to indicate that this phase is the last beat of data.

Returns

None

Example

```plaintext
// Declare a local variable to hold the transaction record.
axi4lite_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Execute the write data phase for the write_trans transaction.
bfm.execute_write_data_phase(write_trans, 0, last); //Note array element 0
```
**get_read_data_phase()**

This blocking task gets a read data phase previously created by the `create_read_transaction()` task.

---

**Note**

The `get_read_data_phase()` sets the RREADY protocol signal at the appropriate time defined by the `data_ready_delay` field and sets the `transaction_done` field to 1 to indicate the whole read transaction has completed.

---

**Prototype**

```systemverilog
task automatic get_read_data_phase
    (axi4_transaction trans
    int index = 0 // Optional
    );
```

**Arguments**

- `trans` (Optional) The `axi4_transaction` record.
- `index` (Optional) Data phase (beat) number. Note: ‘0’ for AXI4-Lite

**Returns**

None

**Example**

```systemverilog
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Get the read data phase for the read_trans transaction.
bfm.get_read_data_phase(read_trans, 0); // Note: array element 0.
```
get_write_response_phase()

This blocking task gets a write response phase previously created by the create_write_transaction() task.

**Note**

The get_write_response_phase() sets the transaction_done field to 1 when the transaction completes to indicate the whole transaction is complete.

Prototype

task automatic get_write_response_phase
    (axi4_transaction trans);

Arguments

trans The axi4_transaction record.

Returns

None

Example

```verilog
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Get the write response phase for the write_trans transaction.
bfm.get_write_response_phase(write_trans);
```
get_read_addr_ready()

This blocking task returns the value of the read address channel ARREADY signal using the ready argument. It will block for one ACLK period.

Prototype

```
task automatic get_read_addr_ready
    (output bit ready

);```

Arguments

ready The value of the ARREADY signal.

Returns

ready

Example

```
// Get the ARREADY signal value
bfm.get_read_addr_ready(ready);
```
**get_read_data_cycle()**

This blocking task waits until the read data channel RVALID signal is asserted.

**Prototype**

```systemverilog
task automatic get_read_data_cycle();
```

**Arguments**

None

**Returns**

None

**Example**

```systemverilog
// Waits until the read data channel RVALID signal is asserted.
bfm.get_read_data_cycle();
```
get_write_addr_ready()

This blocking task returns the value of the write address channel AWREADY signal using the \textit{ready} argument. It will block for one ACLK period.

**Prototype**

```verilog
task automatic get_write_addr_ready
(
   output bit ready
);
```

**Arguments**

- \textit{ready} 
  The value of the AWREADY signal.

**Returns**

None

**Example**

```verilog
// Get the value of the AWREADY signal
bfm.get_write_addr_ready();
```
get_write_data_ready()

This blocking task returns the value of the write data channel WREADY signal using the `ready` argument. It will block for one ACLK period.

**Prototype**
```verilog
task automatic get_write_data_ready
    (output bit ready)
);
```

**Arguments**
- `ready` The value of the WREADY signal.

**Returns** None

**Example**

```verilog
// Get the value of the WREADY signal
bfm.get_write_data_ready();
```
get_write_response_cycle()

This blocking task waits until the write response channel BVALID signal is asserted.

Prototype   task automatic get_write_response_cycle();
Arguments    None
Returns      None

Example

// Wait until the write response channel BVALID signal is asserted.
bfm.get_write_response_cycle();
execute_read_data_ready()

This task executes a read data ready by placing the `ready` argument value onto the RREADY signal. It will block for one ACLK period.

**Prototype**

```
task automatic execute_read_data_ready
    (bit ready);
```

**Arguments**

- `ready` — The value to be placed onto the RREADY signal

**Returns**

None

**Example**

```
// Assert and deassert the RREADY signal
forever begin
    bfm.execute_read_data_ready(1'b0);
    bfm.wait_on(AXI4_CLOCK_POSEDGE);
    bfm.wait_on(AXI4_CLOCK_POSEDGE);
    bfm.execute_read_data_ready(1'b1);
    bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
```
execute_write_resp_ready()

This task executes a write response ready by placing the ready argument value onto the BREADY signal. It will block for one ACLK period.

Prototype

task automatic execute_write_resp_ready
(
    bit ready
);

Arguments
ready The value to be placed onto the BREADY signal

Returns
None

Example

// Assert and deassert the BREADY signal
forever begin
    bfm.execute_write_resp_ready(1'b0);
    bfm.wait_on(AXI4_CLOCK_POSEDGE);
    bfm.wait_on(AXI4_CLOCK_POSEDGE);
    bfm.execute_write_resp_ready(1'b1);
    bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
wait_on()

This blocking task waits for an event(s) on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

**Prototype**

```systemverilog
task automatic wait_on
(
    a4_wait_e phase,
    input int count = 1 //Optional
);
```

**Arguments**

- **phase**: Wait for:
  - AXI4_CLOCK_POSEDGE
  - AXI4_CLOCK_NEGEDGE
  - AXI4_CLOCK_ANYEDGE
  - AXI4_CLOCK_0_TO_1
  - AXI4_CLOCK_1_TO_0
  - AXI4_RESET_POSEDGE
  - AXI4_RESET_NEGEDGE
  - AXI4_RESET_ANYEDGE
  - AXI4_RESET_0_TO_1
  - AXI4_RESET_1_TO_0

- **count**: (Optional) Wait for a number of events to occur set by *count*. (default = 1)

**Returns**

None

**Example**

```systemverilog
bfm.wait_on(AXI4_RESET_POSEDGE);
bfm.wait_on(AXI4_CLOCK_POSEDGE,10);
```
This chapter describes the SystemVerilog slave BFM. Each BFM has an API that contains tasks and functions to configure the BFM and to access the dynamic Transaction Record during the lifetime of the transaction.

**Slave BFM Protocol Support**

This section defines protocol support for various AXI BFMs. The AXI4-Lite slave BFM supports the AMBA AXI4-Lite protocol with restrictions described in “Protocol Restrictions” on page 17.

**Slave Timing and Events**

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI Protocol Specification chapter, which you can use to reference details of the following slave BFM API timing and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the slave BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the test bench and design IP based on using the directives, declarations, options, and initialization files below:

- `timescale directives in design elements`
- Timeprecision declarations in design elements
- Compiler command-line options
- Simulation command-line options
- Local or site-wide simulator initialization files

If there is no timescale directive, the default time unit and time precision are tool specific. Using timeunit and timeprecision declarations are recommended. Refer to the IEEE Standard for SystemVerilog, Section 3.14 for details.
Slave BFM Configuration

The slave BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. It has parameters you can use to configure the widths of the address and data signals, and transaction fields to configure timeout factors, and setup and hold times, and so on.

You can change the address and data signal widths from their default settings by assigning them with new values, usually performed in the top-level module of the test bench. These new values are then passed into the slave BFM using a parameter port list of the slave BFM module. For example, the code extract below shows the slave BFM with the address and data signal widths defined in module top() and passed in to the slave_test_program parameter port list:

```verilog
module top ();

    parameter AXI4_ADDRESS_WIDTH = 24;
    parameter AXI4_RDATA_WIDTH = 16;
    parameter AXI4_WDATA_WIDTH = 16;

    slave_test_program #(AXI4_ADDRESS_WIDTH, AXI4_RDATA_WIDTH, AXI4_WDATA_WIDTH) bfm_slave(....);
```

Table 4-1 lists the parameter names for the address and data signals, and their default values.

<table>
<thead>
<tr>
<th>Signal Width Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_ADDRESS_WIDTH</td>
<td>Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32</td>
</tr>
<tr>
<td>AXI4_RDATA_WIDTH</td>
<td>Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>AXI4_WDATA_WIDTH</td>
<td>Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>index</td>
<td>Ignored for the SystemVerilog slave BFM.</td>
</tr>
<tr>
<td>READ_ACCEPTANCE_CAPABILITY</td>
<td>The maximum number of outstanding read transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
</tbody>
</table>
A slave BFM has configuration fields that you can set with the `set_config()` function to configure timeout factors, setup and hold times, and so on. You can also get the value of a configuration field via the `get_config()` function.

Table 4-2 describes the full list of configuration fields.

### Table 4-2. Slave BFM Configuration

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE_ACCEPTANCE_CAPACITY</td>
<td>The maximum number of outstanding write transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
<tr>
<td>COMBINED_ACCEPTANCE_CAPACITY</td>
<td>The maximum number of outstanding combined read and write transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
<tr>
<td>AXI4_CONFIG_SETUP_TIME</td>
<td>The setup time prior to the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_HOLD_TIME</td>
<td>The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR</td>
<td>The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_BURST_TIMEOUT_FACTOR</td>
<td>The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY</td>
<td>The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods (default 10000).</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY</td>
<td>The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods (default 10000).</td>
</tr>
</tbody>
</table>
### Table 4-2. Slave BFM Configuration (cont.)

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY</td>
<td>The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods (default 10000).</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY</td>
<td>The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods (default 10000).</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY</td>
<td>The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods (default 10000).</td>
</tr>
</tbody>
</table>

### Master Attributes

### Slave Attributes

| AXI4_CONFIG_AXI4LITE_axi4                                                              | Configures the AXI4 slave BFM to be AXI4-Lite compatible. 0 = disabled (default) 1 = enabled                                           |
| AXI4_CONFIG_SLAVE_START_ADDR                                                           | Configures the start address map for the slave.                                                                                       |
| AXI4_CONFIG_SLAVE_END_ADDR                                                             | Configures the end address map for the slave.                                                                                         |
| AXI4_CONFIG_MAX_OUTSTANDING_WR                                                         | Configures the maximum number of outstanding write requests from the master that can be processed by the slave. The slave back-pressures the master by setting the signal AWREADY=0b0 if this value is exceeded. Default = 0. |
| AXI4_CONFIG_MAX_OUTSTANDING_RD                                                          | Configures the maximum number of outstanding read requests from the master that can be processed by the slave. The slave back-pressures the master by setting the signal ARREADY=0b0 if this value is exceeded. Default = 0. |
SystemVerilog Slave BFM

Slave Assertions

Table 4-2. Slave BFM Configuration (cont.)

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE</td>
<td>Holds the number of outstanding write phases from the master that can be processed by the slave. Default = 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE</td>
<td>Holds the number of outstanding read phases to the master that can be processed by the slave. Default = 0.</td>
</tr>
</tbody>
</table>

### Error Detection

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_ENABLE_ALL_ASSERTIONS</td>
<td>Global enable/disable of all assertion checks in the BFM.</td>
</tr>
<tr>
<td></td>
<td>0 = disabled</td>
</tr>
<tr>
<td></td>
<td>1 = enabled (default)</td>
</tr>
<tr>
<td>AXI4_CONFIG_ENABLE_ASSERTION</td>
<td>Individual enable/disable of assertion check in the BFM.</td>
</tr>
<tr>
<td></td>
<td>0 = disabled</td>
</tr>
<tr>
<td></td>
<td>1 = enabled (default)</td>
</tr>
</tbody>
</table>

1. Refer to *Slave Timing and Events* for details of simulator time-steps.

**Slave Assertions**

Each slave BFM performs protocol error checking using the built-in assertions.

**Note**

The built-in BFM assertions are independent of programming language and simulator.

**Assertion Configuration**

By default, all built-in assertions are enabled in the slave AXI4-Lite BFM. To globally disable them in the slave BFM, use the `set_config()` command as the following example illustrates:

```plaintext
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS, 0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of `get_config()` and `set_config()` commands on the respective assertion. For example, to disable assertion checking for the AWADDR signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI4_CONFIG_ENABLE_ASSERTION);

// Assign the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI4_AWADDR_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector);

Note
Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the 
AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion, follow the 
above code sequence and assign the assertion within the 
AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1. For a complete listing of AXI4-Lite 
assertions, refer to “AXI4-Lite Assertions” on page 375.

SystemVerilog Slave API

This section describes the SystemVerilog Slave API.
set_config()

This function sets the configuration of the slave BFM.

Prototype

```systemverilog
function void set_config
(input axi4_config_e config_name,
 input axi4_max_bits_t config_val
);
```

Arguments

- `config_name`: Configuration name:
  - AXI4_CONFIG_SETUP_TIME
  - AXI4_CONFIG_HOLD_TIME
  - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
  - AXI4_CONFIG_AXI4LITE_axi4
  - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
  - AXI4_CONFIG_ENABLE_ASSERTION
  - AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
  - AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
  - AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
  - AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
  - AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
  - AXI4_CONFIG_SLAVE_START_ADDR
  - AXI4_CONFIG_SLAVE_END_ADDR
  - AXI4_CONFIG_MAX_OUTSTANDING_WR
  - AXI4_CONFIG_MAX_OUTSTANDING_RD
  - AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE
  - AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE

- `config_val`: See Slave BFM Configuration for descriptions and valid values.

Returns

None

Example

```systemverilog
set_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR, 1000);
```
get_config()

This function gets the configuration of the slave BFM.

Prototype

function void get_config
{
    input axi4_config_e config_name,
}

Arguments

config_name Configuration name:

- AXI4_CONFIG_SETUP_TIME
- AXI4_CONFIG_HOLD_TIME
- AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI4_CONFIG_AXI4LITE_axi4
- AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI4_CONFIG_ENABLE_ASSERTION
- AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI4_CONFIG_SLAVE_START_ADDR
- AXI4_CONFIG_SLAVE_END_ADDR
- AXI4_CONFIG_MAX_OUTSTANDING_WR
- AXI4_CONFIG_MAX_OUTSTANDING_RD
- AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE
- AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE

Returns

config_val See Slave BFM Configuration for descriptions and valid values.

Example

get_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR);
create_slave_transaction()

This nonblocking function creates a slave transaction. All transaction fields default to legal protocol values, unless previously assigned a value. It returns with the axi4_transaction record.

**Prototype**

```cpp
function automatic axi4_transaction create_write_transaction();
```

**Protocol Transaction Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Start address</td>
</tr>
<tr>
<td>prot</td>
<td>Protection:</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_SEC_DATA; (default)</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_SEC_DATA;</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_NONSEC_DATA;</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_NONSEC_DATA;</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_SEC_INST;</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_SEC_INST;</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_NONSEC_INST;</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_NONSEC_INST;</td>
</tr>
<tr>
<td>data_words</td>
<td>Data words</td>
</tr>
<tr>
<td>write_strobes</td>
<td>Write strobes:</td>
</tr>
<tr>
<td>resp</td>
<td>Burst response:</td>
</tr>
<tr>
<td></td>
<td>AXI4_OKAY;</td>
</tr>
<tr>
<td></td>
<td>AXI4_SLVERR;</td>
</tr>
<tr>
<td></td>
<td>AXI4_DECERR;</td>
</tr>
<tr>
<td>read_or_write</td>
<td>Read or write transaction flag:</td>
</tr>
<tr>
<td></td>
<td>AXI4_TRANS_READ;</td>
</tr>
<tr>
<td></td>
<td>AXI4_TRANS_WRITE</td>
</tr>
</tbody>
</table>

**Operational Transaction Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_write_strobes</td>
<td>Correction of write strobes for invalid byte lanes:</td>
</tr>
<tr>
<td></td>
<td>0 = write_strobes passed through to protocol signals.</td>
</tr>
<tr>
<td></td>
<td>1 = write_strobes auto-corrected for invalid byte lanes (default).</td>
</tr>
<tr>
<td>operation_mode</td>
<td>Operation mode:</td>
</tr>
<tr>
<td></td>
<td>AXI4_TRANSACTION_NON_BLOCKING;</td>
</tr>
<tr>
<td></td>
<td>AXI4_TRANSACTION_BLOCKING; (default)</td>
</tr>
<tr>
<td>write_data_mode</td>
<td>Write data mode:</td>
</tr>
<tr>
<td></td>
<td>AXI4_DATA_AFTER_ADDRESS;</td>
</tr>
<tr>
<td></td>
<td>The master first drives the address phase and, after it completes,</td>
</tr>
<tr>
<td></td>
<td>it drives the corresponding data phases. The master waits for</td>
</tr>
<tr>
<td></td>
<td>AWREADY before asserting WVALID. For a slave designed to wait for WVALID</td>
</tr>
<tr>
<td></td>
<td>before asserting AWREADY, using this mode may cause a deadlock situation.</td>
</tr>
<tr>
<td></td>
<td>This mode will force the data transfer to start after the address transfer</td>
</tr>
<tr>
<td></td>
<td>completes; however, it is recommended that you use the **DATA_WITH_ADDRESS</td>
</tr>
<tr>
<td></td>
<td>along with a data_valid_delay setting instead to avoid the possible</td>
</tr>
<tr>
<td></td>
<td>deadlock situation.</td>
</tr>
<tr>
<td></td>
<td>AXI4_DATA_WITH_ADDRESS; (default)</td>
</tr>
<tr>
<td></td>
<td>The master drives the address and the data phase in a nonblocking process; banned assertion of WVALID on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay.</td>
</tr>
<tr>
<td>address_valid_delay</td>
<td>Address channel ARVALID/AWVALID delay measured in ACLK cycles (default = 0).</td>
</tr>
</tbody>
</table>
SystemVerilog Slave BFM

create_slave_transaction()

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_valid_delay</td>
<td>Write data channel WVALID delay array measured in ACLK cycles for this transaction (default = 0 for all elements).</td>
</tr>
<tr>
<td>write_response</td>
<td>Write response channel BREADY delay measured in ACLK cycles for this transaction (default = 0).</td>
</tr>
<tr>
<td>transaction_done</td>
<td>Write transaction done flag for this transaction.</td>
</tr>
</tbody>
</table>

Returns
The axi4_transaction record.

Example

```verilog
// Create a slave transaction.
trans = bfm.create_slave_transaction();
```
execute_read_data_phase()

This task executes a read data phase (beat) previously created by the `create_slave_transaction()` task. This phase can be blocking (default) or nonblocking, as defined by the transaction record `operation_mode` field.

The `execute_read_data_phase()` task sets the RVALID protocol signal at the appropriate time defined by the transaction record `data_valid_delay` field and sets the `transaction_done` field to 1 to indicate the whole read transaction has completed.

Prototype

```text
task automatic execute_read_data_phase
  (axi4_transaction trans
   int Index = 0 // Optional)
);
```

Arguments

- `trans`: The `axi4_transaction` record.
- `index`: Data phase (beat) number.
  Note: '0' for AXI4-Lite

Returns

None

Example

```text
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a slave transaction and assign it to the local // read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Execute the read data phase for the read_trans transaction.
bfm.execute_read_data_phase(read_trans, 0); //Note: array element 0
```
execute_write_response_phase()

This task executes a write phase previously created by the `create_slave_transaction()` task. This phase can be blocking (default) or nonblocking, as defined by the transaction record `operation_mode` field.

It sets the BVALID protocol signal at the appropriate time defined by the transaction record `write_response_valid_delay` field and sets the `transaction_done` field to 1 on completion of the phase to indicate the whole transaction has completed.

**Prototype**

```plaintext
task automatic execute_write_response_phase
(_transaction trans);
```

**Arguments**

- `trans` The `_transaction` record.

**Returns**

None

**Example**

```plaintext
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a slave transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_slave_transaction();

....

// Execute the write response phase for the write_trans transaction.
bfm.execute_write_response_phase(write_trans);
```
get_write_addr_phase()

**Note**

This blocking task gets a write address phase previously created by the `create_slave_transaction()` function.

**Prototype**

```plaintext
task automatic get_write_addr_phase
    (axi4_transaction trans);
```

**Arguments**

- `trans` The `axi4_transaction` record.

**Returns**

- None

**Example**

```plaintext
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a slave transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_slave_transaction();
....

// Get the write address phase of the write_trans transaction.
bfm.get_write_addr_phase(write_trans);
```
get_read_addr_phase()

This blocking task gets a read address phase previously created by the create_slave_transaction() function.

Example

Prototype

task automatic get_read_addr_phase

(  
  axi4_transaction trans
  )

Arguments

trans The axi4_transaction record.

Returns

None

// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a slave transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_slave_transaction();

....

// Get the read address phase of the read_trans transaction.
bfm.get_read_addr_phase(read_trans);
get_write_data_phase()

This blocking task gets a write data phase previously created by the \texttt{create_slave_transaction()} function.

The \texttt{get_write_data_phase()} sets the WREADY protocol signal at the appropriate time defined by the \texttt{data_ready_delay} field.

**Prototype**

```systemverilog
task automatic get_write_data_phase
    (axi4_transaction trans
    int index = 0, // Optional
    output bit last
    );
```

**Arguments**

- \texttt{trans} The \texttt{axi4_transaction} record.
- \texttt{index} (Optional) Data phase (beat) number. Note: '0' for AXI4-Lite

**Returns**

- \texttt{last} Flag to indicate that this data phase is the last in the burst.

**Example**

```systemverilog
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a slave transaction and assign it to the local // write_trans variable.
write_trans = bfm.create_slave_transaction(0);

....

// Get the write data phase for the write_trans transaction.
bfm.get_write_data_phase(write_trans, 0, last); //Note: array element 0
```
get_read_addr_cycle()

This blocking task waits until the read address channel ARVALID signal is asserted.

Prototype: task automatic get_read_addr_cycle();
Arguments: None
Returns: None

Example

// Waits until the read address channel ARVALID signal is asserted.
bfm.get_read_addr_cycle();
execute_read_addr_ready()

This task executes a read address ready by placing the \textit{ready} argument value onto the ARREADY signal. It will block for one ACLK period.

**Prototype**

```systemverilog
task automatic execute_read_addr_ready
  (  
    bit ready
  );
```

**Arguments**

- **ready**: The value to be placed onto the ARREADY signal.

**Returns**

- None

**Example**

```systemverilog
// Assert and deassert the ARREADY signal
forever begin
  bfm.execute_read_addr_ready(1'b0);

  bfm.wait_on(AXI4_CLOCK_POSEDGE);
  bfm.wait_on(AXI4_CLOCK_POSEDGE);

  bfm.execute_read_addr_ready(1'b1);

  bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
```
get_read_data_ready()

This blocking task returns the read data ready value of the RREADY signal using the \textit{ready} argument. It will block for one ACLK period.

**Prototype**  
\begin{verbatim}
  task automatic get_read_data_ready(
    output bit ready
  );
\end{verbatim}

**Arguments**  
ready The value of the RREADY signal.

**Returns**  
ready

**Example**

\begin{verbatim}
// Get the value of the RREADY signal
bfm.get_read_data_ready();
\end{verbatim}
get_write_addr_cycle()

This blocking task waits until the write address channel AWVALID signal is asserted.

Prototype    task automatic get_write_addr_cycle();
Arguments     None
Returns       None

Example

// Wait for a single write address cycle
bfm.get_write_addr_cycle();
execute_write_addr_ready()

This task executes a write address ready by placing the ready argument value onto the AWREADY signal. It will block for one ACLK period.

Prototype task automatic execute_write_addr_ready
  (    bit ready
  );
Arguments ready The value to be placed onto the AWREADY signal
Returns None

Example

// Assert and deassert the AWREADY signal
forever begin
  bfm.execute_write_addr_ready(1'b0);
  bfm.wait_on(AXI4_CLOCK_POSEDGE);
  bfm.wait_on(AXI4_CLOCK_POSEDGE);
  bfm.execute_write_addr_ready(1'b1);
  bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
get_write_data_cycle()

This blocking task waits for a single write data cycle for which the WVALID signal is asserted. It will block for one ACLK period.

**Prototype**

```
task automatic get_write_data_cycle();
```

**Arguments**

None

**Returns**

None

**Example**

```
// Wait for a single write data cycle
bfm.get_write_data_cycle();
```
execute_write_data_ready()

This task executes a write data ready by placing the ready argument value onto the WREADY signal. It will block for one ACLK period.

Prototype

```
task automatic execute_write_data_ready
  (  
    bit ready
  );
```

Arguments

- ready: The value to be placed onto the WREADY signal

Returns

None

Example

```
// Assert and deassert the WREADY signal
forever begin
  bfm.execute_write_data_ready(1'b0);

  bfm.wait_on(AXI4_CLOCK_POSEDGE);
  bfm.wait_on(AXI4_CLOCK_POSEDGE);

  bfm.execute_write_data_ready(1'b1);

  bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
```
get_write_resp_ready()

This blocking task returns the write response ready value of the BREADY signal using the ready argument. It will block for one ACLK period.

Prototype

```plaintext
task automatic get_write_resp_ready
  (output bit ready);
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready</td>
<td>The value of the BREADY signal.</td>
</tr>
</tbody>
</table>

Returns

```plaintext
readyt
```

Example

```plaintext
// Get the value of the BREADY signal
bfm.get_write_resp_ready();
```
wait_on()

This blocking task waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

**Prototype**

```vhdl
task automatic wait_on
(
    axi4_wait_e phase,
    input int count = 1 //Optional
);
```

**Arguments**

- **phase**
  - AXI4_CLOCK_POSEDGE
  - AXI4_CLOCK_NEGEDGE
  - AXI4_CLOCK_ANYEDGE
  - AXI4_CLOCK_0_TO_1
  - AXI4_CLOCK_1_TO_0
  - AXI4_RESET_POSEDGE
  - AXI4_RESET_NEGEDGE
  - AXI4_RESET_ANYEDGE
  - AXI4_RESET_0_TO_1
  - AXI4_RESET_1_TO_0

- **count** (Optional) Wait for a number of events to occur set by *count*. (default = 1)

**Returns**

None

**Example**

```vhdl
bfm.wait_on(AXI4_RESET_POSEDGE);
bfm.wait_on(AXI4_CLOCK_POSEDGE,10);
```
Helper Functions

AMBA AXI protocols typically provide a start address only in a transaction, with the following addresses for each byte of a data beat calculated. Helper functions provide you with a simple interface to set and get address/data values.

get_write_addr_data()

This nonblocking function returns the actual address $addr$ and $data$ of a particular byte in a write data beat. It also returns the maximum number of bytes ($dynamic_size$) in the write data phase (beat). It is used in a slave test program as a helper function to store a byte of data at a particular address in the slave memory. If the corresponding $index$ does not exist, then this function returns $false$; otherwise, it returns $true$.

Prototype

```plaintext
function bit get_write_addr_data
(
    input axi4_transaction trans,
    input int index = 0,
    output bit [((AXI4_ADDRESS_WIDTH) - 1): 0] addr[],
    output bit [7:0] data[]
);
```

Arguments

- **trans**: The `axi4_transaction` record.
- **index**: Data words array element number. Note: ‘0’ for AXI4-Lite

Returns

- **addr**: Write address.
- **data**: Write data byte.
- **bit**: Flag to indicate existence of $data$;
  - 0 = nonexistent.
  - 1 = exists.

Example

```plaintext
bfm.get_write_addr_data(write_trans, 0, addr, data);
```
get_read_addr()

This nonblocking function returns the address \( \textit{addr} \) of a particular byte in a read transaction. It is used in a slave test program as a helper function to return the address of a data byte in the slave memory. If the corresponding \( \textit{index} \) does not exist, then this function returns \textit{false}; otherwise, it returns \textit{true}.

**Prototype**

```verilog
function bit get_read_addr

\{
    input axi4_transaction trans,
    input int index = 0,
    output bit [((AXI4_ADDRESS_WIDTH) - 1) : 0] addr[
\}
```

**Arguments**

- **trans**
  
  The \textit{axi4\_transaction} record.

- **index**

  Array element number. Note: ‘0’ for AXI4-Lite

- **addr**

  Read address array

**Returns**

- **bit**

  Flag to indicate existence of data;

  \[0\] = nonexistent.

  \[1\] = exists.

**Example**

```verilog
bfm.get_read_addr(read_trans, 0, addr);
```
set_read_data()

This nonblocking function sets a read data in the *axi4_transaction* record *data_words* field. It is used in a slave test program as a helper function to read from the slave memory given the address *addr*, data beat *index*, and the read *data* arguments.

Prototype

```SystemVerilog
function bit set_read_data
    input axi4_transaction trans,
    input int index = 0,
    input bit [((AXI4_ADDRESS_WIDTH) - 1) : 0] addr[],
    input bit [7:0] data[];

Arguments

- **trans**: The *axi4_transaction* record.
- **index**: (Optional) Data byte array element number. Note: ‘0’ for AXI4-Lite
- **addr**: Read address.
- **data**: Read data byte.

Returns

- None

Example

```SystemVerilog
bfm.set_read_data(read_trans, 0, addr, data);
```
SystemVerilog Slave BFM

`set_read_data()`
This chapter describes the SystemVerilog monitor BFM. Each BFM has an API that contains tasks and functions to configure the BFM and to access the dynamic Transaction Record during the lifetime of a transaction.

### Inline Monitor Connection

The connection of a monitor BFM to a test environment differs from that of a master and slave BFM. It is wrapped in an inline monitor interface and connected inline between a master and slave, as shown in Figure 5-1. It has separate master and slave ports and monitors protocol traffic between a master and slave. The monitor itself then has access to all the facilities provided by the monitor BFM.

![Figure 5-1. Inline Monitor Connection Diagram](image)

### Monitor BFM Protocol Support

The AXI4-Lite monitor BFM supports the AMBA AXI4 protocol with restrictions described in “Protocol Restrictions” on page 17.

### Monitor Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI Protocol Specification chapter, which you can use to reference details of the following monitor BFM API timing and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the monitor BFM does not contain any timescale,
Monitor BFM Configuration

The monitor BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. It has parameters you can use to configure the widths of the address and data signals, and transaction fields to configure timeout factors, setup and hold times, and so on.

You can change the address and data signals widths from their default settings by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the monitor BFM via a parameter port list of the monitor BFM module. For example, the code extract below shows the monitor BFM with the address and data signal widths defined in `module top()` and passed in to the `monitor_test_program` parameter port list:

```verilog
module top();
    parameter AXI4_ADDRESS_WIDTH = 24;
    parameter AXI4_RDATA_WIDTH = 16;
    parameter AXI4_WDATA_WIDTH = 16;
    monitor_test_program #(AXI4_ADDRESS_WIDTH, AXI4_RDATA_WIDTH, AXI4_WDATA_WIDTH) bfm_monitor(.);
```

Table 5-1 lists the parameter names for the address and data signals, and their default values.

<table>
<thead>
<tr>
<th>Signal Width Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_ADDRESS_WIDTH</td>
<td>Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.</td>
</tr>
</tbody>
</table>
A monitor BFM has configuration fields that you can set via the `set_config()` function to configure variables such as timeout factors and setup and hold times. You can also get the value of a configuration field via the `get_config()` function. Table 5-2 describes the full list of configuration fields.

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timing Variables</strong></td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_SETUP_TIME</td>
<td>The setup time prior to the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_HOLD_TIME</td>
<td>The hold time after the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR</td>
<td>The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.</td>
</tr>
</tbody>
</table>
Table 5-2. AXI Monitor BFM Configuration (cont.)

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_BURST_TIMEOUT_FACTOR</td>
<td>The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY</td>
<td>The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY</td>
<td>The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY</td>
<td>The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY</td>
<td>The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY</td>
<td>The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods. Default: 10000.</td>
</tr>
</tbody>
</table>

**Slave Attributes**

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_SLAVE_START_ADDR</td>
<td>Configures the start address map for the slave.</td>
</tr>
<tr>
<td>AXI4_CONFIG_SLAVE_END_ADDR</td>
<td>Configures the end address map for the slave.</td>
</tr>
</tbody>
</table>

**Monitor Attributes**

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_AXI4LITE_axi4</td>
<td>Configures the AXI4 monitor BFM to be AXI4-Lite compatible. 0 = disabled (default) 1 = enabled</td>
</tr>
</tbody>
</table>

**Error Detection**

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_ENABLE_ALL_ASSERTIONS</td>
<td>Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)</td>
</tr>
</tbody>
</table>
Monitor Assertions

Each monitor BFM performs protocol error checking using built-in assertions.

**Note**
The built-in BFM assertions are independent of programming language and simulator.

Assertion Configuration

By default, all built-in assertions are enabled in the monitor AXI4-Lite BFM. To globally disable them in the monitor BFM, use the `set_config()` command as the following example illustrates:

```systemverilog
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS, 0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of `get_config()` and `set_config()` commands on the respective assertion. For example, to disable assertion checking for the AWADDR signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```systemverilog
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI4_CONFIG_ENABLE_ASSERTION);

// Assign the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI4_AWADDR_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector);
```

**Note**
Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.
To re-enable the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion within the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI4-Lite assertions, refer to “AXI4-Lite Assertions” on page 375.

**SystemVerilog Monitor API**

This section describes the SystemVerilog Monitor API.

**set_config()**

This function sets the configuration of the monitor BFM.

**Prototype**

```plaintext
function void set_config
  (input axi4_config_e config_name,
   input axi4_max_bits_t config_val
  );
```

**Arguments**

- `config_name`  
  Configuration name:
  - AXI4_CONFIG_SETUP_TIME
  - AXI4_CONFIG_HOLD_TIME
  - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
  - AXI4_CONFIG_AXI4LITE_axi4
  - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
  - AXI4_CONFIG_ENABLE_ASSERTION
  - AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
  - AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
  - AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
  - AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
  - AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
  - AXI4_CONFIG_SLAVE_START_ADDR
  - AXI4_CONFIG_SLAVE_END_ADDR

- `config_val`  
  See “Monitor BFM Configuration” on page 92 for descriptions and valid values.

**Returns**

None

**Example**

```plaintext
set_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR, 1000);
```
get_config()

This function gets the configuration of the monitor BFM.

Prototype

```plaintext
function void get_config
(input axi4_config_e config_name,
);
```

Arguments

<table>
<thead>
<tr>
<th>config_name</th>
<th>Configuration name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_SETUP_TIME</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_HOLD_TIME</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_AXI4LITE_axi4</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_ENABLE_ALL_ASSERTIONS</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_ENABLE_ASSERTION</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_SLAVE_START_ADDR</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_SLAVE_END_ADDR</td>
<td></td>
</tr>
</tbody>
</table>

Returns

| config_val | See “Monitor BFM Configuration” on page 92 for descriptions and valid values. |

Example

```plaintext
get_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR);
```
create_monitor_transaction()

This nonblocking function creates a monitor transaction. All transaction fields default to legal protocol values, unless previously assigned a value. It returns with the axi4_transaction record.

**Prototype**

```plaintext
function automatic axi4_transaction create_monitor_transaction();
```

**Protocol**

**Transaction Fields**

- **addr**: Start address

- **prot**: Protection:
  - AXI4_NORM_SEC_DATA; (default)
  - AXI4_PRIV_SEC_DATA;
  - AXI4_NORM_NONSEC_DATA;
  - AXI4_PRIV_NONSEC_DATA;
  - AXI4_NORM_SEC_INST;
  - AXI4_PRIV_SEC_INST;
  - AXI4_NORM_NONSEC_INST;
  - AXI4_PRIV_NONSEC_INST;

- **data_words**: Data words array.

- **write_strobes**: Write strobes:
  - Each strobe 0 or 1.

- **resp**: Burst response:
  - AXI4_OKAY;
  - AXI4_SLVERR;
  - AXI4_DECERR;

**Operational Transaction Fields**

- **gen_write_strobes**: Generate write strobes flag:
  - 0 = user supplied write strobes.
  - 1 = auto-generated write strobes (default).

- **operation_mode**: Operation mode:
  - AXI4_TRANSACTION_NON_BLOCKING;
  - AXI4_TRANSACTION_BLOCKING; (default)

- **write_data_mode**: Write data mode:
  - AXI4_DATA_AFTER_ADDRESS;
    - The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a data_valid_delay setting instead to avoid the possible deadlock situation.
  - AXI4_DATA_WITH_ADDRESS; (default)
    - The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts RWVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay.

- **address_valid_delay**: Address channel AWVALID delay measured in ACLK cycles for this transaction (default = 0).

- **data_valid_delay**: Write data channel WVALID delay array measured in ACLK cycles for this transaction (default = 0 for all elements).
SystemVerilog Monitor BFM

create_monitor_transaction()

write_response  Write response channel BREADY delay measured in ACLK cycles for
_ready_delay    this transaction (default = 0).
transaction_    Write transaction done flag for this transaction.
done

Returns
The axi4_transaction record

Example

// Create a monitor transaction
trans = bfm.create_monitor_transaction();
get_rw_transaction()

This blocking task gets a complete read or write transaction previously created by the `create_monitor_transaction()` function.

It updates the `axi4_transaction` record for the complete transaction.

Prototype

```plaintext
task automatic get_rw_transaction
  (axi4_transaction trans)
```

Arguments

- `trans` The `axi4_transaction` record.

Returns

None

Example

```plaintext
// Declare a local variable to hold the transaction record.
axi4_transaction monitor_trans;

// Create a monitor transaction and assign it to the local
// monitor_trans variable.
monitor_trans = bfm.create_monitor_transaction();

....

// Get the complete monitor_trans transaction.
bfm.get_rw_transaction(monitor_trans);
```
get_write_addr_phase()

This blocking task gets a write address phase previously created by the create_monitor_transaction() function.

Prototype

```
task automatic get_write_addr_phase
    (axi4_transaction trans);
```

Arguments

- **trans**: The axi4_transaction record.

Returns

None

Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a monitor transaction and assign it to the local // write_trans variable.
write_trans = bfm.create_monitor_transaction();

....

// Get the write address phase of the write_trans transaction.
bfm.get_write_addr_phase(write_trans);
```
get_read_addr_phase()

This blocking task gets a read address phase previously created by the create_monitor_transaction() function.

Prototype

task automatic get_read_addr_phase

axi4_transaction trans

)

Arguments

trans The axi4_transaction record.

Returns

None

Example

// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a monitor transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_monitor_transaction();

....

// Get the read address phase of the read_trans transaction.
bfm.get_read_addr_phase(read_trans);
**get_read_data_phase()**

This blocking task gets a read data phase previously created by the `create_monitor_transaction()` function. The `get_read_data_phase()` sets the `transaction_done` field to 1 to indicate the whole read transaction has completed.

**Prototype**

```systemverilog
task automatic get_read_data_phase
    (axi4_transaction trans
     int Index = 0 // Optional
    );
```

**Arguments**

- `trans` (The `axi4_transaction` record.  
  Note: '0' for AXI4-Lite
- `index` (Optional) Data phase (beat) number.

**Returns**

None

**Example**

```systemverilog
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a monitor transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_monitor_transaction();

....

// Get the read data phase for the read_trans transaction.
bfm.get_read_data_phase(read_trans, 0); //Note: array element 0
```
get_write_data_phase()

This blocking task gets a write data phase previously created by the create_monitor_transaction() function.

Prototype

```plaintext
task automatic get_write_data_phase
    (axi4_transaction trans
     int index = 0, // Optional
     output bit last
    );
```

Arguments

- `trans`: The `axi4_transaction` record.
- `index`: (Optional) Data phase (beat) number.
- `last`: Flag to indicate that this data phase is the last in the burst.

Returns: None

Example

```plaintext
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a monitor transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_monitor_transaction();
....

