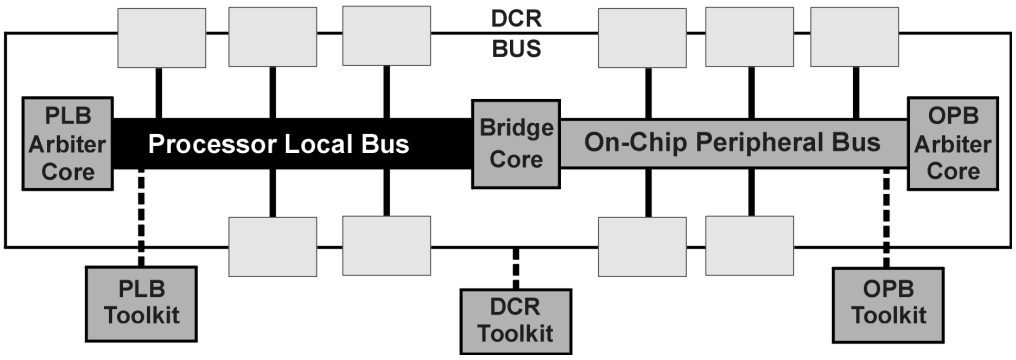
Processor Local Bus Functional Model Toolkit

User's Manual

Version 4.9.2

CoreConnect™

The system on a chip bus standard.



SA-14-2542-11

Eleventh Edition (June 2003)

This edition of Processor Local Bus Functional Model Toolkit User's Manual applies to the IBM PLB toolkit, until otherwise indicated in new versions or application notes.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Address comments about this publication to:

IBM Corporation
Department YM5A
P.O. Box 12195
Research Triangle Park, NC 27709

4 3 2 1

Patents and Trademarks

IBM may have patents or pending patent applications covering the subject matter in this publication. The furnishing of this publication does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904, United States of America.

The following terms are trademarks of IBM Corporation:

IBM
CoreConnect
Other terms which are trademarks are the property of their respective owners.

Contents

Figures

Tables

About This Book

This book begins with an overview followed by detailed information on Processor Local Bus Functional Model Toolkit environment, testbench, bus functional compiler, models and language used in simulation.

The Processor Local Bus Functional Model Toolkit features:

- Unit and subsystem level simulation of logic designs which comply with PLB architecture specifications.
- VHDL and verilog source model solutions with simulator independence.
- Bus functional command definition which generates and responds to different transaction types with varying delays.
- Bus functional compiler which generates model initialization files from bus functional commands.
- Bus protocol checking through the use of general purpose bus monitors.
- Read and write data checking in masters and slaves.
- Model inter-communication bus for event and transaction synchronization.
- Enables peripheral developers to verify and debug designs to assure bus compliance.
- Faster simulation run-times than PPC BFM or FFM to generate bus traffic.
- Allows ‘what if’ simulation scenarios using different master and slave configurations.
- Flexible and user-friendly bus functional language (BFL) for quick generation of a variety of bus transactions.
- Hierarchical solution to verification.

Who Should Use This Book

This book is for hardware, software, and application developers who need to understand Core+ASIC development and system-on-a-chip (SOC) designs. The audience should understand embedded system design, operating systems, and the principles of computer organization.

Since the PLB model toolkit is designed to comply with the architectural specification, toolkit users need to have a working level understanding of the architectural specification to be able to develop test cases and simulate using the bus model toolkit. The user should also be familiar with UNIX type operating systems, basic digital logic design and simulation, and the simulator which is used for the verification process.

Related Publications

The following publications contain related information:

Processor Local Bus Architecture Specifications

On-Chip Peripheral Bus Architecture Specifications

Device Control Register Bus Architecture Specifications

Processor Local Bus Toolkit User’s Manual

On-Chip Peripheral Bus Toolkit User’s Manual

Device Control Register Bus Toolkit User's Manual
Processor Local Bus Arbiter Core User's Manual
On-Chip Peripheral Bus Arbiter Core User's Manual
PLB to OPB Bridge Core User's Manual
OPB to PLB Bridge Core User's Manual

How This Book is Organized

This book is organized as follows:

Chapter 1, "PLB Toolkit Overview"

Chapter 2, "PLB Toolkit Environment"

Chapter 3, "PLB Toolkit Test Bench"

Chapter 4, "PLB Bus Functional Compiler"

Chapter 5, "PLB Bus Models"

Chapter 6, "PLB Bus Functional Language"

Chapter 7, "PLB Bus Timing"

Chapter 8, "PLB Compliance Checks"

To help readers find material in these chapters, the book contains:

- "Contents" on page v
- "Figures" on page ix
- "Tables" on page xi
- "Index" on page 93

Chapter 1. PLB Toolkit Overview

Processor local bus (PLB) functional model toolkit provides unit and system level simulation and verification of ASICs and logic designs which comply with PLB architecture specifications. The toolkit enables the designer to accelerate the design cycle time by identifying and addressing possible problems at an earlier stage of the design cycle.

The processor local bus (PLB) is designed to interface directly with the processor cores. The primary goal of the PLB is to provide a standard interface between the processor cores and integrated bus controllers such that a library of processor cores and bus controllers can be developed for use in the Core+ASIC development. The PLB model toolkit facilitates unit and subsystem level simulation of logic designs which comply with the PLB architecture. See the PLB architecture specifications for more information.

Figure 1 demonstrates how the processor local bus is inter connected for the purpose of Core+ASIC development or system-on-a-chip design.

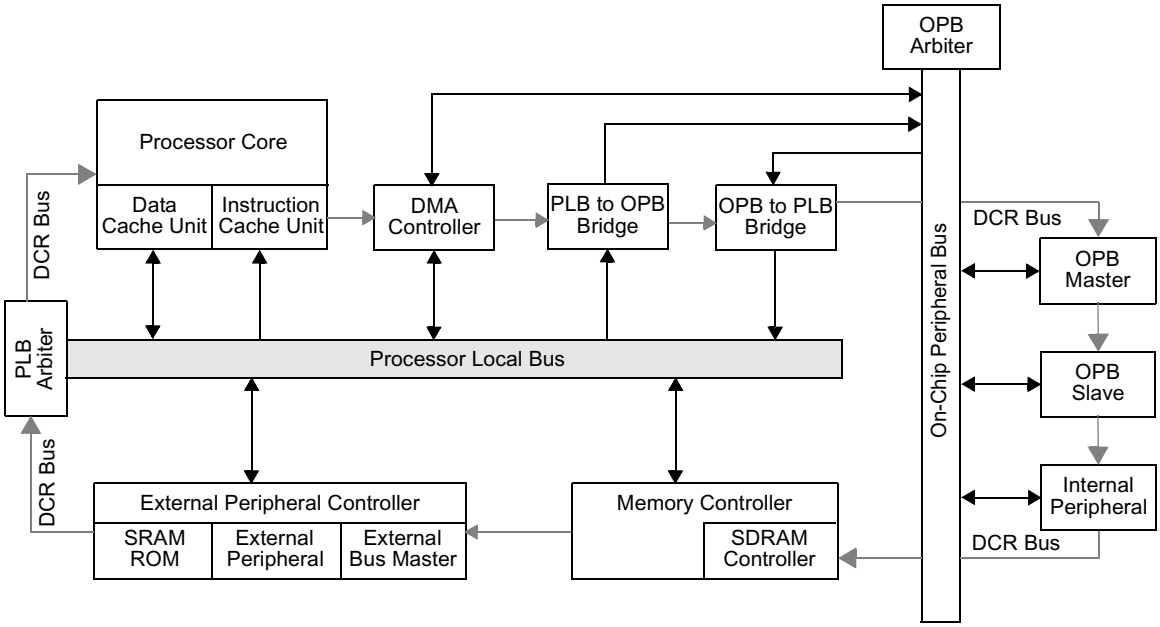


Figure 1. Processor Local Bus Interconnection

As shown in Figure 1, the on-chip bus structure provides a link between the processor core and other peripherals which consist of PLB and OPB master and slave devices.

The processor local bus (PLB) is the high performance bus used to access memory through the bus interface units. The two bus interface units shown above: external peripheral controller and memory controller are the PLB slaves. The processor core has two PLB master connections, one for instruction cache and one for data cache. Attached to the PLB is also the direct memory access (DMA) controller, which is a PLB master device used in data intensive applications to improve data transfer performance.

Lower performance peripherals (such as OPB master, slave, and other internal peripherals) are attached to the on-chip peripheral bus (OPB). A bridge is provided between the PLB and OPB to enable data transfer by PLB masters to and from OPB slaves. In the above example we have two

bridges, a PLB to OPB bridge which is a slave on the PLB and a master on the OPB and an OPB to PLB bridge which is a slave on the OPB and a master on the PLB. OPB peripherals may also comprise DMA peripherals.

The device control register (DCR) bus is used primarily for accessing status and control registers within the various PLB and OPB masters and slaves. It is meant to off-load the PLB from the lower performance status and control read and write transfers. The DCR bus architecture allows data transfers among OPB peripherals to occur independently from, and concurrent with, data transfers between the processor and memory, or among other PLB devices.

The toolkit allows users to initiate PLB master cycles and provide PLB slave responses through a bus functional language which is parameterized according to the architectural specification. Data checking and bus protocol monitoring also provide a way for users to automate the verification of PLB designs under development. Source files in ASCII format are distributed with the toolkit, which include behavioral models, programs, and documentation.

1.1 PLB Toolkit Features

PLB toolkit features:

- Unit and subsystem level simulation of logic designs which comply with PLB architecture specifications.
- VHDL and verilog source model solutions with simulator independence.
- Bus functional command definition which generates and responds to different transaction types with varying delays.
- Bus functional compiler which generates model initialization files from bus functional commands.
- Bus protocol checking through the use of general purpose bus monitors.
- Read and write data checking in masters and slaves.
- Model inter-communication bus for event and transaction synchronization.
- Enables peripheral developers to verify and debug designs to assure bus compliance.
- Faster simulation run-times than PPC BFM or FFM to generate bus traffic.
- Allows ‘what if’ simulation scenarios using different master and slave configurations.
- Flexible and user-friendly bus functional language (BFL) for quick generation of a variety of bus transactions.
- Hierarchical solution to verification.

1.2 PLB Features

The PLB model toolkit enables the user to simulate the following PLB features.

- Separate address bus, read data bus and write data bus for each bus master. Shared address bus, read data bus, and write data bus for all bus slaves.

- Separate read data bus and write data bus transfer qualifiers allow overlap of read and write transfers. Maximum bandwidth of the bus is two transfers per cycle when a read and write operation is overlapped.
- Read operations may be pipelined allowing continuous bus utilization when a master is performing back-to-back read operations. For continuous back-to-back read or write transfers, the maximum bandwidth of the bus is one transfer per cycle.
- Byte enables specify bytes within a word allowing for byte, halfword, and word transfers as well as unaligned halfword transfers and 3-byte transfers.
- Packing and unpacking of data is performed within each PLB slave. All masters and slaves attach to the PLB as 128-bit, 64-bit, and 32-bit devices. Slaves may support 8-bit and 16-bit bus widths if required. PLB protocol does not include support for halfword or byte devices. All transfer requests for bytes within an aligned word will be transferred on the PLB in one data transfer cycle.
- A PLB master may request a16-byte, 32-byte, or 64-byte line of data. These requests will result in data being transferred to the master in 4, 8, or 16 word transfers on the PLB. For non-line transfers, transfer width will match the width of the requested data as indicated by the byte enables.
- PLB slaves provide a read word address(Sl_rdWdAddr) allowing bus slaves to fetch line data in any order (that is, target word first or sequentially). All masters are required to sample the word address for line read operations and steer the word to the appropriate location within the line based on the word address from the slave. See PLB Architecture specification for more information
- Sequential burst protocol provides a mechanism for a master to request a burst of word, halfword, or byte transfers to/from any slave.
- Abort signal provides the master the capability to abort a request late in the cycle.
- Guarded and unguarded memory transfers allow a slave device to enable or disable the prefetching of instructions or data.
- Four levels of request priority for each master allow PLB implementations with various arbitration schemes. Master may lock bus arbitration allowing for atomic operations.
- Slaves provide error reporting and busy acknowledge to each master on the PLB.

1.3 PLB Implementation

The PLB implementation consists of a PLB core to which all masters and slaves are attached. The logic within the PLB core consists of a central bus arbiter and the necessary bus control and gating logic.

The PLB architecture supports up to sixteen master devices. However, PLB core implementations supporting less than sixteen masters are allowed. The PLB architecture also supports any number of slave devices. However, it should be noted that the number of masters and slaves attached to a PLB core in a particular system will directly affect the performance of the PLB core in that system.

Figure 2 shows an example of the PLB connections for a system with three masters and three slaves.

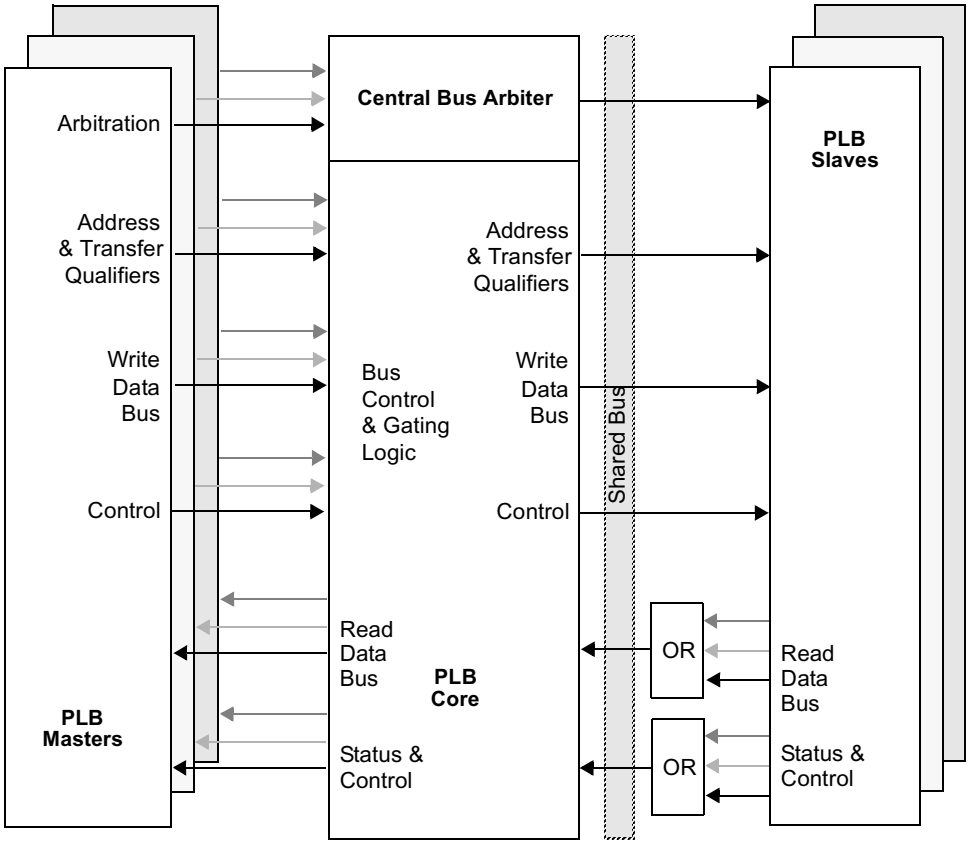


Figure 2. Physical Implementation of the PLB

Chapter 2. PLB Toolkit Environment

The PLB toolkit consists of a test bench, bus functional models, and a bus functional compiler. The test bench instantiates the bus models, a reference PLB core, and design(s) under test.

The toolkit is intended to provide users with an environment in which they may initialize the toolkit models and simulate with the ability to detect error conditions in PLB bus protocol and also in data consistency. Test cases are written using a bus functional language which is described in “PLB Bus Functional Language” on page 29.

Note 1: All user verification procedures and regressions should include a released and qualified PLB core from the IBM Core+ASIC library which will be included in the target product silicon.

Note 2: It is recommended that all simulations use the PLB monitor to ensure PLB compliance. If the monitor is not connected, the master model will not generate any requests and a warning message will be reported by the master.

2.1 Bus Functional Compiler and ReadMe Files

The following files represent the bus functional compiler and a readme file.

- **BFC**

  This Perl program parses the test cases written in the bus functional language and generates command files which are used to control the bus functional models.

- **README**

  This file provides information about the toolkit release including functional support level and any known errata.

- **UPDATE_LOG**

  This file provides information about changes in the toolkit release relative to the previous release.

2.2 Example Test Case File

- **sample.bfl**

  This file is an example PLB BFL file. Note the path in the set_device command may be updated for user simulator hierarchical structure specifications.

- **sample_auto.bfl**

  This example BFL file contains two masters and one slave that demonstrate the use of auto mode. It also shows how to use the move command, which switches a master from command mode to auto mode.

- **mon_init.bfl**

  This example BFL file contains monitor initialization statements for slave address range initialization used in PLB Monitor checking code.

2.3 VHDL/Verilog Files

The following list of files comprise the models and test bench of the PLB toolkit. The files should be compiled in the order specified:

- **plb_pkg.vhd/plb_pkg.inc**

  This HDL file is used by the PLB behavioral models to implement internal functions such as type conversion and other general procedures.

- **plb_dcl.vhd / plb_dcl.inc/ plb_dcl3x.inc**

  This HDL file contains the component and constant declarations for the toolkit behavioral models.

- **plb_master_comp.vhd / plb_master_comp.v/ plb_master_comp3x.v**

  This HDL file represents a PLB master device. It contains decode, execute, and bus logic to interface with the PLB bus. It also contains an internal master memory.

- **plb_slave_comp.vhd / plb_slave_comp.v/ plb_slave_comp3x.v**

  This HDL file represents a PLB slave device. It contains interface with the PLB bus and also an internal memory.

- **plb_monitor_comp.vhd / plb_monitor_comp.v/ plb_monitor_comp3x.v**

  This HDL file performs PLB cycle monitoring and protocol checking

- **plb_master.vhd / plb_master.v**

  This HDL file is a wrapper for the PLB master component which is instantiated in this file. Users should instantiate this entity in test fixtures which include the PLB model toolkit master model.

- **plb_slave.vhd/ plb_slave.v**

  This HDL file is a wrapper for the PLB slave component which is instantiated in this file. Users should instantiate this entity in test fixtures which include the PLB model toolkit slave model.

- **plb_monitor.vhd** / **plb_monitor.v**

  This HDL file is a wrapper for the PLB monitor component which is instantiated in this file. Users should instantiate this entity in test fixtures which include the PLB model toolkit monitor model. It is recommended that all simulations use the PLB monitor to ensure PLB compliance.

- **plb_master3x.vhd / plb_master3x.v**

  This HDL file is a compatibility mode wrapper for the PLB master component which is instantiated in this file. Users should instantiate this entity in test fixtures which include the PLB model toolkit master model being used with 3.x PLB Architecture arbiter designs.

- **plb_slave3x.vhd / plb_slave3x.v**

  This HDL file is a compatibility mode wrapper for the PLB slave component which is instantiated in this file. Users should instantiate this entity in test fixtures which include the PLB model toolkit slave model being used with 3.x PLB Architecture arbiter designs.

- **plb_monitor3x.vhd / plb_monitor3x.v**

This HDL file is a compatibility mode wrapper for the PLB monitor component which is instantiated in this file. Users should instantiate this entity in test fixtures which include the PLB model toolkit monitor model being used with 3.x PLB Architecture arbiter designs. It is recommended that all simulations use the PLB monitor to ensure PLB compliance.