// Get the write data phase for the write_trans transaction.
bfm.get_write_data_phase(write_trans, 0, last); //Note: array element 0
```
get_write_response_phase

This blocking task gets a write response phase previously created by the 
create_monitor_transaction() task.

It sets the transaction_done field to 1 when the transaction completes to indicate the whole 
transaction is complete

Prototype

task automatic get_write_response_phase
    ( 
        axi4_transaction trans
    );

Arguments

trans The axi4_transaction record.

Returns

None

Example

// Declare a local variable to hold the transaction record. 
axi4_transaction write_trans;

// Create a monitor transaction and assign it to the local 
// write_trans variable. 
write_trans = bfm.create_monitor_transaction();

....

// Get the write response phase of the write_trans transaction. 
bfm.get_write_response_phase(write_trans);
get_read_addr_ready()

This blocking task returns the read address ready value of the ARREADY signal using the ready argument. It will block for one ACLK period.

Example

    // Get the ARREADY signal value
    bfm.get_read_addr_ready();
**get_read_data_ready()**

This blocking task returns the read data ready value of the RREADY signal using the ready argument. It will block for one ACLK period.

**Example**

```vhdl
// Get the value of the RREADY signal
bfm.get_read_data_ready();
```
get_write_addr_ready()

This blocking task returns the write address ready value of the AWREADY signal using the \texttt{ready} argument. It will block for one ACLK period.

\textbf{Prototype} \hspace{1em} \texttt{task automatic get_write_addr_ready} \\
\hspace{2em} \texttt{(output bit ready)} \\
\textbf{Arguments} \hspace{1em} \texttt{ready} \hspace{1em} The value of the AWREADY signal. \\
\textbf{Returns} \hspace{1em} None

\textbf{Example}

\begin{verbatim}
// Get the value of the AWREADY signal
bfm.get_write_addr_ready();
\end{verbatim}
get_write_data_ready()

This blocking task returns the write data ready value of the WREADY signal using the `ready` argument. It will block for one ACLK period.

**Prototype**

```verilog
task automatic get_write_data_ready
    (output bit ready);
```

**Arguments**

- `ready` The value of the WREADY signal.

**Returns**

- None

**Example**

```verilog
// Get the value of the WREADY signal bfm.
get_write_data_ready();
```
get_write_resp_ready()

This blocking task returns the write response ready value of the BREADY signal using the ready argument. It will block for one ACLK period.

**Prototype**

```vhdl
task automatic get_write_resp_ready
(
    output bit ready
);
```

**Arguments**

- `ready` The value of the BREADY signal.

**Returns**

None

**Example**

```vhdl
// Get the value of the BREADY signal
bfm.get_write_resp_ready();
```
**wait_on()**

This blocking task waits for an event(s) on the ACLK or ARESETh signals to occur before proceeding. An optional `count` argument waits for the number of events equal to `count`.

**Prototype**

```verbatim
task automatic wait_on
  (axi4_wait_e phase,
   input int count = 1 //Optional)
);
```

**Arguments**

- `phase` Wait for:
  - AXI4_CLOCK_POSEDGE
  - AXI4_CLOCK_NEGEDGE
  - AXI4_CLOCK_ANYEDGE
  - AXI4_CLOCK_0_TO_1
  - AXI4_CLOCK_1_TO_0
  - AXI4_RESET_POSEDGE
  - AXI4_RESET_NEGEDGE
  - AXI4_RESET_ANYEDGE
  - AXI4_RESET_0_TO_1
  - AXI4_RESET_1_TO_0
- `count` (Optional) Wait for a number of events to occur set by `count`. (default = 1)

**Returns**

None

**Example**

```verbatim
bfm.wait_on(AXI4_RESET_POSEDGE);
bfm.wait_on(AXI4_CLOCK_POSEDGE,10);
```
AMBA AXI protocols typically provide a start address only in a transaction, with the following addresses for each byte of a data beat calculated using the size, length, and type transaction fields. Helper functions provide you with a simple interface to set and get address/data values.

**get_write_addr_data()**

This nonblocking function returns the actual address `addr` and `data` of a particular byte in a write data beat. It is used in a monitor test program as a helper function to store a byte of data at a particular address in the monitor memory. If the corresponding `index` does not exist, then this function returns `false`; otherwise, it returns `true`.

**Prototype**

```plaintext
function bit get_write_addr_data
{
    input axi4_transaction trans,
    input int index = 0,
    output bit [((AXI4_ADDRESS_WIDTH) - 1) : 0] addr[],
    output bit [7:0] data[]
};
```

**Arguments**

- `trans`: The `axi4_transaction` record.
- `index`: Array element number. Note: ‘0’ for AXI4-Lite
- `addr`: Write address array
- `data`: Write data array

**Returns**

`bit`: Flag to indicate existence of `index` array element;
- `0` = array element non-existent.
- `1` = array element exists.

**Example**

```plaintext
bfm.get_write_addr_data(write_trans, 0, addr, data);
```
get_read_addr()

This nonblocking function returns the actual address addr of a particular index in a read transaction. It is used in a monitor test program as a helper function to return the address of a byte of data in the monitor memory. If the corresponding index does not exist, then this function returns false; otherwise, it returns true.

Prototype

```systemverilog
function bit get_read_addr
{
    input axi4_transaction trans,
    input int index = 0,
    output bit [((AXI4_ADDRESS_WIDTH) - 1) : 0]
    addr[]
};
```

Arguments

- trans: The axi4_transaction record.
- index: Array element number. Note: '0' for AXI4-Lite
- addr: Read address array

Returns

- bit: Flag to indicate existence of index array element;
  0 = array element non-existent.
  1 = array element exists.

Example

```systemverilog
bfm.get_read_addr(read_trans, 0, addr);
```
**set_read_data()**

This nonblocking function sets the read data in the `axi4_transaction` record `data_words` field. It is used in a monitor test program as a helper function to read from the monitor memory given the address `addr`, data beat `index`, and the read `data` arguments.

**Prototype**

```verbatim
function bit set_read_addr_data
    input axi4_transaction trans,
    input int index = 0,
    input bit [((AXI4_ADDRESS_WIDTH) - 1) : 0] addr[],
    input bit [7:0] data[]
);
```

**Arguments**

- `trans` The `axi4_transaction` record.
- `index` (Optional) Array element number. Note: ‘0’ for AXI4-Lite
- `addr` Read address array
- `data` Read data array

**Returns**

None

**Example**

```verbatim
bfm.set_read_data(read_trans, 0, addr, data);
```
Chapter 6
SystemVerilog Tutorials

This chapter discusses how to use the Mentor Verification IP Altera Edition master and slave BFM models to verify slave and master DUT components.

In the Verifying a Slave DUT tutorial, the slave is an on-chip RAM model that is verified using a master BFM and test program. In the Verifying a Master DUT tutorial, the master issues simple write and read transactions that are verified using a slave BFM and test program.

Following this top-level discussion of how you verify a master and a slave component using the Mentor Verification IP Altera Edition is a brief example of how to run Qsys, the powerful system integration tool in Quartus® II software. This procedure shows you how to use Qsys to create a top-level DUT environment. For more details on this example, refer to “Getting Started with Qsys and the BFMs” on page 359.

Note
Parameters to configure any optional signals, master BFM transaction issuing and slave BFM acceptance capability, are set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362 for details of the Qsys Parameter Editor.

Verifying a Slave DUT

A slave DUT component is connected to a master BFM at the signal-level. A master test program, written at the transaction-level, generates stimulus using the master BFM to verify the slave DUT. Figure 6-1 illustrates a typical top-level test bench environment.

Figure 6-1. Slave DUT Top-Level Test Bench Environment

Top-level file
In this example, the master test program also compares the written data with that read back from the slave DUT, reporting the result of the comparison.

A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

**BFM Master Test Program**

A master test program using the master BFM API is capable of creating a wide range of stimulus scenarios to verify a slave DUT. However, this tutorial restricts the master BFM stimulus to write transactions followed by read transactions to the same address, and then compares the read data with the previously written data.

---

**Note**

For a complete code example of this master test program, refer to the installed Mentor Verification IP Altera Edition kit in the following directory:

```
<install_dir>\mentor_vip_ae\axi4lite\qsys-examples\ex1_back_to_back_sv\master_test_program.sv
```

---

The master test program contains the following elements:

- **A Configuration and Initialization** that creates and executes read and write transactions.
- **Tasks** `handle_write_resp_ready()` and `handle_read_data_ready()` to handle the delay of the write response channel BREADY signal and the read data channel RREADY signals, respectively.
- **Variables** `m_wr_resp_phase_ready_delay` and `m_rd_data_phase_ready_delay` to set the delay of the BREADY and RREADY signals
- **A** `master_ready_delay_mode` variable to configure the behavior of the handshake signals *VALID to *READY delay.

The following sections describe the main tasks and variables.

**master_ready_delay_mode**

The `master_ready_delay_mode` variable holds the configuration that defines the starting point of any delay applied to the RREADY and BREADY signals. It can be configured to the enumerated type values of AXI4_VALID2READY (default) or AXI4_TRANS2READY.

The default configuration (`master_ready_delay_mode = AXI4_VALID2READY`) corresponds to the delay measured from the positive edge of ACLK when *VALID is asserted. Figure 6-2 shows how to achieve a *VALID before *READY handshake, respectively.
The nondefault configuration (master_ready_delay_mode = AXI4_TRANS2READY) corresponds to the delay measured from the completion of a previous transaction phase (*VALID and *READY both asserted). Figure 6-3 shows how to achieve a *READY before *VALID handshake.

**Example 6-1** shows the configuration of the master_ready_delay_mode to its default value.

**Example 6-1. master_ready_delay_mode**

```verilog
// Enum type for master ready delay mode
// AXI4_VALID2READY - Ready delay for a phase will be applied from
// start of phase (Means from when VALID is asserted).
// AXI4_TRANS2READY - Ready delay will be applied from the end of
// previous phase. This might result in ready before valid.
typedef enum bit
{
    AXI4_VALID2READY = 1'b0,
    AXI4_TRANS2READY = 1'b1
} axi4_master_ready_delay_mode_e;
```
// Master ready delay mode selection : default it is VALID2READY
axi4_master_ready_delay_mode_e master_ready_delay_mode = AXI4_VALID2READY;

m_wr_resp_phase_ready_delay

The m_wr_resp_phase_ready_delay variable holds the BREADY signal delay. The delay value extends the length of the write response phase by a number of ACLK cycles. The starting point of the delay is determined by the master_ready_delay_mode variable configuration.

Example 6-2 shows the AWREADY signal delayed by two ACLK cycles. You can edit this variable to change the AWREADY signal delay.

Example 6-2. m_wr_resp_phase_ready_delay

// Variable: m_wr_resp_phase_ready_delay
int m_wr_resp_phase_ready_delay = 2;

m_rd_data_phase_ready_delay

The m_rd_data_phase_ready_delay variable holds the RREADY signal delay. The delay value extends the length of each read data phase (beat) by a number of ACLK cycles. The starting point of the delay is determined by the master_ready_delay_mode variable configuration.

Example 6-3 shows the RREADY signal delayed by two ACLK cycles. You can edit this variable to change the RREADY signal delay.

Example 6-3. m_rd_data_phase_ready_delay

// Variable: m_rd_data_phase_ready_delay
int m_rd_data_phase_ready_delay = 2;

Configuration and Initialization

In an initial block, the master test program defines the transaction variable trans of type axi4_transaction, which hold the record of each transaction during its lifetime, as shown in Example 6-4. The initial wait for the ARESETn signal to be deactivated, followed by a positive ACLK edge, satisfies the protocol requirement detailed in Section A3.1.2 of the AXI Protocol Specification.

Example 6-4. Configuration and Initialization

initial begin
    axi4_transaction trans;
    bit [AXI4_WDATA_WIDTH-1:0] data_word;

    ****************************
    ** Initialization **
Create and Execute Write Transactions

To generate AXI4 protocol traffic, the master test program must create the transaction `trans` before executing it. The code excerpt in Example 6-5 calls the `create_write_transaction()` function, providing only the start address argument of the transaction.

This example has an AXI4 data bus width of 32 bits; therefore, a single beat of data conveys 4 bytes across the data bus. The call to the `set_data_words()` function sets the `data_words` transaction field with the value of 1 on byte lane 1, resulting in a value of 32'h0000_0100. However, the AXI4 protocol permits narrow transfers with the use of the write strobes signal WSTRB to indicate which byte lane contains valid write data, and therefore indicates to the slave DUT which data byte lane will be written into memory. Similarly, you can call the `set_write_strobes()` function to set the `write_strobes` transaction field with the value of 4'b0010, indicating that only valid data is being transferred on byte lane 1. The write transaction `trans` then executes on the protocol signals by calling the `execute_transaction()` function.

All other write transaction fields default to legal protocol values (see `create_write_transaction()` for details).

Example 6-5. Create and Execute Write Transactions

```verbatim
/ ***********************
** Traffic generation: **
/ ***********************
// 4 x Writes
// Write data value 1 on byte lanes 1 to address 1.
trans = bfm.create_write_transaction(1);
trans.set_data_words(32'h0000_0100);
trans.set_write_strobes(4'b0010, 0);
$display (@ $t, master_test_program: Writing data (1) to address (1),
$time);

// By default it will run in Blocking mode
bfm.execute_transaction(trans);
```

In the complete master test program, three subsequent write transactions are created and executed in a similar manner to Example 6-5.

Create and Execute Read Transactions

The code excerpt in Example 6-6 reads the data that has been previously written into the slave memory. The master test program first creates a read transaction `trans` by calling the `create_read_transaction()` function, providing only the start address argument.
The read data is obtained by calling the `get_data_words` function to get the `data_words` transaction field value. The result of the read data is compared with the expected data, and a message displays the transcript.

**Example 6-6. Create and Execute Read Transactions**

```systemverilog
// Read data from address 1.
trans = bfm.create_read_transaction(1);

bfm.execute_transaction(trans);
if (trans.get_data_words == 32'h0000_0100)
    $display (@ %t, master_test_program: Read correct data (1) at
             address (1"h01), $time);
else
    $display (@ %t master_test_program: Error: Expected data (1) at
             address 1, but got %d", $time, trans.get_data_words);
```

In the complete master test program, three subsequent read transactions are created and executed in a similar manner to Example 6-6.

**handle_write_resp_ready()**

The `handle_write_resp_ready()` task handles the BREADY signal for the write response channel. In a `forever` loop, it delays the assertion of the BREADY signal based on the settings of the `master_ready_delay_mode` and `m_wr_resp_phase_ready_delay` as shown in Example 6-7.

If the `master_delay_ready_mode = AXI4_VALID2READY`, then the BREADY signal is immediately deasserted using the nonblocking call to the `execute_write_resp_ready()` task and waits for a write channel response phase to occur with a call to the blocking `get_write_response_cycle()` task. A received write response phase indicates that the BVALID signal has been asserted, triggering the starting point for the delay of the BREADY signal by the number of ACLK cycles defined by `m_wr_resp_phase_ready_delay`. After the delay, another call to the `execute_write_resp_ready()` task to assert the BREADY signal completes the BREADY handling. The `seen_valid_ready` flag is set to indicate the end of a response phase when both BVALID and BREADY are asserted, and the completion of the write transaction.

If the `master_delay_ready_mode = AXI4_TRANS2READY`, then a check of the `seen_valid_ready` flag is performed to indicate that a previous write transaction has completed. If a write transaction is still active (indicated by either BVALID or BREADY not asserted), then the code waits until the previous write transaction has completed. The BREADY signal is deasserted using the nonblocking call to the `execute_write_resp_ready()` task and waits for the number of ACLK cycles defined by `m_wr_resp_phase_ready_delay`. A nonblocking call to the `execute_write_resp_ready()` task to assert the BREADY signal completes the BREADY handling. The `seen_valid_ready` flag is cleared to indicate that only BREADY has been asserted.
Example 6-7. handle_write_resp_ready()

// Task : handle_write_resp_ready
// This method assert/de-assert the write response channel ready signal.
// Assertion and de-assertion is done based on following variable's value:
// m_wr_resp_phase_ready_delay
// master_ready_delay_mode
task automatic handle_write_resp_ready;

// bit seen_valid_ready;

int tmp_ready_delay;
axi4_master_ready_delay_mode_e tmp_mode;

forever
begin
    wait(m_wr_resp_phase_ready_delay > 0);
    tmp_ready_delay = m_wr_resp_phase_ready_delay;
    tmp_mode = master_ready_delay_mode;

    if (tmp_mode == AXI4_VALID2READY)
        begin
            fork
                bfm.execute_write_resp_ready(1'b0);
            join_none
            end

            bfm.get_write_response_cycle;
            repeat(tmp_ready_delay - 1) bfm.wait_on(AXI4_CLOCK_POSEDGE);

            bfm.execute_write_resp_ready(1'b1);
            seen_valid_ready = 1'b1;
        end
    else  // AXI4_TRANS2READY
        begin
            if (seen_valid_ready == 1'b0)
                begin
                    do
                        bfm.wait_on(AXI4_CLOCK_POSEDGE);
                    while (!((bfm.BVALID === 1'b1) && (bfm.BREADY === 1'b1)));
                end

            fork
                bfm.execute_write_resp_ready(1'b0);
            join_none

            repeat(tmp_ready_delay) bfm.wait_on(AXI4_CLOCK_POSEDGE);

            fork
                bfm.execute_write_resp_ready(1'b1);
            join_none
            seen_valid_ready = 1'b0;
        end
endtask
handle_read_data_ready()

The handle_read_data_ready() task handles the RREADY signal for the read data channel. It delays the assertion of the RREADY signal based on the settings of master_ready_delay_mode and m_rd_data_phase_ready_delay. The handle_read_data_ready() task code is similar in operation to the handle_write_resp_ready() task.

Verifying a Master DUT

A master DUT component is connected to a slave BFM at the signal-level. A slave test program, written at the transaction-level, generates stimulus via the slave BFM to verify the master DUT. Figure 6-4 illustrates a typical top-level test bench environment.

![Figure 6-4. Master DUT Top-Level Test Bench Environment](image)

In this example, the slave test program is a simple memory model.

A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

BFM Slave Test Program

The slave test program is a memory model that contains two APIs: a Basic Slave API Definition and an Advanced Slave API Definition.

The Basic Slave API Definition allows you to create a wide range of stimulus scenarios to test a master DUT. This API definition simplifies the creation of slave stimulus based on the default response of OKAY to master read and write transactions. The Advanced Slave API Definition allows you to create additional response scenarios to transactions.
Note

For a complete code example of this slave test program, refer to the installed Mentor Verification IP Altera Edition kit in the following directory:

\<install_dir\\mentor_vip_ae\axi4lite\qsys-examples\ex1_back_to_back_sv\slave_test_program.sv

Basic Slave API Definition

The Basic Slave Test Program API contains the following elements:

- Functions that read and write a byte of data to Internal Memory include `do_byte_read()` and `do_byte_write()`, respectively.
- Functions `set_read_data_valid_delay()` and `set_wr_resp_valid_delay()` to configure the delay of the read data channel RVALID, and write response channel BVALID signals, respectively.
- Variables `m_rd_addr_phase_ready_delay` and `m_wr_addr_phase_ready_delay` to configure the delay of the read/write address channel ARVALID/AWVALID signals, and `m_wr_data_phase_ready_delay` to configure the delay of the write response channel BVALID signal.
- A `slave_ready_delay_mode` variable to configure the behavior of the handshake signals *VALID to *READY delay.

Internal Memory

The internal memory for the slave is defined as a sparse array of 8 bits, so that each byte of data is stored as an address/data pair.

Example 6-8. Internal Memory

```vhdl
// Storage for a memory
bit [7:0] mem [*];
```

do_byte_read()

The `do_byte_read()` function, when called, will read a data byte from the Internal Memory `mem`, given an address location as shown below.

You can edit this function to modify the way the read data is extracted from the internal memory.
Example 6-9. do_byte_read()

```verilog
// Function : do_byte_read
// Function to provide read data byte from memory at
// particular input address
function bit[7:0] do_byte_read(addr_t addr);
    return mem[addr];
endfunction
```

do_byte_write()

The `do_byte_write()` function, when called, writes a data byte to the Internal Memory `mem`, given an address location as shown below.

You can edit this function to modify the way the write data is stored in the internal memory.

Example 6-10. do_byte_write()

```verilog
// Function : do_byte_write
// Function to write data byte to memory at particular
// input address
function void do_byte_write(addr_t addr, bit [7:0] data);
    mem[addr] = data;
endfunction
```
m_rd_addr_phase_ready_delay

The `m_rd_addr_phase_ready_delay` variable holds the ARREADY signal delay. The delay value extends the length of the read address phase by a number of ACLK cycles. The starting point of the delay is determined by the `slave_ready_delay_mode` variable configuration.

Example 6-11 shows the ARREADY signal delayed by two ACLK cycles. You can edit this variable to change the ARREADY signal delay.

Example 6-11. m_rd_addr_phase_ready_delay

```verilog
// Variable : m_rd_addr_phase_ready_delay
int m_rd_addr_phase_ready_delay = 2;
```
m_wr_addr_phase_ready_delay

The `m_wr_addr_phase_ready_delay` variable holds the AWREADY signal delay. The delay value extends the length of the write address phase by a number of ACLK cycles. The starting point of the delay is determined by the `slave_ready_delay_mode` variable configuration.

Example 6-12 shows the AWREADY signal delayed by two ACLK cycles. You can edit this variable to change the AWREADY signal delay.
Example 6-12. m_wr_addr_phase_ready_delay

```SystemVerilog
// Variable : m_wr_addr_phase_ready_delay
int m_wr_addr_phase_ready_delay = 2;
```

m_wr_data_phase_ready_delay

The `m_wr_data_phase_ready_delay` variable holds the WREADY signal delay. The delay value extends the length of each write data phase (beat) in a write data burst by a number of ACLK cycles. The starting point of the delay is determined by the `slave_ready_delay_mode` variable configuration.

Example 6-13 shows the WREADY signal delayed by two ACLK cycles. You can edit this function to change the WREADY signal delay.

Example 6-13. m_wr_data_phase_ready_delay

```SystemVerilog
// Variable : m_wr_data_phase_ready_delay
int m_wr_data_phase_ready_delay = 2;
```

set_read_data_valid_delay()

The `set_read_data_valid_delay()` function, when called, configures the RVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of each read data phase (beat). The delay value of the RVALID signal is stored in the `data_valid_delay` transaction field.

Example 6-14 shows the RVALID signal delay incrementing by an ACLK cycle between each read data phase for the length of the burst. You can edit this function to change the RVALID signal delay.

Example 6-14. set_read_data_valid_delay()

```SystemVerilog
// Function : set_read_data_valid_delay
// This is used to set read data phase valid delays to start driving
// read data/response phases after specified delay.
function void set_read_data_valid_delay(axi4_transaction trans);
    trans.set_data_valid_delay(2);
endfunction
```

set_wr_resp_valid_delay()

The `set_wr_resp_valid_delay()` function, when called, configures the BVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of the write response phase. The delay value of the BVALID signal is stored in the `write_response_valid_delay` transaction field.

Example 6-15 shows the BVALID signal delay set to two ACLK cycles. You can edit this function to change the BVALID signal delay.
Example 6-15. set_wr_resp_valid_delay()

```verilog
define void set_wr_resp_valid_delay(axi4_transaction trans);
  trans.set_write_response_valid_delay(2);
endfunction
```

### slave_ready_delay_mode

The `slave_ready_delay_mode` variable holds the configuration that defines the starting point of any delay applied to the *READY signals. You can configure it to the enumerated type values of AXI4_VALID2READY (default) or AXI4_TRANS2READY.

The default configuration (`slave_ready_delay_mode = AXI4_VALID2READY`) corresponds to the delay measured from the positive edge of ACLK when *VALID is asserted. Figure 6-5 shows how to achieve a *VALID before *READY handshake.

**Figure 6-5. slave_ready_delay_mode = AXI4_VALID2READY**

```
<table>
<thead>
<tr>
<th>ACLK</th>
<th>*VALID</th>
<th>*READY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*valid_delay = 4  *ready_delay = 2

The nondefault configuration (`slave_ready_delay_mode = AXI4_TRANS2READY`) corresponds to the delay measured from the completion of a previous transaction phase (*VALID and *READY both asserted). Figure 6-6 shows how to achieve a *READY before *VALID handshake.
Example 6-16 shows the configuration of the \textit{slave\_ready\_delay\_mode} to its default value.

**Example 6-16. \texttt{slave\_ready\_delay\_mode}**

```systemverilog
// Enum type for slave ready delay mode
// AXI4_VALID2READY - Ready delay for a phase will be applied from
//                     start of phase (Means from when VALID is asserted).
// AXI4_TRANS2READY - Ready delay will be applied from the end of
//                     previous phase. This might result in ready before
//                     valid.
typedef enum bit
{
  AXI4_VALID2READY = 1'b0,
  AXI4_TRANS2READY = 1'b1
} axi4_slave_ready_delay_mode_e;

// Slave ready delay mode selection: default it is AXI4_VALID2READY
axi4_slave_ready_delay_mode_e slave_ready_delay_mode = AXI4_VALID2READY;

**Note**

In addition to the previous variables and procedures, you can configure other aspects of the AXI4-Lite Slave BFM by using the procedures “\texttt{set\_config()}” on page 69 and “\texttt{get\_config()}” on page 70.

**Using the Basic Slave Test Program API**

There are a set of tasks and functions that you can use to create stimulus scenarios based on a memory-model slave with a minimal amount of editing, as described in the Basic Slave API Definition section.

Consider the following configurations when using the slave test program.
Advanced Slave API Definition

**Note**

You are not required to edit the following Advance Slave API unless you require a different response than the default (OKAY) response.

The remaining section of this tutorial presents a walk-through of the Advanced Slave API in the slave test program. It consists of four main tasks: `process_read()`, `process_write()`, `handle_read()`, and `handle_write()` in the slave test program, as shown in Figure 6-7. There are additional `handle_write_addr_ready()`, `handle_read_addr_ready()` and `handle_write_data_ready()` tasks to handle the handshake AWREADY, ARREADY and WREADY signals, respectively.

The Advanced Slave API is capable of handling pipelined transactions. Pipelining can occur when a transaction starts before a previous transaction has completed. Therefore, a write transaction that starts before a previous write transaction has completed can be pipelined. Figure 6-7 shows the write channel with three concurrent `write_trans` transactions, whereby the `get_write_addr_phase[2]`, `get_write_data_phase[1]`, and `execute_write_response_phase[0]` are concurrently active on the write address, data, and response channels, respectively.

Similarly, a read transaction that starts before a previous read transaction has completed can be pipelined. Figure 6-7 shows the read channel with two concurrent `read_trans` transactions, whereby the `get_read_addr_phase[1]` and `execute_read_data_phase[0]` are concurrently active on the read address and data channels, respectively.
Figure 6-7. Slave Test Program Advanced API Tasks

WRITE CHANNEL

READ CHANNEL

= axiLite_transaction

Mentor Verification IP AE AXI4-Lite User Guide, V10.4c
January 2016
Initial Block

In an initial block, the slave test program configures the maximum number of outstanding read and write transactions. The slave test program then starts the processing of any read or write transactions, and the handling of the channel *READY signals in a fork-join block, as shown in Example 6-17.

Example 6-17. Initialization and Transaction Processing

```
initial
begin

  // Traffic generation
  fork
    process_read;
    process_write;
    handle_write_addr_ready;
    handle_read_addr_ready;
    handle_write_data_ready;
  join
end

process_read()
```

The process_read() task loops forever, processing read transactions as they occur from the master. A local transaction variable read_trans of type axi4_transaction is defined to hold a record of the read transaction while it is being processed. A slave transaction is created by calling the create_slave_transaction() function and assigned to the read_trans record.

The subsequent fork-join_none block performs a nonblocking statement so that the process_read() task can begin again to create another read transaction record and get another read address phase before the current read transaction has completed. This permits concurrent read transactions to occur if the master issues a series of read address phases before any previous read transactions have completed.

In the fork-join_none block, the read_trans record is passed into the handle_read() function via the variable t.
Example 6-18. process_read()

```systemverilog
// Task : process_read
// This method keep receiving read address phase and calls another
// method to process received transaction.
task process_read;
    forever
        begin
            axi4_transaction read_trans;

            read_trans = bfm.create_slave_transaction();
            bfm.get_read_addr_phase(read_trans);

            fork
                begin
                    automatic axi4_transaction t = read_trans;
                    handle_read(t);
                end
            join_none
            #0;
        end
    endtask

handle_read()

The `handle_read()` task gets the data from the Internal Memory as a a phase (beat). The `read_trans` argument contains the record of the read transaction up to the point of this task call, namely the content of the read address phase.

The call to `set_read_data_valid_delay()` configures the RVALID signal delay for each phase (beat).

In a loop, the call to the `get_read_addr()` helper function returns the actual address `addr` for a particular byte location. This byte address is used to read the data byte from Internal Memory with the call to the `do_byte_read()` function, assigning the local `mem_data` variable with read data `do_byte_read()`. The call to the `set_read_data()` helper function sets the byte within the read transaction record. The loop continues reading and setting the read data from internal memory for the whole of the read data phase (beat).

The read data phase is executed over the protocol signals by calling the `execute_read_data_phase()`.

Example 6-19. handle_read

```systemverilog
// Task : handle_read
// This method reads data from memory and send read data/response either
// at
// burst or phase level depending upon slave working mode.
task automatic handle_read(input axi4_transaction read_trans);
    addr_t addr[];
```
bit [7:0] mem_data[];

set_read_data_valid_delay(read_trans);
void'(bfm.get_read_addr(read_trans, 0, addr));

mem_data = new[addr.size()];
for(int j = 0; j < addr.size(); j++)
  mem_data[j] = do_byte_read(addr[j]);

bfm.set_read_data(read_trans, 0, addr, mem_data);
bfm.execute_read_data_phase(read_trans);
endtask

process_write()

process_write() is a task that processes read transactions in the slave test program. It works similarly to the process_read() task. The code snippet below demonstrates the process_write() function:

```systemverilog
// Task : process_write
// This method keeps receiving write address phase and calls another method to process received transaction.
task process_write;
  forever
    begin
      axi4_transaction write_trans;

      write_trans = bfm.create_slave_transaction();
      bfm.get_write_addr_phase(write_trans);

      fork
        begin
          automatic axi4_transaction t = write_trans;
          handle_write(t);
        end
      join_none
      #0;
    end
endtask

handle_write()

handle_write() is a task that handles write transactions. It works in a similar way to the handle_read() task. The main difference is that the write transaction handling gets the write data phase and stores it in the slave test program's Internal Memory, adhering to the state of the WSTRB write strobes signal. There is an additional write response phase that is required for the write response channel, as shown in Example 6-21.

Example 6-21. handle_write()

```

```systemverilog
// Task : handle_write
// This method receive write data burst or phases for write transaction
```
// depending upon slave working mode, write data to memory and then send // response
function automatic handle_write(input axi4_transaction write_trans);  // automatic handle_write
    addr_t addr[];
    bit [7:0] data[];
    bit last;

    bfm.get_write_data_phase(write_trans, 0, last);

    void'(bfm.get_write_addr_data(write_trans, 0, addr, data));

    for (int j = 0; j < addr.size(); j++)
        do_byte_write(addr[j], data[j]);

    set_wr_resp_valid_delay(write_trans);

    bfm.execute_write_response_phase(write_trans);
endfunction

handle_write_addr_ready()

The handle_write_addr_ready() task handles the AWREADY signal for the write address channel. In a forever loop, it delays the assertion of the AWREADY signal based on the settings of the slave_ready_delay_mode and m_wr_resp_phase_ready_delay as shown in Example 6-22.

If the slave_delay_ready_mode = AXI4_VALID2READY, then the AWREADY signal is deasserted using the nonblocking call to the execute_write_data_ready() task and waits for a write channel address phase to occur with a call to the blocking get_write_addr_cycle() task. A received write address phase indicates that the AWVALID signal has been asserted, triggering the starting point for the delay of the AWREADY signal by the number of ACLK cycles defined by m_wr_addr_phase_ready_delay. Another call to the execute_write_addr_ready() task to assert the AWREADY signal completes the AWREADY handling. The seen_valid_ready flag is set to indicate the end of an address phase when both AWVALID and AWREADY are asserted.

If the slave_delay_ready_mode = AXI4_TRANS2READY, then a check of the seen_valid_ready flag is performed to indicate that a previous write address phase has completed. If a write address phase is still active (indicated by either AWVALID or AWREADY not asserted), then the code waits until the previous write address phase has completed. The AWREADY signal is then deasserted using the nonblocking call to the execute_write_addr_ready() task and waits for the number of ACLK cycles defined by m_wr_addr_phase_ready_delay. A nonblocking call to the execute_write_addr_ready() task to assert the AWREADY signal completes the AWREADY handling. The seen_valid_ready flag is cleared to indicate that only AWREADY has been asserted.

Example 6-22. handle_write_addr_ready()

    // Task : handle_write_addr_ready
    // This method assert/de-assert the write address channel ready signal.
    // Assertion and de-assertion is done based on m_wr_addr_phase_ready_delay
task automatic handle_write_addr_ready;
  bit seen_valid_ready;
  int tmp_ready_delay;
  axi4_slave_ready_delay_mode_e tmp_mode;

  forever
    begin
      wait(m_wr_addr_phase_ready_delay > 0);
      tmp_ready_delay = m_wr_addr_phase_ready_delay;
      tmp_mode = slave_ready_delay_mode;

      if (tmp_mode == AXI4_VALID2READY)
        begin
          fork
            bfm.execute_write_addr_ready(1'b0);
            join_none

            bfm.get_write_addr_cycle;
            repeat(tmp_ready_delay - 1) bfm.wait_on(AXI4_CLOCK_POSEDGE);

            bfm.execute_write_addr_ready(1'b1);
            seen_valid_ready = 1'b1;
          end
        else  // AXI4_TRANS2READY
          begin
            if (seen_valid_ready == 1'b0)
              begin
                do
                  bfm.wait_on(AXI4_CLOCK_POSEDGE);
                  while (!((bfm.AWVALID === 1'b1) && (bfm.AWREADY === 1'b1)));
                end

                fork
                  bfm.execute_write_addr_ready(1'b0);
                  join_none

                repeat(tmp_ready_delay) bfm.wait_on(AXI4_CLOCK_POSEDGE);
              end
            end
          end
    endtask

handle_read_addr_ready()

The handle_read_addr_ready() task handles the ARREADY signal for the read address channel. In a forever loop, it delays the assertion of the ARREADY signal based on the settings of the slave_ready_delay_mode and m_rd_addr_phase_ready_delay. The handle_read_addr_ready() task code is similar in operation to the handle_write_addr_ready() task.
handle_write_data_ready()

The `handle_write_data_ready()` task handles the WREADY signal for the write data channel. In a forever loop, it delays the assertion of the WREADY signal based on the settings of the `slave_ready_delay_mode` and `m_wr_data_phase_ready_delay`. The `handle_write_data_ready()` task code is similar in operation to the `handle_write_addr_ready()` task.
Chapter 7
VHDL API Overview

This chapter describes the VHDL Application Programming Interface (API) procedures for all the BFM (master, slave, and monitor) components. For each BFM, you can configure protocol transaction fields that execute on the protocol signals and control the operational transaction fields that permit delays between the handshake signals for each of the five address, data, and response channels.

In addition, each BFM API has procedures that wait for certain events to occur on the system clock and reset signals, and procedures to get and set information about a particular transaction.

**Note**
The VHDL API is built on the SystemVerilog API. An internal VHDL to SystemVerilog (SV) wrapper casts the VHDL BFM API procedure calls to the SystemVerilog BFM API tasks and functions.
Figure 7-1. VHDL BFM Internal Structure

Notes: 1. Refer to the `create*_transaction()`
2. Refer to the `execute_transaction(), execute*_phase()`
3. Refer to the `get*()`
**Configuration**

Configuration sets timeout delays, error reporting, and other attributes of the BFM.

Each BFM has a `set_config()` procedure that sets the configuration of the BFM. Refer to the individual BFM API for valid details. Each BFM has a `get_config()` procedure that returns the configuration of the BFM. Refer to the individual BFM API for details.

**set_config()**

For example, the following test program code sets the burst timeout factor for a transaction in the master BFM:

```vhdl
-- Setting the burst timeout factor to 1000
set_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index, axi4_tr_if_0(bfm_index))
```

In the above example, the `bfm_index` specifies the BFM.

**get_config()**

For example, the following test program code gets the protocol signal hold time in the master BFM:

```vhdl
-- Getting the burst timeout factor
get_config(4_CONFIG_HOLD_TIME, config_value, bfm_index, axi4_tr_if_0(bfm_index))
```

In the above example, the `bfm_index` specifies the BFM.

**Creating Transactions**

To transfer information between a master BFM and slave DUT over the protocol signals, you must create a transaction in the master test program. Similarly, to transfer information between a master DUT and a slave BFM, you must create a transaction in the slave test program. To monitor the transfer of information using a monitor BFM, you must create a transaction in the monitor test program.

Creating a transaction also creates a Transaction Record that exists for the life of the transaction. This transaction record can be accessed by the BFM test program during the life of the transaction as it transfers information between the master and slave.

**Transaction Record**

The transaction record contains transaction fields. There are two main types of transaction fields, *protocol* and *operational*.
Protocol fields hold transaction information that is transferred over the protocol signals. For example, the `prot` field is transferred over the AWPROT protocol signals during a write transaction.

Operational fields hold information about how and when the transaction is transferred. Their content is not transferred over protocol signals. For example, the `operation_mode` field controls the blocking/nonblocking operation of the transaction, but is not transferred over the protocol signals.

**Transaction Definition**

The transaction record exists as a SystemVerilog class definition in each BFM. Example 7-1 shows the definition of the `axi4_transaction` class members that form the transaction record.

**Example 7-1. AXI4-Lite Transaction Definition**

```vhdl
// Global Transaction Class
class axi4_transaction;
  // Protocol
  axi4_rw_e read_or_write;
  bit [((`MAX_AXI4_ADDRESS_WIDTH) - 1):0] addr;
  axi4_prot_e prot;
  bit [3:0] region; // Not supported in AXI4-Lite
  axi4_size_e size; // Not supported in AXI4-Lite
  axi4_burst_e burst; // Not supported in AXI4-Lite
  axi4_lock_e lock; // Not supported in AXI4-Lite
  axi4_cache_e cache; // Not supported in AXI4-Lite
  bit [3:0] qos; // Not supported in AXI4-Lite
  bit [((`MAX_AXI4_ID_WIDTH) - 1):0] id // Not supported in AXI4-Lite;
  bit [7:0] burst_length;
  bit [((`MAX_AXI4_USER_WIDTH) - 1):0] addr_user; // Not supported in AXI4-Lite
  bit [((`MAX_AXI4_RDATA_WIDTH > `MAX_AXI4_WDATA_WIDTH) ?
    `MAX_AXI4_RDATA_WIDTH : `MAX_AXI4_WDATA_WIDTH)) - 1]:0] data_words [];
  bit [((`MAX_AXI4_WDATA_WIDTH / 8)) - 1]:0] write_strobes [];
  axi4_response_e resp[];
  int address_valid_delay;
  int data_valid_delay[];
  int write_response_valid_delay;
  int address_ready_delay;
  int data_ready_delay[];
  int write_response_ready_delay;

  // Housekeeping
  bit gen_write_strobes = 1'b1;
  axi4_operation_mode_e operation_mode = AXI4_TRANSACTION_BLOCKING;
  axi4_write_data_mode_e write_data_mode = AXI4_DATA_WITH_ADDRESS;
  bit data_beat_done[]; // Not supported in AXI4-Lite
  bit transaction_done;
  ...

endclass
```
Note

The `axi4_transaction` class code above is shown for information only. Access to each transaction record during its lifetime is performed via the various `set*()` and `get*()` procedures detailed later in this chapter.

---

Table 7-1 describes the transaction fields in the transaction record.

### Table 7-1. Transaction Fields

<table>
<thead>
<tr>
<th>Transaction Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Protocol Transaction Fields</strong></td>
<td></td>
</tr>
<tr>
<td><code>addr</code></td>
<td>A bit vector (of length equal to the ARADDR/AWADDR signal bus width) to hold the start <code>address</code> of the first transfer (beat) of a transaction. The <code>addr</code> value is transferred over the ARADDR or AWADDR signals for a read or write transaction, respectively.</td>
</tr>
<tr>
<td><code>prot</code></td>
<td>An enumeration to hold the <code>protection</code> type of a transaction. The types of <code>protection</code> are as follows:</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_SEC_DATA (default)</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_SEC_DATA</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_NONSEC_DATA</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_NONSEC_DATA</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_SEC_INST</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_SEC_INST</td>
</tr>
<tr>
<td></td>
<td>AXI4_NORM_NONSEC_INST</td>
</tr>
<tr>
<td></td>
<td>AXI4_PRIV_NONSEC_INST</td>
</tr>
<tr>
<td>The <code>prot</code> value is transferred over the ARPROT or AWPROT signals for a read or write transaction, respectively.</td>
<td></td>
</tr>
<tr>
<td><code>data_words</code></td>
<td>A bit vector (of length equal to the greater of the RDATA/WDATA signal bus widths) to hold the <code>data words</code> of the payload. A <code>data_words</code> is transferred over the RDATA or WDATA signals per beat of the read or write data channel, respectively.</td>
</tr>
<tr>
<td><code>write_strobes</code></td>
<td>A bit vector (of length equal to the WDATA signal bus width divided by 8) to hold the write strobes. A <code>write_strobes</code> is transferred over the WSTRB signals per beat of the write data channel.</td>
</tr>
</tbody>
</table>
### VHDL API Overview

#### Creating Transactions

<table>
<thead>
<tr>
<th>Transaction Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resp</td>
<td>An enumeration array to hold the responses of a transaction. The types of response are as follows:</td>
</tr>
<tr>
<td></td>
<td>AXI4_OKAY;</td>
</tr>
<tr>
<td></td>
<td>AXI4_SLVERR;</td>
</tr>
<tr>
<td></td>
<td>AXI4_DECERR;</td>
</tr>
<tr>
<td></td>
<td>A resp value is transferred over the RRESP signals per beat of the read data channel, and over the BRESP signals for a write transaction, respectively.</td>
</tr>
</tbody>
</table>

#### Operational Transaction Fields

<table>
<thead>
<tr>
<th>read_or_write</th>
<th>An enumeration to hold the read or write control flag. The types of read_or_write are:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AXI4_TRANS_READ</td>
</tr>
<tr>
<td></td>
<td>AXI4_TRANS_WRITE</td>
</tr>
<tr>
<td>address_valid_delay</td>
<td>An integer to hold the delay value of the address channel AWVALID and ARVALID signals (measured in ACLK cycles) for a read or write transaction, respectively.</td>
</tr>
<tr>
<td>data_valid_delay</td>
<td>An integer to hold the delay values of the data channel WVALID and RVALID signals (measured in ACLK cycles) for a read or write transaction, respectively.</td>
</tr>
<tr>
<td>write_response_valid_delay</td>
<td>An integer to hold the delay value of the write response channel BVALID signal (measured in ACLK cycles) for a write transaction.</td>
</tr>
<tr>
<td>address_ready_delay</td>
<td>An integer to hold the delay value of the address channel AWREADY and ARREADY signals (measured in ACLK cycles) for a read or write transaction, respectively.</td>
</tr>
<tr>
<td>data_ready_delay</td>
<td>An integer to hold the delay values of the data channel WREADY and RREADY signals (measured in ACLK cycles) for a read or write transaction, respectively.</td>
</tr>
<tr>
<td>write_response_ready_delay</td>
<td>An integer to hold the delay value of the write response channel BREADY signal (measured in ACLK cycles) for a write transaction.</td>
</tr>
<tr>
<td>gen_write_strobes</td>
<td>Automatically correct write strobes flag. Refer to Automatic Correction of Byte Lane Strobes for details.</td>
</tr>
</tbody>
</table>

Table 7-1. Transaction Fields (cont.)
Creating Transactions

The master BFM API allows you to create a master transaction by providing only the address argument for a read, or write, transaction. All other protocol transaction fields automatically default to legal protocol values to create a complete master transaction record. Refer to the `create_read_transaction()` and `create_write_transaction()` procedures for default protocol read and write transaction field values.

The slave BFM API allows you to create a slave transaction by providing no arguments. All protocol transaction fields automatically default to legal protocol values to create a complete slave transaction record. Refer to the `create_slave_transaction()` procedure for default protocol transaction field values.

The monitor BFM API allows you to create a slave transaction by providing no arguments. All protocol transaction fields automatically default to legal protocol values to create a complete

---

### Table 7-1. Transaction Fields (cont.)

<table>
<thead>
<tr>
<th>Transaction Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation_mode</td>
<td>An enumeration to hold the <code>operation mode</code> of the transaction. The two types of <code>operation mode</code> are: AXI4_TRANSACTION_NON_BLOCKING AXI4_TRANSACTION_BLOCKING</td>
</tr>
<tr>
<td>write_data_mode</td>
<td>An enumeration to hold the <code>write data mode</code> control flag. The types of <code>write_data_mode</code> are as follows: AXI4_DATA_AFTER_ADDRESS AXI4_DATA_WITH_ADDRESS (default)</td>
</tr>
<tr>
<td>transaction_done</td>
<td>A bit to hold the <code>done</code> flag for a transaction when it has completed.</td>
</tr>
</tbody>
</table>

The master BFM API allows you to create a master transaction by providing only the address argument for a read, or write, transaction. All other protocol transaction fields automatically default to legal protocol values to create a complete master transaction record. Refer to the `create_read_transaction()` and `create_write_transaction()` procedures for default protocol read and write transaction field values.

The slave BFM API allows you to create a slave transaction by providing no arguments. All protocol transaction fields automatically default to legal protocol values to create a complete slave transaction record. Refer to the `create_slave_transaction()` procedure for default protocol transaction field values.

The monitor BFM API allows you to create a slave transaction by providing no arguments. All protocol transaction fields automatically default to legal protocol values to create a complete
slave transaction record. Refer to the `create_monitor_transaction()` procedure for default protocol transaction field values.

**Note**

If you change a protocol transaction field value from its default, it is then valid for all future transactions until a new value is set.

### create*_transaction()

There are two master BFM API procedures available to create transactions, `create_read_transaction()` and `create_write_transaction()`, a `create_slave_transaction()` slave BFM API procedure, and a `create_monitor_transaction()` monitor BFM API procedure.

For example, to create a simple write transaction with a start address of 1, and a single data phase with a data value of 2, the master BFM test program would contain the following code:

```vhdl
-- Define local variables to hold the transaction ID
-- and data word.
variable tr_id: integer;
variable data_words : std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);

-- Create a master write transaction and set data_word value
create_write_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
data_words(31 downto 0) := x"00000200";
set_data_words(data_words, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

For example, to create a simple slave transaction the slave BFM test program would contain the following code:

```vhdl
-- Define a local variable write_trans to hold the transaction ID
variable write_trans : integer;

-- Create a slave transaction
create_slave_transaction(write_trans, bfm_index, axi4_tr_if_0(bfm_index));
```

In the above examples, the `bfm_index` specifies the BFM.

### Executing Transactions

Executing a transaction in a master/slave BFM test program initiates the transaction onto the protocol signals. Each master/slave BFM API has execution tasks that push transactions into the BFM internal transaction queues. Figure 7-1 on page 138 illustrates the internal BFM structure.
execute_transaction(), execute*_phase()

If the DUT is a slave then the `execute_transaction()` procedure is called in the master BFM test program. If the DUT is a master, then the `execute*_phase()` procedures are is called in the slave BFM test program.

For example, to execute a master write transaction the master BFM test program would contain the following code:

```vhdl
-- By default the execution of a transaction will block
execute_transaction(tr_id, bfm_index, axi4_tr_if_2(bfm_index));
```

For example, to execute a slave write response phase, the slave BFM test program would contain the following code:

```vhdl
-- By default the execution of a phase will block
execute_write_response_phase(write_trans, bfm_index,
axi4_tr_if_2(bfm_index));
```

In the above example, the `bfm_index` specifies the BFM.

Waiting Events

Each BFM API has procedures that block the test program code execution until an event has occurred.

The `wait_on()` procedure blocks the test program until an ACLK or ARESETn signal event has occurred before proceeding.

The `get*_transaction(), get*_phase(), get*_cycle()` procedures block the test program code execution until a complete transaction, phase or cycle has occurred, respectively.

wait_on()

For example, a BFM test program can wait for the positive edge of the `ARESETn` signal using the following code:

```vhdl
-- Block test program execution until the positive edge of the clock
wait_on(AXI4_RESET_POSEDGE, bfm_index, axi4_tr_if_0(bfm_index));
```

In the above example, the `bfm_index` specifies the BFM.

get*_transaction(), get*_phase(), get*_cycle()

For example, a slave BFM test program can use a received write address phase to form the response of the write transaction. The test program gets the write address phase for the transaction by calling the `get_write_addr_phase()` procedure. This task blocks until it has
received the address phase, allowing the test program to then call the
`execute_write_response_phase()` procedure for the transaction, as shown in the slave BFM test
program in Example 7-2 below.

Example 7-2. Slave BFM Test Program Using `get_write_addr_phase()`

```vhdl
create_slave_transaction(write_trans, bfm_index, axi4_tr_if_0(bfm_index));
get_write_addr_phase(write_trans, bfm_index, axi4_tr_if_0(bfm_index));
...
execute_write_response_phase(write_trans, bfm_index, AXI4_PATH_2,
axi4_tr_if_2(bfm_index));
```

In the above example, the `bfm_index` specifies the BFM.

---

Note
---

Not all BFM APIs support the full complement of `get*_transaction()`, `get*_phase()`,
`get*_cycle()` tasks. Refer to the individual master, slave, or monitor BFM API for details.

---

Access Transaction Record
---

Each BFM API has procedures that can access a complete, or partially complete, **Transaction Record**. The `set*()` and `get*()` procedures are used in a test program to set and get information from the transaction record.

**set***()**

For example, to set the WSTRB write strobes signal in the **Transaction Record** of a write transaction, the master test program would use the `set_write_strobes()` procedure, as shown in the code below.

```vhdl
set_write_strobes(2, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

In the above example, the `bfm_index` specifies the BFM.

**get***()**

For example, a slave BFM test program uses a received write address phase to get the AWPROT signal value from the **Transaction Record**, as shown in the following slave BFM test program code.

```vhdl
-- Wait for a write address phase;
get_write_addr_phase(slave_trans, bfm_index, axi4_tr_if_0(bfm_index));
...
```
-- Get the AWPROT signal value of the slave transaction
get_prot(prot_value, slave_trans, bfm_index, axi4_tr_if_0(bfm_index));

In the above example, the bfm_index specifies the BFM.

## Operational Transaction Fields

Operational transaction fields control the way in which a transaction is executed on the protocol signals. These fields also provide an indicator of when a data phase (beat) or transaction is complete.

### Automatic Correction of Byte Lane Strobes

The master BFM permits unaligned and narrow write transfers by using byte lane strobe (WSTRB) signals to indicate which byte lanes contain valid data per data phase (beat).

When you create a write transaction in your master BFM test program, the write_strobes variable is available to store the write strobe values for each write data phase (beat) in the transaction. To assist you in creating the correct byte lane strobes, automatic correction of any previously set write_strobes is performed by default during execution of the write transaction, or write data phase (beat). You can disable this default behavior by setting the transaction field gen_write_strobes = 0, which allows any previously set write_strobes to pass through uncorrected onto the protocol WSTRB signals. In this mode, with the automatic correction disabled, you are responsible for setting the correct write_strobes for the whole transaction.

The automatic correction algorithm performs a bit-wise AND operation on any previously set write_strobes. To do the corrections, the automatic correction algorithm uses the equations described in the AMBA AXI Protocol Specification, Version 2.0, Section A3.4.1, that define valid write data byte lanes for legal protocol. Therefore, if you require automatic generation of all write_strobes, before the write transaction executes, you must set all write_strobes to 1, indicating that all bytes lanes initially contain valid write data, prior to execution of the write transaction. Automatic correction will then set the relevant write_strobes to 0 to produce legal protocol WSTRB signals.

### Operation Mode

By default, each read or write transaction performs a blocking operation that prevents a following transaction from starting until the current active transaction completes.

You can configure this behavior to be nonblocking by setting the operation_mode transaction field to the enumerate type value AXI4_TRANSACTION_NON_BLOCKING instead of the default AXI4_TRANSACTION_BLOCKING.
For example, in a master BFM test program, you can create a transaction by calling the `create_read_transaction()` or `create_write_transaction()` tasks, which creates a transaction record. Before executing the transaction record, you can change the `operation_mode` as follows:

```vhdl
-- Create a write transaction to create a transaction record
create_write_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Change operation_mode to be nonblocking in the transaction record
set_operation_mode(AXI4_TRANSACTION_NON_BLOCKING, tr_id, bfm_index,
                   axi4_tr_if_0(bfm_index));
```

In the above example, the `bfm_index` specifies the BFM.

### Channel Handshake Delay

Each of the five protocol channels have *VALID and *READY handshake signals to control the rate at which information is transferred between a master and slave. Refer to [Handshake Delay](#) for details of the AXI4-Lite BFM API.

### Handshake Delay

The delay between the *VALID and *READY handshake signals for each of the five protocol channels is controlled in a BFM test program using `execute_*_ready()`, `get_*_ready()` and `get_*_cycle()` procedures. The `execute_*_ready()` procedures place a value onto the *READY signals, and the `get_*_ready()` procedures retrieve a value from the *READY signals. The `get_*_cycle()` procedures wait for a *VALID signal to be asserted and are used to insert a delay between the *VALID and *READY signals in the BFM test program.

For example, the master BFM test program code below inserts a specified delay between the read channel RVALID and RREADY handshake signals using the `execute_read_data_ready()` and `get_read_data_cycle()` procedures.

```vhdl
-- Set the RREADY signal to ‘0’.
execute_read_data_ready(0, 1, bfm_index, AXI4_PATH_6,
                        axi4_tr_if_6(bfm_index));

-- Wait for the RVALID signal to be asserted.
get_read_data_cycle(bfm_index, AXI4_PATH_6,
                    axi4_tr_if_6(bfm_index));

-- Add delay between RVALID and RREADY.
for i in 0 to 2 loop
    wait_on(AXI4_CLOCK_POSEDGE, bfm_index, AXI4_PATH_6,
            axi4_tr_if_6(bfm_index));
end loop;
execute_read_data_ready(1, 1, bfm_index, AXI4_PATH_6,
                        axi4_tr_if_6(bfm_index));
```

In this example, the `bfm_index` specifies the BFM.
*VALID Signal Delay Transaction Fields

The transaction record contains a *valid_delay transaction field for each of the five protocol channels to configure the delay value prior to the assertion of the *VALID signal for the channel. The master BFM holds the delay configuration for the *VALID signals that it asserts, and the slave BFM holds the delay configuration for the *VALID signals that it asserts. Table 7-2 specifies which *valid_delay fields are configured by the master and slave BFMs.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Operational Transaction Field</th>
<th>Configuration BFM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWVALID</td>
<td>address_valid_delay</td>
<td>Master</td>
</tr>
<tr>
<td>WVALID</td>
<td>data_valid_delay</td>
<td>Master</td>
</tr>
<tr>
<td>BVALID</td>
<td>write_response_valid_delay</td>
<td>Slave</td>
</tr>
<tr>
<td>ARVALID</td>
<td>address_valid_delay</td>
<td>Master</td>
</tr>
<tr>
<td>RVALID</td>
<td>data_valid_delay</td>
<td>Slave</td>
</tr>
</tbody>
</table>

*READY Handshake Signal Delay Transaction Fields

The transaction record contains a *ready_delay transaction field for each of the five protocol channels to store the delay value between the assertion of the *VALID and *READY handshake signals for the channel. Table 7-3 specifies the *ready_delay field corresponding to the *READY signal delay.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Operational Transaction Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWREADY</td>
<td>address_ready_delay</td>
</tr>
<tr>
<td>WREADY</td>
<td>data_ready_delay</td>
</tr>
<tr>
<td>BREADY</td>
<td>write_response_ready_delay</td>
</tr>
<tr>
<td>ARREADY</td>
<td>address_ready_delay</td>
</tr>
<tr>
<td>RREADY</td>
<td>data_ready_delay</td>
</tr>
</tbody>
</table>

Transaction Done

There is a transaction_done transaction field in each transaction that indicates when the transaction has completed.

In a BFM test program, you call the respective BFM get_transaction_done() procedure to investigate whether a read or write transaction has completed.
This chapter provides information about the VHDL master BFM. The BFM has an API that contains procedures to configure the BFM and to access the dynamic “Transaction Record” on page 28 during the life of the transaction.

Overloaded Procedure Common Arguments

The BFMs use VHDL procedure overloading, which results in the prototype having a number of prototype definitions for each procedure. Their arguments are unique to each procedure and concern the protocol or operational transaction fields for a transaction. These procedures have several common arguments, which can be optional, and include the arguments described below:

- **transaction_id** is an index number that identifies a specific transaction. Each new transaction automatically increments the index number until reaching 255, the maximum value, and then the index number automatically wraps to zero. The transaction_id uniquely identifies each transaction when there are a number of concurrently active transactions.

- **queue_id** is a unique identifier for each queue in a test bench. A queue is used to pass the record of a transaction between the address, data and response channels of a write transaction, and the address and data channels of a read transaction. There is a maximum of eight queues available within an AXI4 BFM-Lite. Refer to “Advanced Slave API Definition” on page 350 for more details on the application of the queue_id.

- **bfm_id** is a unique identification number for each master, slave, and monitor BFM in a multiple BFM test bench.

- **path_id** is a unique identifier for each parallel process in a multiple process test bench. You must specify the path_id for test bench stimulus to replicate the pipelining features of a protocol in a VHDL test bench. If no pipelining is performed in the test bench stimulus (a single process), then specifying the path_id argument for the procedure is optional. There is a maximum of eight paths available within an AXI4 BFM-Lite. Refer to “Advanced Slave API Definition” on page 350 for more details on the application of the path_id.

- **tr_if** is a signal definition that passes the content of a transaction between the VHDL and SystemVerilog environments.
Master BFM Protocol Support

The AXI4-Litemaster BFM supports the AMBA AXI4 protocol with restrictions detailed in “Protocol Restrictions” on page 17.

Master Timing and Events

For detailed timing diagrams of the protocol bus activity and details of the following master BFM API timing and events, refer to the relevant AMBA AXI Protocol Specification chapter.

The AMBA AXI Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the master BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

Master BFM Configuration

The master BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. It has parameters you can use to configure the widths of the address and data signals, and transaction fields to configure timeout factors, setup and hold times, and so on.

You can change the address and data signal widths from their default settings by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the master BFM via a parameter port list of the master BFM component.

Table 8-1 lists the parameter names for the address and data signals, and their default values.

<table>
<thead>
<tr>
<th>Signal Width Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_ADDRESS_WIDTH</td>
<td>Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.</td>
</tr>
<tr>
<td>AXI4_RDATA_WIDTH</td>
<td>Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>AXI4_WDATA_WIDTH</td>
<td>Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>index</td>
<td>Uniquely identifies a Master BFM instance. It must be set to a different value for each Master BFM in the system. Default: 0.</td>
</tr>
</tbody>
</table>
A master BFM has configuration fields that you can set via the \texttt{set_config()}
function to configure timeout factors, setup and hold times, and so on. You can also get the value of a configuration field via the \texttt{get_config()}
procedures. Table \ref{table:bfm-config} describes the full list of configuration fields.

### Table 8-2. Master BFM Configuration

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timing Variables</strong></td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_SETUP_TIME</td>
<td>The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_HOLD_TIME</td>
<td>The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR</td>
<td>The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_BURST_TIMEOUT_FACTOR</td>
<td>The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.</td>
</tr>
</tbody>
</table>
### Table 8-2. Master BFM Configuration (cont.)

<table>
<thead>
<tr>
<th>Configuration Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods. Default: 10000.</td>
</tr>
</tbody>
</table>

**Master Attributes**

<table>
<thead>
<tr>
<th>AXI4_CONFIG_AXI4LITE_axi4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configures the AXI4 master BFM to be AXI4-Lite compatible.</td>
</tr>
<tr>
<td>0 = disabled (default)</td>
</tr>
<tr>
<td>1 = enabled</td>
</tr>
</tbody>
</table>

**Slave Attributes**

<table>
<thead>
<tr>
<th>AXI4_CONFIG_SLAVE_START_ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configures the start address map for the slave.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AXI4_CONFIG_SLAVE_END_ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configures the end address map for the slave.</td>
</tr>
</tbody>
</table>

**Error Detection**

<table>
<thead>
<tr>
<th>AXI4_CONFIG_ENABLE_ALL_ASSERTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global enable/disable of all assertion checks in the BFM.</td>
</tr>
<tr>
<td>0 = disabled</td>
</tr>
<tr>
<td>1 = enabled (default)</td>
</tr>
</tbody>
</table>
Master Assertions

Each master BFM performs protocol error checking via built-in assertions.

**Note**

The built-in BFM assertions are independent of programming language and simulator.

Assertion Configuration

By default, all built-in assertions are enabled in the master BFM. To globally disable them in the master BFM, use the \texttt{set_config()} command as the following example illustrates.

\begin{verbatim}
set_config(AXI4_CONFIG_ENABLE_ASSERTION, 0, bfm_index, axi4_tr_if_0(bfm_index));
\end{verbatim}

Alternatively, you can disable individual built-in assertions by using a sequence of \texttt{get_config()} and \texttt{set_config()} commands on the respective assertion. For example, to disable assertion checking for the AWADDR signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

\begin{verbatim}
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector, bfm_index, axi4_tr_if_0(bfm_index));

-- Assign the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector(AXI4_AWADDR_CHANGED_BEFORE_AWREADY) := '0';

-- Set the new value of the assertion bit vector
set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector, bfm_index, axi4_tr_if_0(bfm_index));
\end{verbatim}
Note

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion within the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to “AXI4-Lite Assertions” on page 375.

VHDL Master API

This section describes the VHDL Master API.

set_config()

This nonblocking procedure sets the configuration of the master BFM.

Prototype

```
procedure set_config
(
  config_name   : in std_logic_vector(7 downto 0);
  config_val    : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)|integer;
  bfm_id        : in integer;
  path_id       : in axi4_path_t; -- optional
  signal tr_if  : inout axi4_vhd_if_struct_t
);
```

Arguments

- **config_name** Configuration name:
  - AXI4_CONFIG_SETUP_TIME
  - AXI4_CONFIG_HOLD_TIME
  - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
  - AXI4_CONFIG_AXI4LITE_axi4
  - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
  - AXI4_CONFIG_ENABLE_ASSERTION
  - AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
  - AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
  - AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
  - AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
  - AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
  - AXI4_CONFIG_SLAVE_START_ADDR
  - AXI4_CONFIG_SLAVE_END_ADDR

- **config_val** Refer to “Master BFM Configuration” on page 152 for description and valid values.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
set_config()

path_id (Optional) Parallel process path identifier:

AXI4_PATH_0
AXI4_PATH_1
AXI4_PATH_2
AXI4_PATH_3
AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Example

set_config(AXI4_MAX_TRANSACTION_TIME_FACTOR, 1000, bfm_index, axi4_tr_if_0(bfm_index));
VHDL Master BFM

**get_config()**

This nonblocking procedure gets the configuration of the master BFM.

### Prototype

```vhdl
procedure get_config
(    config_name   : in std_logic_vector(7 downto 0);
    config_val    : out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)|integer;
    bfm_id        : in integer;
    path_id       : in axi4_path_t; --optional
    signal tr_if  : inaxi4_vhd_if_struct_t
);
```

### Arguments

- **config_name** Configuration name:
  - AXI4_CONFIG_SETUP_TIME
  - AXI4_CONFIG_HOLD_TIME
  - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
  - AXI4_CONFIG_AXI4LITE_axi4
  - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
  - AXI4_CONFIG_ENABLE_ASSERTION
  - AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
  - AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
  - AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
  - AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
  - AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
  - AXI4_CONFIG_SLAVE_START_ADDR
  - AXI4_CONFIG_SLAVE_END_ADDR

- **config_val** Refer to “Master BFM Configuration” on page 152 for description and valid values.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

### Returns

`config_val`

### Example

```vhdl
get_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR, config_value, 
            bfm_index, axi4_tr_if_0(bfm_index));
```
create_write_transaction()

This nonblocking procedure creates a write transaction with a start address \( \text{addr} \) argument. All other transaction fields default to legal protocol values, unless previously assigned a value. It returns with the \( \text{transaction_id} \) argument.

**Prototype**

```vhdl
procedure create_write_transaction
(
    addr            : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)|integer;
    transaction_id  : out integer;
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **addr**
  - Start address
- **transaction_id**
  - Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**
  - BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**
  - (Optional) Parallel process path identifier:
    - AXI4_PATH_0
    - AXI4_PATH_1
    - AXI4_PATH_2
    - AXI4_PATH_3
    - AXI4_PATH_4
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**
  - Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Protocol**

**Transaction Fields**

- **prot**
  - Protection:
    - AXI4_NORM_SEC_DATA; (default)
    - AXI4_PRIV_SEC_DATA;
    - AXI4_NORM_NONSEC_DATA;
    - AXI4_PRIV_NONSEC_DATA;
    - AXI4_NORM_SEC_INST;
    - AXI4_PRIV_SEC_INST;
    - AXI4_NORM_NONSEC_INST;
    - AXI4_PRIV_NONSEC_INST;

- **data_words**
  - Data words.

- **write_strobes**
  - Write strobes:
    - Each strobe 0 or 1.

- **resp**
  - Response:
    - AXI4_OKAY;
    - AXI4_SLVERR;
    - AXI4_DECERR;

**Operational Transaction Fields**

- **gen_write_strobes**
  - Correction of write strobes for invalid byte lanes:
    - \( 0 = \text{write_strobes} \) passed through to protocol signals.
    - \( 1 = \text{write_strobes} \) auto-corrected for invalid byte lanes (default).
VHDL Master BFM

**create_write_transaction()**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| operation_mode    | Operation mode:  
|                   | AXI4_TRANSACTION_NON_BLOCKING;  
|                   | AXI4_TRANSACTION_BLOCKING; (default)  
| write_data_mode   | Write data mode:  
|                   | AXI4_DATA_AFTER_ADDRESS;  
|                   | The master first drives the address phase and, after it completes, it drives the corresponding data phases.  
|                   | The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **DATA_WITH_ADDRESS along with a data_valid_delay setting instead to avoid the possible deadlock situation.  
|                   | AXI4_DATA_WITH_ADDRESS; (default)  
|                   | The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay  
| address_valid_delay| Address channel A*VALID delay measured in ACLK cycles for this transaction (default = 0).  
| data_valid_delay  | Write data channel WVALID delay array measured in ACLK cycles for this transaction (default = 0 for all elements).  
| write_response_read_delay| Write response channel BREADY delay measured in ACLK cycles for this transaction (default = 0).  
| transaction_done  | Write transaction done flag for this transaction.  
| transaction_id    | Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151.  

**Example**

```vhdl
-- Create a write data transaction to start address 16.
-- Returns the transaction ID (tr_id) for this created transaction.
create_write_transaction(16, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
create_read_transaction()

This nonblocking procedure creates a read transaction with a start address `addr` argument. All other transaction parameters default to legal protocol values, unless previously assigned a value. It returns with the `transaction_id` argument.

**Prototype**

```vhdl
procedure create_read_transaction
(
  addr            : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)|integer;
  transaction_id  : out integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `addr` Start address
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `prot` Protection:
  - AXI4_NORM_SEC_DATA; (default)
  - AXI4_PRIV_SEC_DATA;
  - AXI4_NORM_NONSEC_DATA;
  - AXI4_PRIV_NONSEC_DATA;
  - AXI4_NORM_SEC_INST;
  - AXI4_PRIV_SEC_INST;
  - AXI4_NORM_NONSEC_INST;
  - AXI4_PRIV_NONSEC_INST;
- `data_words` Data words.
- `resp` Response:
  - AXI4_OKAY;
  - AXI4_SLVERR;
  - AXI4_DECERR;

**Operational Transaction Fields**

- `operation_mode` Operation mode:
  - AXI4_TRANSACTION_NON_BLOCKING;
  - AXI4_TRANSACTION_BLOCKING; (default)
### VHDL Master BFM

**create_read_transaction()**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address_valid_delay</td>
<td>Address channel A*VALID delay measured in ACLK cycles for this transaction (default = 0).</td>
</tr>
<tr>
<td>data_ready_delay</td>
<td>Read data channel RREADY delay array measured in ACLK cycles for this transaction (default = 0).</td>
</tr>
<tr>
<td>transaction_done</td>
<td>Read transaction done flag for this transaction.</td>
</tr>
<tr>
<td>transaction_id</td>
<td></td>
</tr>
</tbody>
</table>

**Returns**

**Example**

```vhdl
-- Create a read data transaction with start address 16.
-- Returns the transaction ID (tr_id) for this created transaction.
create_read_transaction(16, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_addr()

This nonblocking procedure sets the start address \textit{addr} field for a transaction that is uniquely identified by the \textit{transaction_id} field previously created by either the \textit{create_write_transaction()} or \textit{create_read_transaction()} procedure.

**Prototype**

```
set_addr
(
    addr : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **addr** Start address of transaction.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:  
  - AXI4_PATH_0  
  - AXI4_PATH_1  
  - AXI4_PATH_2  
  - AXI4_PATH_3  
  - AXI4_PATH_4  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the start address to 1 for the tr_id transaction
set_addr(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_addr()

This nonblocking procedure gets the start address $addr$ field for a transaction that is uniquely identified by the $transaction_id$ field previously created by either the $create_write_transaction()$ or $create_read_transaction()$ procedure.

Prototype

```vhdl
get_addr
(   addr : out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **addr**: Start address of transaction.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **addr**

Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

```
-- Get the start address addr of the tr_id transaction
get_addr(addr, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_prot()

This nonblocking procedure sets the protection prot field for a transaction that is uniquely identified by the transaction_id field previously created by either the create_write_transaction() or procedure.

**Prototype**

```vhdl
set_prot
(
    prot: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **prot** Protection:
  - AXI4_NORM_SEC_DATA (default);
  - AXI4_PRIV_SEC_DATA;
  - AXI4_NORM_NONSEC_DATA;
  - AXI4_PRIV_NONSEC_DATA;
  - AXI4_NORM_SEC_INST;
  - AXI4_PRIV_SEC_INST;
  - AXI4_NORM_NONSEC_INST;
  - AXI4_PRIV_NONSEC_INST;

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the protection field to a normal, secure, instruction access
-- for the tr_id transaction.
set_prot(AXI4_NORM_SEC_INST, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_prot()

This nonblocking procedure gets the protection prot field for a transaction that is uniquely identified by the transaction_id field previously created by either the create_write_transaction() or procedure.

Prototype

```vhdl
get_prot
(
    prot: out integer,
    transaction_id  : in integer,
    bfm_id : in integer,
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **prot** Protection:
  - AXI4_NORM_SEC_DATA;
  - AXI4_PRIV_SEC_DATA;
  - AXI4_NORM_NONSEC_DATA;
  - AXI4_PRIV_NONSEC_DATA;
  - AXI4_NORM_SEC_INST;
  - AXI4_PRIV_SEC_INST;
  - AXI4_NORM_NONSEC_INST;
  - AXI4_PRIV_NONSEC_INST;

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **prot**

Example

```vhdl
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the protection field of the tr_id transaction.
get_prot(prot, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
**set_data_words()**

This nonblocking procedure sets a `data_words` field for a write transaction that is uniquely identified by the `transaction_id` field previously created by the `create_write_transaction()` procedure.

**Prototype**

```vhdl
set_data_words(
  data_words: in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0) |
  transaction_id  : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `data_words` Data words.
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns** None

**Example**

```vhdl
-- Create a write transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the data_words field to 2 for the data phase
-- for the tr_id transaction.
set_data_words(2, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_data_words()

This nonblocking procedure gets a data_words field for a transaction that is uniquely identified by the transaction_id field previously created by either the create_write_transaction() or procedure.

Prototype

```vhdl
get_data_words
  (   data_words: out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)  
| integer;  
  transaction_id : in integer;  
  bfm_id : in integer;  
  path_id : in axi4_path_t; --optional  
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **data_words** Data words.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **data_words**

Example

```vhdl
-- Create a read transaction with start address of 1.  
-- Creation returns tr_id to identify the transaction.  
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the data_words field for data phase  
-- of the tr_id transaction.  
get_data_words(data, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_write_strobes()

This nonblocking procedure sets the write_strobes field for a write transaction that is uniquely identified by the transaction_id field previously created by the create_write_transaction() procedure.

Prototype

```vhdl
set_write_strobes
(
    write_strobes : in std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

write_strobes Write strobes.
transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

- AXI4_PATH_0
- AXI4_PATH_1
- AXI4_PATH_2
- AXI4_PATH_3
- AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Example

```vhdl
-- Create a write transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write_strobes field to for the data phase
-- for the tr_id transaction.
set_write_strobes(2, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_strobes()

This nonblocking procedure gets a `write_strobes` field for a write transaction that is uniquely identified by the `transaction_id` field previously created by the `create_write_transaction()` procedure.

**Prototype**

```vhdl
get_write_strobes
  (write_strobes : out std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
   transaction_id : in integer;
   bfm_id : in integer;
   path_id : in axi4_path_t; --optional
   signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `write_strobes` Write strobes.
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`write_strobes`

**Example**

```
-- Create a write transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write_strobes field for the data phase
-- of the tr_id transaction.
get_write_strobes(write_strobe, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```
set_resp()

This nonblocking procedure sets a response \textit{resp} field for a transaction that is uniquely identified by the \textit{transaction_id} field previously created by either the \textit{create_write_transaction()} or procedure.

Prototype

\begin{verbatim}
set_resp (
    resp: in std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
\end{verbatim}

Arguments

\begin{itemize}
\item \textbf{resp} Transaction response:
\begin{itemize}
\item AXI4_OKAY = 0;
\item AXI4_SLVERR = 2;
\item AXI4_DECERR = 3;
\end{itemize}
\item \textbf{transaction_id} Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
\item \textbf{bfm_id} BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
\item \textbf{path_id} (Optional) Parallel process path identifier:
\begin{itemize}
\item AXI4_PATH_0
\item AXI4_PATH_1
\item AXI4_PATH_2
\item AXI4_PATH_3
\item AXI4_PATH_4
\end{itemize}
\item \textbf{tr_if} Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
\end{itemize}

Returns

None

\begin{itemize}
\item \textbf{Note}
\item You would not normally use this procedure in a master test program.
\end{itemize}
VHDL Master BFM

get_resp()

get_resp()

This nonblocking procedure gets a response resp field for a transaction that is identified by the transaction_id field previously created by either the create_write_transaction() or procedure.

Prototype

```vhdl
get_resp
(  resp: out std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) |
   transaction_id : in integer;    
   bfm_id : in integer;            
   path_id : in axi4_path_t;      
   signal tr_if : inout axi4_vhd_if_struct_t  
);
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resp</td>
<td>Transaction response:</td>
</tr>
<tr>
<td></td>
<td>AXI4_OKAY = 0;</td>
</tr>
<tr>
<td></td>
<td>AXI4_SLVERR = 2;</td>
</tr>
<tr>
<td></td>
<td>AXI4_DECERR = 3;</td>
</tr>
<tr>
<td>transaction_id</td>
<td>Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.</td>
</tr>
<tr>
<td>bfm_id</td>
<td>BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.</td>
</tr>
<tr>
<td>path_id</td>
<td>(Optional) Parallel process path identifier:</td>
</tr>
<tr>
<td></td>
<td>AXI4_PATH_0</td>
</tr>
<tr>
<td></td>
<td>AXI4_PATH_1</td>
</tr>
<tr>
<td></td>
<td>AXI4_PATH_2</td>
</tr>
<tr>
<td></td>
<td>AXI4_PATH_3</td>
</tr>
<tr>
<td></td>
<td>AXI4_PATH_4</td>
</tr>
<tr>
<td></td>
<td>Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.</td>
</tr>
<tr>
<td>tr_if</td>
<td>Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.</td>
</tr>
</tbody>
</table>

Returns

resp

Example

```vhdl
-- Create a read transaction with start address of 1.  
-- Creation returns tr_id to identify the transaction.  
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the response field for the data phase  
-- of the tr_id transaction.  
get_resp(read_resp, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_read_or_write()

This nonblocking procedure sets the *read_or_write* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or procedure.