- **plb_complex.vhd / plb_complex.v**

  This HDL file is the PLB test bench. Designs under test may be instantiated in the test bench and simulated with the PLB behavioral models.

2.4 VHDL/Verilog Files for Crossbar Arbiter (with crossbar monitor)

The PLB4 Toolkit supports use of the crossbar arbiter with the cross bar monitor. In this environment, these are the list of files to use and the order they need to be compiled:

- **plb_pkg.vhd/plb_pkg.inc**

  This HDL file is used by the PLB behavioral models to implement internal functions such as type conversion and other general functions.

- **plb_dcl_12m.vhd / plb_dcl_12m.inc**

  This HDL file contains the component and constant declarations for the toolkit behavioral models.

- **plb_master_comp.vhd / plb_master_comp.v**

  This HDL file represents a PLB master device. It contains decode, execute, and bus logic to interface with the PLB bus. It also contains an internal master memory.

- **plb_slave_comp12m.vhd / plb_slave_comp12m.v**

  This HDL file represents a PLB slave device. It contains interface with the PLB bus and also an internal memory.

- **plb_monitor_comp12m.vhd / plb_monitor_comp12m.v**

  This HDL file performs PLB cycle monitoring and protocol checking. It monitors up to 12 masters and 8 slaves which support 8 or 12 masters.

- **plb_master.vhd / plb_master.v**

  This HDL file is a wrapper for the PLB master component which is instantiated in this file. Users should instantiate this entity in test fixtures which include the PLB model toolkit master model.

- **plb_slave_12m.vhd/ plb_slave_12m.v**

  This HDL file is a wrapper for the PLB slave component which is instantiated in this file. It supports 12 masters. Users should instantiate this entity in test fixtures which include the PLB model toolkit slave model.

- **plb_monitor_12m.vhd** / **plb_monitor_12m.v**

  This HDL file is a wrapper for the PLB monitor component which is instantiated in this file. Users should instantiate this entity in test fixtures which include the PLB model toolkit monitor model. It is recommended that all simulations use the PLB monitor to ensure PLB compliance.

- **plb_monitor_xbar.vhd** / **plb_monitor_xbar.v**

This HDL file is a wrapper which instantiates two plb_monitor_12m modules for the crossbar environment. It is recommended that all simulations use the PLB monitor to ensure PLB compliance.

The crossbar arbiter is required to run in the environment with the crossbar monitor. Contact PPCSUPP (ppcsupp@us.ibm.com) for more information on obtaining the arbiter and corresponding test bench.

Chapter 3. PLB Toolkit Test Bench

Figure 3 shows the default configuration of the PLB toolkit test bench environment. It contains instantiations of a PLB master model, PLB slave model, PLB core, and a PLB monitor model. It also contains the distributed multiplexers and logic to implement the PLB bus using AND/OR HDL statements.

Note: Default configuration instantiates four masters and four slaves. The current PLB implementation supports up to eight masters.

Designs under test may be instantiated in the toolkit test bench HDL which replace the default configuration. Also, the bus functional models may be instantiated in other test bench environments which include designs under test.

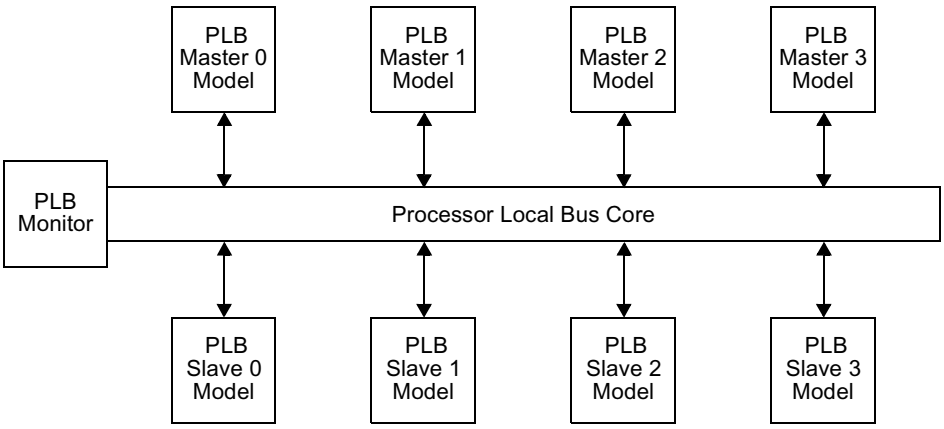


Figure 3. PLB Toolkit Testbench(plb_complex.vhd/.v)

3.1 Programmable PLB Data Bus Width and Automatic Replication

The PLB toolkit test bench uses a constant called PLB_DATA_BUS_WIDTH in declaring the actual PLB arbiter data bus signals. This constant defaults to 128 for the PLB 4.x toolkit, but may be changed to be 64 or 32 for other PLB arbiter implementations. The toolkit will automatically scale the model I/O's and bus operations based on the PLB_DATA_BUS_WIDTH constant.

In addition to programmable arbiter data bus widths, the PLB Test Bench contains logic to automatically perform the necessary Byte Enable(BE) and Data Bus replication as described by the PLB 4.x Bus Architecture specification. This replication is performed when necessary as smaller PLB masters and Slaves are connected to a wider data path PLB Arbiter. The master BE's and write data bus(wrDbus) are replicated to the arbiter, and the slave read data bus(rdDbus) is also replicated to the arbiter when necessary. The test bench uses the msize/ssize signals from the masters and slave, together with other bus control signals to automatically generate a "mirror" signal.

3.2 Instantiating PLB Slave Designs Under Test

When instantiating a PLB slave design under test, connect the PLB signals in the test bench to the new device and remove a PLB slave model instantiation. Ensure that the PLB slave address map has non-overlapping slave address space. The PLB slave model instantiation generics/parameters have a default slave address mapping as follows:

Device 0	00000000-0003FFFF
Device 1	00040000-0007FFFF
Device 2	00080000-000BFFFF
Device 3	000C0000-000FFFFF

The PLB slave model has parameters for the address decode in the HDL file. These may be updated by the user to re-program the default slave address space for a single PLB slave instantiation. When multiple slave models are connected to a single PLB core, the user must ensure that the slave decode maps are mutually exclusive within a test bench. This can be accomplished through the BFL configure command, generic statements in the VHDL instantiations, or Verilog parameters in the model HDL files.

3.3 Instantiating PLB Master Designs Under Test

When instantiating a PLB master design under test, connect the master to the PLB master interface of the PLB core/arbiter. Each master will have an independent PLB interface as defined in the PLB architecture specification.

Each toolkit model uses intercommunication signals called Synch_in(0 to 31) and Synch_out(0 to 31) to synchronize events in the test environment. The Synch_in(0 to 31) is the logical OR of all the Synch_out outputs of each instantiated models in the simulation environment which uses the intercommunication bus. When instantiating toolkit models which need to support the send/wait commands as described in “PLB Bus Functional Language” on page 29, ensure that the intercommunication logic within the test bench is updated to reflect the additional model instantiations.

3.4 VHDL Signal Types

The signal interface for the VHDL bus functional model wrappers and test bench are declared as IEEE std_logic and std_logic_vector signal types. If the bus functional models are integrated with a test bench environment which uses bit and bit_vector types, the wrappers may be eliminated so that type conversions do not have to be included in the model interfaces. When the wrappers are not used, please note that the I/O for each PLB toolkit model component contains primary inputs for configuration signals rather than the VHDL generic declarations which are in each wrapper.

3.5 IEEE Packages

The VHDL version of the PLB core RTL is translated from Verilog to VHDL with a source level language translator. The translated VHDL calls out some IEEE packages such as:

ieee.std_logic_1164.all;

ieee.std_logic_arith.all;

ieee.std_logic_unsigned.all;

Some VHDL compilers and analyzers may require the use of compiler directive switches to resolve overloaded relational operators.

Chapter 4. PLB Bus Functional Compiler

Test cases written in the PLB bus functional languages are parsed by the toolkit bus functional compiler. For VHDL simulators, it generates simulator interface commands to initialize the PLB models within the toolkit test bench. For Verilog simulators, the BFC generates a Verilog initialization file which contains Verilog statements to initialize the bus functional models. These command files are used to load the bus functional model command and data arrays after the test bench is loaded into the simulator.

Multiple bus functional model command sections may be written within the same BFL file, or individual BFL files may be used to initialize each master/slave device. The bus functional compiler will generate a single command file for each BFL file which it processes. If more than one BFL file is used for a single test case, the user should include each command file generated by the BFC when initializing the bus functional models within the test bench.

The bus functional compiler is implemented in Perl, which is an interpreted language distributed under the GNU public license. It is available at no cost and runs on nearly all UNIX or UNIX-like operating systems.

4.1 Simulator Configuration

Since the output of the bus functional compiler is a simulator specific command initialization file, the user should indicate the target simulator being used. This can be done by changing the default simulator command output parameter at the header of the bus functional compiler Perl program. The name of this variable is $target_simulator.

Note: The BFC will not compile a.bfl file if it is not configured properly.

4.2 Declarations File Access

The PLB slave model provided with this toolkit supports more than one internal memory lookup algorithm. In addition, the model memory sizes can be configured by updating the declarations file (plb_dcl.vhd/v). This is described in more detail in “Master Model Operation” on page 17 and “Slave Model Operation” on page 23. Since the memory initialization statements are generated from the BFC when parsing the bus functional language memory initialization constructs, the BFC needs to know the memory configuration parameters before it generates the simulator specific initialization files. Therefore, the BFC reads the plb_dcl.vhd/v file before it processes a test case to determine the memory configuration parameters.

Note: If the BFC file is not executed from the same directory where the plb_dcl.vhd/v file resides, the user must update the $plb_dcl_path at the header of the BFC file. An error message is generated if the declarations file is not found.

4.3 Invoking the Bus Functional Compiler

The bus functional compiler operates on files with the extension ".bfl", and it generates a file with the same name but with a ".cmd" or ".do" or ".v" extension depending on the target simulator. If the ".bfl" extension is omitted, the ".cmd" extension is simply added to the end of the filename.

It is possible to invoke the BFC with multiple command files by specifying each file as an argument (input parameter) to the BFC. When multiple files are used in a single BFC call, the command file which is generated will be named using the first input file name.

To invoke the bus functional compiler, type "BFC filename1.bfl filename2.bfl...".

Note: If the message "perl: not found" or other system error is encountered when invoking the BFC, ensure that the path for the Perl executable is correct on the first line of the BFC source program as required by the Perl interpreter specification. To locate the Perl executable, try using the UNIX command "which perl."

4.4 Initializing the Bus Functional Models

The command files which are generated by the bus functional compiler should be executed at simulation time 0. For the VHDL toolkit this is accomplished with an "include" or "do" type command after the model is loaded by the simulator. For the Verilog toolkit the BFC generates a Verilog initialization command file which should be included when the simulation model is compiled.

Chapter 5. PLB Bus Models

The toolkit contains bus functional models for PLB master, PLB slave, and PLB monitor.

The PLB master and slave models are controlled through a bus functional command interface which is defined in “PLB Bus Functional Language” on page 29. The master model has the ability to issue requests for the bus and generate cycles, and the slave model can be initialized to decode PLB bus cycles and respond to PLB master requests. Both the master and slave models may be initialized to perform different PLB cycles with programmable signal delays.

The master and slave models support multiple modes of data operation. The default mode for the models is to source data from internal data memories which are initialized at model load time. The initialization data is specified through the BFM user interface. The user has the ability to modify the internal memory contents during simulation through the use of bus functional commands described in “PLB Bus Functional Language” on page 29.

Another data mode which is provided in the PLB toolkit is auto_data. When the master and/or slave models are configured for this mode, data is automatically generated and checked as bus transactions are processed in simulation. When the master and slave models are ready to drive data on the PLB bus, they call a pseudo-random number generator to form the necessary data patterns. When data is received by the master and slave models, they call the same pseudo-random number generator to create expect data which is used to compare with the data received from the bus. The source and receive data is synchronized through the formation of a deterministic seed produced with a root_seed and the address of each byte of data. To produce the source and expect data, the models use an addition function for the root_seed and address which produces a unique seed for each byte of data. The root_seed is typically varied by the user on a test case basis.

The PLB Bus Model can be configured to run with a 64-bit and a 128-bit data bus width. It is important to note that the 4X version of the toolkit will not work with a 32-bit data bus width configuration. For applications requiring a 32-bit data bus width there are two options: PLB3X can be used and configured for 32-bits, or PLB4X can be used with manipulated assignments to all unused portions of the 64-bit data connection and 8-bit byte enable connection.

The PLB supports all singles, line, fixed length burst, master and slave terminated burst transfers. However, the PLB only supports a burst size equal to that of the slave. For example: ssize (slave data bus width) = 00, size (burst size) =1010; ssize = 01, size = 1011; ssize = 10, size = 1100.

Another option provided by the 4X toolkit is the ability to run in “3X mode”. This means that the toolkit will run with the functionality and signals from 3X, but will have the additional monitor checks and toolkit enhancements that have been added to the 4X version. This configuration is possible through the use of 3X wrappers. These wrappers allow the user to instantiate 3X devices using the 4X toolkit without having to connect all of the additional 4X signals. Please see the section titled “Running in 4X Bus Models in 3X Mode” for more configuration information.

The PLB monitor model receives all PLB core signals in the PLB test bench environment. It performs bus protocol checking for arbitration and data transfers. Error detection is reported to the user through direct register access and message generation.

5.1 Running the 4X Bus Models in 3X Mode

As discussed previously, the 4X toolkit may be configured to run in 3X mode. This allows the user to have access to all of the functionality and signals from the 3X toolkit, with additional checks from the monitor and improvements to the toolkit itself. In order to run in 3X mode the user must do the following:

- Change the PLB_ARCHITECTURE_VERSION in the plb_dcl to 3.
- Change the PLB_DATA_BUS_WIDTH in the plb_dcl to 64.
- Change the PLB_READ_PIPELINE_DEPTH in the plb_dcl to 2. A larger value can be used when using the arbiter model.
- Instantiate devices using the 3X wrappers, i.e. use plb_master3x/plb_slave3x titles for instantiation.
- Connect the byte enable signals for a 64-bit bus(byte enables are 8-bits for a 64-bit bus as opposed to 4-bits for a 32-bit bus and 16-bits for a128-bit bus).
- Note that ordered, guarded, and compressed are separate in 3X, but are part of the t attribute in 4X.
- Note that merr in 3X is separated into mrderr and mwrerr in 4X.
- Note that mirq does not exist in 3X.

5.2 PLB Master Model

The PLB master model contains logic to automatically request the bus when it has read/write commands to execute. It performs an operation based on the bus functional command which was initialized in the internal master model command array. The PLB arbiter determines which master bus cycle to issue on the slave side of the PLB bus.

Figure 4 demonstrates all PLB master interface input/output signals. See PLB architecture specifications for detailed functional description of the signals.

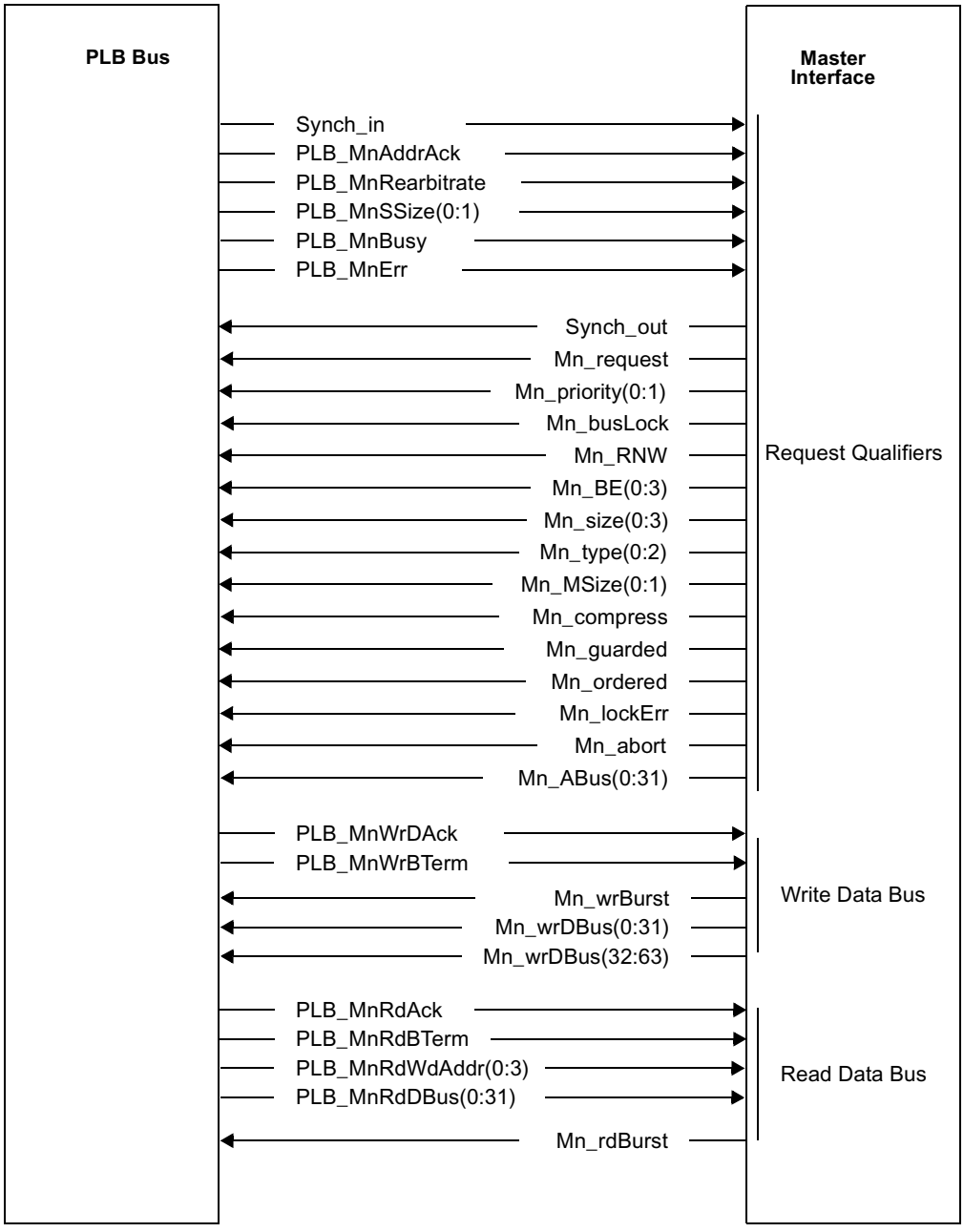


Figure 4. PLB Master Interface

5.2.1 Master Model Operation

The PLB master model operation section discusses the decode unit, internal master data memory, command modes, burst modes, conversion cycles with different PLB device sizes, general purpose registers and ALU instructions, and branch instructions. The operations of PLB master model are shown in Figure 5.

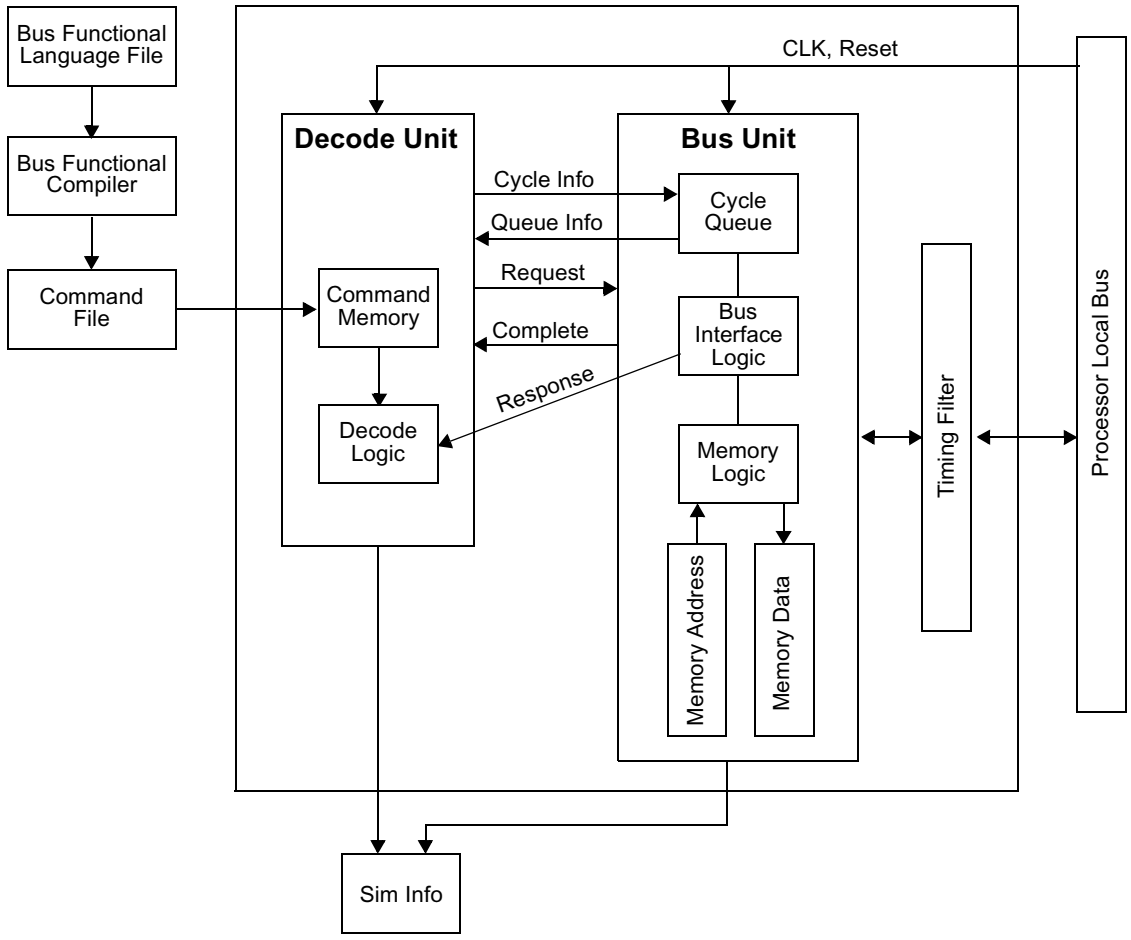


Figure 5. PLB Master Model

5.2.2 Decode Unit

The PLB master model executes bus commands from internal command arrays which are loaded from the command file. The command file is the file generated by the bus functional compiler.

When invoking the simulator, the command file should be included so that the command arrays are initialized when the simulation model is loaded. This is accomplished by using an include file parameter. The master commands are executed sequentially during simulation. The decode unit is designed to decode and queue external bus requests every PLB clock until the bus unit cycle queue is full.

When the decode unit issues a bus request, it continues decoding and executing additional instructions from the command memory as long as it is not waiting on an external event. An example of an external event would be a synchronization signal. In addition, a request delay parameter may suspend the decode unit issuance of commands to the bus unit until an internal command delay counter has expired.

The default size of the bus command array is 1024 entries. This may be increased or decreased by updating the PLB_Master_CMD_Array_Size constant in the declarations HDL file of the toolkit.

5.2.3 Internal Master Data Memory

The PLB master model contains an internal memory array which can be initialized by the user with mem_init bus functional commands. Each entry of the memory array is maintained by the master model during simulation. During this initialization with mem_init commands, a valid bit is set for each address entry. Any bus functional commands which specify an address that matches a valid entry in the memory array will use the memory array for the bus functional command operation. The data used in mem_init statements provides the expect data that is used for the read data comparison checks within the master model. It also provides the data used during a write transfer to a memory address.

For write cycles, the internal master data memory will be used as source data when the bus functional command address matches the command array. For read cycles, the data which is received from the PLB bus will automatically be compared with the internal master data memory.

Error messages will be generated when data miscompares occur. Only the valid bytes for a particular command will be checked. The valid bytes are determined by the transfer size as defined in the PLB architecture.

The memory array is declared as a linear array of 64-bit address/control and 128-bit data for each array entry:

- PLB_Master_Addr_Array(0 to 63) - bits 0 to 61:address field, bit 62:dirty bit, bit 63:valid bit
- PLB_Master_Data_Array(0 to 127) - 4 word data field

The valid bit is set during model initialization for each memory entry used and when a memory entry is allocated in the case of a read memory access miss. When a read command reloads a memory entry with bus data, the dirty bit is set. This may be used with test environments to check if bus commands have executed and caused a master memory update in addition to data miscompares.

The default size of the master memory array is 64k bytes. This may be increased or decreased by updating the PLB_Master_Mem_Size in the declaration HDL file of the toolkit.

5.2.4 Command Modes

This section discusses the possible modes that are available for a PLB master device. PLB masters may be configured to act in either Command mode or Auto mode. A description of these modes are:

- **Command Mode**

This mode enables the user to generate PLB read/write bus transactions and perform other operations using commands which are decoded and executed from within the PLB master model. These commands are parsed by the BFC, loaded into the PLB master model at simulation time 0, and decoded from an internal command array after PLB reset is de-asserted. The following is an example of how to specify a 32-bit single read transaction:

read(addr=00001000,size=0000,be=1111)

Command mode is the default mode for the master model, and the user is not required to specify a BFL configuration parameter for this mode.

- **Auto Mode**

  This mode enables PLB read/write bus cycles to be generated automatically without the need for read/write master commands in the bfl. In this mode, the PLB master model will generate PLB cycles automatically based on an internal random number generator and pre-programmed minimum and maximum delays of the available parameters. The random number generator selects a new delay value between the minimum and maximum using a uniform distribution function. The minimum and maximum range may be initialized by the user through a configuration command. The PLB master model will use auto mode when a configuration command enables it, such as:

  configure(master_auto_mode=true)

5.2.5 Burst Operations

The PLB master model supports the PLB burst protocol. The model accesses the internal master memory during the initial bus request and also during each data phases of the burst transfer. The PLB master model terminates a burst transfer when the user specified number of data transfers has occurred or when a slave terminates the burst transfer with a burst terminate signal. The PLB master model supports a burst_continue_mode where the PLB master will request the bus again if a slave terminate signal is received and the PLB master has more data to transfer. To configure the PLB master model for burst_continue_mode, use the following configuration statement:

configure(burst_continue_mode=true)

The PLB master model also supports the PLB fixed length burst protocol. The master model will automatically de-assert the M_rd/wrBurst signal when: the ‘BE’ parameter is non-zero; the ‘size’ parameter indicates a burst cycle; the corresponding number of data transfers takes place.

Note: For fixed length bursts, the PLB architecture allows the pre-mature de-assertion of the M_rd/wrburst signal before the number of data beat transfers specified by the byte enables. The PLB architecture also allows masters to continue bursting beyond the fixed length burst beat count if a slave burst terminate signal is NOT received during a fixed length burst transfer. To allow verification of this architectural flexibility, the BURST_COUNT parameter can be programmed to a different number of beats than that specified on the byte enables. When the BURST_COUNT parameter is less than the BE data beat count, the burst signal will be pre-maturely de-asserted. When the BURST_COUNT is greater than the BE data beat count, the master will continue bursting IF a slave burst terminate signal was not asserted to the master.

IT IS STRONGLY RECOMMENDED THAT LOGIC DESIGNS WHICH SUPPORT BURST AND FIXED LENGTH BURST BE TESTED THOROUGHLY WITH ALL COMBINATIONS OF BYTE ENABLES, MASTER BURST ABORTS, AND SLAVE BURST TERMINATES TO MAXIMIZE CORE COMPATIBILITY AND COMPLIANCE.

5.2.6 Conversion Cycles with Different PLB Device Sizes

The PLB master model can be configured to be a 32-bit, 64-bit, or 128-bit device. When the PLB master model receives a PLB_MnSSize with PLB_MnAddrAck which represents a slave that is smaller than the configured PLB master size (Mn_Msize(0:1)), the PLB master will automatically generate additional cycles to complete the original requested transaction. These additional cycles can be referred to as “conversion cycles.” For example, a 64-bit PLB master will always generate one additional cycle when a 32-bit slave responds to a single transfer (M_Size=”0000”) read or write cycle. For line transfers, the PLB master completes the request when the appropriate number of words have been transferred as determined by the transaction width (MSize/SSize) and the transfer size (M_Size). See PLB architecture specifications for details on different device widths.

5.2.7 General Purpose and Branch Processor Registers

The PLB master model contains 32 general purpose registers (R0-31), a 32-bit synchronization register (SR) and a 32-bit condition register (CR). These registers can be used by the programmer to implement algorithms or conditions which affect the way in which the PLB master executes commands.

The condition register is updated during ‘compare’ instructions. It provides a mechanism for testing and branching. The bits of the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0)..., CR Field 7 (CR7). Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits. The bits of the CR “fields” are interpreted as follows:

- bit 0: Negative (LT) The result is negative.
- bit 1: Positive (GT) The result is positive.
- bit 2: Zero (EQ) The result is zero.
- bit 3: undefined

The synchronization register latches the Synch_in input. It can subsequently be referenced or reset using an ALU or move instruction.

5.2.8 ALU Instructions

The following instructions are available for arithmetic operations: (a detailed description is provided in “PLB Master Commands” on page 30)

– Add()
– Sub()
– And
– Or
– Shift_Left
– Shift_Right
– Compare()

5.2.9 Branch Instructions

The PLB master model contains a branch instruction so that users can implement loops and forward jumps. The branch instruction uses the condition register to determine whether a jump is taken. Any one of the Condition Register Fields (CR0-CR7) may be used in a branch instruction. Branching between devices is not supported.

5.3 PLB Slave Model

PLB slaves respond to cycles based on requests generated from the PLB core, and they maintain an internal memory which can be initialized through the bus functional language. This memory may be dynamically checked during simulation or when all bus transactions have completed. Figure 6 demonstrates all PLB slave interface input/output signals. See PLB architecture specifications for detailed functional description of the signals.

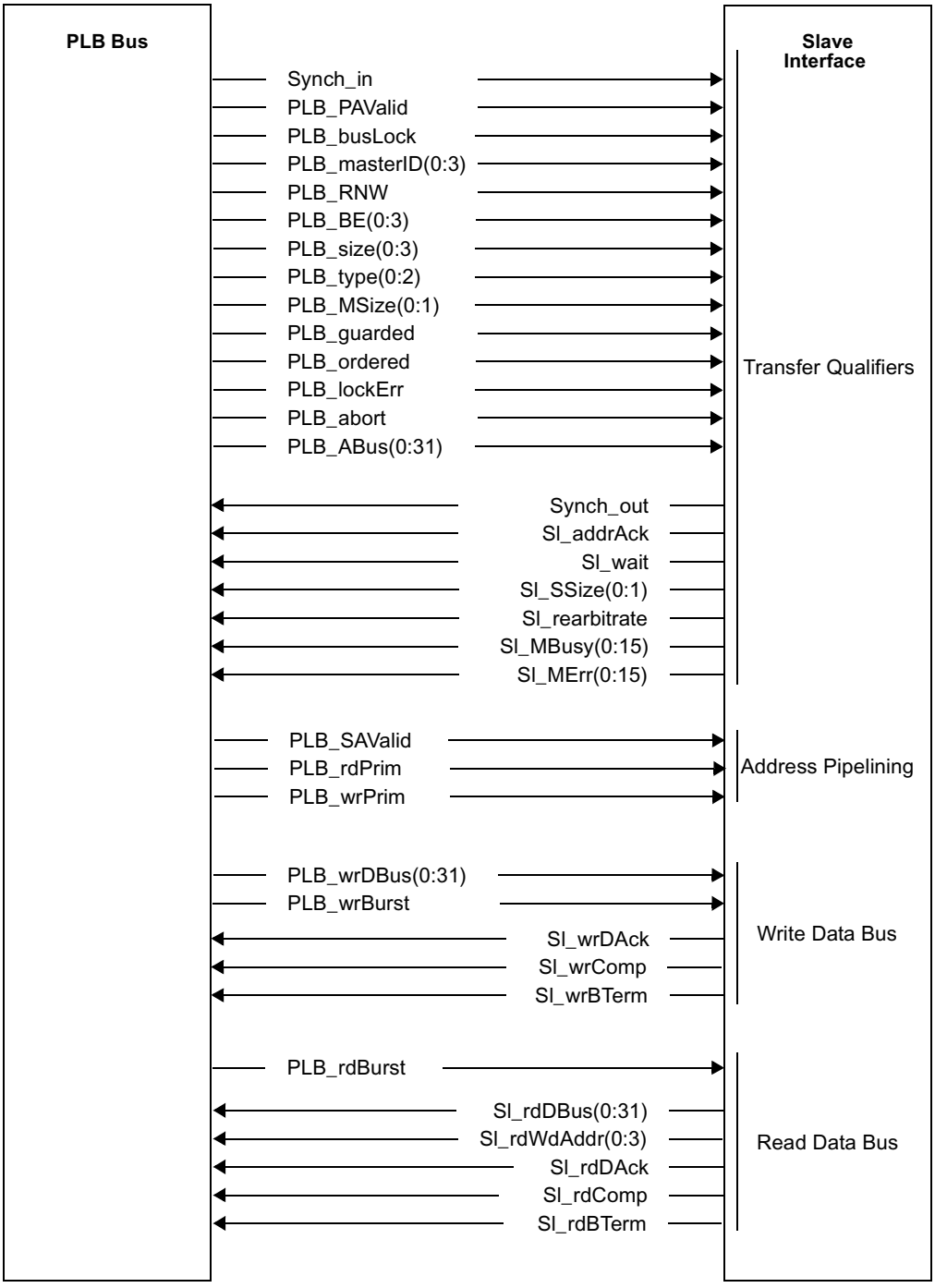


Figure 6. PLB Slave Interface

5.3.1 Slave Model Operation

The PLB slave model operation section discusses the slave bus commands and modes, internal slave data memory structure and look-up algorithms, ordered write cycles (PLB_ordered), internal slave memory checking, burst modes, conversion cycles with different PLB device sizes, and pipeline modes.

5.3.2 Slave Bus Commands and Modes

The PLB slave model responds to PLB memory cycles (PLB_type=000) when there is a valid request on the PLB bus and the PLB address is within the PLB slave's configured address space. Separate read_response and write_response commands allow programmable, overlapped read and write data phases. The slave model stores the read and write response parameters in separate command arrays which allows independent and parallel control of the data buses. Although address phases occur sequentially, it is possible for a slave to perform out of order memory access relative to the address phases depending on how the master cycles are generated and how the slave model is programmed to respond to the bus cycles. The PLB slave model memory uses the Sl_rdDAck and Sl_wrDAck signals to perform internal memory accesses needed to satisfy PLB read and write bus cycle requests.

In the PLB slave model, memory access for reads always occurs one clock before the Sl_rdDAck assertion, and the memory access for writes always occurs in the clock that Sl_wrDAck is asserted. Also, writes have precedence in the memory access logic of the PLB slave memory model. If Sl_rdDAck is asserted in a cycle after a wrDAck, then the read data always reflects the new data written. If a rdDAck is asserted in a cycle before or during a wrDAck, then the read data always reflects the old data. Since the PLB architecture allows out of order rd/wrDAcks with respect to addrAck from the same slave, it is the responsibility of the system designer to resolve data coherency issues when simultaneous write/read cycles occur.

Slave memory and configuration parameters are initialized at simulation time 0 with the command file generated by the PLB bus functional compiler. The slave response commands are initialized in the slave command array at simulation time 0, but they are executed sequentially during simulation. Separate read and write internal command pointers advance when bus transfers complete, a rearbitrate occurs, or a master abort occurs. Intermediate data checking during simulation can be done by using memory check commands which are executed when an intercommunication signal is received by the slave. These memory check commands are independent of the slave response command flow.

The default size of the bus command array is 256 entries. This may be increased or decreased by updating the PLB_slave_CMD_array_size parameter in the PLB_DCL declaration HDL file of the toolkit.

5.3.3 Command Modes

This section discusses the possible modes that are available for a PLB slave device. PLB slaves may be configured to act in either Configuration mode, Command mode, or Auto mode. A description of these modes are:

- **Configuration Mode**

This response mode allows the user to change the slave read/write response on a test case basis. For each PLB master cycle that is received, the PLB slave model uses an internal configuration register to respond to all PLB read or write memory cycles which are active within its address space. This mode is the default configuration of the PLB slave model. The configuration delay values can be initialized using configuration commands such as:

configure(read_aack_delay=2,read_dack_delay=2-1-1-1,write_aack_delay=1,write_dack_delay=1-1-1-1)

This mode is the default mode of the PLB Slave model, and the user is not required to specify a BFL configuration parameter for this mode.

- **Command Mode**

  This response mode enables the user to change the slave read/write response on a transaction basis. For each PLB master cycle that is received, the PLB slave model uses a response command from an internal read or write command array when a PLB read or write memory cycle is active within its address space. The PLB slave model uses command mode instead of configuration mode when read/write response commands are programmed in a test case such as:

  read_response(aack_delay=2,dack_delay=3)

  When the BFC parses “response” commands, it automatically initializes the appropriate internal slave model register to enable “command” mode. The user is not required to specify a BFL configuration parameter to use this mode.

- **Auto Mode**

  This response mode enables various cycle responses without using pre-programmed read/write slave response commands. In this mode, the PLB slave model generates address and data phase completion signals automatically, based on an internal random number generator and pre-programmed minimum and maximum delays. The random number generator selects a new delay value between the minimum and maximum using a uniform distribution function. The minimum and maximum range may be initialized by the user using a configuration command. The PLB slave model requires a configuration command for enabling auto mode. The following configure statement is an example:

  configure(slave_auto_mode=true)

5.3.4 Internal Slave Data Memory Structure and Look-Up Algorithms

The PLB slave model contains an internal memory array which can be initialized by the user with mem_init bus functional commands. Each entry of the memory array represents a 64-bit address and 128-bit data.

- PLB_slave_addr_array(0 to 63) - bits 0 to 61:address, bit 62:dirty bit, bit 63:valid bit
- PLB_slave_data_array(0 to 127) - 4 word data field
- PLB_slave_dirty_array(0 to 15) - bytes written (half word) field

Bits 62 and 63of the PLB_slave_addr_array are not used for addressing bytes within the PLB slave model memory, but they are used by the internal slave model memory interface logic to maintain the status of memory array elements. The valid bit is set during model initialization for each memory entry used and also during simulation when a memory entry is allocated in the case of a write memory access miss. When a write command reloads a memory entry with bus data, the dirty bit is set. This

may be used with test environments to check that bus commands are executed and master memory is updated in addition to data miscompares.