**Prototype**

```vhdl
set_read_or_write
(
    read_or_write: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **read_or_write** Read or write transaction:
  - AXI4_TRANS_READ = 0
  - AXI4_TRANS_WRITE = 1
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Note**

You do not normally use this procedure in a master test program.
get_read_or_write()

This nonblocking procedure gets the read_or_write field for a transaction that is uniquely identified by the transaction_id field previously created by either the create_write_transaction() or procedure.

Prototype

```vhdl
get_read_or_write
(
    read_or_write: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **read_or_write** Read or write transaction:
  - `AXI4_TRANS_READ = 0`
  - `AXI4_TRANS_WRITE = 1`
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **read_or_write**

Example

```vhdl
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the read_or_write field of the tr_id transaction.
get_read_or_write(read_or_write, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_gen_write_strobes()

This nonblocking procedure sets the `gen_write_strobes` field for a write transaction that is uniquely identified by the `transaction_id` field previously created by the `create_write_transaction()` procedure.

Prototype

```vhdl
set_gen_write_strobes(
    gen_write_strobes: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- `gen_write_strobes` Correction of write strobes for invalid byte lanes:
  - 0 = `write_strobes` passed through to protocol signals.
  - 1 = `write_strobes` auto-corrected for invalid byte lanes (default).
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Disable the auto correction of the write strobes for the
-- tr_id transaction.
set_gen_write_strobes(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_gen_write_strobes()

This nonblocking procedure gets the gen_write_strobes field for a write transaction that is uniquely identified by the transaction_id field previously created by the create_write_transaction() procedure.

Prototype

```
get_gen_write_strobes
(
    gen_write_strobes: out integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **gen_write_strobes** Correct write strobes flag:
  - 0 = write_strobes passed through to protocol signals.
  - 1 = write_strobes auto-corrected for invalid byte lanes.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **gen_write_strobes**

Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the auto correction write strobes flag of the tr_id transaction.
get_gen_write_strobes(write_strobes_flag, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```
**set_operation_mode()**

This nonblocking procedure sets the `operation_mode` field for a transaction that is uniquely identified by the `transaction_id` field previously created by either the `create_write_transaction()` or `procedure`.

**Prototype**

```vhdl
set_operation_mode
(  operation_mode: in integer;
  transaction_id  : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **operation_mode**
  - Operation mode:
    - AXI4_TRANSACTION_NON_BLOCKING
    - AXI4_TRANSACTION_BLOCKING (default)
- **transaction_id**
  - Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 151 for more details.
- **bfm_id**
  - BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 151 for more details.
- **path_id**
  - (Optional) Parallel process path identifier:
    - AXI4_PATH_0
    - AXI4_PATH_1
    - AXI4_PATH_2
    - AXI4_PATH_3
    - AXI4_PATH_4
    - Refer to "Overloaded Procedure Common Arguments" on page 151 for more details.
- **tr_if**
  - Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the operation mode field to nonblocking for tr_id transaction.
set_operation_mode(AXI4_TRANSACTION_NON_BLOCKING, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_operation_mode()

This nonblocking procedure gets the operation_mode field for a transaction that is uniquely identified by the transaction_id field previously created by either the create_write_transaction() or procedure.

Prototype

def get_operation_mode(
    operation_mode: out integer,
    transaction_id : in integer,
    bfm_id : in integer,
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
)

Arguments

operation_mode Operation mode:

- AXI4_TRANSACTION_NON_BLOCKING;
- AXI4_TRANSACTION_BLOCKING;

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

(path_id (Optional) Parallel process path identifier:

- AXI4_PATH_0
- AXI4_PATH_1
- AXI4_PATH_2
- AXI4_PATH_3
- AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

operation_mode

Example

-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the operation mode field of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
set_write_data_mode()

This nonblocking procedure sets the `write_data_mode` field for a transaction that is uniquely identified by the `transaction_id` field previously created by either the `create_write_transaction()` or `procedure`.

**Prototype**

```vhdl
set_write_data_mode
(
    write_data_mode: in integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **write_data_mode** Write data mode:
  - `AXI4_DATA_AFTER_ADDRESS;`
    - The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the `DATA_WITH_ADDRESS` along with a `data_valid_delay` setting instead to avoid the possible deadlock situation.
  - `AXI4_DATA_WITH_ADDRESS;` (default)
    - The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on `data_valid_delay`. If `data_valid_delay` is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after `data_valid_delay`.

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
    - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None
Example

-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write data mode field of the address and data phases for the
-- tr_id transaction
set_write_data_mode(AXI4_DATA_WITH_ADDRESS, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
get_write_data_mode()

This nonblocking procedure gets the write_data_mode field for a transaction that is uniquely identified by the transaction_id field previously created by either the create_write_transaction() or procedure.

Prototype

get_write_data_mode
{
    write_data_mode: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
};

Arguments

write_data_mode Write data mode:

AXI4_DATA_AFTER_ADDRESS;
The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the _DATA_WITH_ADDRESS along with a data_valid_delay setting instead to avoid the possible deadlock situation.

AXI4_DATA_WITH_ADDRESS; (default)
The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay.

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

    AXI4_PATH_0
    AXI4_PATH_1
    AXI4_PATH_2
    AXI4_PATH_3
    AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

write_data_mode
Example

-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data mode field of the tr_id transaction
get_write_data_mode(write_data_mode, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
**set_address_valid_delay()**

This nonblocking procedure sets the `address_valid_delay` field for a transaction that is uniquely identified by the `transaction_id` field previously created by either the `create_write_transaction()` or procedure.

**Prototype**

```vhdl
set_address_valid_delay

(address_valid_delay: in integer;
 transaction_id: in integer;
 bfm_id: in integer;
 path_id: in axi4_path_t; --optional
 signal tr_if: inout axi4_vhd_if_struct_t)
```

**Arguments**

- **address_valid_delay** Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction. Default: 0.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the address channel *VALID delay to 3 clock cycles
-- for the tr_id transaction.
set_address_valid_delay(3, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_address_valid_delay()

This nonblocking procedure gets the `address_valid_delay` field for a transaction that is uniquely identified by the `transaction_id` field previously created by either the `create_write_transaction()` or `procedure`.

**Prototype**

```vhdl
get_address_valid_delay(
    address_valid_delay: out integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `address_valid_delay` Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction.
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`address_valid_delay`

**Example**

```vhdl
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write address channel AWVALID delay of the tr_id transaction.
get_address_valid_delay(address_valid_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

---
get_address_ready_delay()

This nonblocking procedure gets the address_ready_delay field for a transaction that is uniquely identified by the transaction_id field previously created by either the create_write_transaction() or procedure.

Prototype

```
get_address_ready_delay(
    address_ready_delay: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **address_ready_delay**: Address channel A*READY delay measured in ACLK cycles for this transaction.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **address_ready_delay**

Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the address channel *READY delay of the tr_id transaction.
get_address_ready_delay(address_ready_delay, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_data_valid_delay()

This nonblocking procedure sets the data_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_write_transaction() procedure.

Prototype

```vhdl
set_data_valid_delay(
    data_valid_delay: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

data_valid_delay Write data channel WVALID delay measured in ACLK cycles for this transaction. Default: 0.

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

- AXI4_PATH_0
- AXI4_PATH_1
- AXI4_PATH_2
- AXI4_PATH_3
- AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Example

```vhdl
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write channel WVALID delay to 3 ACLK cycles for the data
-- phase of the tr_id transaction.
set_data_valid_delay(3, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_data_valid_delay()

This nonblocking procedure gets the data_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by either the create_write_transaction() or procedure.

Prototype

get_data_valid_delay
{
  data_valid_delay : out integer;
  transaction_id : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
};

Arguments

data_valid_delay Data channel array to store *VALID delays measured in ACLK cycles for this transaction.

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

  AXI4_PATH_0
  AXI4_PATH_1
  AXI4_PATH_2
  AXI4_PATH_3
  AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

data_valid_delay

Example

-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the read channel RVALID delay for the data
-- phase of the tr_id transaction.
get_data_valid_delay(data_valid_delay, tr_id, bfm_index,
  axi4_tr_if_0(bfm_index));
get_data_ready_delay()

This nonblocking procedure gets the *data_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *procedure*.

**Prototype**

```vhdl
get_data_ready_delay(
  data_ready_delay: out integer;
  transaction_id  : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **data_ready_delay** Read data channel RREADY delay measured in ACLK cycles for this transaction.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

- **data_ready_delay**

**Example**

```vhdl
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write data channel WREADY delay the data
-- phase of the tr_id transaction.
get_data_ready_delay(data_ready_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```
set_write_response_valid_delay()

This nonblocking procedure sets the write_response_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_write_transaction() procedure.

Prototype

```
set_write_response_valid_delay
(
    write_response_valid_delay: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- `write_response_valid_delay`: Write data channel BVALID delay measured in ACLK cycles for this transaction. Default: 0.
- `path_id`: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if`: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Note

You do not normally use this procedure in a master test program.
**get_write_response_valid_delay()**

This nonblocking procedure gets the `write_response_valid_delay` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_write_transaction()` procedure.

**Prototype**

```
get_write_response_valid_delay(
    write_response_valid_delay: out integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **write_response_valid_delay** Write data channel BVALID delay measured in ACLK cycles for this transaction.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`write_response_valid_delay`

**Example**

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BVALID delay of the tr_id transaction.
get_write_response_valid_delay(write_response_valid_delay, tr_id,
bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_response_ready_delay()

This nonblocking procedure gets the `write_response_ready_delay` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_write_transaction()` procedure.

Prototype

```vhd
get_write_response_ready_delay
{
    write_response_ready_delay: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
};
```

Arguments

- **write_response_ready_delay**: Write data channel BREADY delay measured in ACLK cycles for this transaction.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **write_response_ready_delay**

Example

```vhd
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BREADY delay of the tr_id transaction.
get_write_response_ready_delay(write_resp_ready_delay, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_transaction_done()

This nonblocking procedure sets the transaction_done field for a transaction that is uniquely identified by the transaction_id field previously created by the create_write_transaction() or procedure.

**Prototype**

```vhdl
set_transaction_done
(
    transaction_done : in integer;
    transaction_id   : in integer;
    bfm_id           : in integer;
    path_id          : in axi_path_t; --optional
    signal tr_if     : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **transaction_done**
  - Transaction done flag for this transaction

- **transaction_id**
  - Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id**
  - BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**
  - (Optional) Parallel process path identifier:
    - AXI4_PATH_0
    - AXI4_PATH_1
    - AXI4_PATH_2
    - AXI4_PATH_3
    - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  - Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

...

-- Set the read transaction_done flag of the tr_id transaction.
set_transaction_done(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```


get_transaction_done()

This nonblocking procedure gets the transaction_done field for a transaction that is uniquely identified by the transaction_id field previously created by either the create_write_transaction() procedure.

Prototype

```vhdl
get_transaction_done
(
    transaction_done : out integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_done**: Transaction done flag for this transaction
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **transaction_done**:

Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

```
....
```

```
-- Get the read transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
execute_transaction()

This procedure executes a master transaction that is uniquely identified by the `transaction_id` argument, previously created with either the `create_write_transaction()` or procedure. A transaction can be blocking (default) or nonblocking, based on the setting of the transaction `operation_mode` field.

The results of `execute_transaction()` for write transactions varies based on how write transaction fields are set. If the transaction `gen_write_strobes` field is set, `execute_transaction()` automatically corrects any previously set `write_strobes` field array elements. However, if the `gen_write_strobes` field is not set, then any previously assigned `write_strobes` field array elements will be passed onto the WSTRB protocol signals, which can result in a protocol violation if not correctly set. Refer to “Automatic Correction of Byte Lane Strobes” on page 147 for more details.

If the `write_data_mode` field for a write transaction is set to AXI4_DATA_WITH_ADDRESS, `execute_transaction()` calls the `execute_write_addr_phase()` and `execute_write_data_phase()` procedures simultaneously; otherwise, `execute_write_data_phase()` is called after `execute_write_addr_phase()` so that the write data beat occurs after the write address phase (default). It then calls the `get_write_response_phase()` procedure to complete the write transaction.

For a read transaction, `execute_transaction()` calls the `execute_read_addr_phase()` procedure followed by the `get_read_data_phase()` procedure to complete the read transaction.

**Prototype**

```vhdl
procedure execute_transaction
(
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **transaction_id**
  
  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id**
  
  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**
  
  (Optional) Parallel process path identifier:
  
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  
  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None
Example

-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Execute the tr_id transaction.
execute_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
execute_write_addr_phase()

This procedure executes a master write address phase uniquely identified by the transaction_id argument previously created by the create_write_transaction() procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record operation_mode field.

It sets the AWVALID protocol signal at the appropriate time defined by the transaction record address_valid_delay field.

Prototype

procedure execute_write_addr_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);  

Arguments

transaction_id   Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
bfm_id           BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
path_id          (Optional) Parallel process path identifier:
  AXI4_PATH_0
  AXI4_PATH_1
  AXI4_PATH_2
  AXI4_PATH_3
  AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
tr_if            Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Execute the write address phase for the tr_id transaction.
execute_write_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
execute_read_addr_phase()

This procedure executes a master read address phase uniquely identified by the transaction_id argument previously created by the procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record operation_mode field.

It sets the ARVALID protocol signal at the appropriate time defined by the transaction record address_valid_delay field.

Prototype

```vhdl
procedure execute_read_addr_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
)
```

Arguments

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- None

Example

```vhdl
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Execute the read address phase for the tr_id transaction.
execute_read_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
execute_write_data_phase()

This procedure executes a write data phase that is uniquely identified by the transaction_id argument and previously created by the create_write_transaction() procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record operation_mode field.

The execute_write_data_phase() sets the WVALID protocol signal at the appropriate time defined by the transaction record data_valid_delay field when the phase complete.

Prototype

```vhdl
procedure execute_write_data_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Execute the write data phase for the tr_id transaction.
execute_write_data_phase(tr_id, bfm_index, axi4_, tr_if_0(bfm_index));
```
get_read_data_phase()

This blocking procedure gets a read data phase that is uniquely identified by the transaction_id argument previously created by the procedure. If this is the last phase (beat), then it sets the transaction_done field to 1 to indicate the whole read transaction is complete.

Example

Prototype

procedure get_read_data_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id          : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);

Arguments

transaction_id  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id  (Optional) Parallel process path identifier:

  AXI4_PATH_0
  AXI4_PATH_1
  AXI4_PATH_2
  AXI4_PATH_3
  AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns  None

-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read data phase of the tr_id transaction.
get_read_data_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
get_write_response_phase()

This blocking procedure gets a write response phase that is uniquely identified by the `transaction_id` argument previously created by the `create_write_transaction()` procedure. It sets the `transaction_done` field to 1 when the transaction completes to indicate the whole transaction is complete.

Example

Prototype

```vhdl
procedure get_write_response_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

```
....
```

```
-- Get the write response phase for the tr_id transaction.
get_write_response_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_read_addr_ready()

This blocking procedure returns the value of the read address channel ARREADY signal using the *ready* argument. It will block for one ACLK period.

**Prototype**

```vhdl
procedure get_read_addr_ready
(  ready : out integer;
  bfm_id : in integer;
  path_id : in axi4_adv_path_t;  --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **ready**: The value of the ARREADY signal.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

- **ready**

**Example**

```vhdl
// Get the ARREADY signal value
bfm.get_read_addr_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
get_read_data_cycle()

This blocking procedure waits until the read data channel RVALID signal has been asserted.

Prototype

```
procedure get_read_data_cycle
  (
    bfm_id          : in integer;
    path_id         : in axi4_adv_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
  );
```

Arguments

- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```
// Wait for the RVALID signal to be asserted.
bfm.get_read_data_cycle(bfm_index, axi4_tr_if_0(bfm_index));
```
execute_read_data_ready()

This procedure executes a read data ready by placing the ready argument value onto the RREADY signal. It will block (default) for one ACLK period.

Prototype

```vhdl
procedure execute_read_data_ready
(
  ready : in integer
  non_blocking_mode : in integer; --optional
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **ready**: The value to be placed onto the RREADY signal
- **non_blocking_mode**: (Optional) Nonblocking mode:
  - 0 = Nonblocking
  - 1 = Blocking (default)
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
-- Set the RREADY signal to 1 and block for 1 ACLK cycle
execute_read_data_ready(1, 1, index, AXI4_PATH_6, axi4_tr_if_6(index));
```
get_write_addr_ready()

This blocking procedure returns the value of the write address channel AWREADY signal using the ready argument. It will block for one ACLK period.

**Prototype**

```vhdl
procedure get_write_addr_ready
(   ready : out integer;
    bfm_id  : in integer;
    path_id : in axi4_adv_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **ready** The value of the AWREADY signal.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

- **ready**

**Example**

```vhdl
// Get the AWREADY signal value
bfm.get_write_addr_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
**get_write_data_ready()**

This blocking procedure returns the value of the write data channel WREADY signal using the `ready` argument. It will block for one ACLK period.

**Prototype**

```
procedure get_write_data_ready
(
    ready : out integer;
    bfm_id : in integer;
    path_id : in axi4_adv_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
)
```

**Arguments**

- `ready` The value of the WREADY signal.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`ready`

**Example**

```
// Get the WREADY signal value
bfm.get_write_data_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_response_cycle()

This blocking procedure waits until the write response channel BVALID signal has been asserted.

**Prototype**

```vhdl
procedure get_write_response_cycle
(  bfm_id       : in integer;
  path_id      : in axi4_adv_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **bfm_id**  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**  (Optional) Parallel process path identifier:
  
<table>
<thead>
<tr>
<th>Path Identifier</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_PATH_5</td>
<td></td>
</tr>
<tr>
<td>AXI4_PATH_6</td>
<td></td>
</tr>
<tr>
<td>AXI4_PATH_7</td>
<td></td>
</tr>
</tbody>
</table>

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**  None

**Example**

```vhdl
// Wait for the BVALID signal to be asserted.
bfm.get_write_response_cycle(bfm_index, axi4_tr_if_0(bfm_index));
```
execute_write_resp_ready()

This procedure executes a write response ready by placing the `ready` argument value onto the BREADY signal. It will block for one ACLK period.

**Prototype**

```vhdl
procedure execute_write_resp_ready
(
  ready : in integer;
  non_blocking_mode : in integer; --optional
  bfm_id          : in integer;
  path_id          : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **ready**
  - The value to be placed onto the BREADY signal
- **non_blocking_mode**
  - (Optional) Nonblocking mode:
    - 0 = Nonblocking
    - 1 = Blocking (default)
- **bfm_id**
  - BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**
  - (Optional) Parallel process path identifier:
    - AXI4_PATH_0
    - AXI4_PATH_1
    - AXI4_PATH_2
    - AXI4_PATH_3
    - AXI4_PATH_4
    - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**
  - Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Set the BREADY signal to 1 and block for 1 ACLK cycle
execute_write_resp_ready(1, 1, index, AXI4_PATH_5, axi4_tr_if_5(index));
```
**push_transaction_id()**

This nonblocking procedure pushes a transaction ID into the back of a queue. The transaction is uniquely identified by the `transaction_id` argument previously created by either the `create_write_transaction()` or `create_read_transaction()` procedure. The queue is identified by the `queue_id` argument.

**Prototype**

```vhdl
procedure push_transaction_id
(
    transaction_id  : in integer;
    queue_id  : in integer;
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `queue_id` Queue identifier:
  - AXI4_QUEUE_ID_0
  - AXI4_QUEUE_ID_1
  - AXI4_QUEUE_ID_2
  - AXI4_QUEUE_ID_3
  - AXI4_QUEUE_ID_4
  - AXI4_QUEUE_ID_5
  - AXI4_QUEUE_ID_6
  - AXI4_QUEUE_ID_7
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None
Example

-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Push the transaction record into queue 1 for the tr_id transaction.
push_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
pop_transaction_id()

This nonblocking (unless queue is empty) procedure pops a transaction ID from the front of a queue. The transaction is uniquely identified by the transaction_id argument previously created by either the create_write_transaction() or procedure. The queue is identified by the queue_id argument.

If the queue is empty, then it will block until an entry becomes available.

**Prototype**

```vhdl
procedure pop_transaction_id
(
    transaction_id : in integer;
    queue_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **transaction_id**
  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **queue_id**
  Queue identifier:
  - AXI4_QUEUE_ID_0
  - AXI4_QUEUE_ID_1
  - AXI4_QUEUE_ID_2
  - AXI4_QUEUE_ID_3
  - AXI4_QUEUE_ID_4
  - AXI4_QUEUE_ID_5
  - AXI4_QUEUE_ID_6
  - AXI4_QUEUE_ID_7
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id**
  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**
  (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None
Example

-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Pop the transaction record from queue 1 for the tr_id transaction.
pop_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
print()  

This nonblocking procedure prints a transaction record that is uniquely identified by the `transaction_id` argument previously created by either the `create_write_transaction()` or procedure.

**Prototype**

```vhdl
procedure print
(
    transaction_id  : in integer;
    print_delays   : in integer; --optional
    bfm_id         : in integer;
    path_id        : in axi4_path_t; --optional
    signal tr_if   : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `print_delays`: (Optional) Print delay values flag:
  - `0` = do not print the delay values (default).
  - `1` = print the delay values.
- `path_id`: (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if`: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
```
**destruct_transaction()**

This blocking procedure removes a transaction record for cleanup purposes and memory management that is uniquely identified by the *transaction_id* argument previously created by either the *create_write_transaction()* or procedure.

**Prototype**

```vhdl
procedure destruct_transaction
(  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier: `AXI4_PATH_0` `AXI4_PATH_1` `AXI4_PATH_2` `AXI4_PATH_3` `AXI4_PATH_4`
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
wait_on()

This blocking task waits for an event(s) on the ACLK or ARESETn signals to occur before proceeding. An optional `count` argument waits for the number of events equal to `count`.

**Prototype**

```vhdl
procedure wait_on
(
  phase           : in integer;
  count: in integer; --optional
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **phase**
  
  Wait for:
  
  - AXI4_CLOCK_POSEDGE
  - AXI4_CLOCK_NEGEDGE
  - AXI4_CLOCK_ANYEDGE
  - AXI4_CLOCK_0_TO_1
  - AXI4_CLOCK_1_TO_0
  - AXI4_RESET_POSEDGE
  - AXI4_RESET_NEGEDGE
  - AXI4_RESET_ANYEDGE
  - AXI4_RESET_0_TO_1
  - AXI4_RESET_1_TO_0

- **count**
  
  (Optional) Wait for a number of events to occur set by `count`. (default = 1)

- **bfm_id**
  
  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**
  
  (Optional) Parallel process path identifier:
  
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  
  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
wait_on(AXI4_RESET_POSEDGE, bfm_index, axi4_tr_if_0(bfm_index));
wait_on(AXI4_CLOCK_POSEDGE, 10, bfm_index, axi4_tr_if_0(bfm_index));
```
This chapter provides information about the VHDL slave BFM. The BFM has an API that contains procedures to configure the BFM and to access the “Transaction Record” on page 28 during the lifetime of the transaction.

**Slave BFM Protocol Support**

The AXI4-Lite slave BFM supports the AMBA AXI4 protocol with restrictions detailed in “Protocol Restrictions” on page 17.

**Slave Timing and Events**

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI Protocol Specification chapter, which you can use to reference details of the following slave BFM API timing and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the slave BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

**Slave BFM Configuration**

The slave BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. The BFM has parameters that you can use to configure the widths of the address and data signals and transaction fields to configure timeout factors, setup and hold times, and so on.

You can change the address and data signals widths from their default settings by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the slave BFM via a parameter port list of the slave BFM component.

Table 9-1 lists the parameter names for the address and data signals, and their default values.

---

**Note**

See “Running the Qsys Tool” on page 362 for details of the Qsys Parameter Editor.
VHDL Slave BFM

Slave BFM Configuration

A slave BFM has configuration fields that you can set via the `set_config()` function to configure timeout factors, setup and hold times, and so on. You can also get the value of a configuration field via the `get_config()` procedures. Table 9-2 describes the full list of configuration fields.

Table 9-1. Slave BFM Signal Width Parameters

<table>
<thead>
<tr>
<th>Signal Width Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_ADDRESS_WIDTH</td>
<td>Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.</td>
</tr>
<tr>
<td>AXI4_RDATA_WIDTH</td>
<td>Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>AXI4_WDATA_WIDTH</td>
<td>Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>index</td>
<td>Uniquely identifies a slave BFM instance. It must be set to a different value for each slave BFM in the system. Default: 0.</td>
</tr>
<tr>
<td>READ_ACCEPTANCE_CAPABILITY</td>
<td>The maximum number of outstanding read transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
<tr>
<td>WRITE_ACCEPTANCE_CAPABILITY</td>
<td>The maximum number of outstanding write transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
<tr>
<td>COMBINED_ACCEPTANCE_CAPABILITY</td>
<td>The maximum number of outstanding combined read and write transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
</tbody>
</table>

Table 9-2. Slave BFM Configuration

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing Variables</td>
<td></td>
</tr>
<tr>
<td>AXI4_CONFIG_SETUP_TIME</td>
<td>The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
</tbody>
</table>
### Table 9-2. Slave BFM Configuration (cont.)

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_HOLD_TIME</td>
<td>The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR</td>
<td>The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_BURST_TIMEOUT_FACTOR</td>
<td>The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY</td>
<td>The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods (default 10000).</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY</td>
<td>The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods (default 10000).</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY</td>
<td>The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods (default 10000).</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY</td>
<td>The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods (default 10000).</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY</td>
<td>The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods (default 10000).</td>
</tr>
</tbody>
</table>

### Slave Attributes

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_AXI4LITE_axi4</td>
<td>Configures the AXI4 slave BFM to be AXI4-Lite compatible. 0 = disabled (default) 1 = enabled</td>
</tr>
<tr>
<td>AXI4_CONFIG_SLAVE_START_ADDR</td>
<td>Configures the start address map for the slave.</td>
</tr>
<tr>
<td>AXI4_CONFIG_SLAVE_END_ADDR</td>
<td>Configures the end address map for the slave.</td>
</tr>
</tbody>
</table>
Slave Assertions

The slave BFM performs protocol error checking via built-in assertions.

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_MAX_OUTSTANDING_WR</td>
<td>Configures the maximum number of outstanding write requests from the master that can be processed by the slave. The slave back-pressures the master by setting the signal AWREADY=0b0 if this value is exceeded. Default = 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_OUTSTANDING_RD</td>
<td>Configures the maximum number of outstanding read requests from the master that can be processed by the slave. The slave back-pressures the master by setting the signal ARREADY=0b0 if this value is exceeded. Default = 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE</td>
<td>Holds the number of outstanding write phases from the master that can be processed by the slave. Default = 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE</td>
<td>Holds the number of outstanding read phases to the master that can be processed by the slave. Default = 0.</td>
</tr>
</tbody>
</table>

**Error Detection**

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_ENABLE_ALL_ASSERTIONS</td>
<td>Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)</td>
</tr>
<tr>
<td>AXI_CONFIG_ENABLE_ASSERTION</td>
<td>Individual enable/disable of assertion check in the BFM. 0 = disabled 1 = enabled (default)</td>
</tr>
</tbody>
</table>

1. Refer to Slave Timing and Events for details of simulator time-steps.
Note

The built-in BFM assertions are independent of programming language and simulator.

Assertion Configuration

By default, all built-in assertions are enabled in the slave BFM. To globally disable them in the master BFM, use the `set_config()` command as the following example illustrates.

```vhd
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS, 0, bfm_index, axi4_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of `get_config()` and `set_config()` commands on the respective assertion. For example, to disable assertion checking for the AWADDR signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```vhd
-- Define a local bit vector to hold the value of the assertion bit vector variable config_assert_bitvector : std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);

-- Get the current value of the assertion bit vector get_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector, bfm_index, axi4_tr_if_0(bfm_index));

-- Assign the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion bit to 0 config_assert_bitvector(AXI4_AWADDR_CHANGED_BEFORE_AWREADY) := '0';

-- Set the new value of the assertion bit vector set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector, bfm_index, axi4_tr_if_0(bfm_index));
```

Note

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion within the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to “AXI4-Lite Assertions” on page 375.

VHDL Slave API

This section describes the VHDL Slave API.
**set_config()**

This nonblocking procedure sets the configuration of the slave BFM.

**Prototype**

```vhdl
procedure set_config
(
    config_name   : in std_logic_vector(7 downto 0);
    config_val    : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0) |
                    integer;
    bfm_id        : in integer;
    path_id       : in axi4_path_t; --optional
    signal tr_if  : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **config_name**
  Configuration name:
  - AXI4_CONFIG_SETUP_TIME
  - AXI4_CONFIG_HOLD_TIME
  - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
  - AXI4_CONFIG_AXI4LITE_axi4
  - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
  - AXI4_CONFIG_ENABLE_ASSERTION
  - AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
  - AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
  - AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
  - AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
  - AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
  - AXI4_CONFIG_SLAVE_START_ADDR
  - AXI4_CONFIG_SLAVE_END_ADDR
  - AXI4_CONFIG_MAX_OUTSTANDING_WR
  - AXI4_CONFIG_MAX_OUTSTANDING_RD
  - AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE
  - AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE

- **config_val**
  Refer to “Slave BFM Configuration” on page 215 for description and valid values.

- **bfm_id**
  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**
  (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None
Example

```vhdl
set_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR, 1000, bfm_index,
axi4_tr_if_0(bfm_index));
```
get_config()

This nonblocking procedure gets the configuration of the slave BFM.

Prototype

```vhdl
procedure get_config
(
    config_name   : in std_logic_vector(7 downto 0);
    config_val    : out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)| integer;
    bfm_id        : in integer;
    path_id       : in axi4_path_t; --optional
    signal tr_if  : inout axi4_vhd_if_struct_t
);
```

Arguments

- **config_name** Configuration name:
  - AXI4_CONFIG_SETUP_TIME
  - AXI4_CONFIG_HOLD_TIME
  - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
  - AXI4_CONFIG_AXI4LITE_axi4
  - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
  - AXI4_CONFIG_ENABLE_ASSERTION
  - AXI4_CONFIG_MAX_LATENCY_AWVALID ASSERTION_TO_AWREADY
  - AXI4_CONFIG_MAX_LATENCY_ARVALID ASSERTION_TO_ARREADY
  - AXI4_CONFIG_MAX_LATENCY_RVALID ASSERTION_TO_RREADY
  - AXI4_CONFIG_MAX_LATENCY_BVALID ASSERTION_TO_BREADY
  - AXI4_CONFIG_MAX_LATENCY_WVALID ASSERTION_TO_WREADY
  - AXI4_CONFIG_SLAVE_START_ADDR
  - AXI4_CONFIG_SLAVE_END_ADDR
  - AXI4_CONFIG_MAX_OUTSTANDING_WR
  - AXI4_CONFIG_MAX_OUTSTANDING_RD
  - AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE
  - AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE

- **config_val** Refer to “Slave BFM Configuration” on page 215 for description and valid values.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **config_val**
Example

```vhdl
get_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR, config_value,
            bfm_index, axi4_tr_if_0(bfm_index));
```
create_slave_transaction()

This nonblocking procedure creates a slave transaction. All transaction fields default to legal protocol values, unless previously assigned a value. It returns the `transaction_id` argument.

**Prototype**

```vhdl
procedure create_slave_transaction
(
  transaction_id : out integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `path_id`: (Optional) Parallel process path identifier: `AXI4_PATH_0`, `AXI4_PATH_1`, `AXI4_PATH_2`, `AXI4_PATH_3`, `AXI4_PATH_4`. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Protocol Transaction Fields**

- `addr`: Start address
- `prot`: Protection:
  - `AXI4_NORM_SEC_DATA`; (default)
  - `AXI4_PRIV_SEC_DATA`
  - `AXI4_NORM_NONSEC_DATA`
  - `AXI4_PRIV_NONSEC_DATA`
  - `AXI4_NORM_SEC_INST`
  - `AXI4_PRIV_SEC_INST`
  - `AXI4_NORM_NONSEC_INST`
  - `AXI4_PRIV_NONSEC_INST`
- `data_words`: Data words.
- `write_strobes`: Write strobes:
  - Each strobe 0 or 1.
- `resp`: Response:
  - `AXI4_OKAY`
  - `AXI4_SLVERR`
  - `AXI4_DECERR`
- `read_or_write`: Read or write transaction flag:
  - `AXI_TRANS_READ`
  - `AXI_TRANS_WRITE`
<table>
<thead>
<tr>
<th>Operational Transaction Fields</th>
<th>returns</th>
<th></th>
</tr>
</thead>
</table>
| **gen_write_strobes** | Correction of write strobes for invalid byte lanes:  
0 = write_strobes passed through to protocol signals.  
1 = write_strobes auto-corrected for invalid byte lanes (default). |  |
| **operation_mode** | Operation mode:  
AXI4_TRANSACTION_NON_BLOCKING;  
AXI4_TRANSACTION_BLOCKING; (default) |  |
| **write_data_mode** | Write data mode:  
AXI4_DATA_AFTER_ADDRESS;  
The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **DATA_WITH_ADDRESS along with a data_valid_delay setting instead to avoid the possible deadlock situation.  
AXI4_DATA_WITH_ADDRESS; (default)  
The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay. |  |
| **address_valid_delay** | Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction (default = 0). |  |
| **data_valid_delay** | Write data channel WVALID delay array measured in ACLK cycles for this transaction (default = 0 for all elements). |  |
| **write_response_valid_delay** | Write data channel BVALID delay measured in ACLK cycles for this transaction (default = 0). |  |
| **address_ready_delay** | Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction (default = 0). |  |
| **data_ready_delay** | Read data channel RREADY delay measured in ACLK cycles for this transaction (default = 0). |  |
| **write_response_ready_delay** | Write data channel BREADY delay measured in ACLK cycles for this transaction (default = 0). |  |
| **transaction_done** | Transaction done flag for this transaction |  |
| **transaction_id** | Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151. |  |
Example

-- Create a slave transaction
-- Returns the transaction ID (tr_id) for this created transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_3(bfm_index));
set_addr()

This nonblocking procedure sets the start address \( addr \) field for a transaction that is uniquely identified by the \( transaction\_id \) field previously created by the \( create\_slave\_transaction() \) procedure.

**Prototype**

```vhdl
set_addr
(   addr : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0) |
    integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **addr** Start address of transaction.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4\_PATH\_0
  - AXI4\_PATH\_1
  - AXI4\_PATH\_2
  - AXI4\_PATH\_3
  - AXI4\_PATH\_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Note**

You do not normally use this procedure in a slave test program.
VHDL Slave BFM

get_addr()

get_addr()

This nonblocking procedure gets the start address addr field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```
get_addr
(
    addr : out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **addr** Start address of transaction.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

```
addr
```

Example

```
-- Create a slave transaction. Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the start address addr of the tr_id transaction
get_addr(addr, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_prot()

This nonblocking procedure sets the protection prot field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```
set_prot
(
    prot: in integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **prot** Protection:
  - AXI4_NORM_SEC_DATA (default);
  - AXI4_PRIV_SEC_DATA;
  - AXI4_NORM_NONSEC_DATA;
  - AXI4_PRIV_NONSEC_DATA;
  - AXI4_NORM_SEC_INST;
  - AXI4_PRIV_SEC_INST;
  - AXI4_NORM_NONSEC_INST;
  - AXI4_PRIV_NONSEC_INST;

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.
get_prot()

This nonblocking procedure gets the protection prot field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```
get_prot

    ( prot: out integer;
      transaction_id : in integer;
      bfm_id : in integer;
      path_id : in axi4_path_t; --optional
      signal tr_if : inout axi4_vhd_if_struct_t
    );
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prot</td>
<td>Protection:</td>
</tr>
<tr>
<td>transaction_id</td>
<td>Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.</td>
</tr>
<tr>
<td>bfm_id</td>
<td>BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.</td>
</tr>
<tr>
<td>path_id</td>
<td>(Optional) Parallel process path identifier:</td>
</tr>
<tr>
<td>tr_if</td>
<td>Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.</td>
</tr>
</tbody>
</table>

Returns

prot

Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the protection field of the tr_id transaction.
get_prot(prot, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_data_words()

This nonblocking procedure sets the read data_words field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```vhdl
set_data_words(
    data_words: in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **data_words**: Data words.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the data_words field to 2 for the read data phase (beat)
-- for the tr_id transaction.
set_data_words(2, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_data_words()

This nonblocking procedure gets a data_words field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```vhdl
get_data_words
(
  data_words: out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)
  | integer;
  transaction_id : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **data_words** Data words.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **data_words**

Example

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the data_words field of the data phase (beat)
-- for the tr_id transaction.
get_data_words(data, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_write_strobes()

This nonblocking procedure sets the write_strobes field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```vhdl
set_write_strobes
(
    write_strobes : in std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **write_strobes**: Write strobes array.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  
  AXI4_PATH_0
  AXI4_PATH_1
  AXI4_PATH_2
  AXI4_PATH_3
  AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 123 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.
get_write_strobes()

This nonblocking procedure gets the write_strobes field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```
get_write_strobes
(
  write_strobes : out std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
  transaction_id  : in integer;
  bfm_id   : in integer;
  path_id  : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t;
);
```

Arguments

- **write_strobes** Write strobes array.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **write_strobes**

Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write_strobes field of the data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```
**set_resp()**

This nonblocking procedure sets the response `resp` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_slave_transaction()` procedure.

**Prototype**

```
set_resp
(
    resp: in std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **resp**
  
  Transaction response:
  
  AXI4_OKAY = 0;
  AXI4_SLVERR = 2;
  AXI4_DECERR = 3;

- **transaction_id**
  
  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id**
  
  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**
  
  (Optional) Parallel process path identifier:
  
  AXI4_PATH_0
  AXI4_PATH_1
  AXI4_PATH_2
  AXI4_PATH_3
  AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  
  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the read response to AXI_OKAY for the data phase (beat)
-- for the tr_id transaction.
set_resp(AXI4_OKAY, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_resp()

This nonblocking procedure gets a response `resp` field for a transaction uniquely identified by the `transaction_id` field previously created by the `create_slave_transaction()` procedure.

Prototype

```
get_resp
{
  resp: out std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) |
  integer;
  transaction_id : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
};
```

Arguments

- **resp**: Transaction response:
  - AXI4_OKAY = 0;
  - AXI4_SLAVEERR = 2;
  - AXI4_DECERR = 3;
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

`resp`

Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the response field of the data phase (beat)
-- of the tr_id transaction.
get_resp(read_resp, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_read_or_write()

This procedure sets the read_or_write field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

set_read_or_write

( read_or_write: in integer;
  transaction_id : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);

Arguments

read_or_write Read or write transaction:

  AXI4_TRANS_READ = 0
  AXI4_TRANS_WRITE = 1

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

  AXI4_PATH_0
  AXI4_PATH_1
  AXI4_PATH_2
  AXI4_PATH_3
  AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Note

You do not normally use this procedure in a slave test program.
get_read_or_write()

This nonblocking procedure gets the read_or_write field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

**Prototype**

```vhdl
get_read_or_write
  
  
  
  read_or_write: out integer;
  transaction_id  : in integer;
  bfm_id: in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t

);```

**Arguments**

- **read_or_write** Read or write transaction:
  - AXI4_TRANS_READ = 0
  - AXI4_TRANS_WRITE = 1
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

read_or_write

**Example**

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read_or_write field of tr_id transaction.
get_read_or_write(read_or_write, tr_id, bfm_index,
  axi4_tr_if_0(bfm_index));
```
set_gen_write_strobes()

This nonblocking procedure sets the `gen_write_strobes` field for a write transaction that is uniquely identified by the `transaction_id` field previously created by the `create_slave_transaction()` procedure.

Prototype

```vhdl
set_gen_write_strobes

    (gen_write_strobes: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t);
```

Arguments

- `gen_write_strobes` Correction of write strobes for invalid byte lanes:
  - 0 = `write_strobes` passed through to protocol signals.
  - 1 = `write_strobes` auto-corrected for invalid byte lanes (default).
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.
VHDL Slave BFM

get_gen_write_strobes()

get_gen_write_strobes()

This nonblocking procedure gets the `gen_write_strobes` field for a write transaction that is uniquely identified by the `transaction_id` field previously created by the `create_slave_transaction()` procedure.

Prototype

```vhdl
get_gen_write_strobes (  
    gen_write_strobes: out integer;  
    transaction_id : in integer;  
    bfm_id : in integer;  
    path_id : in axi4_path_t; --optional  
    signal tr_if : inout axi4_vhd_if_struct_t  
);  
```

Arguments

- `gen_write_strobes` Correct write strobes flag:
  - `0 = write_strobes` passed through to protocol signals.
  - `1 = write_strobes` auto-corrected for invalid byte lanes.
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns `gen_write_strobes`

Example

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
....

-- Get the auto correction write strobes flag of the tr_id transaction.
get_gen_write_strobes(write_strobes_flag, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_operation_mode()

This nonblocking procedure sets the operation_mode field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

**Prototype**

```
set_operation_mode  
  (  
    operation_mode: in integer;  
    transaction_id : in integer;  
    bfm_id : in integer;  
    path_id : in axi4_path_t; --optional  
    signal tr_if : inout axi4_vhd_if_struct_t  
  );
```

**Arguments**

- **operation_mode** Operation mode:
  - AXI4_TRANSACTION_NON_BLOCKING;
  - AXI4_TRANSACTION_BLOCKING (default);
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns** None

**Example**

```
-- Create a slave transaction. Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the operation mode to nonblocking for the tr_id transaction.
set_operation_mode(AXI4_TRANSACTION_NON_BLOCKING, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_operation_mode()

This nonblocking procedure gets the `operation_mode` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_slave_transaction()` procedure.

**Prototype**

```vhdl
get_operation_mode
(
  operation_mode: out integer,
  transaction_id : in integer,
  bfm_id : in integer,
  path_id : in axi4_path_t, --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **operation_mode** Operation mode:
  - AXI4_TRANSACTION_NON_BLOCKING;
  - AXI4_TRANSACTION_BLOCKING;

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`operation_mode`

**Example**

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the operation mode of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_write_data_mode()

This nonblocking procedure sets the write_data_mode field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```vhdl
set_write_data_mode
(
    write_data_mode: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- `write_data_mode` Write data mode:
  - AXI4_DATA_AFTER_ADDRESS;
    The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a data_valid_delay setting instead to avoid the possible deadlock situation.
  - AXI4_DATA_WITH_ADDRESS; (default)
    The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay.

- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151

- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.
VHDL Slave BFM
get_write_data_mode()

get_write_data_mode()

This nonblocking procedure gets the write_data_mode field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype
get_write_data_mode
{
    write_data_mode: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
};

Arguments
write_data_mode Write data mode:
AXI4_DATA_AFTER_ADDRESS;
The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a data_valid_delay setting instead to avoid the possible deadlock situation.

AXI4_DATA_WITH_ADDRESS; (default)
The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay.

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:
AXI4_PATH_0
AXI4_PATH_1
AXI4_PATH_2
AXI4_PATH_3
AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns
write_data_mode
Example

-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data mode of the tr_id transaction
get_write_data_mode(write_data_mode, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
set_address_valid_delay()

This nonblocking procedure sets the address_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

**Prototype**

```
set_address_valid_delay(
    address_valid_delay: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **address_valid_delay**
  Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction. Default: 0.

- **transaction_id**
  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id**
  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**
  (Optional) Parallel process path identifier:
  
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

---

**Note**

You do not normally use this procedure in a slave test program.
get_address_valid_delay()

This nonblocking procedure gets the address_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

**Prototype**

```vhdl
get_address_valid_delay
(
    address_valid_delay: out integer;
    transaction_id     : in integer;
    bfm_id             : in integer;
    path_id            : in axi4_path_t; --optional
    signal tr_if       : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `address_valid_delay` Address channel ARVALID/AWVALID delay in ACLK cycles for this transaction.
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`address_valid_delay`

**Example**

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the address channel delay of the tr_id transaction.
get_address_valid_delay(address_valid_delay, tr_id, bfm_index,
                        axi4_tr_if_0(bfm_index));
```

---

**Mentor Verification IP AE AXI4-Lite User Guide, V10.4c**

January 2016
get_address_ready_delay()

This nonblocking procedure gets the `address_ready_delay` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_slave_transaction()` procedure.

**Prototype**

```vhdl
get_address_ready_delay
(
  address_ready_delay: out integer;
  transaction_id : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `address_ready_delay` Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction.
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`address_ready_delay`

**Example**

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the address channel *READY delay of the tr_id transaction.
get_address_ready_delay(address_ready_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_valid_delay()

This nonblocking procedure sets the data_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```vhdl
set_data_valid_delay(
    data_valid_delay: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- data_valid_delay: Read data channel array to hold RVALID delays measured in ACLK cycles for this transaction. Default: 0.
- path_id: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- tr_if: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify the transaction.
create_write_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the read channel RVALID delay to 3 ACLK cycles for the data phase (beat) of the tr_id transaction.
set_data_valid_delay(3, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_data_valid_delay()

This nonblocking procedure sets the data_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```vhdl
get_data_valid_delay(
    data_valid_delay: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

data_valid_delay Data channel array to hold RVALID/WVALID delays measured in ACLK cycles for this transaction.

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

- AXI4_PATH_0
- AXI4_PATH_1
- AXI4_PATH_2
- AXI4_PATH_3
- AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

data_valid_delay

Example

```vhdl
-- Create a slave transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
....
-- Get the write channel WVALID delay for the data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_data_ready_delay()

This nonblocking procedure gets the data_ready_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

get_data_ready_delay

(data_ready_delay: out integer;
 transaction_id : in integer;
 bfm_id : in integer;
 path_id : in axi4_path_t; --optional
 signal tr_if : inout axi4_vhd_if_struct_t
);

Arguments

data_ready_delay Data channel array to hold RREADY/WREADY delay measured in ACLK cycles for this transaction.

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:
   AXI4_PATH_0
   AXI4_PATH_1
   AXI4_PATH_2
   AXI4_PATH_3
   AXI4_PATH_4
   Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Example

-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the read data channel RREADY delay for the
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
set_write_response_valid_delay()

This nonblocking procedure sets the `write_response_valid_delay` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_slave_transaction()` procedure.

**Prototype**

```vhdl
set_write_response_valid_delay(
    write_response_valid_delay: in integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `write_response_valid_delay`: Write data channel BVALID delay measured in ACLK cycles for this transaction. Default: 0.
- `path_id`: (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if`: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write response channel BVALID delay to 3 ACLK cycles for the tr_id transaction.
set_write_response_valid_delay(3, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_response_valid_delay()

This nonblocking procedure gets the write_response_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```vhdl
get_write_response_valid_delay(
    write_response_valid_delay: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- `write_response_valid_delay`: Write data channel BVALID delay measured in ACLK cycles for this transaction.
- `path_id`: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if`: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- `write_response_valid_delay`

Example

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BVALID delay of the tr_id transaction.
get_write_response_valid_delay(write_response_valid_delay, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_response_ready_delay()

This nonblocking procedure gets the write_response_ready_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```vhd
get_write_response_ready_delay

write Response Ready Delay: out integer;
transaction_id : in integer;
bfm_id : in integer;
path_id : in axi4_path_t; --optional
signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **write_response_ready_delay**: Write data channel BREADY delay measured in ACLK cycles for this transaction.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **write_response_ready_delay**

Example

```vhd
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write response channel BREADY delay of the tr_id transaction.
get_write_response_ready_delay(write_resp_ready_delay, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_transaction_done()

This nonblocking procedure sets the transaction_done field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

Prototype

```vhdl
set_transaction_done
  (  
     transaction_done : in integer;
     transaction_id   : in integer;
     bfm_id           : in integer;
     path_id          : in axi4_path_t; --optional
     signal tr_if     : inout axi4_vhd_if_struct_t  
  );
```

Arguments

- **transaction_done** Transaction done flag for this transaction
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Set the slave transaction_done flag of the tr_id transaction.
set_transaction_done(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_transaction_done()

This nonblocking procedure gets the transaction_done field for a transaction that is uniquely identified by the transaction_id field previously created by the create_slave_transaction() procedure.

**Prototype**

```
get_transaction_done
(
    transaction_done : out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **transaction_done**: Transaction done flag for this transaction.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

- **transaction_done**

**Example**

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
execute_read_data_phase()

This procedure executes a read data phase that is uniquely identified by the transaction_id argument previously created by the create_slave_transaction() procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record operation_mode field.

The execute_read_data_phase() sets the RVALID protocol signal at the appropriate time defined by the transaction record data_valid_delay field and sets the transaction_done field to 1 when the phase completes.

Prototype

procedure execute_read_data_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);

Arguments

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

    AXI4_PATH_0
    AXI4_PATH_1
    AXI4_PATH_2
    AXI4_PATH_3
    AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Example

-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Execute the read data phase for the tr_id transaction.
execute_read_data_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
execute_write_response_phase()

This procedure executes a write response phase that is uniquely identified by the `transaction_id` argument previously created by the `create_slave_transaction()` procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record `operation_mode` field.

It sets the BVALID protocol signal at the appropriate time defined by the transaction record `write_response_valid_delay` field. It also sets the `transaction_done` field on completion.

**Prototype**

```vhdl
procedure execute_write_response_phase
(
    transaction_id  : in integer;
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_2(bfm_index));

....

-- Execute the write response phase of the tr_id transaction.
execute_write_response_phase(tr_id, bfm_index, axi4_tr_if_2(bfm_index));
```
get_write_addr_phase()

This blocking procedure gets a write address phase uniquely identified by the transaction_id argument previously created by the create_slave_transaction() procedure.

Prototype

```
procedure get_write_addr_phase
{
    transaction_id  : in integer;
    bfm_id          : in integer;
    path_id         : in axi4_path_t; -- Optional
    signal tr_if    : inout axi4_vhd_if_struct_t
};
```

Arguments

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```
-- Create a slave transaction. Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write address phase of the tr_id transaction.
get_write_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_read_addr_phase()

This blocking procedure gets a read address phase uniquely identified by the `transaction_id` argument previously created by the `create_slave_transaction()` procedure.

**Prototype**

```
procedure get_read_addr_phase
(  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; -- Optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `path_id`: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if`: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read address phase of the tr_id transaction.
get_read_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_data_phase()

This blocking procedure gets a write data phase that is uniquely identified by the transaction_id argument previously created by the create_slave_transaction() procedure.

Prototype

procedure get_write_data_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id          : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);

Arguments

transaction_id  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id  (Optional) Parallel process path identifier:

  AXI4_PATH_0
  AXI4_PATH_1
  AXI4_PATH_2
  AXI4_PATH_3
  AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data phase for the first beat of the tr_id transaction.
get_write_data_phase(tr_id, last, bfm_index, axi4_tr_if_0(bfm_index));
 VHDL Slave BFM
 get_read_addr_cycle()

get_read_addr_cycle()

This blocking procedure waits until the read address channel ARVALID signal is asserted.

Prototype

procedure get_read_addr_cycle
(bfm_id : in integer;
path_id : in axi4_adv_path_t; --optional
signal tr_if : inout axi4_vhd_if_struct_t
);

Arguments

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

  AXI4_PATH_5
  AXI4_PATH_6
  AXI4_PATH_7

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Example

// Wait for the ARVALID signal to be asserted.
bfm.get_read_addr_cycle(bfm_index, axi4_tr_if_0(bfm_index));
execute_read_addr_ready()

This procedure executes a read address ready by placing the \textit{ready} argument value onto the ARREADY signal. It will block (default) for one ACLK period.

\textbf{Prototype}

\begin{verbatim}
procedure execute_read_addr_ready
  (  
    ready : in integer;  
    bfm_id : in integer;  
    path_id : in axi4_path_t; --optional  
    signal tr_if : inout axi4_vhd_if_struct_t
  );
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \texttt{transaction_id} Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
  \item \texttt{non\_blocking\_mode} (Optional) Nonblocking mode:
    \begin{itemize}
      \item 0 = Nonblocking
      \item 1 = Blocking (default)
    \end{itemize}
  \item \texttt{bfm\_id} BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
  \item \texttt{path\_id} (Optional) Parallel process path identifier:
    \begin{itemize}
      \item AXI4\_PATH\_0
      \item AXI4\_PATH\_1
      \item AXI4\_PATH\_2
      \item AXI4\_PATH\_3
      \item AXI4\_PATH\_4
    \end{itemize}
    Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
  \item \texttt{tr\_if} Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
\end{itemize}

\textbf{Returns}

None

\textbf{Example}

\begin{verbatim}
-- Set the ARREADY signal to 1 and block for 1 ACLK cycle
execute_read_addr_ready(1, 1, index, AXI4\_PATH\_6, axi4\_tr\_if\_6(index));
\end{verbatim}
get_read_data_ready()

This blocking procedure returns the value of the read data channel RREADY signal using the `ready` argument. It will block for one ACLK period.

Prototype

```vhd
procedure get_read_data_ready
(
  ready : out integer;
  bfm_id : in integer;
  path_id : in axi4_adv_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready</td>
<td>The value of the RREADY signal.</td>
</tr>
<tr>
<td>bfm_id</td>
<td>BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.</td>
</tr>
<tr>
<td>path_id</td>
<td>(Optional) Parallel process path identifier:</td>
</tr>
<tr>
<td></td>
<td>AXI4_PATH_5</td>
</tr>
<tr>
<td></td>
<td>AXI4_PATH_6</td>
</tr>
<tr>
<td></td>
<td>AXI4_PATH_7</td>
</tr>
<tr>
<td>tr_if</td>
<td>Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.</td>
</tr>
</tbody>
</table>

Returns

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready</td>
<td></td>
</tr>
</tbody>
</table>

Example

```vhd
// Get the RREADY signal value
bfm.get_read_data_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_addr_cycle()

This blocking procedure waits until the write address channel AWVALID signal is asserted.

**Prototype**

```vhdl
procedure get_write_addr_cycle
(
  bfm_id : in integer;
  path_id : in axi4_adv_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `path_id`: (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- `tr_if`: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
// Wait for the AWVALID signal to be asserted.
bfm.get_write_addr_cycle(bfm_index, axi4_tr_if_0(bfm_index));
```
VHDL Slave BFM

**execute_write_addr_ready()**

This procedure executes a write address ready by placing the `ready` argument value onto the AWREADY signal. It will block for one ACLK period.

**Prototype**

```vhdl
procedure execute_write_addr_ready
(
    ready : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `transaction_id` (Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.)
- `index` (Optional) Data phase (beat) number.
- `bfm_id` (BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.)
- `path_id` (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
  (Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.)
- `tr_if` (Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.)

**Returns**

None

**Example**

```vhdl
-- Set the AWREADY signal to 1 and block for 1 ACLK cycle
execute_write_addr_ready(1, 1, index, AXI4_PATH_5, axi4_tr_if_5(index));
```
get_write_data_cycle()

This blocking procedure waits until the write data channel WVALID signal is asserted.

Prototype

```vhdl
procedure get_write_data_cycle
(  
  bfm_id : in integer;  
  path_id : in axi4_adv_path_t; --optional  
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **bfm_id**
  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**
  (Optional) Parallel process path identifier:
  
  AXI4_PATH_5
  AXI4_PATH_6
  AXI4_PATH_7

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
// Wait for the WVALID signal to be asserted.
bfm.get_write_data_cycle(bfm_index, axi4_tr_if_0(bfm_index));
```
execute_write_data_ready()

This procedure executes a write data ready by placing the ready argument value onto the WREADY signal. It blocks for one ACLK period.

**Prototype**

```vhdl
procedure execute_write_data_ready
(
    ready : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `non_blocking_mode` (Optional) Nonblocking mode:
  - 0 = Nonblocking
  - 1 = Blocking (default)
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns** None

**Example**

```vhdl
-- Set the WREADY signal to 1 and block for 1 ACLK cycle
execute_write_data_ready(1, 1, index, AXI4_PATH_7, axi4_tr_if_7(index));
```
get_write_resp_ready()

This blocking procedure returns the value of the write response channel BREADY signal using the `ready` argument. It blocks for one ACLK period.

**Prototype**

```vhdl
procedure get_write_resp_ready
(
    ready : out integer;
    bfm_id : in integer;
    path_id : in axi4_adv_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `ready` - The value of the RREADY signal.
- `bfm_id` - BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` - (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` - Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`ready`

**Example**

```vhdl
// Get the BREADY signal value
bfm.get_write_resp_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
push_transaction_id()

This nonblocking procedure pushes a transaction ID into the back of a queue. The transaction is uniquely identified by the transaction_id argument previously created by the create_slave_transaction() procedure. The queue is identified by the queue_id argument.

Prototype

```vhdl
procedure push_transaction_id
  ( transaction_id : in integer;
    queue_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
  );
```

Arguments

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **queue_id**: Queue identifier:
  - AXI4_QUEUE_ID_0
  - AXI4_QUEUE_ID_1
  - AXI4_QUEUE_ID_2
  - AXI4_QUEUE_ID_3
  - AXI4_QUEUE_ID_4
  - AXI4_QUEUE_ID_5
  - AXI4_QUEUE_ID_6
  - AXI4_QUEUE_ID_7
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None
Example

-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Push the transaction record into queue 1 for the tr_id transaction.
push_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
pop_transaction_id()

This nonblocking (unless queue is empty) procedure pops a transaction ID from the front of a queue. The transaction is uniquely identified by the transaction_id argument previously created by the create_slave_transaction() procedure. The queue is identified by the queue_id argument.

If the queue is empty, then it will block until an entry becomes available.

Prototype

procedure pop_transaction_id
(
    transaction_id : in integer;
    queue_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);

Arguments

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

queue_id Queue identifier:

AXI4_QUEUE_ID_0
AXI4_QUEUE_ID_1
AXI4_QUEUE_ID_2
AXI4_QUEUE_ID_3
AXI4_QUEUE_ID_4
AXI4_QUEUE_ID_5
AXI4_QUEUE_ID_6
AXI4_QUEUE_ID_7

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

AXI4_PATH_0
AXI4_PATH_1
AXI4_PATH_2
AXI4_PATH_3
AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None
Example

-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Pop the transaction record from queue 1 for the tr_id transaction.
pop_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index, axi4_tr_if_0(bfm_index));
print()

This nonblocking procedure prints a transaction record that is uniquely identified by the transaction_id argument previously created by the create_slave_transaction() procedure.

Prototype

procedure print
(
    transaction_id  : in integer;
    print_delays : in integer; --optional
    bfm_id       : in integer;
    path_id      : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);

Arguments

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

print_delays (Optional) Print delay values flag:

    0 = do not print the delay values (default).
    1 = print the delay values.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

    AXI4_PATH_0
    AXI4_PATH_1
    AXI4_PATH_2
    AXI4_PATH_3
    AXI4_PATH_4

    Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Example

-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
destruct_transaction()

This blocking procedure removes a transaction record for cleanup purposes and memory management, uniquely identified by the transaction_id argument previously created by the create_slave_transaction() procedure.

Prototype

```
procedure destruct_transaction
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
wait_on()

This blocking procedure waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional count argument waits for the number of events equal to count.

Prototype

```
procedure wait_on

   (phase           : in integer;
    count: in integer; --optional
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

- **phase**
  - Wait for:
    - AXI4_CLOCK_POSEDGE
    - AXI4_CLOCK_NEGEDGE
    - AXI4_CLOCK_ANYEDGE
    - AXI4_CLOCK_0_TO_1
    - AXI4_CLOCK_1_TO_0
    - AXI4_RESET_POSEDGE
    - AXI4_RESET_NEGEDGE
    - AXI4_RESET_ANYEDGE
    - AXI4_RESET_0_TO_1
    - AXI4_RESET_1_TO_0

- **count**
  - (Optional) Wait for a number of events to occur set by count. (default = 1)

- **bfm_id**
  - BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**
  - (Optional) Parallel process path identifier:
    - AXI4_PATH_0
    - AXI4_PATH_1
    - AXI4_PATH_2
    - AXI4_PATH_3
    - AXI4_PATH_4
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  - Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```
wait_on(AXI_RESET_POSEDGE, bfm_index, axi4_tr_if_0(bfm_index));
wait_on(AXI_CLOCK_POSEDGE, 10, bfm_index,
       axi4_tr_if_0(bfm_index));
```
AMBA AXI protocols typically provide a start address only in a transaction, with the following addresses for each byte of data calculated. Helper functions provide you with a simple interface to set and get actual address/data values.

get_write_addr_data()

This nonblocking procedure returns the actual address \( addr \) and data of a particular byte in a write data beat. It also returns the maximum number of bytes \( (\text{dynamic}\_\text{size}) \) in the write data phase (beat). It is used in a slave test program as a helper procedure to store a byte of data at a particular address in the slave memory.

Prototype

```
procedure get_write_addr_data
(
    transaction_id  : in integer;
    byte_index      : in integer;
    dynamic_size    : out integer;
    addr            : out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);
    data            : out std_logic_vector(7 downto 0);
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t);
```

Arguments

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **byte_index** Data byte number in a data phase (beat)
- **dynamic_size** Number of data bytes in a data phase (beat).
- **addr** Data byte address.
- **data** Write data byte.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
VHDL Slave BFM

get_write_addr_data()

--- Returns ---

dynamic_size
addr
data

--- Example ---

-- Wait for a write data phase to complete for the write_trans transaction.
get_write_data_phase(write_trans, index, AXI4_PATH_1, axi4_tr_if_1(index));

-- Get the address, first data byte and byte length for the data phase (beat).
get_write_addr_data(write_trans, 0, byte_length, addr, data, index, AXI4_PATH_1, axi4_tr_if_1(index));

-- Store the first data byte into the slave memory using the slave test program do_byte_write procedure.
do_byte_write(addr, data);

-- Get the remaining bytes of the write data phase (beat) and store them in the slave memory.
if byte_length > 1 then
    for j in 1 to byte_length-1 loop
        get_write_addr_data(write_trans, j, byte_length, addr, data, index, AXI4_PATH_1, axi4_tr_if_1(index));
do_byte_write(addr, data);
    end loop;
end if;
get_read_addr()

This nonblocking procedure returns the actual address addr a particular byte in a read data transaction. It also returns the maximum number of bytes (dynamic_size) in the read data phase (beat). It is used in a slave test program as a helper procedure to return the address of a data byte in the slave memory.

Prototype

```vhdl
procedure get_read_addr
(
  transaction_id  : in integer;
  byte_index      : in integer;
  dynamic_size    : out integer;
  addr            : out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **byte_index**: Data byte number in a data phase (beat)
- **dynamic_size**: Number of data bytes in a data phase (beat).
- **addr**: Data byte address.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **dynamic_size**:
- **addr**: 
Example

-- Get the byte address and number of bytes in the data phase (beat).
get_read_addr(read_trans, 0, byte_length, addr, index, AXI4_PATH_4,
        axi4_tr_if_4(index));

-- Retrieve the first data byte from the slave memory using the
-- slave test program do_byte_read procedure.
do_byte_read(addr, data);

-- Set the first read data byte for the read_trans transaction.
set_read_data(read_trans, 0, byte_length, addr, data, index,
        AXI4_PATH_4, axi4_tr_if_4(index));

-- Loop for the number of bytes in the data phase (beat)
-- given by the byte_length.
if byte_length > 1 then
    for j in 1 to byte_length-1 loop
        -- Get the next read data byte address.
        get_read_addr(read_trans, j, byte_length, addr, index,
           AXI4_PATH_4, axi4_tr_if_4(index));

        -- Retrieve the next data byte from the slave memory using the
        -- slave test program do_byte_read procedure.
do_byte_read(addr, data);

        -- Set the next read data byte for the read_trans transaction.
        set_read_data(read_trans, j, byte_length, addr, data, index,
           AXI4_PATH_4, axi4_tr_if_4(index));
    end loop;
end if;
set_read_data()

This nonblocking procedure sets a read data byte in a read transaction prior to execution. It is used in a slave test program as a helper procedure to set the read data retrieved from the slave memory into the relevant byte of a read data phase.

Prototype

```vhdl
procedure set_read_data
(
    transaction_id : in integer;
    byte_index      : in integer;
    dynamic_size    : in integer;
    addr            : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);
    data            : in std_logic_vector(7 downto 0);
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **byte_index**: Data byte index number of a particular data phase (beat).
- **dynamic_size**: Maximum number of bytes in a particular data phase (beat).
- **addr**: Read address.
- **data**: Read data byte.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **None**
Example

-- Get the byte address and number of bytes in the data phase (beat).
get_read_addr(read_trans, 0, byte_length, addr, index, AXI4_PATH_4,
             axi4_tr_if_4(index));

-- Retrieve the first data byte from the slave memory using the
-- slave test program do_byte_read procedure.
do_byte_read(addr, data);

-- Set the first read data byte for the read_trans transaction.
set_read_data(read_trans, 0, byte_length, addr, data, index,
              AXI4_PATH_4, axi4_tr_if_4(index));

-- Loop for the number of bytes in the data phase (beat)
-- given by the byte_length.
if byte_length > 1 then
  for j in 1 to byte_length-1 loop
    -- Get the next read data byte address.
    get_read_addr(read_trans, j, byte_length, addr, index,
                 AXI4_PATH_4, axi4_tr_if_4(index));

    -- Retrieve the next data byte from the slave memory using the
    -- slave test program do_byte_read procedure.
do_byte_read(addr, data);

    -- Set the next read data byte for the read_trans transaction.
    set_read_data(read_trans, j, byte_length, addr, data, index,
                 AXI4_PATH_4, axi4_tr_if_4(index));
  end loop;
end if;
This chapter provides information about the VHDL monitor BFM. Each BFM has an API containing procedures that configure the BFM and access the dynamic Transaction Record during the lifetime of a transaction.

**Inline Monitor Connection**

The connection of a monitor BFM to a test environment differs from that of a master and slave BFM. It is wrapped in an inline monitor interface and connected inline, between a master and slave, as shown in Figure 10-1. It has separate master and slave ports and monitors protocol traffic between a master and slave. By construction, the monitor has access to all the facilities provided by the monitor BFM.

![Figure 10-1. Inline Monitor Connection Diagram](image)

**Monitor BFM Protocol Support**

The AXI4-Lite monitor BFM supports the AMBA AXI4-Lite protocol with restrictions detailed in “Protocol Restrictions” on page 17.

**Monitor Timing and Events**

For detailed timing diagrams of the protocol bus activity and details of the following monitor BFM API timing and events, refer to the relevant AMBA AXI Protocol Specification chapter, The AMBA AXI Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the monitor BFM does not
contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

### Monitor BFM Configuration

The monitor BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. The BFM has parameters you can use to configure the widths of the address and data signals, and transaction fields to configure timeout factors, and setup and hold times, and so on.

You can change the address and data signal widths from their default settings by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the monitor BFM via a parameter port list of the monitor BFM component.

Table 10-1 lists the parameter names for the address and data, and their default values.

<table>
<thead>
<tr>
<th>Signal Width Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_ADDRESS_WIDTH</td>
<td>Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.</td>
</tr>
<tr>
<td>AXI4_RDATA_WIDTH</td>
<td>Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>AXI4_WDATA_WIDTH</td>
<td>Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.</td>
</tr>
<tr>
<td>index</td>
<td>Uniquely identifies a monitor BFM instance. It must be set to a different value for each monitor BFM in the system. Default: 0.</td>
</tr>
<tr>
<td>READ_ACCEPTANCE_CAPABILITY</td>
<td>The maximum number of outstanding read transactions that can be accepted by the monitor BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
<tr>
<td>WRITE_ACCEPTANCE_CAPABILITY</td>
<td>The maximum number of outstanding write transactions that can be accepted by the monitor BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362. for details. Default: 16.</td>
</tr>
</tbody>
</table>
A monitor BFM has configuration fields that you can set via the `set_config()` function to configure timeout factors, setup and hold times, and so on. You can also get the value of a configuration field via the `get_config()` function. The full list of configuration fields is described in Table 10-2.

### Table 10-2. Monitor BFM Configuration

<table>
<thead>
<tr>
<th>Configuration Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4_CONFIG_SETUP_TIME</td>
<td>The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_HOLD_TIME</td>
<td>The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals. Default: 0.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR</td>
<td>The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_BURST_TIMEOUT_FACTOR</td>
<td>The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY</td>
<td>The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY</td>
<td>The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods. Default: 10000.</td>
</tr>
<tr>
<td>AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY</td>
<td>The maximum timeout duration from the assertion of RVVALID to the assertion of RREADY in clock periods. Default: 10000.</td>
</tr>
</tbody>
</table>
The monitor BFM performs protocol error checking via built-in assertions.

Note

The built-in BFM assertions are independent of programming language and simulator.
Assertion Configuration

By default, all built-in assertions are enabled in the monitor BFM. To globally disable them in the monitor BFM, use the `set_config()` command as the following example illustrates:

```vhdl
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS, 0, bfm_index, axi4_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of `get_config()` and `set_config()` commands on the respective assertion. For example, to disable assertion checking for the AWADDR signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```vhdl
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector, bfm_index, axi4_tr_if_0(bfm_index));

-- Assign the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector(AXI4_AWADDR_CHANGED_BEFORE_AWREADY) := '0';

-- Set the new value of the assertion bit vector
set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector, bfm_index, axi4_tr_if_0(bfm_index));
```

**Note**

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion, following the above code sequence, assign the assertion within the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to “AXI4-Lite Assertions” on page 375.

VHDL Monitor API

This section describes the VHDL Monitor API.
set_config()

This nonblocking procedure sets the configuration of the monitor BFM.

Prototype

```vhdl
procedure set_config
(
    config_name   : in std_logic_vector(7 downto 0);
    config_val    : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)|integer;
    bfm_id        : in integer;
    path_id       : in axi4_path_t; --optional
    signal tr_if  : inout axi4_vhd_if_struct_t
);
```  

Arguments

cfg_name

- **config_name**
  - Configuration name:
    - AXI4_CONFIG_SETUP_TIME
    - AXI4_CONFIG_HOLD_TIME
    - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
    - AXI4_CONFIG_AXI4LITE_axi4
    - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
    - AXI4_CONFIG_ENABLE_ASSERTION
    - AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
    - AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
    - AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
    - AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
    - AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
    - AXI4_CONFIG_SLAVE_START_ADDR
    - AXI4_CONFIG_SLAVE_END_ADDR

config_val

Refer to “Monitor BFM Configuration” on page 284 for description and valid values.

bfm_id

BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id

(Optional) Parallel process path identifier:

- AXI4_PATH_0
- AXI4_PATH_1
- AXI4_PATH_2
- AXI4_PATH_3
- AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if

Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
set_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR, 1000, bfm_index, axi4_tr_if_0(bfm_index));
```
get_config()

This nonblocking procedure gets the configuration of the monitor BFM.

Prototype

```vhdl
procedure get_config
(
    config_name : in std_logic_vector(7 downto 0);
    config_val  : out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)|integer;
    bfm_id      : in integer;
    path_id     : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **config_name** Configuration name:
  - AXI4_CONFIG_SETUP_TIME
  - AXI4_CONFIG_HOLD_TIME
  - AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
  - AXI4_CONFIG_AXI4LITE_axi4
  - AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
  - AXI4_CONFIG_ENABLE_ASSERTION
  - AXI4_CONFIG_MAX_LATENCY_AWVALID_ ASSERTION_TO_AWREADY
  - AXI4_CONFIG_MAX_LATENCY_ARVALID_ ASSERTION_TO_ARREADY
  - AXI4_CONFIG_MAX_LATENCY_RVALID_ ASSERTION_TO_RREADY
  - AXI4_CONFIG_MAX_LATENCY_BVALID_ ASSERTION_TO_BREADY
  - AXI4_CONFIG_MAX_LATENCY_WVALID_ ASSERTION_TO_WREADY
  - AXI4_CONFIG_SLAVE_START_ADDR
  - AXI4_CONFIG_SLAVE_END_ADDR

- **config_val** Refer to “Monitor BFM Configuration” on page 284 for description and valid values.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **config_val**

Example

```vhdl
get_config(AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR, config_value,
           bfm_index, axi4_tr_if_0(bfm_index));
```
create_monitor_transaction()

This nonblocking procedure creates a monitor transaction. All transaction fields default to legal protocol values, unless previously assigned a value. It returns with the transaction_id argument.

Prototype

```vhdl
procedure create_monitor_transaction
(
  transaction_id  : out integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

transaction_id  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id  (Optional) Parallel process path identifier:

- AXI4_PATH_0
- AXI4_PATH_1
- AXI4_PATH_2
- AXI4_PATH_3
- AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Transaction Fields

addr  Start address

prot  Protection:

- AXI4_NORM_SEC_DATA; (default)
- AXI4_PRIV_SEC_DATA;
- AXI4_NORM_NONSEC_DATA;
- AXI4_PRIV_NONSEC_DATA;
- AXI4_NORM_SEC_INST;
- AXI4_PRIV_SEC_INST;
- AXI4_NORM_NONSEC_INST;
- AXI4_PRIV_NONSEC_INST;

data_words  Data words.

write_strobes  Write strobes array:

- Each element 0 or 1.

resp  Response:

- AXI4_OKAY;
- AXI4_SLVERR;
- AXI4_DECERR;

read_or_write  Read or write transaction flag:

- AXI_TRANS_READ;
- AXI_TRANS_WRITE
### Operational Transaction Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_write_strobes</td>
<td>Correction of write strobes for invalid byte lanes:</td>
</tr>
<tr>
<td></td>
<td>0 = write_strobes passed through to protocol signals.</td>
</tr>
<tr>
<td></td>
<td>1 = write_strobes auto-corrected for invalid byte lanes (default).</td>
</tr>
<tr>
<td>operation_mode</td>
<td>Operation mode:</td>
</tr>
<tr>
<td></td>
<td>AXI4 TRANSACTION_NON_BLOCKING;</td>
</tr>
<tr>
<td></td>
<td>AXI4 TRANSACTION_BLOCKING (default);</td>
</tr>
<tr>
<td>write_data_mode</td>
<td>Write data mode:</td>
</tr>
<tr>
<td></td>
<td>AXI4_DATA_AFTER_ADDRESS;</td>
</tr>
<tr>
<td></td>
<td>The master first drives the address phase and, after it completes, it drives the corresponding</td>
</tr>
<tr>
<td></td>
<td>data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait</td>
</tr>
<tr>
<td></td>
<td>for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode</td>
</tr>
<tr>
<td></td>
<td>will force the data transfer to start after the address transfer completes; however, it is</td>
</tr>
<tr>
<td></td>
<td>recommended that you use the **_DATA_WITH_ADDRESS along with a data_valid_delay setting instead to</td>
</tr>
<tr>
<td></td>
<td>avoid the possible deadlock situation.</td>
</tr>
<tr>
<td></td>
<td>AXI4_DATA_WITH_ADDRESS; (default)</td>
</tr>
<tr>
<td></td>
<td>The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and</td>
</tr>
<tr>
<td></td>
<td>then asserts WVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID</td>
</tr>
<tr>
<td></td>
<td>and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay.</td>
</tr>
<tr>
<td>address_valid_delay</td>
<td>Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction. Default: 0.</td>
</tr>
<tr>
<td>data_valid_delay</td>
<td>Data channel RVALID/WVALID delay measured in ACLK cycles for this transaction. Default: 0.</td>
</tr>
<tr>
<td>write_response_valid_delay</td>
<td>Write response channel BVALID delay measured in ACLK cycles for this transaction. Default: 0.</td>
</tr>
<tr>
<td>address_ready_delay</td>
<td>Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction. Default: 0.</td>
</tr>
<tr>
<td>data_ready_delay</td>
<td>Data channel RREADY/WREADY delay measured in ACLK cycles for this transaction. Default: 0.</td>
</tr>
<tr>
<td>write_response_ready_delay</td>
<td>Write data channel BREADY delay measured in ACLK cycles for this transaction. Default: 0.</td>
</tr>
<tr>
<td>transaction_done</td>
<td>Transaction done flag for this transaction</td>
</tr>
<tr>
<td>transaction_id</td>
<td>Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151.</td>
</tr>
</tbody>
</table>

**Returns**
Example

-- Create a monitor transaction
-- Returns the transaction ID (tr_id) for this created transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_3(bfm_index));
**set_addr()**

This nonblocking procedure sets the start address `addr` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_monitor_transaction()` procedure.

**Prototype**

```vhdl
set_addr
  (addr : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0) |
   integer;
  transaction_id : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `addr` Start address of transaction.
- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

---

**Note**

You do not normally use this procedure in a monitor test program.
get_addr()

This nonblocking procedure gets the start address addr field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype  
get_addr  
   (addr : out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0) | integer;  
   transaction_id : in integer;  
   bfm_id : in integer;  
   path_id : in axi4_path_t; --optional  
   signal tr_if : inout axi4_vhd_if_struct_t  
);  

Arguments  
addr Start address of transaction.  
transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.  
bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.  
path_id (Optional) Parallel process path identifier:  
   AXI4_PATH_0  
   AXI4_PATH_1  
   AXI4_PATH_2  
   AXI4_PATH_3  
   AXI4_PATH_4  
   Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.  
tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.  

Returns  
addr Start address of transaction.