For each byte that is written to the slave memory array during simulation of PLB write cycles to the slave, the PLB_slave_dirty_array is updated to indicate that the corresponding byte has been updated. This feature can be used in simulation environments which need to detect or count exactly which bytes were written to in the slave so that a more comprehensive checking algorithm may be implemented. In addition, there are two integer variables within the slave memory process which accumulate the total number of bytes which have been read from the slave memory array and also the total number of bytes which have been written to the array. These two variables are called:

– slave_mem/total_bytes_written

– slave_mem/total_bytes_read

The default size of the slave memory array is 64k bytes. This may be increased or decreased by updating the PLB_slave_mem_array_size parameter in the PLB_DCL declaration HDL file of the toolkit.

The PLB slave model supports two different memory look-up algorithms used to access the internal memory arrays: fully associative and direct mapped. Each of these is described below:

- **Fully associative, linear search look-up mode**

The default mode for the PLB slave model is linear search mode. This algorithm provides the flexibility to initialize the slave model with many non-contiguous data patterns within an entire 64-bit address range. In the linear look-up mode, each array element status is maintained by the slave. Initialization with mem_init commands causes the valid bit to be set. Any bus cycles which have an address that matches a valid entry in the memory array will use the memory array for the bus functional command operation. For write cycles, the internal master data memory will be updated when Sl_wrDAck is asserted. For read cycles, data from the internal slave memory is loaded into a buffer one clock before Sl_rdDAck assertion.

- **Direct mapped mode**

When the PLB slave model is programmed in direct mapped mode, a subfield of the address (PLB_Abus) is used to directly reference the PLB slave internal memory. The direct mapped mode is useful when simulations contain many contiguous slave address, as the lookup algorithm is more efficient than the linear search mode.

Note: The internal slave address subfield is automatically calculated by the model as the slave memory configuration parameters are set in the plb_dcl.vhd file. The user is not required to configure these indices within the toolkit.

5.3.5 Ordered Write Cycles (PLB_ordered)

When the slave model acknowledges a write cycle when PLB_TAttribute[8] signal is active, the PLB slave model will delay all subsequent internal data accesses until the previous write cycle is internally complete. The PLB slave model, however, may acknowledge additional cycles (assert Sl_addrAck) during ordered writes.

5.3.6 Internal Slave Memory Checking)

Mem_check slave commands allow automatic slave data memory lookup and compares without performing PLB bus operations. Error messages will be generated when data miscompares occur with the mem_check commands. When the slave is programmed with mem_check commands, it waits to receive a synchronization signal and then performs the data memory lookup. The order in which the mem_check commands appear in a BFL file are independent of the slave read_response, write_response, and mem_init commands.

– Slave_check_addr_array(0 to 31) - bits 0 to 29:address, bits 30.31 are unused

– Slave_check_data_array(0 to 31) - 4 byte data field

The default size of the slave check memory array is also 128 entries. This may be increased or decreased by updating the slave_check_array_size in the declaration HDL file of the toolkit.

5.3.7 Burst Modes

The PLB slave model supports the PLB burst protocol. The model accesses the internal slave memory during the initial bus request and also during each data phases of the burst transfer. The PLB Slave model supports two types of counting mechanisms for asserting sl_rd/wrBterm. It can count by the number of clocks from the start of the data phase of the transaction, or by the number of sl_rd/wrDack assertions from the start of the transaction. The PLB slave will assert the burst terminate signal when “burst_term” parameter is used and the corresponding count parameter has expired. The PLB slave model also supports the PLB fixed length burst protocol. The slave model will automatically assert the Sl_rd/wrBterm signal when the “fixed_burst_mode” is enabled, the PLB_BE signal is non-zero, the PLB_size signal indicates a burst cycle, and the corresponding number of data transfers has occurred. To enable fixed length burst in the slave model, use the following slave configuration statement:

– configure(fixed_burst_mode=1) -- Default is 0 (disabled)

Note: The PLB Architecture allows slave designs to assert sl_rdBterm anywhere from one clock after sl_AddrAck (or PLB_rdPrim) to sl_rdComp for read transactions, and together with sl_AddrAck (or one clock after PLB_wrPrim) to sl_wrComp for write transactions. It is common for PLB Slave designs to assert sl_rd/wrBterm together with the second to last sl_rd/wrDack assertions. However, in order to achieve 100 percent functional verification of a PLB master design, the Burst_Term_Mode=Clk should be used to assert the sl_rd/wrbTerm signals in every possible combination of assertions for the entire set of burst transactions.

5.3.8 Conversion Cycles with Different PLB Device Sizes

The PLB slave model can be configured to be a 32 bit, 64 bit, or 128 bit device. When the PLB slave model asserts Sln_AddrAck with a PLB_MSize which represents a master that is smaller than the configured PLB slave size (Sln_SSize(0:1)), the PLB slave will automatically adjust the data bus access and internal memory access to correspond to the transaction size as indicated by the PLB_MSize and Sl_SSize values during the Sl_addrAck assertion. For example, if a 32 bit PLB master initiates a 4 word read line transfer to a 64 bit slave, the slave will drive 4 Sl_rdDack signals and drive bits 0:31 of the Sl_rdDBus. However, if a 64 bit PLB master initiates a 4 word read line transfer to a 64 bit slave, the slave will drive 2 Sl_rdDAck signals and drive bits 0:63 of the Sl_rdDBus. To configure data bus size of the PLB slave model, use the following command:

– configure(SSize=xx)

where “xx” is “00” for a 32 bit device, “01” for a 64 bit device, or “10” for a 128 bit device.

5.3.9 Pipeline Modes

The PLB slave model samples PLB_SAValid to detect the occurrence of pipelined address cycles on the PLB. From a test case, the user may selectively disable read or write address pipelining in the PLB slave model by setting the read/write_addr_pipeline_mode parameters. For example:

```
configure(read_addr_pipeline_disable=1)   -- disable address pipelining for read cycles
                                          -- by ignoring PLB_SAValid during read cycles
```

Note: Although address pipelining can be disabled in the PLB slave model by user configuration commands, and also by simply not connecting the PLB_SAValid signal at the slave interfaces, it is important to test pipelining conditions whenever a target system (or follow-on systems) includes pipelining slaves. Therefore, complete and thorough verification with pipelining enabled and multiple slave devices is recommended for most functional verification efforts.

5.4 PLB Monitor

The PLB monitor is a model which samples all PLB signals in every clock cycle. It checks for violations of the PLB architectural specification during simulation. When a warning or error condition occurs by the PLB arbiter, master, or slave device, the model assigns an error register and reports the corresponding condition to the user with a display message.

Figure 7 demonstrates all PLB monitor interface input/output signals. See PLB architecture specifications for detailed functional description of the signals.

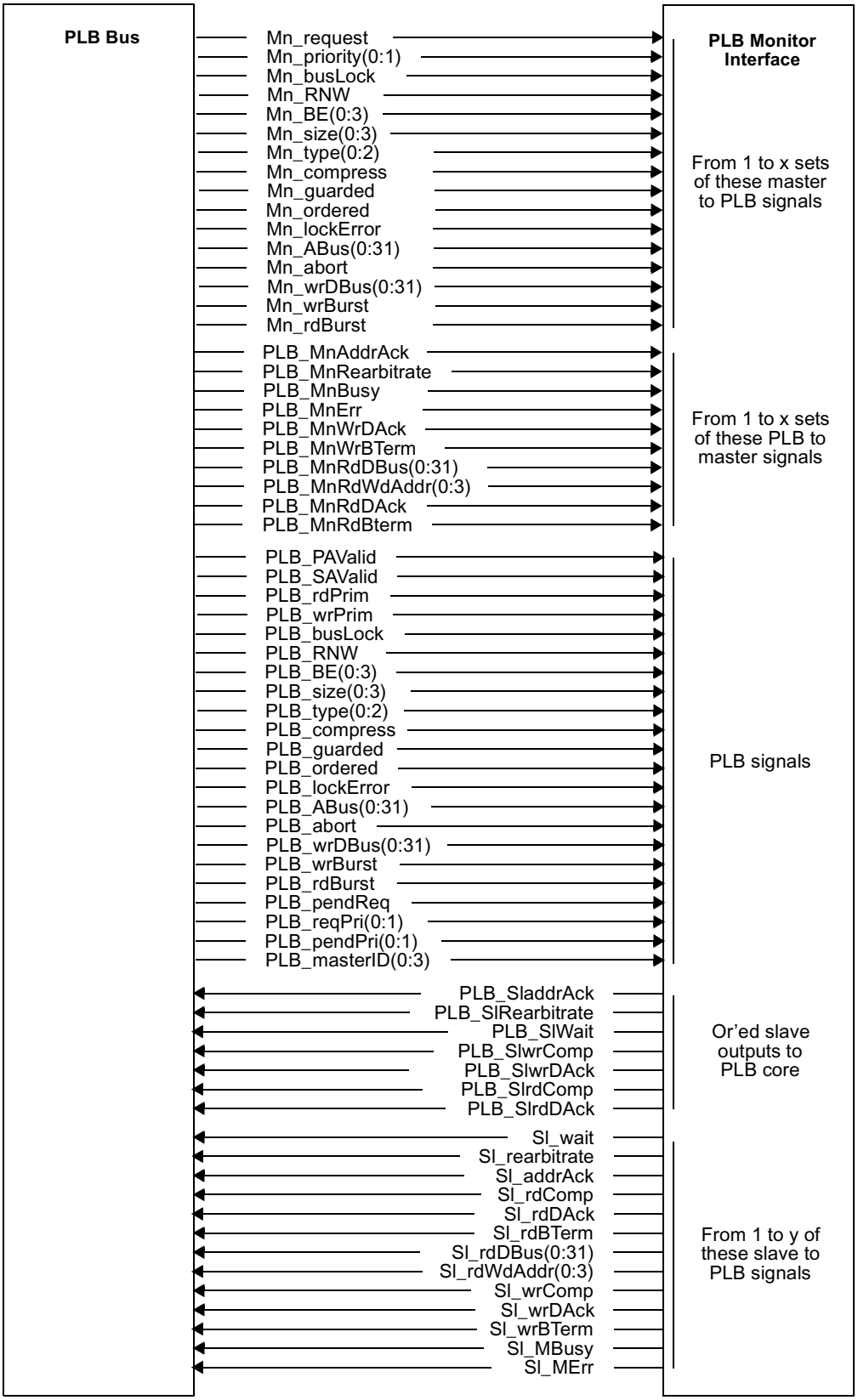


Figure 7. PLB Monitor Interface

Chapter 6. PLB Bus Functional Language

The bus functional models in the PLB model toolkit can be controlled through command files which contain information on how to initiate and respond to bus cycles. The command files also contain model configuration information. The general form of a bus functional language command is specified as follows: *command (parameters)*

Parameters define attributes for bus cycles and configurations. A bus command may have more than one parameter and have the following format: *parameter* = *value*. A parameter value may be either a scalar or enumerated type (that is, a string).

Note 1: Bus functional language (BFL) comments are delineated using the "--" or "//" string. All characters after the comment delineator are ignored by the BFL parser. An end of line character terminates a comment.

Note 2: Bus functional language is case insensitive.

Note 3: A * denotes an optional parameter, otherwise the parameter is required.

Note 4: There is no restriction on the order in which parameters may be specified within a command.

Note 5: If multiple parameters are specified for the same command, they must be separated by commas.

6.1 BFM Device Configuration Commands

This section discusses set_device (), path = [string], and device_type = [string] configuration commands.

6.1.1 Set_Device () Command

The set_device command selects a model to initialize. Within a BFL file (test case), there may be only one set_device () command used per model initialization.

- Path = [String]

  The path = [string] specifies the path of the model within the test bench hierarchy. The commands following the set_device command are used to initialize the specified device. Additional set_device commands are used to specify the initialization of other devices.

- Device_Type = [String]

  The device_type = [string] specifies the type of model being initialized. The valid strings for this toolkit are PLB_master, PLB_slave, or PLB_monitor. Any mismatch in the device_type parameter and the model being initialized will cause an initialization error.

  Example: set_device(path=/plb_complex/m0,device_type=plb_master)

6.2 PLB Alias Commands

This section discusses set_alias () and name = value alias commands.

6.2.1 Set_Alias () Command

The set_alias is an optional command which sets up an alias to be used for string substitution in the bus functional command parameters. The aliases must be set up before they are used. Multiple aliases may be set up. It is possible to have alias files separate from BFL command files by invoking the BFC with multiple BFL files. The syntax for the set_alias command is set_alias(target_name=string_name). When an alias is established, the BFC automatically substitutes the string_name for every occurrence of the target_name.

Example: set_alias(TARGET_REG = 01020304)

In this example, the BFC automatically replaces every occurrence of the string “TARGET_REG” with string “01020304”.

6.3 PLB Master Commands

The PLB master command types include configuration, bus cycle, and internal memory access commands.

6.3.1 Configure () Command

The configure command allows the user to configure different PLB model attributes. One or more parameters can be used with a single “configure” command, and multiple “configure” commands may be used in a single test case. It is important to note that the configure commands are only executed at time 0 in simulation.

6.3.1.1 BFL Mode Configuration Parameters

- *** msize = [2 bit]**

  This parameter specifies the Mn_Msize signals of the PLB. The valid size combinations are listed in the PLB architecture specifications under transfer qualifier signals. This parameter may also be used in each read/write master command. However, when this parameter is used in a configuration statement it is retained throughout the simulation as a master configuration parameter.

  Example: configure(msize=01).

- *** burst_continue_mode = [boolean] default:false**

  When this parameter is set to true and a burst transfer is terminated early, the master will automatically issue additional requests to satisfy the burst count. Otherwise, the master command which was terminated early by a slave or other cycle terminating condition such as master abort or rearbitrate will be considered complete.

- *** msg_disable = [bit] default:0**

  This parameter disables all message generation of the PLB master model, including read data error message generation.

6.3.1.2 Automatic Command Mode Configuration Parameters

- *** master_auto_mode = [boolean] default:false**

This parameter specifies whether the PLB master behavior should automatically generate random bus requests. When the master model is configured for automatic mode, there is no need to initialize read/write commands since they are ignored while the bus cycle generation is enabled.

- **auto_max_cycle = [integer] default:0**

  This parameter specifies the maximum number of cycles that will be generated automatically and complete on the bus. If the default value of “0” is used, cycles will be produced until the simulation is interrupted/stopped.

- **auto_synch_level = [integer] default:none**

  This parameter specifies the level on the synch bus that will be asserted when the maximum number of automatically generated cycles on the bus have completed. The maximum number of automatically generated cycles is determined by the **auto_max_cycle** value.

- *** seed = [integer] default:1**

  This parameter specifies the seed to be used for the random features of the model. The master model currently uses uniform distribution functions implemented with exclusive-or algorithms to produce random values for parameters which are changed by the model when configured to automatic mode. The user may change the initial seed to meet the random number sequences which are generated, which in turn will produce various master read and write sequences by the model.

- *** master_addr_LO_x = [32/64 bit], * master_addr_HI_x = [32/64 bit]**

  These parameters are used in master_auto_mode and they override the default generic address parameters for the PLB master models. The user may provide up to two non-contiguous address ranges for the same master by changing the ‘x’ from 0 to 1. Note that the addresses specified are **starting** addresses for generated transactions in the automatic mode. If the master_addr_HI_x=000003FF, then this will be a viable starting address for any transaction. For a size value greater than a single byte, the requested address will go outside of the boundary specified by master_addr_HI_x. In order to avoid generating transactions that go beyond the boundary of the address range, specify master_addr_HI_x such that a transfer size equal to the largest value of size_max (including the burst_count_max, if applicable) can be completed without going outside of the address range.

- *** msize_min/max = [integer] default:0/1**

  These parameters specify the range for Msize when master_auto_mode is true. The model automatically assigns a random value to the Msize between (and including) the minimum and maximum values. If the Msize parameter is also specified in a configure statement for the same master, the minimum and maximum values will not be used even if the master is configured for auto_mode.

- *** size_min/max = [integer] default:0/6**

  These parameters specify the range for m_size when master_auto_mode is true. The model automatically assigns a random value to m_size between (and including) the minimum and maximum values. All non-valid sizes are automatically adjusted down to a size of 0000 because some sizes are reserved. This means that is a user specifies a range of 0 to 12 that sizes of 4, 5, 6, and 7 are adjusted down to a single(size=0000) transfer. Also, burst sizes are adjusted according to the master size(msize). This means that a 32 bit master will produce only fullword bursts, a 64 bit master will produce only doubleword bursts, and a 128 bit master will produce only quadword bursts.

- *** size0_min/max = [integer] default:0/3 * size1_min/max = [integer] default:8/12**

  These parameters should be used instead of size_min/max when a user wants to avoid having the model adjust all invalid sizes down to a single(size=0000) transfer(see description of size_min/max). These parameters provide the user with the ability to provide split ranges for the m_size parameter. Thus a user could do the following:

  configure(size0_min=0,size0_max=1,size1_min=10,size1_max=10)

  This would produce singles transfers(size=0000), 4-word line transfers(size=0001), and fullword burst transfers(size=1010).

- *** cmd_min/cmd_max = [1,2] 1:read 2:write default:1/2**

  These parameters specify the random generation of reads and/or writes. If one is specified for both then only reads will be produced, and if two is specified for both then only writes will be produced.

- **single_probability/line_probability/burst_probability = [integer] default:33/33/34**

  These parameters specify the percentage of single, line, and burst transfers that will be generated during the entire random simulation. The total of the three should add up to one hundred. For example, a user could specify the following:

  configure(single_probability=50, line_probability=25, burst_probability=25)

  This would generate single transfers 50% of the time, line transfers 25% of the time, and burst transfers 25% of the time. All three parameters should be specified in order to correctly override the defaults. This is true even if one of the values is set to zero.

- **line4wd_8wd_16wd = [integer-integer-integer] default:33-33-34**

  This parameter specifies the percentage of 4wd, 8wd, and 16wd line transfers that will be generated during an entire random simulation in which the line_probability is greater than zero. The total of the three integer values should add up to one hundred. For example, a user could specify the following:

  configure(line4wd_8wd_16wd=40-30-30)

  Out of all line transfers generated, 40% would be 4wd line transfers, 30% would be 8wd line transfers, and 30% would be 16wd line transfers. This parameter should be used when the line_probability is greater than zero or it is ignored.

- *** burst_count_min/max = [integer] default:1/32**

  These parameters specify the range for burst_count when master_auto_mode is true and burst_probability is greater than zero or size_min/max include burst transfers. The model automatically assigns a random value to burst_count between (and including) the minimum and maximum values.

- **fixed_burst_probability = [integer] default:4**

  This parameter specifies the probability of fixed burst transfers out of all burst transfers when master_auto_mode is true and burst_probability is greater than zero. For example, a user could specify the following:

  configure(single_probability=20,line_probability=20,burst_probability=60,fixed_burst_probability=5
)

  This configure would generate a fixed burst transfer for 1 out of every 5 burst transfers.

- **fixed_burst_count_min/fixed_burst_count_max = [integer] default:2/255**

  These parameters specify the range for the fixed burst transfers when master_auto_mode is true, burst_probability is greater than zero or size_min/max include burst transfers, and fixed_burst_probability is greater than zero. For example, a user could specify the following:

  configure(msize=01, single_probability=20, line_probability=20, burst_probability=60, fixed_burst_probability=5, fixed_burst_count_min=2, fixed_burst_count_max=8)

  This configure would randomly generate fixed bursts of two to eight doublewords. Note: the bursts would be doublewords b/c the msize specifies a 64-bit master.