Example  
-- Create a monitor transaction. Creation returns tr_id to identify the transaction.  
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the start address addr of the tr_id transaction  
get_addr(addr, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
**set_prot()**

This nonblocking procedure sets the protection *prot* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

**Prototype**

```vhdl
set_prot
(
    prot: in integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **prot**  Burst protection:
  - AXI4_NORM_SEC_DATA (default);
  - AXI4_PRIV_SEC_DATA;
  - AXI4_NORM_NONSEC_DATA;
  - AXI4_PRIV_NONSEC_DATA;
  - AXI4_NORM_SEC_INST;
  - AXI4_PRIV_SEC_INST;
  - AXI4_NORM_NONSEC_INST;
  - AXI4_PRIV_NONSEC_INST;

- **transaction_id**  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id**  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**  (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

- None

---

**Note**

You do not normally use this procedure in a monitor test program.
get_prot()

This nonblocking procedure gets the protection prot field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

get_prot
(
   prot: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi4_path_t; --optional
   signal tr_if : inout axi4_vhd_if_struct_t
);

Arguments

prot
Burst protection:
   AXI4_NORM_SEC_DATA;
   AXI4_PRIV_SEC_DATA;
   AXI4_NORM_NONSEC_DATA;
   AXI4_PRIV_NONSEC_DATA;
   AXI4_NORM_SEC_INST;
   AXI4_PRIV_SEC_INST;
   AXI4_NORM_NONSEC_INST;
   AXI4_PRIV_NONSEC_INST;

transaction_id
Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id
BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id
(Optional) Parallel process path identifier:
   AXI4_PATH_0
   AXI4_PATH_1
   AXI4_PATH_2
   AXI4_PATH_3
   AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if
Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

prot

Example

-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the protection field of the tr_id transaction.
get_prot(prot, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
set_data_words()

This nonblocking procedure sets the data_words field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```
set_data_words
(
    data_words: in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)
    | integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **data_words** Data words.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.
VHDL Monitor BFM

get_data_words()  

This nonblocking procedure gets a data_words field for a transaction that is uniquely identified
by the transactionid field previously created by the create_monitor_transaction() procedure.

Prototype

```vhdl
get_data_words
(
  data_words: out std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0)
| integer;
  transaction_id : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
) ;
```

Arguments

- **data_words**  Data words.
- **transaction_id**  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**  (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **data_words**

Example

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

```vhdl
-- Get the data_words field of the data phase (beat)
-- for the tr_id transaction.
get_data_words(data, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_write_strobes()

This nonblocking procedure sets the write_strobes field array elements for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure and uniquely identified by the transaction_id field.

Prototype

```vhdl
set_write_strobes
(
    write_strobes : in std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **write_strobes** Write strobes.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.
get_write_strobes()

This nonblocking procedure gets the write_strobes field array elements for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```vhdl
get_write_strobes
(
    write_strobes : out std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **write_strobes** Write strobes array.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **write_strobes**

Example

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write_strobes field of the data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
**set_resp()**

This nonblocking procedure sets the response `resp` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_monitor_transaction()` procedure.

**Prototype**

```vhdl
set_resp(
    resp: in std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **resp**
  
  Transaction response:
  
  ```vhdl
  AXI4_OKAY = 0;
  AXI4_SLVERR = 2;
  AXI4_DECERR = 3;
  ```

- **transaction_id**

  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id**

  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**

  (Optional) Parallel process path identifier:
  
  ```vhdl
  AXI4_PATH_0
  AXI4_PATH_1
  AXI4_PATH_2
  AXI4_PATH_3
  AXI4_PATH_4
  ```

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**

  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Note**

You do not normally use this procedure in a monitor test program.
get_resp()

This nonblocking procedure gets a response resp field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

**Prototype**

```vhd
get_resp
    (    
        resp: out std_logic_vector (AXI4_MAX_BIT_SIZE-1 downto 0) | integer;
        transaction_id : in integer;
        bfm_id : in integer;
        path_id : in axi4_path_t; --optional
        signal tr_if : inout axi4_vhd_if_struct_t
    );
```

**Arguments**

- **resp**
  - Transaction response:
    ```vhd
    AXI4_OKAY = 0;
    AXI4_SLVERR = 2;
    AXI4_DECERR = 3;
    ```
- **transaction_id**
  - Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**
  - BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**
  - (Optional) Parallel process path identifier:
    ```vhd
    AXI4_PATH_0
    AXI4_PATH_1
    AXI4_PATH_2
    AXI4_PATH_3
    AXI4_PATH_4
    ```
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**
  - Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

- **resp**

**Example**

```vhd
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the response field of the data phase (beat)
-- of the tr_id transaction.
get resp(read resp, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_read_or_write()

This procedure sets the read_or_write field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype  

```vhdl
set_read_or_write
(  
    read_or_write: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments  

- **read_or_write**  
  Read or write transaction:
  - AXI4_TRANS_READ = 0
  - AXI4_TRANS_WRITE = 1

- **transaction_id**  
  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id**  
  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id**  
  (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**  
  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns  

None

Note  

You do not normally use this procedure in a monitor test program.
get_read_or_write()

This nonblocking procedure gets the read_or_write field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

get_read_or_write
(
    read_or_write: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);

Arguments

read_or_write Read or write transaction:

    AXI4_TRANS_READ = 0
    AXI4_TRANS_WRITE = 1

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

path_id (Optional) Parallel process path identifier:

    AXI4_PATH_0
    AXI4_PATH_1
    AXI4_PATH_2
    AXI4_PATH_3
    AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

read_or_write

Example

    -- Create a monitor transaction. Creation returns tr_id to identify
    -- the transaction.
    create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

    ....

    -- Get the read_or_write field of tr_id transaction.
    get_read_or_write(read_or_write, tr_id, bfm_index,
    axi4_tr_if_0(bfm_index));
set_gen_write_strobes()

This nonblocking procedure sets the `gen_write_strobes` field for a write transaction that is uniquely identified by the `transaction_id` field previously created by the `create_monitor_transaction()` procedure.

Prototype

```vhdl
set_gen_write_strobes
(
    gen_write_strobes: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **gen_write_strobes** Correction of write strobes for invalid byte lanes:
  - 0 = `write_strobes` passed through to protocol signals.
  - 1 = `write_strobes` auto-corrected for invalid byte lanes (default).
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - `AXI4_PATH_0`
  - `AXI4_PATH_1`
  - `AXI4_PATH_2`
  - `AXI4_PATH_3`
  - `AXI4_PATH_4`
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.
get_gen_write_strobes()

This nonblocking procedure gets the gen_write_strobes field for a write transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```vhd
get_gen_write_strobes
(
    gen_write_strobes: out integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **gen_write_strobes**: Correct write strobes flag:
  - 0 = write_strobes passed through to protocol signals.
  - 1 = write_strobes auto-corrected for invalid byte lanes.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- gen_write_strobes

Example

```vhd
-- Create a monitor transaction. Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the auto correction write strobes flag of the tr_id transaction.
get_gen_write_strobes(write_strobes_flag, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_operation_mode()

This nonblocking procedure sets the operation_mode field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```vhdl
set_operation_mode
(
    operation_mode: in integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **operation_mode** Operation mode:
  - AXI4_TRANSACTION_NON_BLOCKING;
  - AXI4_TRANSACTION_BLOCKING (default);
  - AXI4_TRANSACTION_ATOMIC

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the operation mode to nonblocking for the tr_id transaction.
set_operation_mode(AXI4_TRANSACTION_NON_BLOCKING, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_operation_mode()

This nonblocking procedure gets the *operation_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```vhdl
get_operation_mode
(
    operation_mode: out integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **operation_mode** Operation mode:
  
  - AXI4_TRANSACTION_NON_BLOCKING;
  - AXI4_TRANSACTION_BLOCKING;

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

**operation_mode**

Example

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify -- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the operation mode of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_write_data_mode()

This nonblocking procedure sets the write_data_mode field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```vhdl
set_write_data_mode
(
    write_data_mode: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);```

Arguments

- **write_data_mode**: Write data mode:
  - **AXI4_DATA_AFTER_ADDRESS**;
    - The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **DATA_WITH_ADDRESS** along with a data_valid_delay setting instead to avoid the possible deadlock situation.
  - **AXI4_DATA_WITH_ADDRESS**; (default)
    - The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay.

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional): Parallel process path identifier:
  - **AXI4_PATH_0**
  - **AXI4_PATH_1**
  - **AXI4_PATH_2**
  - **AXI4_PATH_3**
  - **AXI4_PATH_4**
  - Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

**Note**

You do not normally use this procedure in a monitor test program.
get_write_data_mode()

This nonblocking procedure gets the write_data_mode field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure

Prototype

```vhdl
get_write_data_mode
(
    write_data_mode: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **write_data_mode** Write data mode:
  - AXI4_DATA_AFTER_ADDRESS; The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a data_valid_delay setting instead to avoid the possible deadlock situation.
  - AXI4_DATA_WITH_ADDRESS; (default) The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on data_valid_delay. If data_valid_delay is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after data_valid_delay.

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

```vhdl
write_data_mode
```
Example

-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data mode of the tr_id transaction
get_write_data_mode(write_data_mode, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
set_address_valid_delay()

This nonblocking procedure sets the address_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```vhdl
set_address_valid_delay(
    address_valid_delay: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **address_valid_delay**: Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction. Default: 0.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.
get_address_valid_delay()

This nonblocking procedure gets the address_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype  

```
get_address_valid_delay  
    (  
        address_valid_delay: out integer;  
        transaction_id : in integer;  
        bfm_id : in integer;  
        path_id : in axi4_path_t; --optional  
        signal tr_if : inout axi4_vhd_if_struct_t  
    );
```

Arguments  

- **address_valid_delay**: Address channel ARVALID/AWVALID delay in ACLK cycles for this transaction.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns  

- **address_valid_delay**

Example

```
-- Create a monitor transaction. Creation returns tr_id to identify  
-- the transaction.  
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the address channel delay of the tr_id transaction.  
get_address_valid_delay(address_valid_delay, tr_id, bfm_index,  
axi4_tr_if_0(bfm_index));
```
get_address_ready_delay()

This nonblocking procedure gets the address_ready_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```
get_address_ready_delay
(
    address_ready_delay: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address_ready_delay</td>
<td>Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction.</td>
</tr>
<tr>
<td>transaction_id</td>
<td>Transaction identifier. Refer to &quot;Overloaded Procedure Common Arguments&quot; on page 151 for more details.</td>
</tr>
<tr>
<td>bfm_id</td>
<td>BFM identifier. Refer to &quot;Overloaded Procedure Common Arguments&quot; on page 151 for more details.</td>
</tr>
<tr>
<td>path_id</td>
<td>(Optional) Parallel process path identifier: AXI4_PATH_0, AXI4_PATH_1, AXI4_PATH_2, AXI4_PATH_3, AXI4_PATH_4</td>
</tr>
<tr>
<td>tr_if</td>
<td>Transaction signal interface. Refer to &quot;Overloaded Procedure Common Arguments&quot; on page 151 for more details.</td>
</tr>
</tbody>
</table>

Returns

`address_ready_delay`

Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the address channel *READY delay of the tr_id transaction.
get_address_ready_delay(address_ready_delay, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
set_data_valid_delay()

This nonblocking procedure sets the `data_valid_delay` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_monitor_transaction()` procedure.

**Prototype**

```vhdl
set_data_valid_delay
(
    data_valid_delay: in integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **data_valid_delay**: Data channel array to hold RVALID/WVALID delays measured in ACLK cycles for this transaction. Default: 0.
- **transaction_id**: Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to "Overloaded Procedure Common Arguments" on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 151 for more details.

**Returns**

None

---

**Note**

You do not normally use this procedure in a monitor test program.
get_data_valid_delay()

This nonblocking procedure sets the data_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```vhdl
get_data_valid_delay(
    data_valid_delay: out integer,
    transaction_id : in integer,
    bfm_id : in integer,
    path_id : in axi4_path_t; --optional
);
```

Arguments

- `data_valid_delay`: Data channel array to hold RVALID/WVALID delays measured in ACLK cycles for this transaction.
- `path_id`: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if`: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns `data_valid_delay`

Example

```vhdl
-- Create a monitor transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
....

-- Get the write channel WVALID delay for the data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```
get_data_ready_delay()

This nonblocking procedure gets the data_ready_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype  
get_data_ready_delay  
{  
data_ready_delay: out integer;  
transaction_id : in integer;  
bfm_id : in integer;  
path_id : in axi4_path_t; --optional  
signal tr_if : inout axi4_vhd_if_struct_t  
};

Arguments  
data_ready_delay Data channel array to hold RREADY/WREADY delay measured in ACLK cycles for this transaction.  
transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.  
bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.  
path_id (Optional) Parallel process path identifier:  

\begin{verbatim}  
AXI4_PATH_0  
AXI4_PATH_1  
AXI4_PATH_2  
AXI4_PATH_3  
AXI4_PATH_4  
\end{verbatim}  

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.  
tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Example

-- Create a monitor transaction. Creation returns tr_id to identify  
-- the transaction.  
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));  

-- Get the read data channel RREADY delay for the  
-- data phase (beat) of the tr_id transaction.  
get_data_ready_delay(data_ready_delay, 0, tr_id, bfm_index,  
axi4_tr_if_0(bfm_index));
get_write_response_valid_delay()

This nonblocking procedure gets the write_response_valid_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```vhdl
get_write_response_valid_delay(
    write_response_valid_delay: out integer;
    transaction_id : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **write_response_valid_delay**: Write data channel BVALID delay measured in ACLK cycles for this transaction.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

**write_response_valid_delay**

Example

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BVALID delay of the tr_id transaction.
get_write_response_valid_delay(write_response_valid_delay, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_response_ready_delay()

This nonblocking procedure gets the write_response_ready_delay field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```
get_write_response_ready_delay(
    write_response_ready_delay: out integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **write_response_ready_delay** Write data channel BREADY delay measured in ACLK cycles for this transaction.
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

write_response_ready_delay

Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BREADY delay of the tr_id transaction.
get_write_response_ready_delay(write_resp_ready_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```
set_transaction_done()

This nonblocking procedure sets the `transaction_done` field for a transaction that is uniquely identified by the `transaction_id` field previously created by the `create_monitor_transaction()` procedures.

**Prototype**

```vhdl
set_transaction_done
(
  transaction_done : in integer;
  transaction_id  : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **transaction_done** Transaction `done` flag for this transaction
- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Note**

You do not normally use this procedure in a monitor test program.
get_transaction_done()

This nonblocking procedure gets the transaction_done field for a transaction that is uniquely identified by the transaction_id field previously created by the create_monitor_transaction() procedure.

Prototype

```
get_transaction_done(
    transaction_done : out integer;
    transaction_id  : in integer;
    bfm_id : in integer;
    path_id : in axi4_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_done**: Transaction done flag for this transaction.
- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

**transaction_done**

Example

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_data_phase()

This blocking procedure gets a read data phase that is uniquely identified by the transaction_id argument previously created by the create_monitor_transaction() procedure.

The get_read_data_phase() sets the transaction_done field to 1 to indicate the whole read transaction has completed.

Prototype

procedure get_read_data_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; --optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);

Arguments

- path_id: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- tr_if: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read data phase of the tr_id transaction.
get_read_data_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
get_write_response_phase()

This blocking procedure gets a write response phase that is uniquely identified by the transaction_id argument previously created by the create_monitor_transaction() procedure.

It sets the transaction_done field to 1 when the phase completes to indicate the whole transaction has completed.

Prototype

```
procedure get_write_response_phase
(
    transaction_id  : in integer;
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response phase for the tr_id transaction.
get_write_response_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_addr_phase()

This blocking procedure gets a write address phase that is uniquely identified by the transaction_id argument previously created by the create_monitor_transaction() procedure.

Prototype

```vhdl
procedure get_write_addr_phase
(
  transaction_id : in integer;
  bfm_id          : in integer;
  path_id         : in axi4_path_t; -- Optional
  signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_id**: Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write address phase of the tr_id transaction.
get_write_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_read_addr_phase()

This blocking procedure gets a read address phase that is uniquely identified by the transaction_id argument previously created by the create_monitor_transaction() procedure.

Prototype  

procedure get_read_addr_phase
{
    transaction_id : in integer;
    bfm_id          : in integer;
    path_id         : in axi4_path_t; -- Optional
    signal tr_if   : inout axi4_vhd_if_struct_t
};

Arguments  

transaction_id Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
bfm_id BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
path_id (Optional) Parallel process path identifier:

    AXI4_PATH_0
    AXI4_PATH_1
    AXI4_PATH_2
    AXI4_PATH_3
    AXI4_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

tr_if Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns None

Example

-- Create a monitor transaction. Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read address phase of the tr_id transaction.
get_read_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
get_write_data_phase()

This blocking procedure gets a write data phase that is uniquely identified by the transaction_id argument previously created by the create_monitor_transaction() procedure.

Prototype

```vhdl
procedure get_write_data_phase
(
    transaction_id  : in integer;
    bfm_id          : in integer;
    path_id          : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data phase for the first beat of the tr_id transaction.
get_write_data_phase(tr_id, last, bfm_index, axi4_tr_if_0(bfm_index));
```
get_rw_transaction()

This blocking procedure gets a complete read/write transaction that is uniquely identified by the transaction_id argument previously created by the create_monitor_transaction() procedure.

Prototype

`procedure get_rw_transaction`

```
  transaction_id: in integer;
  bfm_id: in integer;
  path_id: in axi4_path_t; --optional
  signal tr_if: inout axi4_vhd_if_struct_t
);
```

Arguments

- path_id: (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- tr_if: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the complete tr_id transaction.
get_rw_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
get_read_addr_ready()

This blocking procedure returns the value of the read address channel ARREADY signal using the ready argument. It will block for one ACLK period.

Prototype

```vhdl
procedure get_read_addr_ready
(
  ready : out integer;
  bfm_id : in integer;
  path_id : in axi4_adv_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **ready**: The value of the ARREADY signal.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **ready**

Example

```vhdl
// Get the ARREADY signal value
bfm.get_read_addr_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
get_read_data_ready()

This blocking procedure returns the value of the read data channel RREADY signal using the 
ready argument. It will block for one ACLK period.

Prototype

```vhd
procedure get_read_data_ready
(
    ready : out integer;
    bfm_id : in integer;
    path_id : in axi4_adv_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **ready**: The value of the RREADY signal.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **ready**

Example

```vhd
// Get the RREADY signal value
bfm.get_read_data_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_addr_ready()

This blocking procedure returns the value of the write address channel AWREADY signal using the *ready* argument. It will block for one ACLK period.

**Prototype**

```vhdl
procedure get_write_addr_ready
(  ready : out integer;
  bfm_id : in integer;
  path_id : in axi4_adv_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- **ready** The value of the AWREADY signal.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`ready`

**Example**

```vhdl
// Get the WREADY signal value
bfm.get_write_addr_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
get_write_data_ready()

This blocking procedure returns the value of the write data channel WREADY signal using the ready argument. It will block for one ACLK period.

Prototype

```vhdl
procedure get_write_data_ready
(
    ready : out integer;
    bfm_id : in integer;
    path_id : in axi4_adv_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **ready**: The value of the WREADY signal.
- **bfm_id**: BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**: (Optional) Parallel process path identifier:
  - AXI4_PATH_5
  - AXI4_PATH_6
  - AXI4_PATH_7

Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**: Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

- **ready**

Example

```vhdl
// Get the WREADY signal value
bfm.get_write_data_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
get_writeResp_ready()

This blocking procedure returns the value of the write response channel BREADY signal using the `ready` argument. It blocks for one ACLK period.

**Prototype**

```vhdl
procedure get_write_resp_ready
(
    ready : out integer;
    bfm_id : in integer;
    path_id : in axi4_adv_path_t; --optional
    signal tr_if : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `ready` The value of the RREADY signal.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - `AXI4_PATH_5`
  - `AXI4_PATH_6`
  - `AXI4_PATH_7`
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

`ready`

**Example**

```vhdl
// Get the BREADY signal value
bfm.get_write_resp_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```
push_transaction_id()

This nonblocking procedure pushes a transaction record into the back of a queue. The transaction is uniquely identified by the transaction_id argument previously created by the create_monitor_transaction() procedure. The queue is identified by the queue_id argument.

Prototype

```vhdl
procedure push_transaction_id
(
  transaction_id : in integer;
  queue_id : in integer;
  bfm_id : in integer;
  path_id : in axi4_path_t; --optional
  signal tr_if : inout axi4_vhd_if_struct_t
);
```

Arguments

- **transaction_id** Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **queue_id** Queue identifier:
  - AXI4_QUEUE_ID_0
  - AXI4_QUEUE_ID_1
  - AXI4_QUEUE_ID_2
  - AXI4_QUEUE_ID_3
  - AXI4_QUEUE_ID_4
  - AXI4_QUEUE_ID_5
  - AXI4_QUEUE_ID_6
  - AXI4_QUEUE_ID_7
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id** BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if** Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None
Example

-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Push the transaction record into queue 1 for the tr_id transaction.
push_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
pop_transaction_id()

This nonblocking (unless queue is empty) procedure pops a transaction record from the front of a queue. The transaction is uniquely identified by the `transaction_id` argument previously created by the `get_rw_transaction()` procedure. The queue is identified by the `queue_id` argument.

If the queue is empty, then it will block until an entry becomes available.

**Prototype**

```vhdl
procedure pop_transaction_id
(
    transaction_id  : in integer;
    queue_id  : in integer;
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `queue_id` Queue identifier:
  
  - AXI4_QUEUE_ID_0
  - AXI4_QUEUE_ID_1
  - AXI4_QUEUE_ID_2
  - AXI4_QUEUE_ID_3
  - AXI4_QUEUE_ID_4
  - AXI4_QUEUE_ID_5
  - AXI4_QUEUE_ID_6
  - AXI4_QUEUE_ID_7
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None
Example

-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
....

-- Pop the transaction record from queue 1 for the tr_id transaction.
pop_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
**print()**

This nonblocking procedure prints a transaction record, that is uniquely identified by the `transaction_id` argument previously created by the `create_monitor_transaction()` procedure.

**Prototype**

```vhdl
procedure print
(
    transaction_id  : in integer;
    print_delays : in integer; --optional
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**

- `transaction_id` Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `print_delays` (Optional) Print delay values flag:
  - 0 = do not print the delay values (default).
  - 1 = print the delay values.
- `bfm_id` BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `path_id` (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- `tr_if` Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**

None

**Example**

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
```
destruct_transaction()

This blocking procedure removes a transaction record, for cleanup purposes and memory management, that is uniquely identified by the transaction_id argument previously created by the create_monitor_transaction() procedure.

**Prototype**  
```vhdl
procedure destruct_transaction
(
    transaction_id  : in integer;
    bfm_id          : in integer;
    path_id         : in axi4_path_t; --optional
    signal tr_if    : inout axi4_vhd_if_struct_t
);
```

**Arguments**  
- **transaction_id**  Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **bfm_id**  BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **path_id**  (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4
  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.
- **tr_if**  Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

**Returns**  None

**Example**

```vhdl
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```
wait_on()

This blocking procedure waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional count argument waits for the number of events equal to count.

Prototype

```vhdl
procedure wait_on
  (phase           : in integer;
   count: in integer; --optional
   bfm_id          : in integer;
   path_id         : in axi4_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
  );
```

Arguments

- **phase**
  - Wait for:
    - AXI4_CLOCK_POSEDGE
    - AXI4_CLOCK_NEGEDGE
    - AXI4_CLOCK_ANYEDGE
    - AXI4_CLOCK_0_TO_1
    - AXI4_CLOCK_1_TO_0
    - AXI4_RESET_POSEDGE
    - AXI4_RESET_NEGEDGE
    - AXI4_RESET_ANYEDGE
    - AXI4_RESET_0_TO_1
    - AXI4_RESET_1_TO_0

- **count** (Optional) Wait for a number of events to occur set by count. (default = 1)

- **bfm_id**
  - BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **path_id** (Optional) Parallel process path identifier:
  - AXI4_PATH_0
  - AXI4_PATH_1
  - AXI4_PATH_2
  - AXI4_PATH_3
  - AXI4_PATH_4

  Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

- **tr_if**
  - Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 151 for more details.

Returns

None

Example

```vhdl
wait_on(AXI4_RESET_POSEDGE, bfm_index, axi4_tr_if_0(bfm_index));
wait_on(AXI4_CLOCK_POSEDGE, 10, bfm_index,
       axi4tr_if_0(bfm_index));
```
VHDL Monitor BFM

wait_on()
This chapter discusses how to use the Mentor Verification IP Altera Edition master and slave BFM s to verify slave and master components, respectively.

In the Verifying a Slave DUT tutorial, the slave is an on-chip RAM model that is verified using a master BFM and test program. In the Verifying a Master DUT tutorial, the master issues simple write and read transactions that are verified using a slave BFM and test program.

Following this top-level discussion of how you verify a master and a slave component using the Mentor Verification IP Altera Edition is a brief example of how to run Qsys, the powerful system integration tool in the Quartus II software. This procedure shows you how to use Qsys to create a top-level DUT environment. For more details on this example, refer to “Getting Started with Qsys and the BFM s” on page 359.

Note

Parameters to configure any optional signals, master BFM transaction issuing, and slave BFM acceptance capability, are set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 362 for details of the Qsys Parameter Editor.

Verifying a Slave DUT

A slave DUT component is connected to a master BFM at the signal-level. A master test program, written at the transaction-level, generates stimulus via the master BFM to verify the slave DUT. Figure 11-1 illustrates a typical top-level test bench environment.

Figure 11-1. Slave DUT Top-Level Test Bench Environment

Top-level file

Master test program

Master BFM

On-chip RAM slave
In this example, the master test program also compares the written data with that read back from the slave DUT, reporting the result of the comparison.

A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

**BFM Master Test Program**

A master test program using the master BFM API is capable of creating a wide range of stimulus scenarios to verify a slave DUT. However, this tutorial restricts the master BFM stimulus to write transactions followed by read transactions to the same address, and then compares the read data with the previously written data.

**Note**

For a complete code example of this master test program, refer to the installed Mentor Verification IP Altera Edition kit in the following directory:

`<install_dir>/mentor_vip_ae\axi4lite\qsys-examples\ex1_back_to_back_vhd\master_test_program.vhd`

The master test program contains the following:

- A `create_transactions` process that creates and executes read and write transactions.
- Processes `handle_write_resp_ready` and `handle_read_data_ready` to handle the write response channel BREADY and read data channel RREADY signals, respectively.
- Variables `m_wr_resp_phase_ready_delay` and `m_rd_data_phase_ready_delay` to set the delay of the BREADY and RREADY signals.

The following sections describe the main processes and variables:

**m_wr_resp_phase_ready_delay**

The `m_wr_resp_phase_ready_delay` variable holds the BREADY signal delay. The delay value extends the length of the write response phase by a number of ACLK cycles.

**Example 11-1** below shows the AWREADY signal delayed by two ACLK cycles. You can edit this variable to change the AWREADY signal delay.

**Example 11-1. m_wr_resp_phase_ready_delay**

```vhdl
-- Variable : m_wr_resp_phase_ready_delay
signal m_wr_resp_phase_ready_delay :integer := 2;
```

**m_rd_data_phase_ready_delay**

The `m_rd_data_phase_ready_delay` variable holds the RREADY signal delay. The delay value extends the length of each read data phase (beat) by a number of ACLK cycles.
Example 11-2 below shows the RREADY signal delayed by two ACLK cycles. You can edit this variable to change the RREADY signal delay.

Example 11-2. m_rd_data_phase_ready_delay

```vhdl
-- Variable : m_rd_data_phase_ready_delay
signal m_rd_data_phase_ready_delay : integer := 2;
```

**Configuration and Initialization**

The master test process creates and executes read and write transactions. The whole process runs concurrently with other processes in the test program, using the `path_id = AXI4_PATH_0`. See “Overloaded Procedure Common Arguments” on page 151 for details of `path_id`.

The process waits for the ARESETn signal to be deasserted, followed by a positive ACLK edge, as shown in Example 11-3. This satisfies the protocol requirements in Section A3.1.2 of the AXI Protocol Specification.

Example 11-3. Configuration and Initialization

```vhdl
-- Master test
process
variable tr_id: integer;
variable data_words : std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);
variable lp: line;
begin
    wait_on(AXI4_RESET_0_TO_1, index, axi4_tr_if_0(index));
    wait_on(AXI4_CLOCK_POSEDGE, index, axi4_tr_if_0(index));
```

**Create and Execute Write Transactions**

To generate AXI4-Lite protocol traffic, the master test program must create a transaction before executing it. The code shown in Example 11-4 calls the `create_write_transaction()` procedure, providing only the start address argument of the transaction.

This example has an AXI4-Lite write data bus width of 32 bits; therefore, a single beat of data conveys 4 bytes across the data bus. The call to the `set_data_words()` procedure sets the first element of the `data_words` transaction field with the value 1 on byte lane 1, with a result of `x"0000_0100"`. However, the AXI4-Lite protocol permits narrow transfers with the use of the write strobes signal WSTRB to indicate which byte lane contains valid write data, and therefore indicates to the slave DUT which data byte lane will be written into memory. The write strobes WSTRB signal indicates to the slave which byte lane contains valid write data to be written to the slave memory. Similarly, you can call the `set_write_strobes()` procedure to set the first element of the `write_strobes` transaction field with the value 2, indicating that only byte lane 1 contains valid data. Calling the `execute_transaction()` procedure executes the transaction on the protocol signals.
All other transaction fields default to legal protocol values (see `create_write_transaction()` procedure for details).

**Example 11-4. Create and Execute Write Transactions**

```vhdl
-- 4 x Writes
-- Write data value 1 on byte lanes 1 to address 1.
create_write_transaction(1, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"00000100";
set_data_words(data_words, tr_id, index, axi4_tr_if_0(index));
set_write_strobes(2, tr_id, index, axi4_tr_if_0(index));
report "master_test_program: Writing data (1) to address (1)";

-- By default it will run in Blocking mode
execute_transaction(tr_id, index, axi4_tr_if_0(index));
```

In the complete master test program, three subsequent write transactions are created and executed in a similar manner to Example 11-4.

**Create and Execute Read Transactions**

The code excerpt in **Example 11-5** reads the data that has been previously written into the slave memory. The master test program first creates a read transaction by calling the `create_read_transaction()` procedure, providing only the start address argument.

The read transaction is then executed on the protocol signals by calling the `execute_transaction()` procedure.

The read data is obtained using the `get_data_words()` procedure to get the `data_words` transaction field value. The result of the read data is compared with the expected data, and a message displays the transcript.

**Example 11-5. Create and Execute Read Transactions**

```vhdl
--4 x Reads
--Read data from address 1.
create_read_transaction(1, tr_id, index, axi4_tr_if_0(index));
execute_transaction(tr_id, index, axi4_tr_if_0(index));

get_data_words(data_words, tr_id, index, axi4_tr_if_0(index));
if(data_words(31 downto 0) = x"00000100") then
  report "master_test_program: Read correct data (1) at address (1)";
else
  hwrite(lp, data_words(31 downto 0));
  report "master_test_program: Error: Expected data (1) at address 1, but got " & lp.all;
end if;
```

In the complete master test program, three subsequent read transactions are created and executed in a similar manner to Example 11-5.
handle_write_resp_ready

The handle write response ready process handles the BREADY signal for the write response channel. The whole process runs concurrently with other processes in the test program, using the \texttt{path\_id = AXI4\_PATH\_5}. See “Overloaded Procedure Common Arguments” on page 151 for details of \texttt{path\_id}, as shown in the Example 11-6.

The initial wait for the ARESETn signal to be deactivated, followed by a positive ACLK edge, satisfies the protocol requirement detailed in Section A3.1.2 of the AXI Protocol Specification.

The BREADY signal is deasserted using the nonblocking call to the \texttt{execute\_write\_resp\_ready()} procedure and waits for a write channel response phase to occur with a call to the blocking \texttt{get\_write\_response\_cycle()} procedure. A received write response phase indicates that the BVALID signal has been asserted, triggering the starting point for the delay of the BREADY signal. In a \texttt{loop} it delays the assertion of BREADY based on the setting of the \texttt{m\_wr\_resp\_phase\_ready\_delay} variable. After the delay, another call to the \texttt{execute\_write\_resp\_ready()} procedure to assert the BREADY signal completes the BREADY handling.

**Example 11-6. Process handle_write_resp_ready**

```vhdl
-- handle_write_resp_ready : write response ready through path 5.
-- This method assert/de-assert the write response channel ready signal.
-- Assertion and de-assertion is done based on following variable's value:
-- m_wr_resp_phase_ready_delay
process
  variable tmp_ready_delay : integer;
begin
  wait_on(AXI4_RESET_0_TO_1, index, AXI4_PATH_5, axi4_tr_if_5(index));
  wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_5, axi4_tr_if_5(index));
  loop
    wait until m_wr_resp_phase_ready_delay > 0;
    tmp_ready_delay := m_wr_resp_phase_ready_delay;
    execute_write_resp_ready(0, 1, index, AXI4_PATH_5,
      axi4_tr_if_5(index));
    get_write_response_cycle(index, AXI4_PATH_5, axi4_tr_if_5(index));
    if(tmp_ready_delay > 1) then
      for i in 0 to tmp_ready_delay-2 loop
        wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_5,
          axi4_tr_if_5(index));
      end loop;
    end if;
    execute_write_resp_ready(1, 1, index, AXI4_PATH_5,
      axi4_tr_if_5(index));
  end loop;
  wait;
end process;
```

handle_read_data_ready

The handle read data ready process handles the RREADY signal for the read data channel. It delays the assertion of the RREADY signal based on the setting of the
Verifying a Master DUT

A master DUT component is connected to a slave BFM at the signal-level. A slave test program, written at the transaction-level, generates stimulus via the slave BFM to verify the master DUT. Figure 11-2 illustrates a typical top-level test bench environment.

In this example, the slave test program is a simple memory model.

A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

BFM Slave Test Program

The slave test program is a memory model that contains two APIs:

- **Basic Slave API Definition**

  The Basic Slave API Definition allows you to create a wide range of stimulus scenarios to test a master DUT. This API definition simplifies the creation of slave stimulus based on the default response of OKAY to master read and write transactions.

- **Advanced Slave API Definition**

  The Advanced Slave API Definition allows you to create additional response scenarios to transactions.
basic_slave_API_definition

The Basic Slave Test Program API contains the following elements:

- Procedures `m_wr_addr_phase_ready_delay` and `do_byte_write()` that read and write a byte of data to Internal Memory, respectively.
- Procedures `set_read_data_valid_delay()` and `set_wr_resp_valid_delay()` to configure the delay of the read data channel RVALID, and write response channel BVALID signals, respectively.
- Variables `m_wr_addr_phase_ready_delay` and `m_rd_addr_phase_ready_delay` to configure the delay of the read/write address channel AWVALID/ARVALID signals, and `m_wr_data_phase_ready_delay` to configure the delay of the write response channel BVALID signal.

Internal Memory

The internal memory for the slave is defined as an array of 8 bits, so that each byte of data is stored as an address/data pair.

**Example 11-7. Internal Memory**

```
type memory_t is array (0 to 2**16-1) of std_logic_vector(7 downto 0);
shared variable mem : memory_t;
```

do_byte_read()

The `do_byte_read()` procedure reads a data byte from the Internal Memory `mem` given an address location `addr`, as shown below.

You can edit this procedure to modify the way the read data is extracted from the internal memory.

```
-- Procedure : do_byte_read
-- Procedure to provide read data byte from memory at particular input
-- address
procedure do_byte_read(addr : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0); data : out std_logic_vector(7 downto 0)) is
begin
  data := mem(to_integer(addr));
end do_byte_read;
```
do_byte_write()

The `do_byte_write()` procedure when called writes a data byte to the Internal Memory `mem` given an address location `addr`, as shown below.

You can edit this procedure to modify the way the write data is stored in the internal memory.

```vhdl
-- Procedure : do_byte_write
-- Procedure to write data byte to memory at particular input address
procedure do_byte_write(addr : in std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);  data : in std_logic_vector(7 downto 0)) is
begin
    mem(to_integer(addr)) := data;
end do_byte_write;
```

m_wr_addr_phase_ready_delay

The `m_wr_addr_phase_ready_delay` variable holds the AWREADY signal delay. The delay value extends the length of the write address phase by a number of ACLK cycles. The starting point of the delay is determined by the assertion of the AWVALID signal.

Example 11-8 shows the AWREADY signal delayed by two ACLK cycles. You can edit this variable to change the AWREADY signal delay.

```vhdl
Example 11-8. m_wr_addr_phase_ready_delay

-- Variable : m_wr_addr_phase_ready_delay
signal m_wr_addr_phase_ready_delay : integer := 2;
```

m_rd_addr_phase_ready_delay

The `m_rd_addr_phase_ready_delay` variable holds the ARREADY signal delay. The delay value extends the length of the read address phase by a number of ACLK cycles. The starting point of the delay is determined by the assertion of the ARVALID signal.

Example 11-9 shows the ARREADY signal delayed by two ACLK cycles. You can edit this variable to change the ARREADY signal delay.

```vhdl
Example 11-9. m_rd_addr_phase_ready_delay

-- Variable : m_rd_addr_phase_ready_delay
signal m_rd_addr_phase_ready_delay : integer := 2;
```

m_wr_data_phase_ready_delay

The `m_wr_data_phase_ready_delay` variable holds the WREADY signal delay. The delay value extends the length of each write data phase (beat) by a number of ACLK cycles. The starting point of the delay is determined by the assertion of the WVALID signal.
Example 11-10 shows the WREADY signal delayed by two ACLK cycles. You can edit this function to change the WREADY signal delay.

Example 11-10. m_wr_data_phase_ready_delay

```vhdl
-- Variable : m_wr_data_phase_ready_delay
signal m_wr_data_phase_ready_delay : integer := 2;
```

set_wr_resp_valid_delay()

The set_wr_resp_valid_delay() procedure has two prototypes (path_id is optional), and configures the BVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of the write response phase. The delay value of the BVALID signal is stored in the write_response_valid_delay transaction field.

Example 11-11 shows the BVALID signal delay set to two ACLK cycles. You can edit this function to change the BVALID signal delay.

Example 11-11. set_wr_resp_valid_delay()

```vhdl
-- Procedure : set_wr_resp_valid_delay
-- This is used to set write response phase valid delay to start driving
-- write response phase after specified delay.
procedure set_wr_resp_valid_delay(id : integer; path_id : in axi4_path_t;
signal tr_if : inout axi4_vhd_if_struct_t) is
begin
  set_write_response_valid_delay(2, id, index, path_id, tr_if);
end set_wr_resp_valid_delay;
```

set_read_data_valid_delay()

The set_read_data_valid_delay() procedure has two prototypes (path_id is optional), and configures the RVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of a read data phase (beat). The delay value of the RVALID signal is stored in the data_valid_delay transaction field.

The code below shows the RVALID signal delay set to two ACLK periods. You may edit this function to change the RVALID signal delay.

Example 11-12. set_read_data_valid_delay()

```vhdl
-- Procedure : set_read_data_valid_delay
-- This will set the ready delay for write data phase
procedure set_read_data_valid_delay(id : integer; signal tr_if : inout
axi4_vhd_if_struct_t) is
  variable burst_length : integer;
begin
  set_data_valid_delay(2, id, index, tr_if);
end set_read_data_valid_delay;
```
Notes

In addition to the above variables and procedures, you can configure other aspects of the AXI4-Lite Slave BFM by using these procedures: “set_config()” on page 220 and “get_config()” on page 222.