- **fixed_burst_abort_probability = [integer] default:4**

  This parameter species the probability of early terminated(also called aborted) fixed burst transfers when master_auto_mode is true, burst_probability is greater than zero or size_min/max includes burst transfers, and fixed_burst_probability is greater than zero. For example, a user could specify the following:

  configure(msize=01,single_probability=20,line_probability=20,burst_probability=60,fixed_burst_probability=5,fixed_burst_count_min=2,fixed_burst_count_max=8,fixed_burst_abort_probability=3)

  This configure would randomly generate fixed bursts of two to eight doublewords with one out of every three fixed bursts terminating early. Note: the bursts would be doublewords b/c the msize specifies a 64-bit master.

- *** req_delay_min/max = [integer] default:0/20**

  These parameters specify the range for req_delay when master_auto_mode is true. The model automatically assigns a random value to the req_delay between (and including) the minimum and maximum values.

- *** abort_probability = [integer] default:5**

  This parameter specifies the probability that M_abort will be asserted when master_auto_mode is true. For example, the probability that M_abort will be asserted is approximately 1 out of every 5 (default value) master requests when master_auto_mode is enabled.

- *** abort_delay_min/max = [integer] default:0/17**

  These parameters specify the range for abort_delay when master_auto_mode is true. The model automatically assigns a random value to the abort_delay between (and including) the minimum and maximum values. This parameter is generated for approximately 1 out of every abort_probability cycles when master_auto_mode is enabled.

- *** lock_probability = [integer] default:5**

  This parameter specifies the probability that M_busLock will be asserted when master_auto_mode is true. For example, the probability that M_busLock will be asserted is approximately 1 out of every 5 (default value) master requests when master_auto_mode is enabled.

- *** unlock_delay_min/max = [integer] default:1/50**

  These parameters specify the range for unlock_delay when master_auto_mode is true. The model automatically assigns a random value to the unlock_delay between (and including) the minimum and maximum values when the lock_delay parameter is selected. This parameter is generated for approximately 1 out of every five cycles when master_auto_mode is enabled.

- *** priority_min/max = [integer:0,1,2,3] default:0/3**

These parameters specify the range for m_priority when master_auto_mode is true. The range maximum must not exceed a value of three because the m_priority is only two-bits. Therefore valid priorities are 00, 01,10, and 11.

- *** merr_read_data_check_disable = [integer:0,1] default:0**

  This parameter is used to disable the read data checking in the master when a PLB_MrdErr (signal driven by the slave to indicate the slave has encountered an error during a read transfer initiated by the master) is also asserted. To disable this check so that the master will not report read data errors, set merr_read_data_check_disable=1.

- * **read_check_disable = [integer:0,1] default:0**

  This parameter is used to disable the read data checking in the master. Disabling this check (setting read_check_disable=1) is useful when the slave in the testbench is not a toolkit slave and may have different data than the toolkit master, which will cause read data comparison errors. Note that if this parameter is set equal to 1, it may mask a real problem in the design.

- * **ordered_probabilty = [integer:0-100] default:0**

  This parameter corresponds to the percentage of write requests in which the master will assert the signal Mn_Tattribute bit 8 (Mn_Ordered) for the duration of the master request phase. For example, if ordered_probability=75, then Mn_Tattribute(8) will be asserted for 75 percent of the write requests from the master.

6.3.1.3 Automatic Data Mode Configuration Parameters

- *** master_auto_data = [boolean] default:false**

  This parameter specifies whether the PLB master behavior should automatically generate data. When the master model is configured for automatic data mode, there is no need to initialize internal memory data since it is generated and checked using the BFM data generation function described in “PLB Bus Functional Compiler” on page 12.

- *** data_seed = [integer] default:1**

  This parameter specifies the seed to be used for the auto data feature of the model. The slave model forms a seed using this root_seed and the byte address. The user may change the initial seed to meet the random number sequences which are generated, which in turn will produce various slave data.

6.3.2 Mem_Init () Command

The mem-init command initializes the internal master memory and is only executed at simulation time 0.

- **addr = [4/8 byte]**

  This parameter specifies a master 32-bit or 64-bit address for data array initialization.

- **data = [4/8 byte]**

  This parameter specifies the master memory data to initialize in the corresponding PLB device.

  Examples:

  mem_init(addr=00001000,data=01020304)

  mem_init(addr=00000100,data=01020304_00607080)

6.3.3 Reg_Init () Command

The reg_init command initializes an internal master register and is only executed at simulation time 0.

- **DST = [4 byte]**

  This parameter specifies the master 32-bit register to initialize. DST may be CR, SR, or R0-31

  Example: reg_init(R0=01020304)

6.3.4 Read () and Write () Bus Cycle Commands

The read/write bus cycle commands initiate read or write cycles on the PLB bus by causing the bus master to assert the M_request signal. These commands are executed sequentially, therefore the completion of the cycles on the PLB bus influence the simulation time at which the commands are decoded. The following is a list of possible read/write parameters:

- **addr = [4/8 byte]**

  This parameter specifies the 32-bit or 64-bit address.

- **tattr/tattribute= [4 hex] default:0000**

  This parameter enables and specifies the tattribute value.

  Example: read(req_delay=3,addr=00001000,size=0000,be=1111,tattr=A001)

- *** be = [4/8/16 bit]**

  This parameter specifies the Mn_BE (byte-enable) signals of the PLB. The valid byte-enable combinations are listed in the PLB architecture specifications under transfer qualifier signals. This parameter is required for all nonburst and non-line transfers.

- **size = [4 bit]**

  This parameter specifies the Mn_size signals of the PLB. The valid size combinations are listed in the PLB architecture specifications under transfer qualifier signals.

- *** msize = [2 bit]**

  This parameter specifies the Mn_MSize signals of the PLB. The valid size combinations are listed in the PLB architecture specifications under transfer qualifier signals. Note that this parameter will be ignored if MSize was previously defined as a configuration parameter.

- *** req_delay = [integer]**

  This parameter specifies the number of clock cycles for the master to wait before it asserts its M_request signal for the corresponding read or write cycle.

- *** lock = [1 bit] default:0**

  This parameter specifies whether to assert Mn_busLock with Mn_request. If the parameter is not specified then lock is not asserted. The PLB will be locked when a valid PLB_MnaddrAck is asserted. When using the lock parameter, an unlock parameter is required in the same instruction. The lock signal is automatically deasserted if a rearbitrate occurs.

- *** unlock_mode = [enumerated type] default:cycle**

This parameter specifies the mode in which lock is deasserted. This may be specified as clock mode or cycle mode. Clock mode specifies the deassertion of lock as the number of clocks from when lock was previously asserted. Cycle mode specifies the number of cycles to expire before lock is deasserted. In cycle mode, lock is deasserted when Mn_request is deasserted during the last locked cycle.

- *** unlock_delay = [integer:range 1 to 255]**

  This parameter specifies the number of clocks to deassert lock after it was previously asserted, or the number of cycles to elapse before it is deasserted. The mode for the deassertion of lock is specified with the unlock_mode parameter. A cycle is defined by the assertion of PLB_MAddrAck, PLB_MTimeout or M_abort. Note that if two commands are given, both with an unlock_delay > 1, the unlock_delay from the first command will be used to deassert M_buslock and not the second. For example:

  ```
  write(addr=08f9f360,size=0001,priority=11,lock=1,unlock_delay=4,unlock_mode=cycle)
  read(addr=08f9f460,size=0001,priority=11,lock=1,unlock_delay=3,unlock_mode=cycle)
  ```

  In this case, the M_buslock will be deasserted 4 cycles after the write request. The unlock_delay of 3 will be ignored.

- *** type = [3 bit] default:000**

  This parameter specifies the Mn_type signals of the PLB. The valid size combinations are listed in the PLB architecture specifications under transfer qualifier signals. The default value is a memory transfer.

- *** compress = [1 bit] default:0**

  This parameter enables the assertion of the Mn_Compress signal for the duration of the master request phase.

- *** guarded = [1 bit] default:0**

  This parameter enables the assertion of the Mn_Guarded signal for the duration of the master request phase.

- *** ordered = [1 bit] default:0**

  This parameter enables the assertion of the Mn_Ordered signal for the duration of the master request phase.

- *** lockErr = [1 bit] default:0**

  This parameter enables the assertion of the Mn_lockErr signal or the duration of the master request phase.

- **burst_count = [integer]**

  This parameter specifies the total number of data transfers when a burst cycle is used with the size parameter. The master will continue to execute cycles until an internal transfer counter equals the burst_count parameter. Note that this parameter is only used when a burst cycle is specified with the size parameter. For a burst of one data transfer, the rd/wr_burst signals will not be asserted, and the burst_count parameter should be set to ‘1’ or simply omitted from the command.

Note: For fixed length bursts, the PLB Architecture allows the pre-mature de-assertion of the M_rd/wrburst signal before the number of data beat transfers specified by the byte enables. The PLB Architecture also allows Masters to continue bursting beyond the fixed length burst beat count if a slave burst terminate signal is NOT received during a fixed length burst transfer.

To allow verification of this architectural flexibility, the BURST_COUNT parameter can be programmed to a different number of beats than that specified on the byte enables. When the BURST_COUNT parameter is less than the BE data beat count, the burst signal will be prematurely de-asserted. When the BURST_COUNT is greater than the BE data beat count, the master will continue bursting IF a slave burst terminate signal was not asserted to the master.

IT IS STRONGLY RECOMMENDED THAT LOGIC DESIGNS WHICH SUPPORT BURST AND FIXED LENGTH BURST BE TESTED THOROUGHLY WITH ALL COMBINATIONS OF BYTE ENABLES, MASTER BURST ABORTS, AND SLAVE BURST TERMINATES TO MAXIMIZE CORE COMPATIBILITY AND COMPLIANCE

- *** burst_deassert_delay = [integer]**

  This parameter specifies the wait period for number of clocks, after address acknowledge, before de-asserting the burst signal. When this parameter is used instead of the burst_count parameter, the PLB master model asserts its M_wr/rdBurst signal and deasserts it after the number of clocks have been specified by this parameter; the clock count starts when the transaction has been address acknowledged. The master model waits for a final data acknowledge with its burst signal low and continues to the next instruction. When this parameter is used together with the burst_count parameter, the master will wait the specified number of clocks after the next to last rd/wrDAck is sampled active to deassert the Mn_rd/wrBurst signal.

- *** abort_delay = [integer]**

  This parameter specifies the number of clocks to wait before Mn_abort is asserted with respect to the assertion of Mn_request. If the parameter is not specified, Mn_abort is not asserted. If PLB_MAddrAck is sampled active before the abort signal is asserted, the data phase is processed as it would be without the abort.

- *** priority = [2 bit] default:00**

  This parameter specifies the Mn_priority signals of the PLB.

- *** rdword_mode = [1 bit] 0=sequential,1=target word first**

  This parameter specifies the address intermediation order for automatically checking the PLB_Mn_rdWdAddr signals during line transfers. The valid modes are sequential and target word first. An error message will be generated if a comparison error occurs.

  Example: read(req_delay=3,addr=00001000,size=0000,be=1111)

- *** data = [ALU Register]**

  When this optional parameter is used, data is loaded into the specified PLB master register directly instead of being loaded into the internal PLB master model memory. Also, no data comparison checking is performed with the data received from the PLB bus.

6.3.5 Mem_Update () Command

The mem_update command updates master memory during the decode and execution of master bus commands. It is a serializing instruction meaning that all outstanding bus cycles complete before the internal memory array is updated.

- **addr = [4 byte]**

  This parameter specifies the 32-bit address of the internal memory data value to be updated.

- **data = [4 byte] or [CRx,Rx,SR]**

This parameter specifies the data which will be used to update the PLB master internal memory. An immediate 4-byte data value or an internal register may be used.

Example: mem_update(addr=00001000,data=05060708)

6.3.6 Reg_Update () Command

The reg_update command updates an internal master register during the decode and execution of master bus commands. Each reg_update command executes in one PLB clock cycle.

- **DST = [4 byte]**

  This parameter specifies the 32-bit destination register of the internal register to be updated.

  Example: reg_update(R0=05060708)

6.3.7 Send () Command

The send command causes a vectored intercommunication signal to be asserted at the corresponding level parameter and does not directly cause PLB bus activity. The synch_Out signal is asserted for one clock per send instruction and may be used to communicate between multiple model instantiations in order to coordinate bus activity. The send is executed sequentially along with the read/write commands and is a serializing instruction, meaning that the model waits for all previously issued bus cycles to complete on the bus before sending its intercommunication signal.

- **level = [integer: range 0 to 31]**

  The level parameter allows for one or more send signals to be asserted on the synch_Out bus. Multiple levels may be used in the same clock by listing more than one level. Note that it is possible for a user to use the same intercommunication level with multiple masters, however the assertion of the same level in the same clock by multiple masters will not be distinguished on the intercommunication send vector.

- *** req_delay = [integer]**

  This parameter specifies how many clock cycles the PLB master needs to wait before it asserts the synch signal for the corresponding send instruction. The counter starts when all cycles on the bus are complete since the send instruction is a serializing instruction.

  Example: send(level=1)

6.3.8 Wait () Command

The wait command causes the bus master to suspend instruction decode until an intercommunication signal is received. The wait instruction is a serializing instruction and is executed sequentially along with the read/write commands. In addition, all previously issued bus cycles need to complete on the bus before the intercommunication signal on the synch_In bus is recognized.

- **level = [integer: range 0 to 31]**

  The level parameter specifies the synch_In signal to be sampled before instruction execution continues.

  Example: wait(level=1)

6.3.9 Branch () Command

The branch command enables the user to re-direct the command decode location within the PLB master decode unit internal command array. It also enables looping and jumping. Branch targets are established by specifying label attributes with a PLB master command, such as:

```
T1:read(req_delay=3,addr=00001000,size=0000,be=1111)   -- establish a branch target called T1
```

Note: The ":" between the label and master command delineates the two BFL constructs. Each branch instruction executes in one PLB clock cycle.

- **SRC**

  This parameter specifies the condition register to determine whether the branch is taken. A condition register is used in this comparison.

- **OP**

  This parameter specifies the comparison operator to compare the condition register. The valid conditions are:

  LT: less than

  GT: greater than

  EQ: equal

  NL: not less than

  NG: not greater than

  NE: not equal

- **Label**

  This parameter specifies the BFL destination of the branch if the condition register matches the operator parameter. The label must be defined in the test case using the label command attribute.

  Example: branch (CR3, LT, Label1)

6.3.10 Move () Command

The move command enables the user to move a 32-bit internal register or internal memory value to an internal register. Register to Register Move instructions execute in one clock cycle. Internal memory to register Move instructions are decoded in one clock, and the register update occurs in the next clock.

Note: Register data is always available in the next clock for following instructions.

- **DST=SRC**

  The SRC parameter specifies the internal register or 32-bit internal memory value to be used as the source of the move instruction. The DST parameter specifies the internal register which is updated with the SRC parameter value. The DST may be CR, SR, or R0-31.

  Example: move(CR,R0)

  Example: move(CR,12345678)

6.3.11 Compare () Command

The compare command enables the user to compare two values and store the results in the condition register. Each compare instruction executes in one PLB clock cycle.

- **SRC (two are required)**

  This parameter specifies the internal register value to be used as a source of the compare instruction. Each compare instruction requires two SRC parameters. Valid values are SR,R0-31.

- **DST**

  This parameter specifies the condition register destination of the compare instruction. Valid values are CR0-7.

  Example: compare(R0,R1,CR1)

6.3.12 Add () Command

The add command updates an internal master register with the result of an addition between an internal register and an immediate value. Each add command executes in one PLB clock cycle.

- **DST, [hex value up to 4 bytes]**

  These two parameters specify the 32-bit internal destination register to be updated and the immediate hexadecimal value to be added to the destination register. The DST must be R0-31, SR, or CR.

  Example: add(R0,01)

6.3.13 Sub () Command

The sub command updates an internal master register with the result of a subtraction between an internal register and an immediate value. Each sub command executes in one PLB clock cycle.

- **DST, [hex value up to 4 bytes]**

  These two parameters specify the 32-bit internal destination register to be updated and the immediate hexadecimal value to be subtracted from the destination register. The DST must be R0-31, SR, or CR.

  Example: sub(R0,01)

6.3.14 AND () Command

The AND command updates an internal master register with the result of a bit-wise AND operation between an internal register and an immediate value. Each AND command executes in one PLB clock cycle.

- **DST, [hex value up to 4 bytes]**

  These two parameters specify the 32-bit internal destination register to be updated and the immediate hexadecimal value to be AND'd with the destination register. The DST must be R0- 31, SR, or CR.

  Example: and(R0,FFFFFFF0)

6.3.15 OR () Command

The OR command updates an internal master register with the result of a bit-wise OR operation between an internal register and an immediate value. Each AND command executes in one PLB clock cycle.

- **DST, [hex value up to 4 bytes]**

  These two parameters specify the 32-bit internal destination register to be updated and the immediate hexadecimal value to be OR'd with the destination register. The DST must be R0- 31, SR, or CR.

  Example: or(R0,01)

6.3.16 Shift_left() Command

The Shift_left command updates an internal master register with the result of a bit-wise Shift_left operation between an internal register and an immediate value. Each Shift_left command executes in one PLB clock cycle.

- **DST, [integer value]**

  These two parameters specify the 32-bit internal destination register to be updated and immediate integer value to indicate the number of bit positions to shift the specified register. The DST must be R0- 31, SR, or CR.

  Example: shift_left(R0,1)

6.3.17 Shift_right() Command

The Shift_right command updates an internal master register with the result of a bit-wise Shift_right operation between an internal register and an immediate value. Each Shift_right command executes in one PLB clock cycle.

- **DST, [integer value]**

  These two parameters specify the 32-bit internal destination register to be updated and immediate integer value to indicate the number of bit positions to shift the specified register. The DST must be R0- 31, SR, or CR.

  Example: shift_right(R0,1)

6.4 PLB Slave Commands

The PLB slave commands are categorized into model configuration, memory initialization commands, bus response commands, and memory comparison/check commands. The model configuration commands are used to characterize the slave model by controlling the way in which it responds to PLB cycles. The memory initialization commands are used to set up the slave memory before simulation starts. The memory comparison/check commands can be used to compare expected data with slave memory data during simulation. The bus response commands are used to change the slave bus response parameters such as wait states and termination types.

6.4.1 Configure () Command

The configure command allows the user to configure different PLB model attributes. One or more parameters can be used with a single “configure” command, and multiple “configure” commands may be used in a single test case. It is important to note that the configure commands are only executed at time 0 in simulation.

6.4.1.1 BFL Mode Configuration Parameters

- *** aack_delay = [integer] default=0**

  This parameter specifies the number of clocks to wait before asserting Sl_addrAck from the time PLB_PAValid or PLB_SAValid is active. For example, an aack_delay value of 0 will assert Sl_addrAck together with PLB_PAValid and an aack_delay of 1 will assert Sl_addrAck one clock after PLB_PAValid is recognized.

- *** read_aack_delay = [integer] default=0**

  This parameter specifies the number of clocks to wait before asserting Sl_addrAck from the time PLB_PAValid or PLB_SAValid is active. For example, an read_aack_delay value of 0 will assert Sl_addrAck together with PLB_PAValid and an read_aack_delay of 1 will assert Sl_addrAck one clock after PLB_PAValid is recognized.

- *** write_aack_delay = [integer] default=0**

  This parameter specifies the number of clocks to wait before asserting Sl_addrAck from the time PLB_PAValid or PLB_SAValid is active. For example, an write_aack_delay value of 0 will assert Sl_addrAck together with PLB_PAValid and an write_aack_delay of 1 will assert Sl_addrAck one clock after PLB_PAValid is recognized.

- *** SSize = [2 bit]**

  This parameter specifies the Sln_SSize signals of the PLB. The valid size combinations are listed in the PLB architecture specifications. This parameter may also be used in each read/write slave response command. However, when this parameter is used in a configuration statement it is retained throughout the simulation as a slave configuration parameter.

  Example: configure(ssize=01)

- * **read_dack_delay** = **[integer] default:0**

  This parameter specifies the number of clocks to wait before asserting the first data acknowledge control signal for a read_response. This parameter is relative to the earliest legal assertion of Sl_rdDAck.

- *** write_dack_delay = [integer] default:0**

  This parameter specifies the number of clocks to wait before asserting the first data acknowledge control signal for a write_response. This parameter is relative to the earliest legal assertion of Sl_wrDAck.

- *** burst_term_mode = [enumerated type] default: cycle**

  This parameter specifies whether the PLB slave behavior should terminate burst cycles based on the number of data transfers which have occurred or the number of clocks from the time the burst transaction starts. The valid types are cycle or clk.

- *** rdWord_mode = [1 bit] 0=sequential,1=target word first default:0**

This parameter specifies the address incrementation order for the slave during line transfers. The valid modes are sequential and target word first. When this parameter is used in the configuration statement of the slave, the rdWord_mode will be constant for the entire simulation session.

- *** fixed_burst_mode = [1 bit] 0=disabled, 1=enabled default:0**

  This parameter configures the slave to automatically assert the Sl_rd/wrBterm signal when a fixed length burst cycle is in progress and the corresponding number of data transfer has occurred on the bus.

- *** wait_disable = [1 bit] 0=enabled,1=disabled default:0**

  This parameter specifies if the Sl_wait signal can be asserted. When this parameter is used in the configuration statement of the slave, the wait_disable will be constant for the entire simulation session.

- *** wait_mode = [1 bit] 0=assert on address,1=delay mode default:0**

  This parameter specifies how the Sl_wait signal is asserted. When assert on address mode is used, the slave will assert Sl_wait whenever the PLB_Abus is within the slave address range. When delay mode is used, Sl_wait will be asserted based on both an address match and the wait parameter of the read/write response commands.

- *** read_addr_pipeline_disable = [1 bit] 0=SAValid recognized,1=SAValid ignored default:0**

  This parameter specifies whether the PLB core secondary address valid control signal is recognized by the PLB slave model during secondary read requests. When programmed to recognize PLB_SAValid, the PLB slave model asserts addrAck during PLB_SAValid and queues secondary read requests which are processed after primary requests are complete. The PLB_rdPrim and PLB wrPrim signals are used to switch secondary requests to primary status within the model. When this parameter is used in the configuration statement of the slave, the read_addr_pipeline_mode will be constant for the entire simulation session.

- *** write_addr_pipeline_disable = [1 bit] 0=SAValid recognized,1=SAValid ignored default:0**

  This parameter specifies whether the PLB core secondary address valid control signal is recognized by the PLB slave model during secondary write requests. When programmed to recognize PLB_SAValid, the PLB slave model asserts addrAck during PLB_SAValid and queues secondary write requests which are processed after primary requests are complete. The PLB_rdPrim and PLB wrPrim signals are used to switch secondary requests to primary status within the model. When this parameter is used in the configuration statement of the slave, the write_addr_pipeline_mode will be constant for the entire simulation session.

- *** data_pipeline_mode = [1 bit] 0=overlapped DAck mode,1=sequential DAck mode default:0**

  This parameter specifies whether the PLB slave model may assert read and write data acknowledge signals simultaneously. When programmed for sequential DAck mode, the slave model will not assert read and write DAcks in the same clock. When this parameter is used in the configuration statement of the slave, the data_pipeline_mode will be constant for the entire simulation session.

- *** msg_disable = [bit] 0=disabled 1=enabled default:0**

  This parameter disables all message generation of the PLB slave model, including data error message generation.

6.4.1.2 Automatic Command Mode Configuration Parameters

- *** slave_auto_mode = [boolean] false=disabled true=enabled default:false**

  This parameter specifies whether the PLB slave behavior should automatically respond to bus cycles with varied acknowledge delays and termination types. The valid parameter types are true and false. When the slave model is configured for automatic mode, there is no need to initialize read/write_response commands since they are ignored while auto mode is enabled.

- *** seed = [integer] default:1**

  This parameter specifies the seed to be used for the random features of the model. The slave model currently uses a uniform distribution function implemented with exclusive-or algorithms to produce random values for parameters which are varied by the model when configured in automatic mode. The user may change the initial seed to change the random number sequences which are generated, which in turn will produce varying slave response sequences by the model.

- *** slave_addr_LO_x = [32 bit], * slave_addr_HI_x = [32 bit]**

  These parameters allow a user to override the default generic address decode parameters for the PLB device slaves. The user may provide up to two non-contiguous address ranges for the same slave by changing the ‘x’ from 0 to 1. If more non-contiguous slaves address ranges are necessary, the user may instantiate additional PLB device models in the test bench. It is important that there are no overlapping memory address areas between multiple slaves, since both slaves would attempt to respond to a single cycle. If both address parameters are equal, they are not used as a valid address decode.

- *** read_aack_min/max = [integer] default:0/12**

  These parameters specify the range for aack_delay during read cycles when slave_auto_mode is true. The model automatically assigns a random delay to the aack_delay between (and including) the minimum and maximum values.

- *** write_aack_min/max = [integer] default:0/12**

  These parameters specify the range for aack_delay during write cycles when slave_auto_mode is true. The model automatically assigns a random delay to the aack_delay between (and including) the minimum and maximum values.

- *** read_rearb_min/max = [integer] default:12/15**

  These parameters specify the range for rearb_delay during read cycles when slave_auto_mode is true. The model automatically assigns a random delay to the rearb_delay between (and including) the minimum and maximum values.

- *** write_rearb_min/max = [integer] default:12/15**

  These parameters specify the range for rearb_delay during write cycles when slave_auto_mode is true. The model automatically assigns a random delay to the rearb_delay between (and including) the minimum and maximum values.

- *** read_wait_min/max = [integer] default:12/15**

  These parameters specify the range for wait_delay during read cycles when slave_auto_mode is true. The model automatically assigns a random delay to the wait_delay between (and including) the minimum and maximum values.

- *** write_wait_min/max = [integer] default:12/15**

These parameters specify the range for wait_delay during write cycles when slave_auto_mode is true. The model automatically assigns a random delay to the wait_delay between (and including) the minimum and maximum values.

- *** read_dack_min/max = [integer] default:0/15**

  These parameters specify the range for DAck_delay during read cycles when slave_auto_mode is true. The model automatically assigns a random delay to the DAck_delay between (and including) the minimum and maximum values.

- *** write_dack_min/max = [integer] default:0/15**

  These parameters specify the range for DAck_delay during write cycles when slave_auto_mode is true. The model automatically assigns a random delay to the DAck_delay between (and including) the minimum and maximum values.

- *** read_comp_min/max = [integer] default:0/2**

  These parameters specify the range for comp_delay during read cycles when slave_auto_mode is true. The model automatically assigns a random delay to the comp_delay between (and including) the min and max values.

- *** write_comp_min/max = [integer] default:0/2**

  These parameters specify the range for comp_delay during write cycles when slave_auto_mode is true. The model automatically assigns a random delay to the comp_delay between (and including) the minimum and maximum values.

- *** read_burst_term_min/max = [integer] default:7/10**

  These parameters specify the range for burst_term delay during read cycles when slave_auto_mode is true. The model automatically assigns a random delay to the burst_term delay between (and including) the minimum and maximum values.This parameter is used in conjunction with the burst_term_mode parameter. The delay will count either from clocks (if burst_term_mode=clk) or data cycles (if burst_term_mode=cycle).

- *** write_burst_term_min/max = [integer] default:7/10**

  These parameters specify the range for burst_term delay during write cycles when slave_auto_mode is true. The model automatically assigns a random delay to the burst_term delay between (and including) the minimum and maximum values.This parameter is used in conjunction with the burst_term_mode parameter. The delay will count either from clocks (if burst_term_mode=clk) or data cycles (if burst_term_mode=cycle).

6.4.1.3 Automatic Data Mode Configuration Parameters

- *** slave_auto_data = [boolean] false=disabled true=enabled default:false**

  This parameter specifies whether the PLB slave behavior should automatically generate data. When the slave model is configured for automatic data mode, there is no need to initialize internal memory data since it is generated and checked using the BFM data generation function described in “PLB Bus Functional Compiler” on page 12.

- *** root_seed = [integer] default:1**

This parameter specifies the seed to be used for the auto data feature of the model. The slave model forms a seed using this root_seed and the byte address. The user may change the initial seed to change the random number sequences which are generated, which in turn will produce varying slave data.

6.4.2 Mem_Init () Command

The mem_init command initializes slave memory at simulation time 0.

- **addr = [4 byte]**

  This parameter specifies the 32-bit address.

- **data = [4 byte]**

  This parameter specifies the slave memory data to initialize in the corresponding PLB device.

6.4.3 Read_Response (), Write_Response () Commands

These commands specify attributes for a slave cycle read or write response when an address decode is true. Read_response commands are used to initialize the PLB slave response.

- *** aack_delay = [integer] default=0**

  This parameter specifies the number of clocks to wait before asserting Sl_addrAck from the time PLB_PAValid or PLB_SAValid is active. For example, an aack_delay value of 0 will assert Sl_addrAck together with PLB_PAValid and an aack_delay of 1 will assert Sl_addrAck one clock after PLB_PAValid is recognized.

- *** rearb_delay = [integer] default=0**

  This parameter specifies the number of clocks to wait before asserting the Sl_rearbitrate signal from the clock that PLB_PAValid was previously asserted. No Sl_addrAck is asserted if the rearb_delay parameter is used. Note that rearb_delay is not a valid configuration.

- *** dack_delay = [integer] default=0**

  This parameter specifies the number of clocks to wait before asserting the first data acknowledge control signal for a read_response or write_response. This parameter is relative to the earliest legal assertion of Sl_rdDAck and Sl_wrDAck. For example, a DAck_delay value of 0 for a read_response causes the Sl_rdDAck signal to be asserted 2 clocks after Sl_addrAck is asserted, which is the earliest legal PLB slave data response time for the first Sl_rdDAck. A DAck_delay value of 0 for a write_response causes the Sl_wrDAck signal to be asserted together with the Sl_addrAck assertion, which is the earliest legal PLB slave data response time for the first Sl_wrDAck.

  For line and burst transfers, the dack_delay parameter can be used to specify delays between dacks for a multiple data beat transfer. This is accomplished with the following syntax:

  read_response(dack_delay=2*) or configure(dack_delay=2*)

  These bfl statements will produce a delay of 2 between dacks for a line or burst transfer.

  Another available option is to specify different delays between dacks for a multiple data beat transfer. This is accomplished with the following syntax:

  read_response(dack_delay=2-3-4-5)

This will delay the first dack by 2 in relation to addrack or rdprim, the second dack will be delayed by 3 from the first dack, the third dack will be delayed by 4 from the second dack, and the fourth dack will be delayed by 5 from the third dack. Please note that up to 8 values may be used (dack_delay=x-x-x-x-x-x-x-x) and any dacks beyond 8 will default to the dack delay value of the eighth dack.

- *** comp_delay = [integer] default=0**

  This parameter specifies the number of clocks to wait before asserting the Sl_rdComp or Sl_wrComp signals relative to the last Sl_rdDAck or Sl_wrDAck. For reads, a value of 0 will cause Sl_rdComp to be asserted 1 clock before the last Sl_rdDAck. For writes, a value of 0 will cause Sl_wrComp to be asserted together with the last Sl_wrDAck. A non-zero value will cause a per clock assertion delay relative to the fastest possible assertion time.

- *** wait_delay = [integer:range 0 to 255]**

  This parameter specifies the number of clocks to wait before Sl_wait is asserted with respect to the assertion of PLB_PAValid or PLB_SAValid. If the parameter is not specified and wait_disable is 0, Sl_wait is asserted whenever the PLB_ABus is within the slave address range.

- *** rdWord_mode = [1 bit] 0=sequential,1=target word first**

  This parameter specifies the address incrementation order for the slave during line transfers. The valid modes are sequential and target word first. If this parameter is used in the configuration statement of the slave, the rdWord_mode response parameter will be ignored.

- *** burst_term = [32 bits] or [integer] default:0**

  This parameter specifies the internal burst address or data acknowledge cycle to assert Sl_rdBTerm or Sl_wrBTerm signal. If the slave is configured in burst address terminate mode, a 32-bit address should be specified, otherwise the cycle integer should be specified.

- *** SSize = [2 bit]**

  This parameter specifies the Sln_SSize signals of the PLB. The valid size combinations are listed in the PLB architecture specifications. Note that this parameter will be ignored if the SSize parameter was previously defined as a configuration parameter.

- *** level = [integer: range 0 to 31]**

  This parameter specifies which synchronization signal to send when the slave responds to a read or write.

6.4.4 Merr_Init () Commands

The merr_init commands allow the user to generate a master error for read and write transactions based on a specified address. If the user wishes to generate separate read and write errors, the commands: merr_init_r(addr=xxxx) and merr_init_w(addr=xxxx) can be used. The merr_init_r command generates a master read error based on a specified address and the merr_init_w command generates a master write error based on a specified address. When merr_init is used, an error is generated for both reads and writes at the specified address.

- **addr = [4 byte]**

  The address that when read from or written to should cause a master error. For PLB4X this will cause the assertion of the Sl_MRdErr/Sl_MWrErr signal, whereas for PLB3X this will cause the assertion of Sl_MErr.

6.4.5 Mem_Check () Command

The mem_check command automatically compares slave memory with the specified comparison data. If there is a data comparison error, an error message is generated and the corresponding error detection bit is set in the PLB device.

- **level = [integer: range 0 to 31]**

  This parameter specifies which synchronization signal to wait for before checking the slave memory data.

- **addr = [4 byte]**

  This parameter specifies the 32-bit address.

- **data = [4 byte]**

  This parameter specifies the slave memory data to check in the corresponding PLB device.

6.5 PLB Monitor Commands

The PLB Monitor contains registers for address mapping of slave instantiations. Some of the monitor protocol checks use the address map to provide accurate messaging and perform the actual protocol checks. It is required for the user to set up the Monitor register initializations for the slave address map.

6.5.1 Configure () Command

The configure command allows the user to configure different PLB model attributes. One or more parameters can be used with a single “configure” command, and multiple “configure” commands may be used in a single test case. It is important to note that the configure commands are only executed at time 0 in simulation.

6.5.1.1 Configuration Parameters

- **enable_timeout_report=[bit] 0=disabled 1=enabled default:disabled**

  This parameter enables the monitor to give an error message specifying that a timeout occurred on the bus.

- **addr_check_enable=[1 bit] 0=disabled 1=enabled default:0**

  This parameter enables monitor checks that use the slave address ranges as part of the check. The monitor will default this parameter to disabled because the monitor cannot check slave address related items unless it knows the address ranges for each instantiated slave. If the default slave values of the toolkit are being used then there is no need to specify the slave address ranges in a configure statement because a generic load of the address ranges is performed automatically. However, if a user chooses to vary the slave address ranges in their environment then the monitor must be configured for these values (see slavex_addr_lo_x / slavex_addr_hi_x configure parameters below).In either case, the addr_check_enable must be turned on by the user through monitor bfl.

- **busy_cycle_max=[integer] 0=disabled default:100**

This parameter specifies the maximum number of cycles that the PLB_MnBusy is allowed to be active before issuing a warning (1.22.3). When the value is 0, the warning is disabled.

- **SLAVE0_ADDR_LO_0 = [4 byte], SLAVE0_ADDR_HI_0 = [4 byte], SLAVE0_ADDR_LO_1 = [4 byte], SLAVE0_ADDR_HI_1 = [4 byte],**

  Address map range 0/1 for slave instantiation 0.

- **SLAVE1_ADDR_LO_0 = [4 byte], SLAVE1_ADDR_HI_0 = [4 byte], SLAVE1_ADDR_LO_1 = [4 byte], SLAVE1_ADDR_HI_1 = [4 byte]**

  Address map range 0/1 for slave instantiation 1.

- **SLAVE2_ADDR_LO_0 = [4 byte], SLAVE2_ADDR_HI_0 = [4 byte], SLAVE2_ADDR_LO_1 = [4 byte], SLAVE2_ADDR_HI_1 = [4 byte]**

  Address map range 0/1 for slave instantiation 2.

- **SLAVE3_ADDR_LO_0 = [4 byte], SLAVE3_ADDR_HI_0 = [4 byte], SLAVE3_ADDR_LO_1 = [4 byte], SLAVE3_ADDR_HI_1 = [4 byte]**

  Address map range 0/1 for slave instantiation 3.

- **SLAVE4_ADDR_LO_0 = [4 byte], SLAVE4_ADDR_HI_0 = [4 byte], SLAVE4_ADDR_LO_1 = [4 byte], SLAVE4_ADDR_HI_1 = [4 byte]**

  Address map range 0/1 for slave instantiation 4.

- **SLAVE5_ADDR_LO_0 = [4 byte], SLAVE5_ADDR_HI_0 = [4 byte], SLAVE5_ADDR_LO_1 = [4 byte], SLAVE5_ADDR_HI_1 = [4 byte]**

  Address map range 0/1 for slave instantiation 5.

- **SLAVE6_ADDR_LO_0 = [4 byte], SLAVE6_ADDR_HI_0 = [4 byte], SLAVE6_ADDR_LO_1 = [4 byte], SLAVE6_ADDR_HI_1 = [4 byte]**

  Address map range 0/1 for slave instantiation 6.

- **SLAVE7_ADDR_LO_0 = [4 byte], SLAVE7_ADDR_HI_0 = [4 byte], SLAVE7_ADDR_LO_1 = [4 byte], SLAVE7_ADDR_HI_1 = [4 byte]**

  Address map range 0/1 for slave instantiation 7.

  Example:

```
set_device (path=/plb_complex/mon,device_type=plb_monitor)
  configure(SLAVE0_ADDR_LO_0=00000000)    -- slave 0 range
  configure(SLAVE0_ADDR_HI_0=0003FFFF)
  configure(SLAVE0_ADDR_LO_1=FFFF0000)    -- slave 0 range hi
  configure(SLAVE0_ADDR_HI_1=FFFFFFFF)
  configure(SLAVE1_ADDR_LO_0=00040000)    -- slave 1 range ebiu
  configure(SLAVE1_ADDR_HI_0=0007FFFF)
  configure(SLAVE2_ADDR_LO_0=00080000)    -- slave 2 range
  configure(SLAVE2_ADDR_HI_0=000BFFFF)
  configure(SLAVE3_ADDR_LO_0=000C0000)    -- slave 3 range
```

configure(SLAVE3_ADDR_HI_0=000FFFFF)

- **ssize = [2 bit]**
- **msize = [2 bit]**

6.5.2 Read / Write() Commands

The following set of parameters can be used to create expect values for transactions that occur on the bus. If a value is specified, but mismatches when checked against the actual transaction that occurred on the bus, then the monitor will generate an error message.