Using the Basic Slave Test Program API

There is a set of variables and procedures that you can use to create stimulus scenarios based on a memory-model slave with a minimal amount of editing, as described in “Basic Slave API Definition” on page 347.

Consider the following configuration when using the slave test program.

Advanced Slave API Definition

Notes

You are not required to edit the following Advance Slave API unless you require a different response than the default (OKAY) response.

The remaining section of this tutorial presents a walk-through of the Advanced Slave API in the slave test program. It consists of five main processes—process_write, process_read, handle_write, handle_response, and handle_read—in the slave test program, as shown in Figure 11-3. There are additional handle_write_addr_ready, handle_read_addr_ready, and handle_write_data_ready processes to handle the handshake AWREADY, ARREADY, and WREADY signals, respectively.

The Advanced Slave API is capable of handling pipelined transactions. Pipelining can occur when a transaction starts before a previous transaction has completed. Therefore, a write transaction that starts before a previous write transaction has completed can be pipelined. Figure 11-3 shows the write channel having three concurrent write_trans transactions, whereby the get_write_addr_phase[2], get_write_data_phase[1] and execute_write_response_phase[0] are concurrently active on the write address, data and response channels, respectively.

Similarly, a read transaction that starts before a previous read transaction has completed can be pipelined. Figure 11-3 shows the read channel having two concurrent read_trans transactions, whereby the get_read_addr_phase[1] and execute_read_data_phase[0] are concurrently active on the read address and data channels, respectively.
The `process_read` process creates a slave transaction and receives the read address phase. It uses unique path and queue identifiers to work concurrently with other processes.

The maximum number of outstanding read transactions is configured before the processing of read transactions begins an ACLK period after the ARESETn signal is inactive, as shown in Example 11-13.

Each slave transaction has a unique `transaction_id` number associated with it that is automatically incremented for each new slave transaction created. In a loop, the `create_slave_transaction()` procedure call returns the `transaction_id` for the slave BFM, indexed by the `index` argument. A `read_trans` variable is previously defined to hold the `transaction_id`.

A call to the `get_read_addr_phase()` procedure blocks the code until a read address phase has completed. The call to the `push_transaction_id()` procedure pushes `read_trans` into the AXI4_QUEUE_ID_1 queue.
The loop completes and restarts by creating a new slave transaction and blocks for another write address phase to occur.

**Example 11-13. process_read**

```vhdl
-- process_read : read address phase through path 3
-- This process keep receiving read address phase and push
-- the transaction into queue through push_transaction_id API.
process
    variable read_trans: integer;
begin
    wait_on(AXI4_RESET_0_TO_1, index, AXI4_PATH_3,
            axi4_tr_if_3(index));
    wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_3,
            axi4_tr_if_3(index));
    loop
        create_slave_transaction(read_trans, index, AXI4_PATH_3,
                                 axi4_tr_if_3(index));
        get_read_addr_phase(read_trans, index, AXI4_PATH_3,
                             axi4_tr_if_3(index));
        get_config(AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE,
                    tmp_config_num_outstanding_rd_phase, index,
                    AXI4_PATH_3, axi4_tr_if_3(index));
        push_transaction_id(read_trans, AXI4_QUEUE_ID_1, index,
                             AXI4_PATH_3, axi4_tr_if_3(index));
    end loop;
    wait;
end process;
```

**handle_read**

The handle_read process gets read data from the Internal Memory as a phase (beat). It uses unique path and queue identifiers to work concurrently with other processes.

In a loop, the pop_transaction_id() procedure call returns the transaction_id from the queue for the slave BFM, indexed by the index argument, as shown in Example 11-14. A read_trans variable is previously defined to hold the transaction_id. If the queue is empty, then pop_transaction_id() will block until content is available.

The call to set_read_data_valid_delay() configures the RVALID signal delay.

In a loop, the call to the get_read_addr() helper procedure returns the actual address addr for a particular byte location and the byte_length of the data phase (beat). This byte address is used to read the data byte from Internal Memory with the call to do_byte_read(), and the set_read_data() helper procedure sets the byte in the read transaction record. If the returned byte_length>1, then the code performs in the byte_length loop the reading and setting of the read data from internal memory for the whole of the read data phase (beat).

The read data phase is executed over the protocol signals by calling execute_read_data_phase(). The loop completes and restarts by waiting for another transaction_id to be placed into the queue.
Example 11-14. handle_read

end process;-- handle_read : read data and response through path 4
-- This process reads data from memory and send read data/response
process
    variable read_trans: integer;
    variable byte_length : integer;
    variable addr : std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);
    variable data : std_logic_vector(7 downto 0);
begin
    loop
        pop_transaction_id(read_trans, AXI4_QUEUE_ID_1, index, AXI4_PATH_4, axi4_tr_if_4(index));
        set_read_data_valid_delay(read_trans, AXI4_PATH_4, axi4_tr_if_4(index));
        get_read_addr(read_trans, 0, byte_length, addr, index, AXI4_PATH_4, axi4_tr_if_4(index));
        do_byte_read(addr, data);
        set_read_data(read_trans, 0, byte_length, addr, data, index, AXI4_PATH_4, axi4_tr_if_4(index));
        if byte_length > 1 then
            for j in 1 to byte_length-1 loop
                get_read_addr(read_trans, j, byte_length, addr, index, AXI4_PATH_4, axi4_tr_if_4(index));
                do_byte_read(addr, data);
                set_read_data(read_trans, j, byte_length, addr, data, index, AXI4_PATH_4, axi4_tr_if_4(index));
            end loop;
        end if;
    execute_read_data_phase(read_trans, index, AXI4_PATH_4, axi4_tr_if_4(index));
    end loop;
end process;

process_write

The process_write process works in a similar way as previously described for process_read. It uses unique path and queue identifiers to work concurrently with other processes, as shown in Example 11-15.

Example 11-15. process_write

-- process_write : write address phase through path 0
-- This process keep receiving write address phase and push the
-- transaction into queue through push_transaction_id API.
process
    variable write_trans : integer;
begin
    loop
        create_slave_transaction(write_trans, index, axi4_tr_if_0(index));
        get_write_addr_phase(write_trans, index, AXI4_PATH_4, axi4_tr_if_0(index));
        wait;
    wait;
end process;
get_config(AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE,
    tmp_config_num_outstanding_wr_phase, index,
    AXI4_PATH_3, axi4_tr_if_0(index));
    push_transaction_id(write_trans, AXI4_QUEUE_ID_0, index,
        axi4_tr_if_0(index));
    end loop;
    wait;
end process;

**handle_write**

The `handle_write` process works in a similar way to that previously described for `handle_read`. The main difference is that the write transaction handling gets the write data and stores it in the slave test program `Internal Memory`, and adhering to the state of the WSTRB write strobes signals. There is an additional `pop_transaction_id()` into a queue so that the process can send a write response phase for the transaction, as shown in Example 11-16.
Example 11-16. handle_write

-- handle_write : write data phase through path 1
-- This method receive write data phase for write transaction
process
variable write_trans: integer;
variable byte_length : integer;
variable addr : std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);
variable data : std_logic_vector(7 downto 0);
variable last : integer := 0;
begnin
loop
pop_transaction_id(write_trans, AXI4_QUEUE_ID_0, index, AXI4_PATH_1, axi4_tr_if_1(index));

get_write_data_phase(write_trans, 0, last, index, AXI4_PATH_1, axi4_tr_if_1(index));
get_write_addr_data(write_trans, 0, 0, byte_length, addr, data, index, AXI4_PATH_1, axi4_tr_if_1(index));
do_byte_write(addr, data);
if byte_length > 1 then
  for j in 1 to byte_length-1 loop
    get_write_addr_data(write_trans, 0, j, byte_length, addr, data, index, AXI4_PATH_1, axi4_tr_if_1(index));
do_byte_write(addr, data);
  end loop;
end if;
push_transaction_id(write_trans, AXI4_QUEUE_ID_2, index, AXI4_PATH_1, axi4_tr_if_1(index));
end loop;
wait;
end process;

handle_response

The handle_response process sends a response back to the master to complete a write transaction. It uses unique path and queue identifiers to work concurrently with other processes.

In a loop, the pop_transaction_id() procedure call returns the transaction_id from the queue for the slave BFM, indexed by the index argument, as shown in Example 11-17. A write_trans variable is previously defined to hold the transaction_id. If the queue is empty, then push_transaction_id() will block until content is available.

The call to set_wr_resr_valid_delay() sets the BVALID signal delay for the response prior to calling execute_write_response_phase() to execute the response over the protocol signals.
Example 11-17. handle_response

```vhdl
-- handle_response : write response phase through path 2
-- This method sends the write response phase
process
  variable write_trans: integer;
  begin
    loop
      pop_transaction_id(write_trans, AXI4_QUEUE_ID_2, index, AXI4_PATH_2, axi4_tr_if_2(index));
      set_wr_resp_valid_delay(write_trans, AXI4_PATH_2, axi4_tr_if_2(index));
      execute_write_response_phase(write_trans, index, AXI4_PATH_2, axi4_tr_if_2(index));
      tmp_config_num_outstanding_wr_phase :=
        tmp_config_num_outstanding_wr_phase - 1;
    end loop;
    wait;
  end process;
```

handle_write_addr_ready

The `handle_write_addr_ready` process handles the AWREADY signal for the write address channel. It uses a unique path identifier to work concurrently with other processes.

The handling of the AWREADY signal begins an ACLK period after the ARESETn signal is inactive, as shown in Example 11-18. In a loop, the AWREADY signal is deasserted using the nonblocking call to the `execute_write_addr_ready()` procedure and blocks for a write channel address phase to occur with a call to the blocking `get_write_addr_cycle()` procedure. A received write address phase indicates that the AWVALID signal has been asserted, triggering the starting point for the delay of the AWREADY signal by the number of ACLK cycles defined by `m_wr_addr_phase_ready_delay`. Another call to the `execute_write_addr_ready()` procedure to assert the AWREADY signal completes the AWREADY handling.
Example 11-18. handle_write_addr_ready

-- handle_write_addr_ready : write address ready through path 5
-- This method assert/de-assert the write address channel ready signal.
-- Assertion and de-assertion is done based on
m_wr_addr_phase_ready_delay
  process
    variable tmp_ready_delay : integer;
    begin
      wait_on(AXI4_RESET_0_TO_1, index, AXI4_PATH_5, axi4_tr_if_5(index));
      wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_5, axi4_tr_if_5(index));
      loop
        wait until m_wr_addr_phase_ready_delay > 0;
        tmp_ready_delay := m_wr_addr_phase_ready_delay;
        execute_write_addr_ready(0, 1, index, AXI4_PATH_5,
                        axi4_tr_if_5(index));
        get_write_addr_cycle(index, AXI4_PATH_5, axi4_tr_if_5(index));
        if(tmp_ready_delay > 1) then
          for i in 0 to tmp_ready_delay-2 loop
            wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_5,
                        axi4_tr_if_5(index));
          end loop;
        end if;
        execute_write_addr_ready(1, 1, index, AXI4_PATH_5,
                        axi4_tr_if_5(index));
      end loop;
      wait;
    end process;

handle_read_addr_ready

The handle_read_addr_ready process handles the ARREADY signal for the read address channel. It uses a unique path identifier to work concurrently with other processes. The handle_read_addr_ready process code works in a similar way to that previously described for the handle_write_addr_ready process.

handle_write_data_ready

The handle_write_data_ready process handles the WREADY signal for the write data channel. It uses a unique path identifier to work concurrently with other processes.

The handle_write_data_ready process code works in a similar way to that previously described for the handle_write_addr_ready process.
VHDL Tutorials

Verifying a Master DUT
Chapter 12
Getting Started with Qsys and the BFMs

Note
A license is required to access the Mentor Graphics VIP AE Bus Functional Models and Inline Monitor. See “Mentor VIP AE License Requirements” on page 18 for details.

This example shows you how to use the Qsys tool in Quartus II software to create a top-level design environment. You will use the ex1_back_to_back_sv, a SystemVerilog example from the $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/axi4lite/qsys-examples directory in the Altera Complete Design Suite (ACDS) installation.

You will do the following tasks to set up the design environment:

1. Create a work directory.
2. Copy the example to the work directory.
3. Invoke Qsys from the Quartus II software Tools menu.
4. Generate a top-level netlist.
5. Run simulation by referencing the README text file and command scripts for your simulation environment.

Setting Up Simulation from a UNIX Platform

The following steps outline how to set up the simulation environment from a UNIX platform.

1. Create a work directory into which you copy the example directory qsys-examples, which contains the directory ex1_back_to_back_sv from the Installation.
   
   a. Using the mkdir command, create the work directory into which you will copy the qsys-examples directory.

      mkdir axi4lite-qsys-examples

   b. Using the cp command, copy the qsys-examples directory from the Installation directory into your work directory.

      cp -r $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/axi4lite/
      qsys-examples/* axi4lite-qsys-examples/
Getting Started with Qsys and the BFMs

Setting Up Simulation from the Windows GUI

2. Using the `cd` command, change the directory path to your local path where the example resides.

   ```
cd axi4lite-qsys-examples/ex1_back_to_back_sv
   ```

3. Open the Qsys tool. Refer to “Running the Qsys Tool” on page 362 for details.

Setting Up Simulation from the Windows GUI

The following steps outline how to set up the simulation environment from a Windows GUI. This example uses the Windows 7 platform.

1. Create a work folder into which you copy the contents of the `qsys-examples` folder, which includes the `ex1_back_to_back_sv` folder from the Installation.

   a. Using the GUI, select a location for your work folder, then click the `New folder` option on the window’s menu bar to create and name a work folder. For this example, name the work folder `axi4lite-qsys-examples`. Refer to Figures 12-1 and 12-2 below.

   **Figure 12-1. Copy the Contents of `qsys-examples` from the Installation Folder**

   b. Copy the contents of the `qsys-examples` folder from the Installation folder to your work folder.

   Open the `Installation` and work folders. In the `Installation` folder, double-click the `qsys-examples` folder to select and open it. When the folder opens, type `CTRL/A` to
select the contents of the directory, then right-click to display the drop-down menu and select Copy from the drop-down menu.

Go to the open work folder. Double-click on the folder.

When the folder opens, right-click inside the work folder and select Paste from the drop-down menu to copy the contents of the qsys-examples folder to the new axi4lite-qsys-examples work folder.

Paste the qsys-examples from the Installation folder into the axi4lite-qsys-examples work folder (refer to Figure 12-2).

**Figure 12-2. Paste qsys-examples from Installation to Work Folder**

![Figure 12-2](image)

**Note**

Alternatively, open both folders, the Installation folder containing the qsys-examples folder and the new axi4lite-qsys-examples work folder. Use the Windows select, drag, and drop functions to select the contents of the qsys-examples folder in the Installation folder, and then drag the contents to and drop it in the new axi4lite-qsys-examples work folder.

2. After creating the new axi4lite-qsys-examples work folder and copying the contents of the qsys-examples to it, open the Qsys tool. Refer to “Running the Qsys Tool” on page 362 for details.
Running the Qsys Tool

1. Open Qsys in the Quartus II software menu.
   Start the Quartus II software. When the Quartus II GUI appears, select Tools>Qsys (refer to Figure 12-3).

   **Figure 12-3. Select Qsys from the Quartus II Software Top-Level Menu**

   ![Figure 12-3](image)

2. From the Qsys open window, use the File>Open command to open and select the file `ex1_back_to_back_sv.qsys`. This Qsys file is in the directory `axi4lite-qsys-examples/ex1_back_to_back_sv` (refer to Figure 12-4).

   Select and open the `ex1_back_to_back_sv.qsys` example.

   **Figure 12-4. Open the `ex1_back_to_back_sv.qsys` Example**

   ![Figure 12-4](image)

   **Note**
   If you open the Qsys tool in a subsequent session, a Qsys dialog asks you if you want to open this file.
3. Qsys displays the connectivity of the selected example as shown in Figure 12-5.

Figure 12-5. Show System With Qsys Interconnect

Note

If you are using VHDL, you must select each BFM and verify that the index number specified for the BFM is correct. An information dialog displays the properties of the BFM when you select it. Ensure that the specified BFM index is correct in this dialog. If you do not know the correct index number, check the VHDL code for the BFM.
4. Click the System drop-down menu on the main Qsys toolbar, and select Show System With Qsys Interconnect to open the System With Qsys Interconnect options window, as shown in Figure 12-6.

5. Select a BFM within the Hierarchy pane of the System With Qsys Interconnect window, in this case the mgc_axi4lite_slave_0. Click the Parameters tab to reveal the parameter editor to review and change the selected BFM parameters.

---

Note

Placing the mouse pointer over a parameter name, or its value, opens a documentation popup for the parameter. Parameter documentation is also available by clicking the Documentation button.

---

6. Close the System With Qsys Interconnect window after your parameter edits are complete.

7. Click the Generate drop-down menu on the main Qsys toolbar, and select Generate HDL to open the Generation options window.

8. Specify the Generation window options, as shown in Figure 12-7.
   a. Synthesis section
      i. Set the Create HDL design files for synthesis to None to inhibit the generation of synthesis files.
      ii. Uncheck the Create block symbol file (.bsf) check box.
   b. Simulation section
      i. Set the Create simulation model to Verilog.
c. Change the path of the example. In the Path field of the Output Directory section, ensure the path correctly specifies the subdirectory `ex1_back_to_back_sv`, which is the subdirectory containing the example that you just copied into a temporary directory.

**Note**

If the subdirectory name of the example is duplicated in the Path field, you must remove one of the duplicated subdirectory names. To reset the path, double-click the square browse button to the right of the Path field and locate the correct path of the example.

The path name of the example specified in the Path field of the Output Directory section **must be correct before** generating the HDL for the example.

9. Click the *Generate* button on the bottom right side of the window, as shown in Figure 12-7.

**Figure 12-7. Qsys Generation Window Options**

10. Refer to “Running a Simulation” on page 365 to start simulation.

**Running a Simulation**

The choice of simulator determines the process that you follow to run a simulation. The process for each simulator is detailed in the following sections:
For each simulator, a README text file and a command script file is provided in the installed Mentor VIP AE directory location `axi4lite/qsys-examples/ex1_back_to_back_sv`.

Table 12-1 details the README text file instructions to load a model into the simulator, and the script command file to start the simulation.

### Table 12-1. SystemVerilog README Files and Script Names for all Simulators

<table>
<thead>
<tr>
<th>Simulator</th>
<th>README File</th>
<th>Script File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Questa Simulation</td>
<td>README-Questa.txt</td>
<td>example.do</td>
</tr>
<tr>
<td>ModelSim Simulation</td>
<td>README-ModelSim.txt</td>
<td>example.do</td>
</tr>
<tr>
<td>IES Simulation</td>
<td>README-IUS.txt</td>
<td>example-ius.sh</td>
</tr>
<tr>
<td>VCS Simulation</td>
<td>README-VCS.txt</td>
<td>example-vcs.sh</td>
</tr>
</tbody>
</table>

### Note

The VHDL example `axi4lite/qsys-examples/ex1_back_to_back_vhd` has equivalent README text files and command script files. The process to follow for VHDL simulation is similar to that for SystemVerilog simulation.

### ModelSim Simulation

You can run a ModelSim simulation from a GUI interface or a command line. Before starting a simulation, you must do the following:

- Check that the `$QUARTUS_ROOTDIR` environment variable points to the Quartus II software directory in the Quartus II software installation. The example command script `example.do` requires this variable to locate the installed Mentor VIP AE BFMs during simulation.

- Ensure that the environment variable `MvcHome` points to the location of the installed Mentor VIP AE BFM. You can set the location of `MvcHome` using one of the following options:
  - To set the `MvcHome` variable in the `modelsim.ini` file, refer to “Editing the modelsim.ini File” on page 369.”
  - To specify the `-mvchome` option on the command line, refer to “Starting a Simulation from a UNIX Command Line” on page 368.”
The following sections outline how to run a ModelSim simulation from either a GUI or a command line.

**Starting a Simulation from the ModelSim GUI**

To start a simulation with the ModelSim simulator GUI:

1. Start the ModelSim GUI.
   
   ```
   vsim -mvchome $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common
   ```

2. Change the directory to the work directory that contains the example to be simulated with method (a) or (b) below.
   
   a. From the File menu, click the Change Directory option. When the Browse for Folder dialog appears, select the work directory that contains the example.

   ![Figure 12-8. Select the Work Directory](image)

   b. In the ModelSim Transcript window, change to the work directory containing the example to simulate:

   ```
   vsim> cd axi4lite-qsys-examples/ex1_back_to_back_sv
   ```

3. Run the example.do script within the Transcript window by typing the following command to compile and elaborate the test programs:

   ```
   vsim> do example.do
   ```
Getting Started with Qsys and the BFMs
Setting Up Simulation from the Windows GUI

Note
For details about the processing performed by the example.do script, refer to “ModelSim Example Script Processing” on page 368.

4. In the Transcript window, start the simulation and run to completion:

   vsim> run -all

Starting a Simulation from a UNIX Command Line

To start a simulation with the ModelSim simulator from a UNIX command line:

1. Change the directory to the work directory containing the example to be simulated.

   cd axi4lite-qsys-examples/ex1_back_to_back_sv

2. In a shell, start the ModelSim simulator with the example.do script.

   vsim -mvchome $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common -gui -do example.do

   Note
   For details about the processing performed by the example.do script, refer to “ModelSim Example Script Processing” on page 368.

3. In the Transcript window, start the simulation and run to completion.

   vsim> run -all

ModelSim Example Script Processing

The example.do script described below is contained in the installed Mentor VIP AE directory location axi4lite/qsys-examples/ex1_back_to_back_sv.

The Mentor VIP AE BFMs for AXI4-Lite are compiled.

   set TOP_LEVEL_NAME top
   set QSYS_SIMDIR   simulation

   source $QSYS_SIMDIR/mentor/msim_setup.tcl
   if {![info exists env(MENTOR_VIP_AE)]}
   {
      set env(MENTOR_VIP_AE) $env{QUARTUS_ROOTDIR}/../ip/altera/mentor_vip_ae
   }

   ensure_lib libraries
   ensure_lib libraries/work
   vmap work libraries/work

   vlog -work work -sv \
       $env(MENTOR_VIP_AE)/common/questa_mvc_swapi.svh \
The two **tcl** alias commands `dev_com` and `com` compile the required design files. These alias commands are defined in the `msim_setup.tcl` simulation script generated by Qsys, along with the simulation model files:

```tcl
# Compile device library files
dev_com

# Compile Qsys-generated design files
com
```

The three example test programs are compiled:

```tcl
# Compile example test program files
vlog master_test_program.sv
vlog slave_test_program.sv
vlog monitor_test_program.sv
```

The example top-level file is compiled:

```tcl
# Compile top-level design file
vlog top.sv
```

Simulation starts with the `elab` alias defined in the `msim_setup.tcl` simulation script generated by Qsys:

```tcl
# Simulate
elab
```

### Editing the `modelsim.ini` File

The ModelSim simulator does not have a default installation directory path defined for the environment variable `MvcHome`; therefore, you must define a path for this variable.

**Note**

Setting `MvcHome` within the `modelsim.ini` file eliminates the need to specify the `-mvchome` option on the `vsim` command line.

To provide the installation directory path of the Mentor VIP AE for running a ModelSim simulation:

1. Edit the `modelsim.ini` file and find the section that starts with `[vsim]`.
2. Search for `MvcHome`. If it is not already defined in the `modelsim.ini` file, you must add it. You can add this variable at any location in the `[vsim]` section.
If the `modelsim.ini` file is read-only, you must modify the permissions of the file to allow write access.

3. Add or change the `MvcHome` path to point to the location where the Mentor VIP AE is installed. Do not forget the `common` subdirectory.

   \[
   \text{MvcHome} = \$QUARTUS\_ROOTDIR/./ip/altera/mentor\_vip\_ae/common
   \]

**Note**

Do not use the ModelSim `vmap` command to specify the installed location of the Mentor VIP AE because this places the definition of the environment variable `MvcHome` in the `[library]` section of `modelsim.ini`. For example, do not use the command `vmap MvcHome $QUARTUS\_ROOTDIR/./ip/altera/mentor\_vip\_ae/common`.

### Questa Simulation

To run a Questa simulation, follow the process detailed in “ModelSim Simulation” on page 366.

### Cadence IES Simulation

Before starting a Cadence IES simulation, you must do the following:

- Check that the `$QUARTUS\_ROOTDIR` environment variable points to the Quartus II software directory in the Quartus II software installation. The example script `example-ius.sh` requires this variable to locate the Mentor VIP AE BFMs during simulation.

- Set the environment variable `CDS\_ROOT` to the installation directory of the IES Verilog compiler `ncvlog`. The `cds\_root` command returns the installation directory of the specified tool `ncvlog`.

  \[
  \text{setenv CDS\_ROOT} \ `\text{cds\_root ncvlog}`
  \]

### Starting a Simulation from a UNIX Command Line

To start a simulation with the Cadence IES simulator from a UNIX command line:

1. Change the directory to the work directory containing the example to be simulated:

   \[
   \text{cd axi4lite-qsys-examples/ex1\_back\_to\_back\_sv}
   \]

2. Start the Cadence IES simulator with the `example-ius.sh` script.

   - For a 32-bit simulation, execute this command:

     \[
     \text{sh example-ius.sh } 32
     \]

   - For a 64-bit simulation, execute this command:

     \[
     \text{sh example-ius.sh } 64
     \]
Note

For details about the process steps performed by the example-ius.sh script, refer to “Cadence IES Example Script Processing” on page 371.

Cadence IES Example Script Processing

The example-ius.sh script described below is contained in the installed Mentor VIP AE directory location axi4lite/qsys-examples/ex1_back_to_back_sv.

The Mentor VIP AE BFMs for AXI4-Lite are compiled. The ncsim_setup.sh simulation script is generated by Qsys, along with the simulation model files.

```
#!/bin/sh

# Usage: <command> [32|64]
# 32 bit mode is run unless 64 is passed in as the first argument.

MENTOR_VIP_AE=${MENTOR_VIP_AE:-$QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae}

if [ "$1" == "64" ]
then
    export QUESTA_MVC_GCC_LIB=$MENTOR_VIP_AE/common/questa_mvc_core/linux_x86_64_gcc-4.4_ius
    export INCA_64BIT=1
else
    export QUESTA_MVC_GCC_LIB=$MENTOR_VIP_AE/common/questa_mvc_core/linux_gcc-4.4_ius
fi

export LD_LIBRARY_PATH=$QUESTA_MVC_GCC_LIB:$LD_LIBRARY_PATH

cd simulation/cadence
# Run once, just to execute the 'mkdir' for the libraries.
source ncsim_setup.sh SKIP_DEV_COM=1 SKIP_COM=1 SKIP_ELAB=1 SKIP_SIM=1

# Compile VIP
ncvlog -sv \
    "$MENTOR_VIP_AE/common/questa_mvc_svapi.svh" \
    "$MENTOR_VIP_AE/axi4/bfm/mgc_common_axi4.sv" \
    "$MENTOR_VIP_AE/axi4/bfm/mgc_axi4_monitor.sv" \
    "$MENTOR_VIP_AE/axi4/bfm/mgc_axi4_master.sv" \
    "$MENTOR_VIP_AE/axi4/bfm/mgc_axi4_slave.sv"

The three example test programs are compiled:

```
ncvlog -sv ../../../master_test_program.sv
ncvlog -sv ../../../monitor_test_program.sv
ncvlog -sv ../../../slave_test_program.sv
```
The example top-level file is compiled:

```bash
# Compile the top
ncvlog -sv ../../top.sv
```

Elaboration and simulation starts with the `ncsim_setup.sh` command. The Cadence IES simulator requires the SystemVerilog library path `-sv_lib` to be passed to the simulator.

```bash
# Elaborate and simulate
source ncsim_setup.sh \ 
  USER_DEFINED_ELAB_OPTIONS=""-timescale 1ns/1ns"" \ 
  USER_DEFINED_SIM_OPTIONS=""-MESSAGES \ 
    -sv_lib $QUESTA_MVC_GCC_LIB/libaxi_IN_SystemVerilog_IUS_full"" \ 
  TOP_LEVEL_NAME=top"
```

**Synopsys VCS Simulation**

Before starting a Synopsys VCS simulation, you must do the following:

- Ensure that the `$QUARTUS_ROOTDIR` environment variable points to the Quartus II software directory in the Quartus II software installation. The example script `example-vcs.sh` requires this variable to locate the Mentor VIP AE BFMs during simulation.

- Set the environment variable `VCS_HOME` to the installation directory of the VCS Verilog compiler.

  ```bash
  setenv VCS_HOME <Installation-of-VCS>
  ```

**Starting a Simulation from a UNIX Command Line**

To start a simulation with the Synopsys VCS simulator from a UNIX command line:

1. Change the directory to the work directory containing the example to be simulated:

   ```bash
   cd axi4lite-qsys-examples/ex1_back_to_back_sv
   ```

2. Start the Synopsys VCS simulator with the `example-vcs.sh` script. For

   - For a 32-bit simulation, execute this command:
     ```bash
     sh example-vcs.sh 32
     ```
   - For a 64-bit simulation, execute this command:
     ```bash
     sh example-vcs.sh 64
     ```

**Note**

For details about the process steps performed by the `example-vcs.sh` script, refer to “Synopsys VCS Example Script Processing” on page 373.
Synopsys VCS Example Script Processing

The example-vcs.sh script described below is contained in the installed Mentor VIP AE directory location axi4lite/qsys-examples/ex1_back_to_back_sv.

The Mentor VIP AE BFMs for AXI4-Lite are compiled. The vcs_setup.sh simulation script is generated by Qsys, along with the simulation model files.

```bash
#!/bin/sh

# Usage: <command> [32|64]
# 32 bit mode is run unless 64 is passed in as the first argument.

MENTOR_VIP_AE=${MENTOR_VIP_AE:-$QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae}

if [ "$1" == "64" ]
then
  export RUN_64bit=-full64
  export VCS_TARGET_ARCH=`getsimarch 64`
  export LD_LIBRARY_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_64-shared/lib64
  export QUESTA_MVC_GCC_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_64-shared
  export QUESTA_MVC_GCC_LIB=${MENTOR_VIP_AE}/common/questa_mvc_core/linux_x86_64_gcc-4.7.2_vcs
else
  export RUN_64bit=
  export LD_LIBRARY_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_32-shared/lib
  export QUESTA_MVC_GCC_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_32-shared
  export QUESTA_MVC_GCC_LIB=${MENTOR_VIP_AE}/common/questa_mvc_core/linux_gcc-4.7.2_vcs
fi

cd simulation/synopsys/vcs
rm -rf csnc simv simv.daidir transcript ucli.key vc_hdrs.h

# VCS accepts the -LDFLAGS flag on the command line, but the shell quoting
# is too difficult. Just set the LDFLAGS ENV variable for the compiler to
# pick up. Alternatively, use the VCS command line option '-file' with the
# LDFLAGS set (this avoids shell quoting issues).
# vcs-switches.f:
# -LDFLAGS "-L ${QUESTA_MVC_GCC_LIB} -Wl,-rpath ${QUESTA_MVC_GCC_LIB}
# -laxi4_IN_SystemVerilog_VCS_full"
export LDFLAGS="-L ${QUESTA_MVC_GCC_LIB} -Wl, -rpath ${QUESTA_MVC_GCC_LIB}
-laxi4_IN_SystemVerilog_VCS_full"

USER_DEFINED_ELAB_OPTIONS=""

$RUN_64bit \n  +systemverilogext.sv +vpi +acc +vcs+lic+wait \n  -cpp ${QUESTA_MVC_GCC_PATH}/xbin/g++ \n  \n  $MENTOR_VIP_AE/common/questa_mvc_svapi.svh \n  $MENTOR_VIP_AE/axi4/bfm/mgc_common_axi4.sv \n  $MENTOR_VIP_AE/axi4/bfm/mgc_axi4_monitor.sv \n  $MENTOR_VIP_AE/axi4/bfm/mgc_axi4_inline_monitor.sv \n  $MENTOR_VIP_AE/axi4/bfm/mgc_axi4_slave.sv \n  $MENTOR_VIP_AE/axi4/bfm/mgc_axi4_master.sv \n  
```

Mentor Verification IP AE AXI4-Lite User Guide, V10.4c
January 2016
The three example test programs and top-level file are compiled:

```
  ../../../master_test_program.sv \
  ../../../monitor_test_program.sv  \
  ../../../slave_test_program.sv  \
  ../../../top.sv  ""
```

Elaboration and simulation starts with the `vcs_setup.sh` command:

```
source vcs_setup.sh \ 
    USER_DEFINED_ELAB_OPTIONS="$USER_DEFINED_ELAB_OPTIONS" \ 
    USER_DEFINED_SIM_OPTIONS="'-l transcript'" \ 
    TOP_LEVEL_NAME=top
```
AXI4-Lite Assertions

The AXI4-Lite master, slave, and monitor BFM s all support error checking with the firing of one or more assertions when a property defined in the AMBA AXI Protocol Specification has been violated. Each assertion can be individually enabled/disabled using the `set_config()` function for a particular BFM. The property covered for each assertion is noted in under the Property Reference column. The reference number refers to the section number in the AMBA AXI Protocol Specification.

Note

The AXI4-Lite BFM assertions cover the full AXI4 protocol.

### Table A-1. AXI4 Assertions

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60000</td>
<td>AXI4_ADDRESS_WIDTH_EXCEEDS_64</td>
<td>AXI4 supports up to 64-bit addressing.</td>
<td>A10.3.1</td>
</tr>
<tr>
<td>AXI4-60001</td>
<td>AXI4_ADDR_FOR_READ_BURST ACROSS_4K_BOUNDARY</td>
<td>This read transaction has crossed a 4KB boundary.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60002</td>
<td>AXI4_ADDR_FOR_WRITE_BURST ACROSS_4K_BOUNDARY</td>
<td>This write transaction has crossed a 4KB boundary.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60003</td>
<td>AXI4_ARADDR_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARADDR has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60004</td>
<td>AXI4_ARADDR_FALLS_IN_REGION_HOLE</td>
<td>The ARADDR value cannot be decoded to a region in the region map.</td>
<td>A8.2.1</td>
</tr>
<tr>
<td>AXI4-60005</td>
<td>AXI4_ARADDR_UNKN</td>
<td>ARADDR has an X value/ARADDR has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60006</td>
<td>AXI4_ARBURST_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARBURST has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60007</td>
<td>AXI4_ARBURST_UNKN</td>
<td>ARBURST has an X value/ARBURST has a Z value.</td>
<td></td>
</tr>
</tbody>
</table>
Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60008</td>
<td>AXI4_ARCACHE_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARCACHE has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60009</td>
<td>AXI4_ARCACHE_UNKN</td>
<td>ARCACHE has an X value/ARCACHE has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60010</td>
<td>AXI4_ARID_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARID has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60011</td>
<td>AXI4_ARID_UNKN</td>
<td>ARID has an X value/ARID has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60012</td>
<td>AXI4_ARLEN_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARLEN has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60013</td>
<td>AXI4_ARLEN_UNKN</td>
<td>ARLEN has an X value/ARLEN has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60014</td>
<td>AXI4_ARLOCK_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARLOCK has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60015</td>
<td>AXI4_ARLOCK_UNKN</td>
<td>ARLOCK has an X value/ARLOCK has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60016</td>
<td>AXI4_ARPROT_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARPROT has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60017</td>
<td>AXI4_ARPROT_UNKN</td>
<td>ARPROT has an X value/ARPROT has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60018</td>
<td>AXI4_ARQOS_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARQOS has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60019</td>
<td>AXI4_ARQOS_UNKN</td>
<td>ARQOS has an X value/ARQOS has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60020</td>
<td>AXI4_ARREADY_NOT_ASSERTED_AFTER_ARVALID</td>
<td>Once ARVALID has been asserted ARREADY should be asserted in config_max_latency_ARVALID_assertion_to_ARREADY clock periods.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60021</td>
<td>AXI4_ARREADY_UNKN</td>
<td>ARREADY has an X value/ARREADY has a Z value.</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60022</td>
<td>AXI4_ARREGION_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARREGION has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60023</td>
<td>AXI4_ARREGION_MISMATCH</td>
<td>The ARREGION value does not match the value defined in the region map.</td>
<td>A8.2.1</td>
</tr>
<tr>
<td>AXI4-60024</td>
<td>AXI4_ARREGION_UNKN</td>
<td>ARREGION has an X value/ARREGION has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60025</td>
<td>AXI4_ARSIZE_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARSIZE has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60026</td>
<td>AXI4_ARSIZE_UNKN</td>
<td>ARSIZE has an X value/ARSIZE has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60027</td>
<td>AXI4_ARUSER_CHANGED_BEFORE_ARREADY</td>
<td>The value of ARUSER has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60028</td>
<td>AXI4_ARUSER_UNKN</td>
<td>ARUSER has an X value/ARUSER has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60029</td>
<td>AXI4_ARVALID_DEASSERTED_BEFORE_ARREADY</td>
<td>ARVALID has been de-asserted before ARREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60030</td>
<td>AXI4_ARVALID_HIGH_ON_FIRST_CLOCK</td>
<td>A master interface must begin driving ARVALID high only at a rising clock edge after ARESETn is HIGH.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60031</td>
<td>AXI4_ARVALID_UNKN</td>
<td>ARVALID has an X value/ARVALID has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60032</td>
<td>AXI4_AWADDR_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWADDR has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60033</td>
<td>AXI4_AWADDR_FALLS_IN_REGION_HOLE</td>
<td>The addr value cannot be decoded to a region in the region map.</td>
<td>A8.2.1</td>
</tr>
<tr>
<td>AXI4-60034</td>
<td>AXI4_AWADDR_UNKN</td>
<td>AWADDR has an X value/AWADDR has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60035</td>
<td>AXI4_AWBURST_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWBURST has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60036</td>
<td>AXI4_AWBURST_UNKN</td>
<td>AWBURST has an X value/AWBURST has a Z value.</td>
<td></td>
</tr>
</tbody>
</table>
### AXI4-Lite Assertions

#### Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60037</td>
<td>AXI4_AWCACHE_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWCACHE has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60038</td>
<td>AXI4_AWCACHE_UNKN</td>
<td>AWCACHE has an X value/AWCACHE has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60039</td>
<td>AXI4_AWID_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWID has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60040</td>
<td>AXI4_AWID_UNKN</td>
<td>AWID has an X value/AWID has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60041</td>
<td>AXI4_AWLEN_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWLEN has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60042</td>
<td>AXI4_AWLEN_UNKN</td>
<td>AWLEN has an X value/AWLEN has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60043</td>
<td>AXI4_AWLOCK_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWLOCK has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60044</td>
<td>AXI4_AWLOCK_UNKN</td>
<td>AWLOCK has an X value/AWLOCK has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60045</td>
<td>AXI4_AWPROT_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWPROT has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60046</td>
<td>AXI4_AWPROT_UNKN</td>
<td>AWPROT has an X value/AWPROT has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60047</td>
<td>AXI4_AWQOS_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWQOS has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60048</td>
<td>AXI4_AWQOS_UNKN</td>
<td>AWQOS has an X value/AWQOS has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60049</td>
<td>AXI4_AWREADY_NOT_ASSERTED_AFTER_AWVALID</td>
<td>Once AWVALID has been asserted AWREADY should be asserted in (config_max_latency_AWVALID_assertion_to_AWREADY) clock periods.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60050</td>
<td>AXI4_AWREADY_UNKN</td>
<td>AWREADY has an X value/AWREADY has a Z value.</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60051</td>
<td>AXI4_AWREGION_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWREGION has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60052</td>
<td>AXI4_AWREGION_MISMATCH</td>
<td>The AWREGION value does not match the value defined in the region map.</td>
<td>A8.2.1</td>
</tr>
<tr>
<td>AXI4-60053</td>
<td>AXI4_AWREGION_UNKN</td>
<td>AWREGION has an X value/AWREGION has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60054</td>
<td>AXI4_AWSIZE_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWSIZE has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60055</td>
<td>AXI4_AWSIZE_UNKN</td>
<td>AWSIZE has an X value/AWSIZE has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60056</td>
<td>AXI4_AWUSER_CHANGED_BEFORE_AWREADY</td>
<td>The value of AWUSER has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60057</td>
<td>AXI4_AWUSER_UNKN</td>
<td>AWUSER has an X value/AWUSER has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60058</td>
<td>AXI4_AWVALID_DEASSERTED_BEFORE_AWREADY</td>
<td>AWVALID has been de-asserted before AWREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60059</td>
<td>AXI4_AWVALID_HIGH_ON_FIRST_CLOCK</td>
<td>A master interface must begin driving AWVALID high only at a rising clock edge after ARESETn is HIGH.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60060</td>
<td>AXI4_AWVALID_UNKN</td>
<td>AWVALID has an X value/AWVALID has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60061</td>
<td>AXI4_BID_CHANGED_BEFORE_BREADY</td>
<td>The value of BID has changed from its initial value between the time BVALID was asserted and before BREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60062</td>
<td>AXI4_BID_UNKN</td>
<td>BID has an X value/BID has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60063</td>
<td>AXI4_BREADY_NOT_ASSERTED_AFTER_BVALID</td>
<td>When BVALID has been asserted BREADY should be asserted in config_max_latency_BVALID_assertion_to_BREADY clock periods.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60064</td>
<td>AXI4_BREADY_UNKN</td>
<td>BREADY has an X value/BREADY has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60065</td>
<td>AXI4_BRESP_CHANGED_BEFORE_BREADY</td>
<td>The value of BRESP has changed from its initial value between the time BVALID was asserted and before BREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
</tbody>
</table>
### Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60066</td>
<td>AXI4_BRESP_UNKN</td>
<td>BRESP has an X value/BRESP has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60067</td>
<td>AXI4_BUSER_CHANGED_BEFORE_BREADY</td>
<td>The value of BUSER has changed from its initial value between the time BVALID was asserted and before BREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60068</td>
<td>AXI4_BUSER_UNKN</td>
<td>BUSER has an X value/BUSER has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60069</td>
<td>AXI4_BVALID_DEASSERTED_BEFORE_BREADY</td>
<td>BVALID has been de-asserted before BREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60070</td>
<td>AXI4_BVALID_HIGH_EXITING_RESET</td>
<td>BVALID should have been driven low when exiting reset.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60071</td>
<td>AXI4_BVALID_UNKN</td>
<td>BVALID has an X value/BVALID has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60072</td>
<td>AXI4_DEC_ERR_RESP_FOR_READ</td>
<td>No slave at the address for this read transfer (signaled by AXI4_DECERR).</td>
<td></td>
</tr>
<tr>
<td>AXI4-60073</td>
<td>AXI4_DEC_ERR_RESP_FOR_WRITE</td>
<td>No slave at the address for this write transfer (signaled by AXI4_DECERR).</td>
<td></td>
</tr>
<tr>
<td>AXI4-60074</td>
<td>AXI4_EXCLUSIVE_READ_ACCESS_MODIFIABLE</td>
<td>The modifiable bit (bit 1 of the cache parameter) should not be set for an exclusive read access.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60075</td>
<td>AXI4_EXCLUSIVE_READ_BYTES_TRANSFER_EXCEEDS_128</td>
<td>Number of bytes in an exclusive read transaction must be less than or equal to 128.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60076</td>
<td>AXI4_EXCLUSIVE_READ_BYTES_TRANSFER_NOT_POWER_OF_2</td>
<td>Number of bytes of an exclusive read transaction is not a power of 2.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60077</td>
<td>AXI4_EXCLUSIVE_READ_LENGTH_EXCEEDS_16</td>
<td>Exclusive read accesses are not permitted to use a burst length greater than 16.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60078</td>
<td>AXI4_EXCLUSIVE_WR_ADDRESS_NOTSAME_AS_RD</td>
<td>Exclusive write does not match the address of the previous exclusive read to this id.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60079</td>
<td>AXI4_EXCLUSIVE_WR_BURST_NOTSAME_AS_RD</td>
<td>Exclusive write does not match the burst setting of the previous exclusive read to this id.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60080</td>
<td>AXI4_EXCLUSIVE_WR_CACHE_NOTSAME_AS_RD</td>
<td>Exclusive write does not match the cache setting of the previous exclusive read to this id (see the ARM AXI4 compliance-checker AXI4_RECM_EXCL_MATCH assertion code).</td>
<td></td>
</tr>
<tr>
<td>AXI4-60081</td>
<td>AXI4_EXCLUSIVE_WRITE_ACCESS_MODIFIABLE</td>
<td>The modifiable bit (bit 1 of the cache parameter) should not be set for an exclusive write access.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Name</td>
<td>Description</td>
<td>Property Ref</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>AXI4-60082</td>
<td>AXI4_EXCLUSIVE_WR_LENGTH_NOTSAME_AS_RD</td>
<td>Exclusive write does not match the length of the previous exclusive read to this id.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60083</td>
<td>AXI4_EXCLUSIVE_WR_PROT_NOTSAME_AS_RD</td>
<td>Exclusive write does not match the prot setting of the previous exclusive read to this id.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60084</td>
<td>AXI4_EXCLUSIVE_WR_REGION_NOTSAME_AS_RD</td>
<td>Exclusive write does not match the region setting of the previous exclusive read to this id.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60085</td>
<td>AXI4_EXCLUSIVE_WR_SIZE_NOTSAME_AS_RD</td>
<td>Exclusive write does not match the size of the previous exclusive read to this id.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60086</td>
<td>AXI4_EX_OKAY_RESPONSE_NORMAL_READ</td>
<td>Slave has responded AXI4_EXOKAY to a non exclusive read transfer.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60087</td>
<td>AXI4_EX_OKAY_RESPONSE_NORMAL_WRITE</td>
<td>Slave has responded AXI4_EXOKAY to a non exclusive write transfer.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60088</td>
<td>AXI4_EX_RD_EXOKAY_RESPEXPECTED_OKAY</td>
<td>Expected AXI4_OKAY response to this exclusive read (because the parameters did not meet the restrictions) but got AXI4_EXOKAY.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60089</td>
<td>AXI4_EX_RD_EXOKAY_RESPSLAVE_WITHOUT_EXCLUSIVE_ACCESS</td>
<td>Response for an exclusive read to a slave which does not support exclusive access should be AXI4_OKAY but it returned AXI4_EXOKAY.</td>
<td>A7.2.5</td>
</tr>
<tr>
<td>AXI4-60090</td>
<td>AXI4_EX_RD_OKAY_RECVEXPECTED_EXOKAY</td>
<td>Expected AXI4_EXOKAY response to this exclusive read (because the parameters met the restrictions) but got AXI4_OKAY.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60091</td>
<td>AXI4_EX_RD_WHEN_EX_NOTENABLED</td>
<td>An exclusive read should not be issued when exclusive transactions are not enabled.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60092</td>
<td>AXI4_EX_WRITE_BEFORE_EX_READRESPONSE</td>
<td>Exclusive write has occurred with no previous exclusive read.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60093</td>
<td>AXI4_EX_WRITE_EXOKAY_RESPEXPECTED_OKAY</td>
<td>Exclusive write has not been successful but slave has responded with AXI4_EXOKAY.</td>
<td>A7.2.2</td>
</tr>
<tr>
<td>AXI4-60094</td>
<td>AXI4_EX_WRITE_EXOKAY_RESP_SLAVE_WITHOUT_EXCLUSIVE_ACCESS</td>
<td>Response for an exclusive write to a slave which does not support exclusive access should be AXI4_OKAY but it returned AXI4_EXOKAY.</td>
<td>A7.2.5</td>
</tr>
</tbody>
</table>
### AXI4-Lite Assertions

#### Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60095</td>
<td>AXI4_EX_WRITE_OKAY_RESP_EXPECTED_EXOKAY</td>
<td>An AXI4_OKAY response to an exclusive write occurred but an AXI4_EXOKAY response had been expected. If the slave has multiple interfaces to the system this check should be disabled as it is possible for this response to occur as a result of activity on another port.</td>
<td>A7.2.2</td>
</tr>
<tr>
<td>AXI4-60096</td>
<td>AXI4_EX_WR_WHEN_EX_NOT_ENABLED</td>
<td>An exclusive write should not be issued when exclusive transactions are not enabled.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60097</td>
<td>AXI4_ILLEGAL_ARCACHE_VALUE_FOR_CACHEABLE_ADDRESS_REGION</td>
<td>For a read from a cacheable address region one of bits 2 or 3 of the cache parameter must be HIGH.</td>
<td>A4.5</td>
</tr>
<tr>
<td>AXI4-60098</td>
<td>AXI4_ILLEGAL_ARCACHE_VALUE_FOR_NON_CACHEABLE_ADDRESS_REGION</td>
<td>For a read from a non-cacheable address region bits 2 and 3 of the cache parameter must be LOW.</td>
<td>A4.5</td>
</tr>
<tr>
<td>AXI4-60099</td>
<td>AXI4_ILLEGAL_AWCACHE_VALUE_FOR_CACHEABLE_ADDRESS_REGION</td>
<td>For a write to a cacheable address region one of bits 2 or 3 of the cache parameter must be HIGH.</td>
<td>A4.5</td>
</tr>
<tr>
<td>AXI4-60100</td>
<td>AXI4_ILLEGAL_AWCACHE_VALUE_FOR_NON_CACHEABLE_ADDRESS_REGION</td>
<td>For a write to a non-cacheable address region bits 2 and 3 of the cache parameter must be LOW.</td>
<td>A4.5</td>
</tr>
<tr>
<td>AXI4-60101</td>
<td>AXI4_ILLEGAL_LENGTH_FIXED_READ_BURST</td>
<td>In the last read address phase burst_length has an illegal value for a burst of type AXI4_FIXED.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60102</td>
<td>AXI4_ILLEGAL_LENGTH_FIXED_WRITE_BURST</td>
<td>In the last write address phase burst_length has an illegal value for a burst of type AXI4_FIXED.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60103</td>
<td>AXI4_ILLEGAL_LENGTH_WRAPPING_READ_BURST</td>
<td>In the last read address phase burst_length has an illegal value for a burst of type AXI4_WRAP.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60104</td>
<td>AXI4_ILLEGAL_LENGTH_WRAPPING_WRITE_BURST</td>
<td>In the last write address phase burst_length has an illegal value for a burst of type AXI4_WRAP.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60105</td>
<td>AXI4_ILLEGAL_RESPONSE_EXCLUSIVE_READ</td>
<td>Response for an exclusive read should be either AXI4_OKAY or AXI4_EXOKAY.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60106</td>
<td>AXI4_ILLEGAL_RESPONSE_EXCLUSIVE_WRITE</td>
<td>Response for an exclusive write should be either AXI4_OKAY or AXI4_EXOKAY.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60107</td>
<td>AXI4_INVALID_REGION_CARDINALITY</td>
<td>The configuration parameter config_slave_regions does not lie in the range 1-16 inclusive.</td>
<td>A8.2.1</td>
</tr>
<tr>
<td>AXI4-60108</td>
<td>AXI4_INVALID_WRITE_STROBES_ONAligned_WRITE_TRANSFER</td>
<td>Write strobe(s) incorrect for address/size of an aligned transaction.</td>
<td>A3.4.3</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Name</td>
<td>Description</td>
<td>Property Ref</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>AXI4-60109</td>
<td>AXI4_INVALID_WRITE_STROBES_ON_UNALIGNED_WRITE_TRANSFER</td>
<td>Write strobe(s) incorrect for address/size of an unaligned transaction.</td>
<td>A3.4.3</td>
</tr>
<tr>
<td>AXI4-60110</td>
<td>AXI4_MINIMUM_SLAVE_ADDRESS_SPACE_VIOLATION</td>
<td>The minimum address space occupied by a single slave device is 4 kilobytes.</td>
<td>A10.3.2</td>
</tr>
<tr>
<td>AXI4-60111</td>
<td>AXI4_NON_INCREASING_REGION_SPECIFICATION</td>
<td>A region address-range has an upper bound smaller than the lower bound.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60112</td>
<td>AXI4_NON_ZERO_ARQOS</td>
<td>The master is configured to not participate in the Quality-of-Service scheme but ARQOS is not 4'b0000 as it should be.</td>
<td>A8.1.2</td>
</tr>
<tr>
<td>AXI4-60113</td>
<td>AXI4_NON_ZERO_AWQOS</td>
<td>The master is configured to not participate in the Quality-of-Service scheme but AWQOS is not 4'b0000 as it should be.</td>
<td>A8.1.2</td>
</tr>
<tr>
<td>AXI4-60114</td>
<td>AXI4_OVERLAPPING_REGION</td>
<td>An address-range in the region map overlaps with another address in the region map.</td>
<td>A8.2.1</td>
</tr>
<tr>
<td>AXI4-60115</td>
<td>AXI4_PARAM_READ_DATA_BUS_WIDTH</td>
<td>The value of AXI4_RDATA_WIDTH must be one of 8,16,32,64,128,256,512, or 1024.</td>
<td>A1.3.1</td>
</tr>
<tr>
<td>AXI4-60116</td>
<td>AXI4_PARAM_READ_REORDERING_DEPTH_EQUALS_ZERO</td>
<td>The user-supplied config_read_data_reordering_depth should be greater than zero.</td>
<td>A5.3.1</td>
</tr>
<tr>
<td>AXI4-60117</td>
<td>AXI4_PARAM_READ_REORDERING_DEPTH_EXCEEDS_MAX_ID</td>
<td>The user-supplied config_read_data_reordering_depth exceeds the maximum possible value as defined by the AXI4_ID_WIDTH parameter.</td>
<td>A5.3.1</td>
</tr>
<tr>
<td>AXI4-60118</td>
<td>AXI4_PARAM_WRITE_DATA_BUS_WIDTH</td>
<td>The value of AXI4_WDATA_WIDTH must be one of 8,16,32,64,128,256,512, or 1024.</td>
<td>A1.3.1</td>
</tr>
<tr>
<td>AXI4-60119</td>
<td>AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_12</td>
<td>The RA bit of the cache parameter should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60120</td>
<td>AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_13</td>
<td>The RA bit of the cache parameter should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60121</td>
<td>AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_4</td>
<td>The RA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60122</td>
<td>AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_5</td>
<td>The RA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
</tbody>
</table>
### AXI4-Lite Assertions

#### Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60123</td>
<td>AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_8</td>
<td>The RA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60124</td>
<td>AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_9</td>
<td>The RA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60125</td>
<td>AXI4_READ_BURST_LENGTH_VIOLATION</td>
<td>The burst_length implied by the number of beats actually read does not match the burst_length defined by the master_read_addr_channel_phase.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60126</td>
<td>AXI4_READ_BURST_MAXIMUM_LENGTH_VIOLATION</td>
<td>256 read data beats were seen without RLAST.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60127</td>
<td>AXI4_READ_BURST_SIZE_VIOLATION</td>
<td>In this read transaction, size has been set too high for the defined data buswidth.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60128</td>
<td>AXI4_READ_DATA_BEFORE_ADDRESS</td>
<td>An unexpected read response has occurred (there are no outstanding read transactions with this id).</td>
<td>A3.3.1</td>
</tr>
<tr>
<td>AXI4-60129</td>
<td>AXI4_READ_DATA_CHANGED_BEFORE_RREADY</td>
<td>The value of RDATA has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60130</td>
<td>AXI4_READ_DATA_UNKN</td>
<td>RDATA has an X value/RDATA has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60131</td>
<td>AXI4_READ_EXCLUSIVE_ENCODING_VIOLATION</td>
<td>A read-only interface does not support exclusive accesses.</td>
<td>A10.2.2</td>
</tr>
<tr>
<td>AXI4-60132</td>
<td>AXI4_READ_REORDERING_VIOLATION</td>
<td>The arrival of a read response has exceeded the read reordering depth.</td>
<td>A5.3.1</td>
</tr>
<tr>
<td>AXI4-60133</td>
<td>AXI4_READ_RESP_CHANGED_BEFORE_RREADY</td>
<td>The value of RRESP has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60134</td>
<td>AXI4_READ_TRANSFER_EXCEEDS_ADDRESS_SPACE</td>
<td>This read transfer runs off the edge of the address space defined by AXI4_ADDRESS_WIDTH.</td>
<td>A10.3.1</td>
</tr>
<tr>
<td>AXI4-60135</td>
<td>AXI4_REGION_SMALLER_THAN_4KB</td>
<td>An address-range in the region map is smaller than 4kB.</td>
<td>A8.2.1</td>
</tr>
<tr>
<td>AXI4-60136</td>
<td>AXI4_RESERVED_ARBURST_ENCODING</td>
<td>The reserved encoding of 2'b11 should not be used for ARBURST.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60137</td>
<td>AXI4_RESERVED_AWBURST_ENCODING</td>
<td>The reserved encoding of 2'b11 should not be used for AWBURST.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60138</td>
<td>AXI4_RID_CHANGED_BEFORE_RREADY</td>
<td>The value of RID has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
</tbody>
</table>
Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60139</td>
<td>AXI4_RID_UNKN</td>
<td>RID has an X value/RID has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60140</td>
<td>AXI4_RLAST_CHANGED_BEFORE_RREADY</td>
<td>The value of RLAST has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60141</td>
<td>AXI4_RLAST_UNKN</td>
<td>RLAST has an X value/RLAST has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60142</td>
<td>AXI4_RREADY_NOT_ASSERTED_AFTER_RVALID</td>
<td>Once RVALID has been asserted RREADY should be asserted in config_max_latency_RVALID_assertion_to_RREADY clock periods.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60143</td>
<td>AXI4_RREADY_UNKN</td>
<td>RREADY has an X value/RREADY has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60144</td>
<td>AXI4_RRESP_UNKN</td>
<td>RRESP has an X value/RRESP has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60145</td>
<td>AXI4_RUSER_CHANGED_BEFORE_RREADY</td>
<td>The value of RUSER has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60146</td>
<td>AXI4_RUSER_UNKN</td>
<td>RUSER has an X value/RUSER has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60147</td>
<td>AXI4_RVALID_DEASSERTED_BEFORE_RREADY</td>
<td>RVALID has been de-asserted before RREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60148</td>
<td>AXI4_RVALID_HIGH_EXITING_RESET</td>
<td>RVALID should have been driven low when exiting reset.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60149</td>
<td>AXI4_RVALID_UNKN</td>
<td>RVALID has an X value/RVALID has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60150</td>
<td>AXI4_SLV_ERR_RESP_FOR_READ</td>
<td>Slave has detected an error for this read transfer (signaled by AXI4_SLVERR).</td>
<td></td>
</tr>
<tr>
<td>AXI4-60151</td>
<td>AXI4_SLV_ERR_RESP_FOR_WRITE</td>
<td>Slave has detected an error for this write transfer (signaled by AXI4_SLVERR).</td>
<td></td>
</tr>
<tr>
<td>AXI4-60152</td>
<td>AXI4_TIMEOUT_WAITING_FOR_READ_RESPONSE</td>
<td>Timed-out waiting for a read response.</td>
<td>A4.6</td>
</tr>
<tr>
<td>AXI4-60153</td>
<td>AXI4_TIMEOUT_WAITING_FOR_WRITE_RESPONSE</td>
<td>Timed-out waiting for a write response.</td>
<td>A4.6</td>
</tr>
<tr>
<td>AXI4-60154</td>
<td>AXI4_UNALIGNED_ADDRESS_FOR_EXCLUSIVE_READ</td>
<td>Exclusive read accesses must have address aligned to the total number of bytes in the transaction.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60155</td>
<td>AXI4_UNALIGNED_ADDR_FOR_WRAPPING_READ_BURST</td>
<td>Wrapping bursts must have address aligned to the start of the read transfer.</td>
<td>A3.4.1</td>
</tr>
</tbody>
</table>
### Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60156</td>
<td>AXI4_UNALIGNED_ADDR_FOR_WRAPPING_WRITE_BURST</td>
<td>Wrapping bursts must have address aligned to the start of the write transfer.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60157</td>
<td>AXI4_WDATA_CHANGED_BEFORE_WREADY_ON_INVALID_LANE</td>
<td>On a lane whose strobe is 0, the value of WDATA has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60158</td>
<td>AXI4_WDATA_CHANGED_BEFORE_WREADY_ON_VALID_LANE</td>
<td>On a lane whose strobe is 1, the value of WDATA has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60159</td>
<td>AXI4_WLAST_CHANGED_BEFORE_WREADY</td>
<td>The value of WLAST has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60160</td>
<td>AXI4_WLAST_UNKN</td>
<td>WLAST has an X value/WLAST has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60161</td>
<td>AXI4_WREADY_NOT_ASSERTED_AFTER_WVALID</td>
<td>Once WVALID has been asserted, WREADY should be asserted in config_max_latency_WVALID_assertion_to_WREADY clock periods.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60062</td>
<td>AXI4_WREADY_UNKN</td>
<td>WREADY has an X value/WREADY has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60163</td>
<td>AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_12</td>
<td>The WA bit of the cache parameter should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60164</td>
<td>AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_13</td>
<td>The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60165</td>
<td>AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_4</td>
<td>The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60166</td>
<td>AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_5</td>
<td>The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60167</td>
<td>AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_8</td>
<td>The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60168</td>
<td>AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_9</td>
<td>The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.</td>
<td>A4.4</td>
</tr>
<tr>
<td>AXI4-60169</td>
<td>AXI4_WRITE_BURST_LENGTH_VIOLATION</td>
<td>The number of data beats in a write transfer should match the value given by AWLEN.</td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Name</td>
<td>Description</td>
<td>Property Ref</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>AXI4-60170</td>
<td>AXI4_WRITE_STROBES_LENGTH_VIOLATION</td>
<td>The size of the write_strobes array in a write transfer should match the value given by AWLEN.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60171</td>
<td>AXI4_WRITE_USER_DATA_LENGTH_VIOLATION</td>
<td>The size of the wdata_user_data array in a write transfer should match the value given by AWLEN.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60172</td>
<td>AXI4_WRITE_BURST_MAXIMUM_LENGTH_VIOLATION</td>
<td>256 write data beats were seen without WLAST.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60173</td>
<td>AXI4_WRITE_BURST_SIZE_VIOLATION</td>
<td>In this write transaction size has been set too high for the defined data buswidth.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60174</td>
<td>AXI4_WRITE_DATA_BEFORE_ADDRESS</td>
<td>A write data beat has occurred before the corresponding address phase.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60175</td>
<td>AXI4_WRITE_DATA_UNKN_ON_INVALID_LANE</td>
<td>On a lane whose strobe is 0 WDATA has an X value/WDATA has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60176</td>
<td>AXI4_WRITE_DATA_UNKN_ON_VALID_LANE</td>
<td>On a lane whose strobe is 1 WDATA has an X value/WDATA has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60177</td>
<td>AXI4_WRITE_EXCLUSIVE_ENCODING_VIOLATION</td>
<td>A write-only interface does not support exclusive accesses.</td>
<td>A10.2.3</td>
</tr>
<tr>
<td>AXI4-60178</td>
<td>AXI4_WRITE_RESPONSE_WITHOUT_ADDR_DATA</td>
<td>An unexpected write response has occurred (there are no outstanding write transactions with this id).</td>
<td></td>
</tr>
<tr>
<td>AXI4-60179</td>
<td>AXI4_WRITE_STROBE_FIXED_BURST_VIOLATION</td>
<td>Write strobe(s) incorrect for the address/size of a fixed transfer.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60180</td>
<td>AXI4_WRITE_TRANSFER_EXCEEDS_ADDRESS_SPACE</td>
<td>This write transfer runs off the edge of the address space defined by AXI4_ADDRESS_WIDTH.</td>
<td>A10.3.1</td>
</tr>
<tr>
<td>AXI4-60181</td>
<td>AXI4_WRONG_ARREGION_FOR_SLAVE_WITH_SINGLE_ADDRESS_DECODE</td>
<td>The region value should be 4'b0000 for a read from a slave with a single address decode in the region map.</td>
<td>A8.2.1</td>
</tr>
<tr>
<td>AXI4-60182</td>
<td>AXI4_WRONG_AWREGION_FOR_SLAVE_WITH_SINGLE_ADDRESS_DECODE</td>
<td>The region value should be 4'b0000 for a write to a slave with a single address decode in the region map.</td>
<td>A8.2.1</td>
</tr>
<tr>
<td>AXI4-60183</td>
<td>AXI4_WSTRB_CHANGED_BEFORE_WREADY</td>
<td>The value of WSTRB has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60184</td>
<td>AXI4_WSTRB_UNKN</td>
<td>WSTRB has an X value/WSTRB has a Z value.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60185</td>
<td>AXI4_WUSER_CHANGED_BEFORE_WREADY</td>
<td>The value of WUSER has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
</tbody>
</table>
Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60186</td>
<td>AXI4_WUSER_UNKN</td>
<td>WUSER has an X value/WUSER has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60187</td>
<td>AXI4_WVALID_DEASSERTED_BEFORE_WREADY</td>
<td>WVALID has been de-asserted before WREADY was asserted.</td>
<td>A3.2.1</td>
</tr>
<tr>
<td>AXI4-60188</td>
<td>AXI4_WVALID_HIGH_ON_FIRST_CLOCK</td>
<td>A master interface must begin driving WVALID high only at a rising clock edge after ARESETn is HIGH.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60189</td>
<td>AXI4_WVALID_UNKN</td>
<td>WVALID has an X value/WVALID has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60190</td>
<td>MVC_FAILED_POSTCONDITION</td>
<td>A postcondition failed.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60191</td>
<td>MVC_FAILED_RECOGNITION</td>
<td>An item failed to be recognized.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60192</td>
<td>AXI4_TIMEOUT_WAITING_FOR_WRITE_DATA</td>
<td>Timed-out waiting for a data phase in write data burst.</td>
<td>A4.6</td>
</tr>
<tr>
<td>AXI4-60193</td>
<td>AXI4_EXCL_RD_WHILE_EXCL_WR_IN_PROGRESSSAME_ID</td>
<td>Master starts an exclusive read burst while exclusive write burst with same ID tag is in progress.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60194</td>
<td>AXI4_EXCL_WR_WHILE_EXCL_RD_IN_PROGRESSSAME_ID</td>
<td>Master starts an exclusive write burst while exclusive read burst with same ID tag is in progress.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60195</td>
<td>AXI4_DEC_ERR_ILLEGAL_FOR_MAPPED_SLAVE_ADDR</td>
<td>Slave receives a burst to a mapped address but responds with DECERR (signaled by AXI4_DECERR).</td>
<td>A3.4.4</td>
</tr>
<tr>
<td>AXI4-60196</td>
<td>AXI4_AWVALID_HIGH_DURING_RESET</td>
<td>AWVALID asserted during the reset state.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60197</td>
<td>AXI4_WVALID_HIGH_DURING_RESET</td>
<td>WVALID asserted during the reset state.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60198</td>
<td>AXI4_BVALID_HIGH_DURING_RESET</td>
<td>BVALID asserted during the reset state.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60199</td>
<td>AXI4_ARVALID_HIGH_DURING_RESET</td>
<td>ARVALID asserted during the reset state.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60200</td>
<td>AXI4_RVALID_HIGH_DURING_RESET</td>
<td>RVALID asserted during the reset state.</td>
<td>A3.1.2</td>
</tr>
<tr>
<td>AXI4-60201</td>
<td>AXI4_ARESETn_SIGNAL_Z</td>
<td>Reset signal has a Z value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60202</td>
<td>AXI4_ARESETn_SIGNAL_X</td>
<td>Reset signal has an X value.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60203</td>
<td>AXI4_TIMEOUT_WAITING_FOR_WRITE_ADDR_AFTER_DATA</td>
<td>Timed-out waiting for a write address phase to be coming after data.</td>
<td>A2.2</td>
</tr>
<tr>
<td>AXI4-60204</td>
<td>AXI4_EXCLUSIVE_WRITE_BYTES_TRANSFER_EXCEEDS_128</td>
<td>Number of bytes in an exclusive write transaction must be less than or equal to 128.</td>
<td>A7.2.4</td>
</tr>
</tbody>
</table>
### Table A-1. AXI4 Assertions (cont.)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Name</th>
<th>Description</th>
<th>Property Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4-60205</td>
<td>AXI4_EXCLUSIVE_WRITE_BYTES_TRANSFER_NOT_POWER_OF_2</td>
<td>Number of bytes of an exclusive write transaction is not a power of 2.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60206</td>
<td>AXI4_UNALIGNED_ADDRESS_FOR_EXCLUSIVE_WRITE</td>
<td>Exclusive write accesses must have address aligned to the total number of bytes in the transaction.</td>
<td>A7.2.4</td>
</tr>
<tr>
<td>AXI4-60207</td>
<td>AXI4_RLAST_VIOLATION</td>
<td>RLAST signal should be asserted along with the final transfer of the read data burst.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60208</td>
<td>AXI4_WLAST_ASSERTED_DURING_DATA_PHASE_OTHER_THAN_LAST</td>
<td>Wlast must only be asserted during the last data phase.</td>
<td>A3.4.1</td>
</tr>
<tr>
<td>AXI4-60209</td>
<td>AXI4_READ_ISSUING_CAPABILITY_VIOLATION</td>
<td>The number of outstanding Read transactions exceeded the maximum Read issuing capability.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60210</td>
<td>AXI4_WRITE_ISSUING_CAPABILITY_VIOLATION</td>
<td>The number of outstanding Write transactions exceeded the maximum Write issuing capability.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60211</td>
<td>AXI4_COMBINED_ISSUING_CAPABILITY_VIOLATION</td>
<td>The number of outstanding Read and Write transactions exceeded the maximum combined issuing capability.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60212</td>
<td>AXI4_READ_ACCEPTANCE_CAPABILITY_VIOLATION</td>
<td>The number of outstanding Read transactions exceeded the maximum Read acceptance capability.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60213</td>
<td>AXI4_WRITE_ACCEPTANCE_CAPABILITY_VIOLATION</td>
<td>The number of outstanding Write transactions exceeded the maximum Write acceptance capability.</td>
<td></td>
</tr>
<tr>
<td>AXI4-60214</td>
<td>AXI4_COMBINED_ACCEPTANCE_CAPABILITY_VIOLATION</td>
<td>The number of outstanding Read and Write transactions exceeded the maximum combined acceptance capability.</td>
<td></td>
</tr>
</tbody>
</table>
Third-Party Software for Mentor Verification IP Altera Edition

This section provides information on open source and third-party software that may be included in the Mentor Verification IP Altera Edition software product.

This software application may include GNU GCC version 4.5.0 third-party software. GNU GCC version 4.5.0 is distributed under the terms of the GNU General Public License version 3.0 and is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at: <path to legal directory>/legal/gnu_gpl_3.0.pdf. Portions of this software may be subject to the GNU Free Documentation License version 1.2. You can view a copy of the GNU Free Documentation License version 1.2 at: <path to legal directory>/legal/gnu_free_doc_1.2.pdf. Portions of this software may be subject to the Boost License version 1.0. You can view a copy of the Boost License v1.0 at: <path to legal directory>/legal/boost_1.0.pdf. To obtain a copy of the GNU GCC version 4.5.0 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed GNU GCC version 4.5.0 and valid for as long as Mentor Graphics offers customer support for this Mentor Graphics product. GNU GCC version 4.5.0 may be subject to the following copyrights:


Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

© 2004 Ami Tavory and Vladimir Dreizin, IBM-HRL.

Permission to use, copy, modify, sell, and distribute this software is hereby granted without fee, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation. None of the above authors, nor IBM Haifa Research Laboratories, make any representation about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

© 1994 Hewlett-Packard Company

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

© 1992, 1993 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS `AS IS' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)

HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1992, 1993, 1994 Henry Spencer. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.

2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.

3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.

4. This notice may not be removed or altered.
1. ORDERS, FEES AND PAYMENT.

1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer’s appointed third party buying agent) places and
Mentor Graphics accepts purchase orders pursuant to this Agreement (“Order(s)”), each Order will constitute a contract
between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of
this Agreement, any applicable addenda and the applicable quotation, whether or not these documents are referenced on the
Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in
writing by an authorized representative of Customer and Mentor Graphics.

1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such
invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half
percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight,
insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable
invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor
Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax and service tax.
Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such
taxes imposed on payments by Customer hereunder will be Customer’s sole responsibility. If Customer appoints a third
party to place purchase orders and/or make payments on Customer’s behalf, Customer shall be liable for payment under
Orders placed by such third party in the event of default.

1.3. All Products are delivered FCA factory (Incoterms 2000), freight prepaid and invoiced to Customer, except Software
delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor
Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price
of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or
convenient for use in filing or perfecting such security interest. Mentor Graphics’ delivery of Software by electronic means
is subject to Customer’s provision of both a primary and an alternate e-mail address.

2. GRANT OF LICENSE. The software installed, downloaded, or otherwise acquired by Customer under this Agreement,
including any updates, modifications, revisions, copies, documentation and design data (“Software”) are copyrighted, trade
secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all
rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license
fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as
provided in Subsection 5.2); (b) for Customer’s internal business purposes; (c) for the term of the license; and (d) on the
computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius.
Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a
Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of
employment is the site where the Software is authorized for use. Mentor Graphics’ standard policies and programs, which vary
depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of
Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for
a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar
devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and
revisions. For the avoidance of doubt, if Customer requests any change or enhancement to Software, whether in the course of
receiving support or consulting services, evaluating Software, performing beta testing or otherwise, any inventions, product
improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics’ sole discretion) will be the exclusive property of Mentor Graphics.

3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics’ Embedded Software Channel (“ESC”), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are linked into a composite program as an integral part of Customer’s compiled computer program, provided that Customer distributes these files only in conjunction with Customer’s compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics’ real-time operating systems or other embedded software products into Customer’s products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

4. **BETA CODE.**

4.1. Portions or all of certain Software may contain code for experimental testing and evaluation (“Beta Code”), which may not be used without Mentor Graphics’ explicit authorization. Upon Mentor Graphics’ authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer’s use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer’s use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer’s evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer’s feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. **RESTRICTIONS ON USE.**

5.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer’s employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use it except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any source code from Software. Log files, data files, rule files and script files generated by or for the Software (collectively “Files”), including without limitation files containing Standard Verification Rule Format (“SVRF”) and Tcl Verification Format (“TVF”) which are Mentor Graphics’ proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Software or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Software, or disclose to any third party the results of, or information pertaining to, any benchmark.

5.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer’s employees or contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.

5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise (“Attemped Transfer”), without Mentor Graphics’ prior written consent and payment of Mentor Graphics’ then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics’ prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics’ option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer’s permitted successors in interest and assigns.
5.4. The provisions of this Section 5 shall survive the termination of this Agreement.

6. SUPPORT SERVICES. To the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics’ then current End-User Support Terms located at http://supportnet.mentor.com/about/legal/.

7. AUTOMATIC CHECK FOR UPDATES; PRIVACY. Technological measures in Software may communicate with servers of Mentor Graphics or its contractors for the purpose of checking for and notifying the user of updates and to ensure that the Software in use is licensed in compliance with this Agreement. Mentor Graphics will not collect any personally identifiable data in this process and will not disclose any data collected to any third party without the prior written consent of Customer, except to Mentor Graphics’ outside attorneys or as may be required by a court of competent jurisdiction.

8. LIMITED WARRANTY.

8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer’s requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS’ ENTIRE LIABILITY AND CUSTOMER’S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS’ OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED “AS IS.”

8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

9. LIMITATION OF LIABILITY. EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS’ OR ITS LICENSORS’ LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. HAZARDOUS APPLICATIONS. CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING ITS PRODUCTS USED IN APPLICATIONS WHERE THE FAILURE OR INACCURACY OF ITS PRODUCTS MIGHT RESULT IN DEATH OR PERSONAL INJURY (“HAZARDOUS APPLICATIONS”). NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. INDEMNIFICATION. CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING ATTORNEYS’ FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10. THE PROVISIONS OF THIS SECTION 11 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

12. INFRINGEMENT.

12.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics’ obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
12.2. If a claim is made under Subsection 12.1 Mentor Graphics may, at its option and expense, (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

12.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics’ licensors who do not provide such indemnification to Mentor Graphics’ customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

12.4. THIS SECTION 12 IS SUBJECT TO SECTION 9 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS FOR DEFENSE, SETTLEMENT AND DAMAGES, AND CUSTOMER’S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

13. TERMINATION AND EFFECT OF TERMINATION. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term.

13.1. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer’s obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

13.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer’s possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.

14. EXPORT. The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products and information about the products to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

15. U.S. GOVERNMENT LICENSE RIGHTS. Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

16. THIRD PARTY BENEFICIARY. Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

17. REVIEW OF LICENSE USAGE. Customer will monitor the access to and use of Software. With prior written notice and during Customer’s normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer’s software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer’s compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FLEXlm or FLEXnet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics’ request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 17 shall survive the termination of this Agreement.

18. CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION. The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the United States. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, USA, if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre (“SIAC”) to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. This section shall not
restrict Mentor Graphics’ right to bring an action against Customer in the jurisdiction where Customer’s place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

19. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

20. **MISCELLANEOUS.** This Agreement contains the parties’ entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 100615, Part No. 246066