- **addr = [4/8 byte]**

  Specifies the expected address value for a bus transaction.

- **be = [4/8/16 bit]**

  Specifies the expected byte enable value for a bus transaction.

- **size = [4 bit]**

  Specifies the expected size value for a bus transaction.

- **aack_delay = [integer]**

  Specifies the expected aack_delay value for a bus transaction.

- **abort_delay = [integer]**

  Specifies the expected abort_delay value for a bus transaction.

- **dack_delay = [integer]**

  Specifies the expected dack_delay for a bus transaction.

6.5.2.1 Sample Monitor Read/Write BFL

The following example shows bfl for a master, slave, and monitor device. The master will perform four writes followed by four reads. The slave will respond to each with the specified aack and dack delays. The monitor will check that each transaction completes with the expected values that are provided by the monitor bfl.

- **PLB Master 0**

  set_device (path=/plb_complex/m0,device_type=plb_master)

  configure(msize=00)

  mem_init(addr=00021000,data=00112233)

  mem_init(addr=00021004,data=44556677)

  mem_init(addr=00021008,data=AABBCCDD)

  mem_init(addr=0002100C,data=AABBCCDD)

  write(addr=00021000,size=0000,be=1111)

  write(addr=00021004,size=0000,be=1111)

  write(addr=00021008,size=0000,be=1111)

write(addr=0002100C,size=0000,be=1111)

read(addr=00021000,size=0000,be=1111)

read(addr=00021004,size=0000,be=1111)

read(addr=00021008,size=0000,be=1111)

read(addr=0002100C,size=0000,be=1111)

- **PLB Slave 0**

  set_device (path=/plb_complex/s0,device_type=plb_slave) -- Initialize device 0

  configure(ssize=00)

  mem_init(addr=00021000,data=AABBCCDD)

  mem_init(addr=00021004,data=AABBCCDD)

  mem_init(addr=00021008,data=AABBCCDD)

  mem_init(addr=0002100C,data=AABBCCDD)

  write_response(aack_delay=4,dack_delay=3)

  write_response(aack_delay=3,dack_delay=3)

  write_response(aack_delay=2,dack_delay=3)

  write_response(aack_delay=1,dack_delay=3)

  read_response(aack_delay=4,dack_delay=2)

  read_response(aack_delay=3,dack_delay=2)

  read_response(aack_delay=2,dack_delay=2)

  read_response(aack_delay=1,dack_delay=2)

- **PLB Monitor BFL**

  set_device(path=/plb_complex/mon,device_type=plb_monitor)

  configure(msize=00,ssize=00)

  write(addr=00021000,size=0000,be=1111,aack_delay=4, dack_delay=3)

  write(addr=00021004,size=0000,be=1111,aack_delay=3, dack_delay=3)

  write(addr=00021008,size=0000,be=1111,aack_delay=2, dack_delay=3)

  write(addr=0002100C,size=0000,be=1111,aack_delay=1, dack_delay=3)

  read(addr=00021000,size=0000,be=1111,aack_delay=4, dack_delay=2)

  read(addr=00021004,size=0000,be=1111,aack_delay=3, dack_delay=2)

  read(addr=00021008,size=0000,be=1111,aack_delay=2, dack_delay=2)

  read(addr=0002100C,size=0000,be=1111,aack_delay=1, dack_delay=2)

6.5.3 Report() Command

This command allows the user to output the transactions that have been recorded by the monitor. Without any arguments, this command simply outputs the contents of the record arrays.

- *** level = [integer: range 0 to 31] default: none**

  The level parameter indicates to output the contents of the record arrays when the level is received through the use of a send command. Multiple levels may be used in the same clock by listing more than one level. The report command outputs the transactions that have been recorded by the monitor. If no transactions have occurred across the bus, then the report statement will not output anything. Proper placement of report and/or send level statements will ensure that pertinent transaction information is not lost during simulation.

Note: The number of transactions the monitor may record is determined by the PLB_MONITOR_RECORD_ARRAY_SIZE (default value is 16). Once the arrays have been filled to capacity, the loss of transactions will occur with each transaction that is recorded. The user is responsible for determining the frequency to use the report command.

Chapter 7. PLB Bus Timing

This section on PLB timing provides various examples with timing diagrams consistent with PLB architecture specifications.

7.1 Read Transfers

Figure 8 shows the operation of a single read data transfer on the PLB.

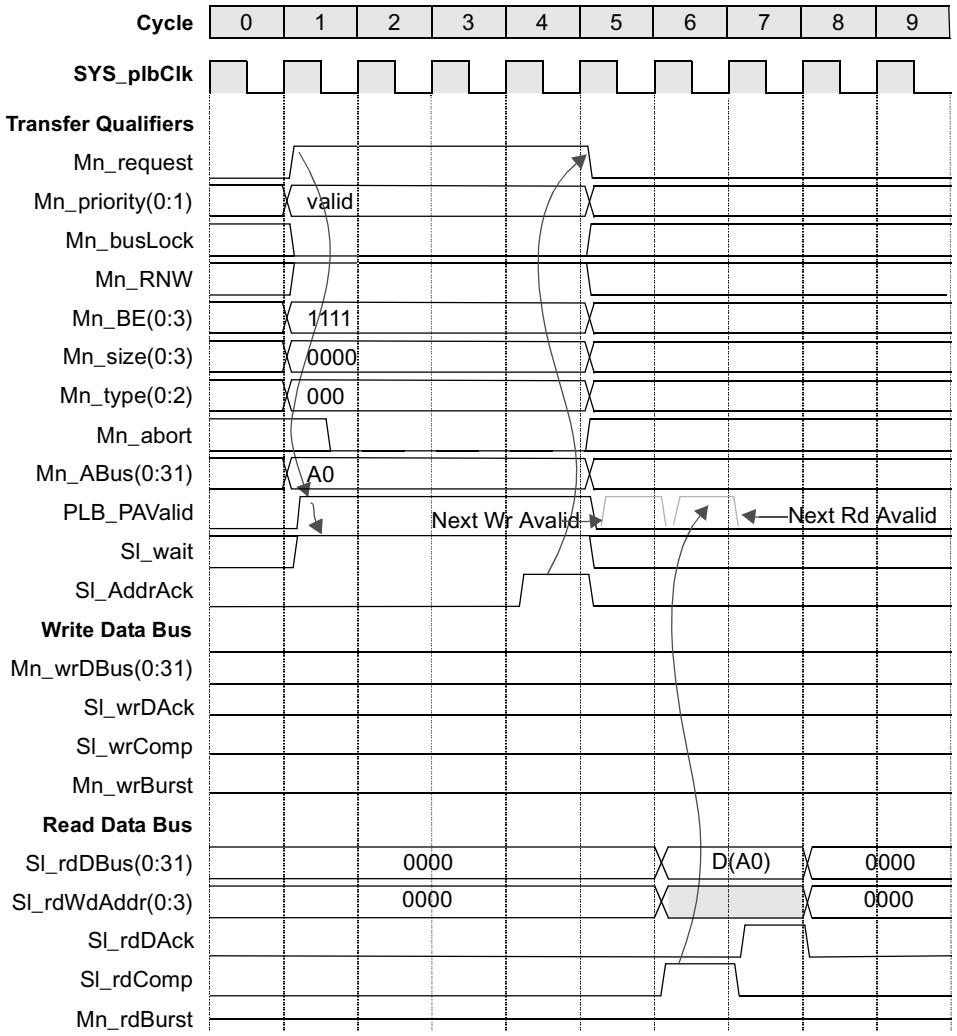


Figure 8. Read Transfers

```
set_device (path=/plb_complex/plb_master0,device_type=plb_master)
   mem_init(addr=00000000,data=00010203)
   read(addr=00000000,size=0000,be=1111)
```

```
    send(level=0)
set_device (path=/plb_complex/plb_slave0,device_type=plb_slave)    -- Initialize device 0
    mem_init(addr=00000000,data=00010203)
    read_response(aack_delay=3,dack_delay=1)
    mem_check(level=0,addr=00000000,data=00010203)
```

7.2 Write Transfers

Figure 9 shows the operation of a single write data transfer on the PLB.

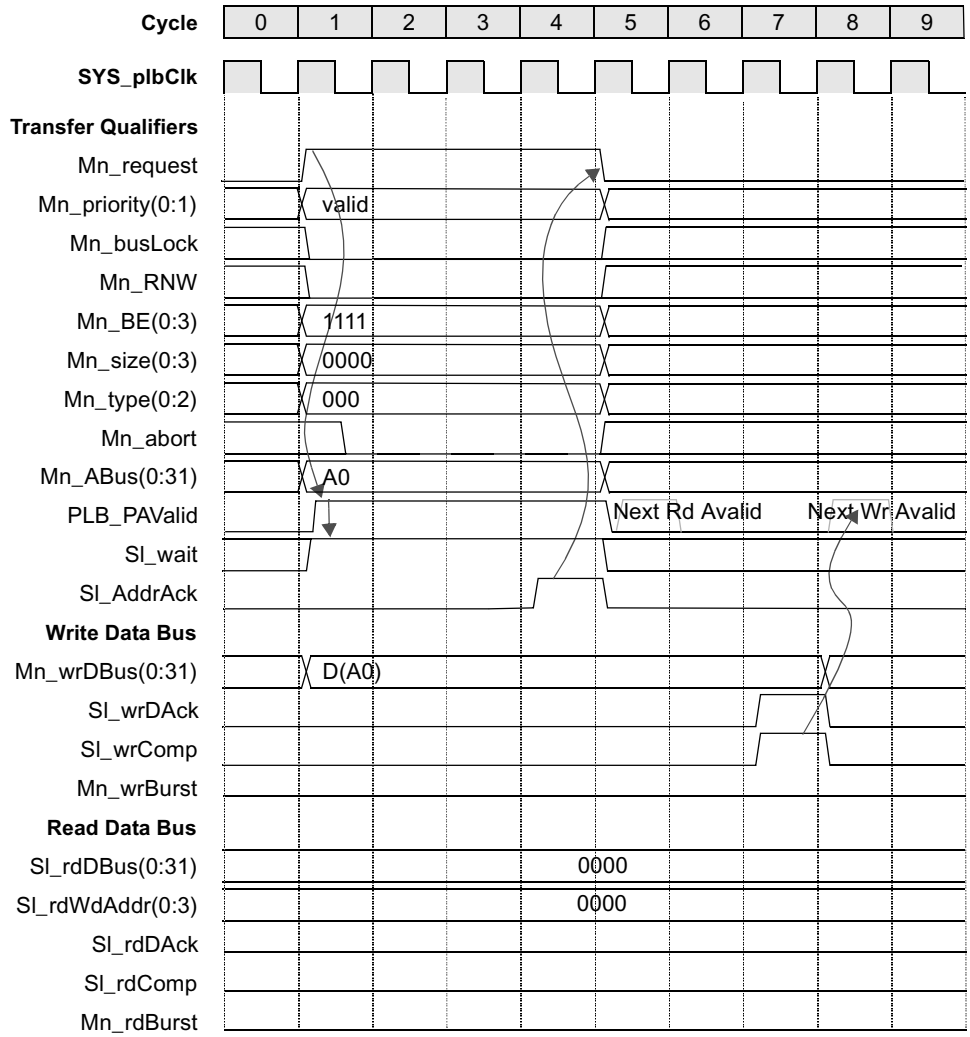


Figure 9. Write Transfers

```
set_device (path=/plb_complex/plb_master0,device_type=plb_master)
   mem_init(addr=00000000,data=00010203)
   write(addr=00000000,size=0000,be=1111)
   send(level=0)
set_device (path=/plb_complex/plb_slave0,device_type=plb_slave)    -- Initialize device 0
   mem_init(addr=00000000,data=00010203)
   write_response(aack_delay=3,dack_delay=3)
   mem_check(level=0,addr=00000000,data=00010203)
```

7.3 Transfer Abort

Figure 10 shows a transfer aborted by the master in the same clock cycle the request was being acknowledged by the slave.

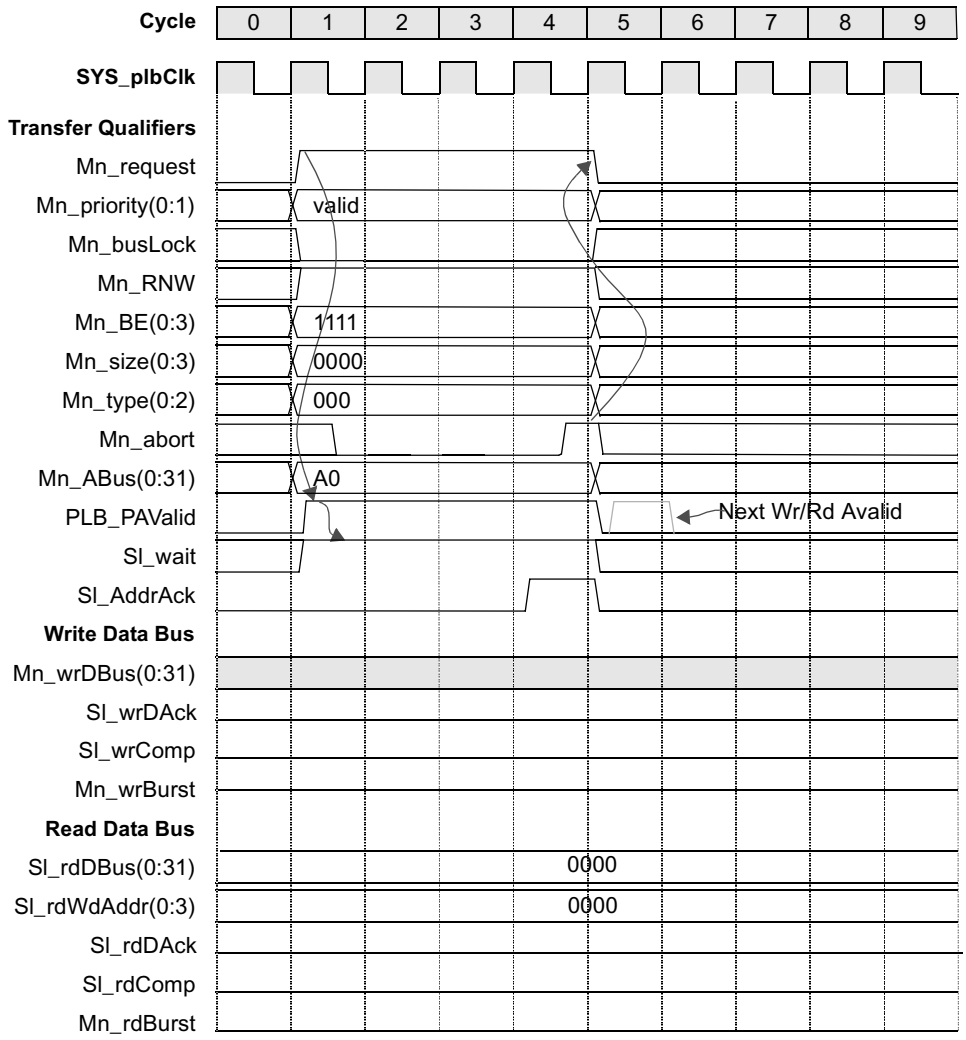


Figure 10. Transfer Abort

```
set_device (path=/plb_complex/plb_master0,device_type=plb_master)
   mem_init(addr=00000000,data=00010203)
   write(addr=00000000,size=0000,be=1111,abort=3)
   send(level=0)
set_device (path=/plb_complex/plb_slave0,device_type=plb_slave)    -- Initialize device 0
   mem_init(addr=00000000,data=00010203)
   write_response(aack_delay=3,dack_delay=0)
   mem_check(level=0,addr=00000000,data=00010203)
```

7.4 Back-to-Back Read Transfers

Figure 11 shows the operation of several back-to-back single read transfers on the PLB.

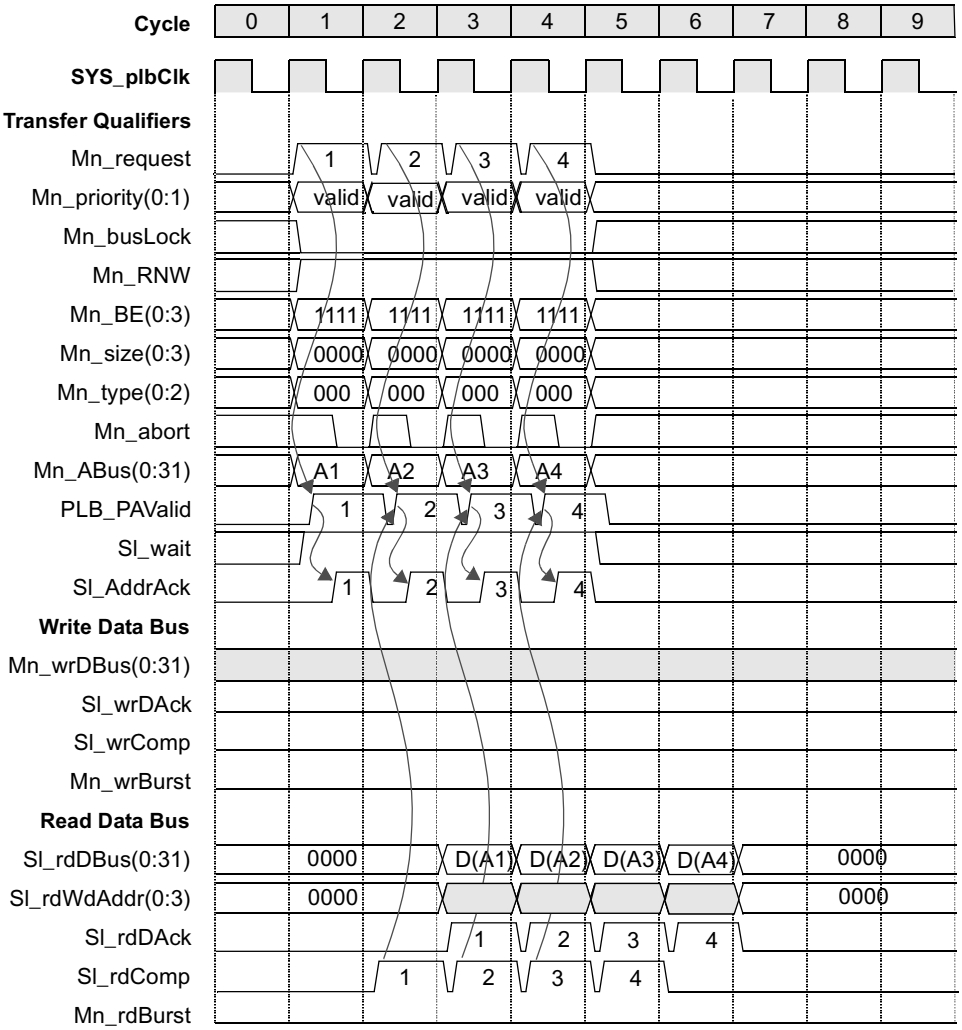


Figure 11. Back-to-Back Read Transfers

```
set_device (path=/plb_complex/plb_master0,device_type=plb_master)
   mem_init(addr=00000000,data=00010203)
   mem_init(addr=00000004,data=04050607)
   mem_init(addr=00000008,data=08090a0b)
   mem_init(addr=0000000c,data=0c0d0e0f)
   read(addr=00000000,size=0000,be=1111)
   read(addr=00000004,size=0000,be=1111)
   read(addr=00000008,size=0000,be=1111)
   read(addr=0000000c,size=0000,be=1111)
```

```
    send(level=0)
set_device (path=/plb_complex/plb_slave0,device_type=plb_slave)    -- Initialize device 0
    mem_init(addr=00000000,data=00010203)
    mem_init(addr=00000004,data=04050607)
    mem_init(addr=00000008,data=08090a0b)
    mem_init(addr=0000000c,data=0c0d0e0f)
    read_response(aack_delay=0,dack_delay=0)
    read_response(aack_delay=0,dack_delay=0)
    read_response(aack_delay=0,dack_delay=0)
    read_response(aack_delay=0,dack_delay=0)
    mem_check(level=0,addr=00000000,data=00010203)
    mem_check(level=0,addr=00000004,data=04050607)
    mem_check(level=0,addr=00000008,data=08090a0b)
    mem_check(level=0,addr=0000000c,data=0c0d0e0f)
```

7.5 Back-to-Back Write Transfers

Figure 12 shows the operation of several back-to-back single write transfers on the PLB.

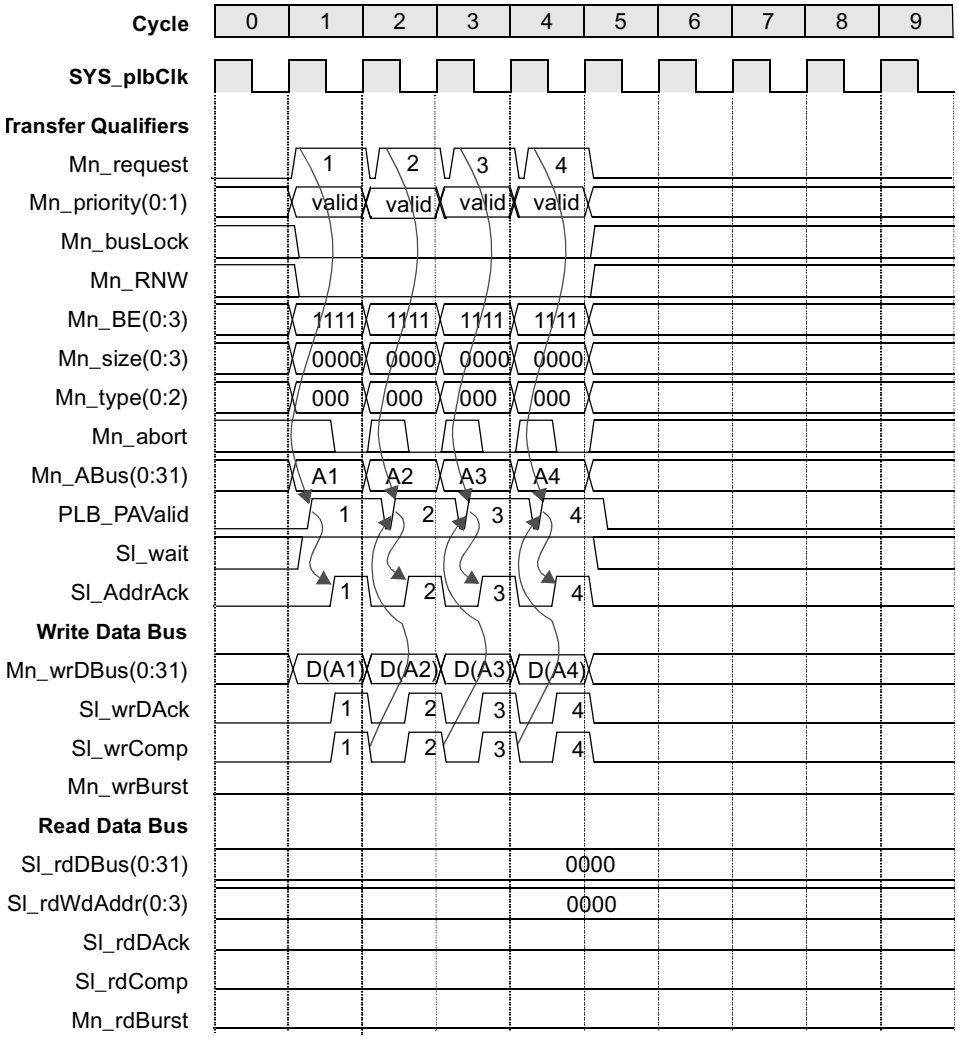


Figure 12. Back-to-Back Write Transfers

```
set_device (path=/plb_complex/plb_master0,device_type=plb_master)
   mem_init(addr=00000000,data=00010203)
   mem_init(addr=00000004,data=04050607)
   mem_init(addr=00000008,data=08090a0b)
   mem_init(addr=0000000c,data=0c0d0e0f)
   write(addr=00000000,size=0000,be=1111)
   write(addr=00000004,size=0000,be=1111)
   write(addr=00000008,size=0000,be=1111)
```

```
    write(addr=0000000c,size=0000,be=1111)
    send(level=0)
set_device (path=/plb_complex/plb_slave0,device_type=plb_slave)    -- Initialize device 0
    mem_init(addr=00000000,data=00010203)
    mem_init(addr=00000004,data=04050607)
    mem_init(addr=00000008,data=08090a0b)
    mem_init(addr=0000000c,data=0c0d0e0f)
    write_response(aack_delay=0,dack_delay=0)
    write_response(aack_delay=0,dack_delay=0)
    write_response(aack_delay=0,dack_delay=0)
    write_response(aack_delay=0,dack_delay=0)
    mem_check(level=0,addr=00000000,data=00010203)
    mem_check(level=0,addr=00000004,data=04050607)
    mem_check(level=0,addr=00000008,data=08090a0b)
    mem_check(level=0,addr=0000000c,data=0c0d0e0f)
```

7.6 Back-to-Back Read - Write - Read - Write Transfers

Figure 13 shows the operation of several back-to-back single read and write transfers on the PLB.

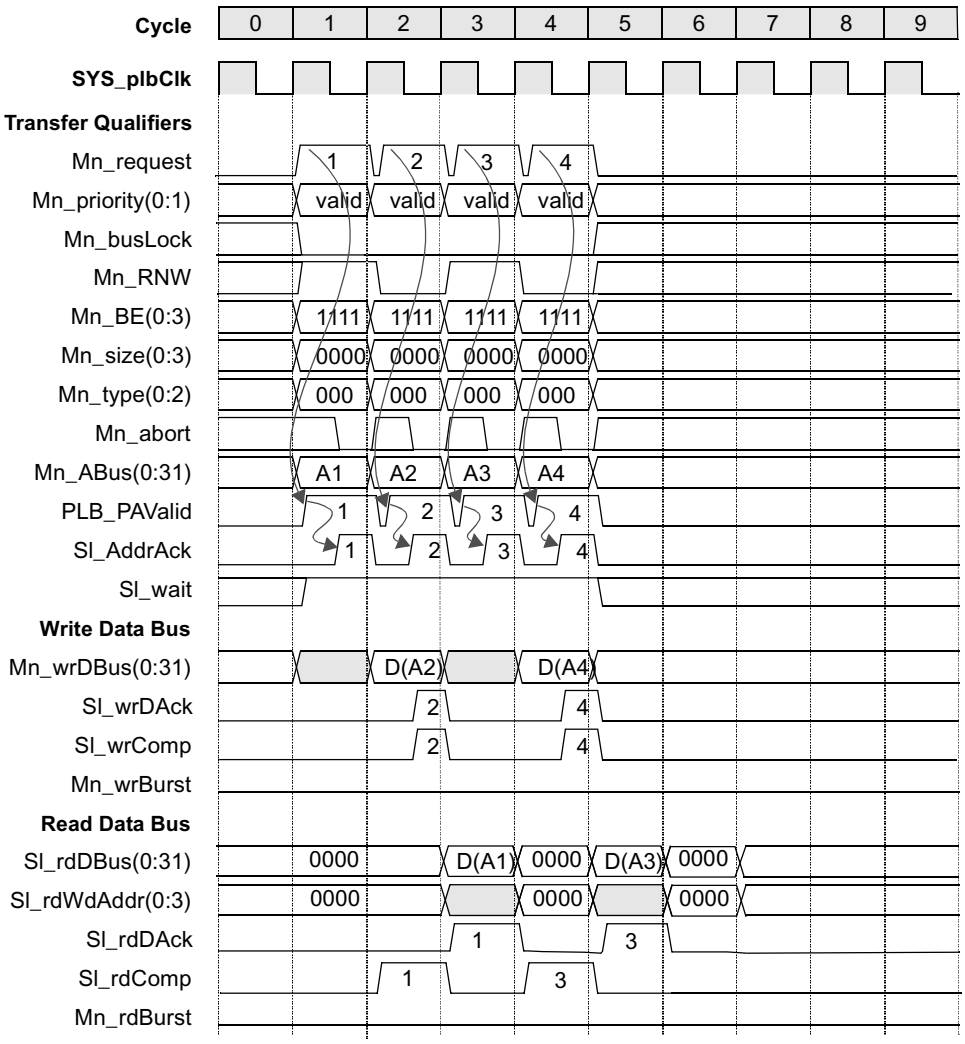


Figure 13. Back-to-Back Read - Write - Read - Write

Set_device (path=/plb_complex/plb_master0,device_type=plb_master)

mem_init(addr=00000000,data=00010203)

mem_init(addr=00000004,data=04050607)

mem_init(addr=00000008,data=08090a0b)

mem_init(addr=0000000c,data=0c0d0e0f)

read(addr=00000000,size=0000,be=1111)

write(addr=00000004,size=0000,be=1111)

read(addr=00000008,size=0000,be=1111)

```
    write(addr=0000000c,size=0000,be=1111)
    send(level=0)
Set_device (path=/plb_complex/plb_slave0,device_type=plb_slave)    -- Initialize device 0
    mem_init(addr=00000000,data=00010203)
    mem_init(addr=00000004,data=04050607)
    mem_init(addr=00000008,data=08090a0b)
    mem_init(addr=0000000c,data=0c0d0e0f)
    read_response(aack_delay=0,dack_delay=0)
    write_response(aack_delay=0,dack_delay=0)
    read_response(aack_delay=0,dack_delay=0)
    write_response(aack_delay=0,dack_delay=0)
    mem_check(level=0,addr=00000000,data=00010203)
    mem_check(level=0,addr=00000004,data=04050607)
    mem_check(level=0,addr=00000008,data=08090a0b)
    mem_check(level=0,addr=0000000c,data=0c0d0e0f)
```

7.7 Four-word Line Read Transfers

Figure 14 shows the operation of a single four word line read from a slave device which is capable of providing data in a single clock cycle.

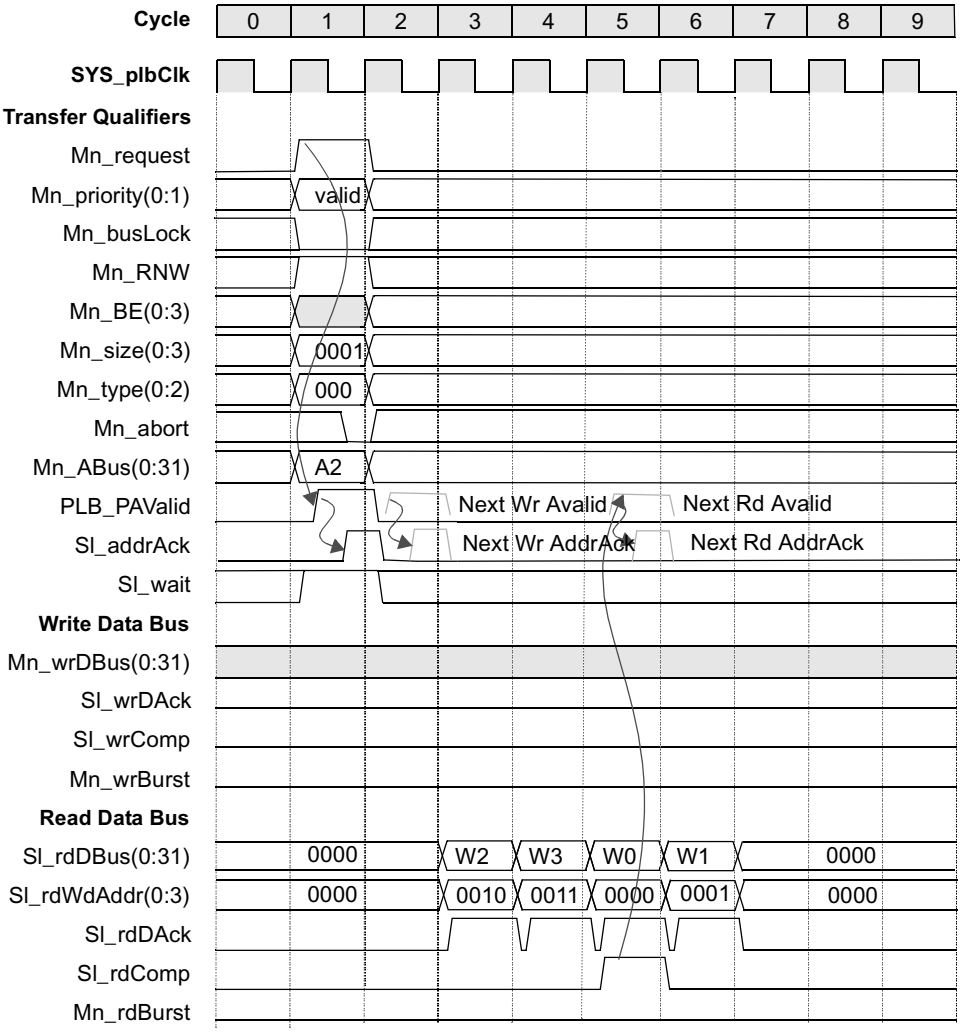


Figure 14. Four Word Line Read

Set_device (path=/plb_complex/plb_master0,device_type=plb_master)

mem_init(addr=00000000,data=00010203)

mem_init(addr=00000004,data=04050607)

mem_init(addr=00000008,data=08090a0b)

mem_init(addr=0000000c,data=0c0d0e0f)

read(addr=00000008,size=0001,be=1111)

send(level=0)

Set_device (path=/plb_complex/plb_slave0,device_type=plb_slave)Initialize device 0

```
mem_init(addr=00000000,data=00010203)
mem_init(addr=00000004,data=04050607)
mem_init(addr=00000008,data=08090a0b)
mem_init(addr=0000000c,data=0c0d0e0f)
read_response(aack_delay=0,dack_delay=0,rdword_mode=1)
mem_check(level=0,addr=00000000,data=00010203)
mem_check(level=0,addr=00000004,data=04050607)
mem_check(level=0,addr=00000008,data=08090a0b)
mem_check(level=0,addr=0000000c,data=0c0d0e0f)
```

7.8 Four-word Line Write Transfers

Figure 15 shows the operation of a single four-word line write to a slave device which is capable of latching data every clock cycle from the PLB.

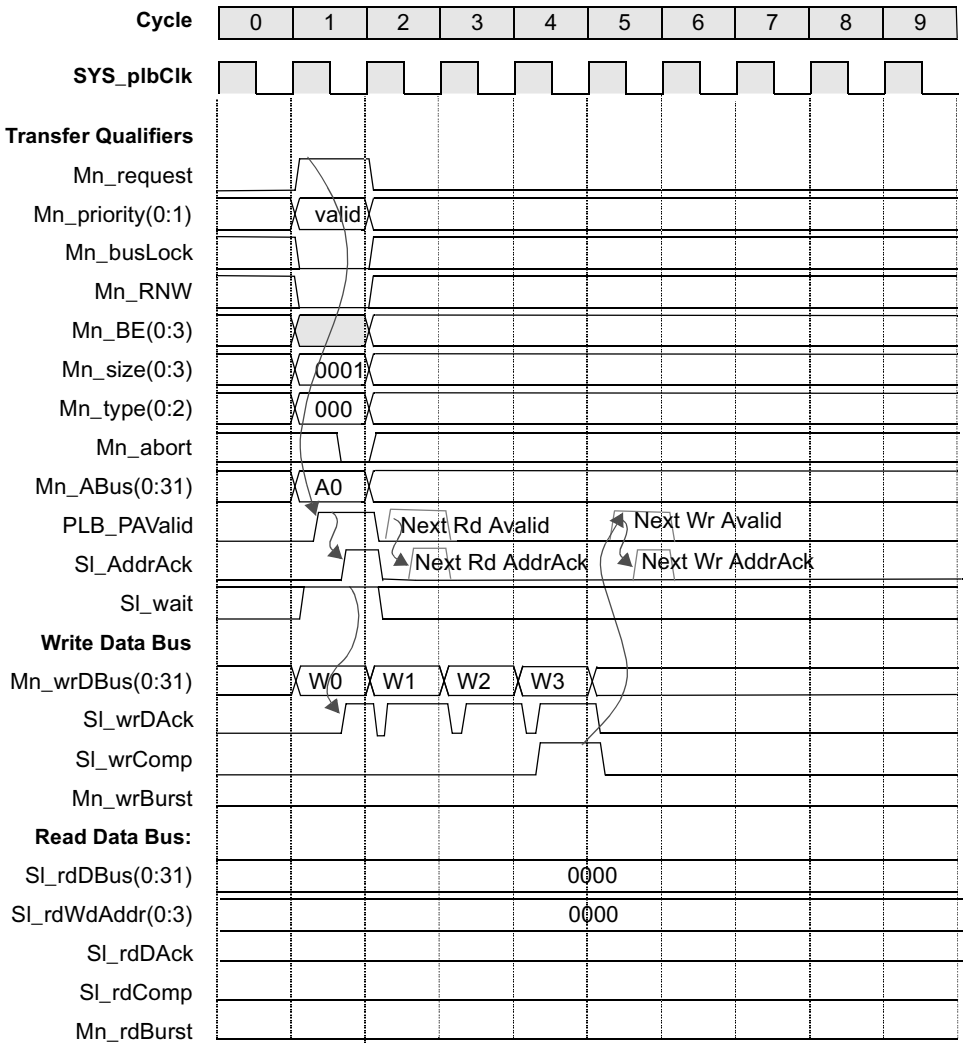


Figure 15. Four Word Line Write

```
set_device (path=/plb_complex/plb_master0,device_type=plb_master)
   mem_init(addr=00000000,data=00010203)
   mem_init(addr=00000004,data=04050607)
   mem_init(addr=00000008,data=08090a0b)
   mem_init(addr=0000000c,data=0c0d0e0f)
   write(addr=00000000,size=0001,be=1111)
   send(level=0)
```

```
set_device (path=/plb_complex/plb_slave0,device_type=plb_slave)    -- Initialize device 0
   mem_init(addr=00000000,data=00010203)
   mem_init(addr=00000004,data=04050607)
   mem_init(addr=00000008,data=08090a0b)
   mem_init(addr=0000000c,data=0c0d0e0f)
   write_response(aack_delay=0,dack_delay=0)
   mem_check(level=0,addr=00000000,data=00010203)
   mem_check(level=0,addr=00000004,data=04050607)
   mem_check(level=0,addr=00000008,data=08090a0b)
   mem_check(level=0,addr=0000000c,data=0c0d0e0f)
```

7.9 Four-word Line Read Followed By Four-word Line Write Transfers

Figure 16 shows the operation of a four-word line read followed immediately by a four-word line write on the PLB.

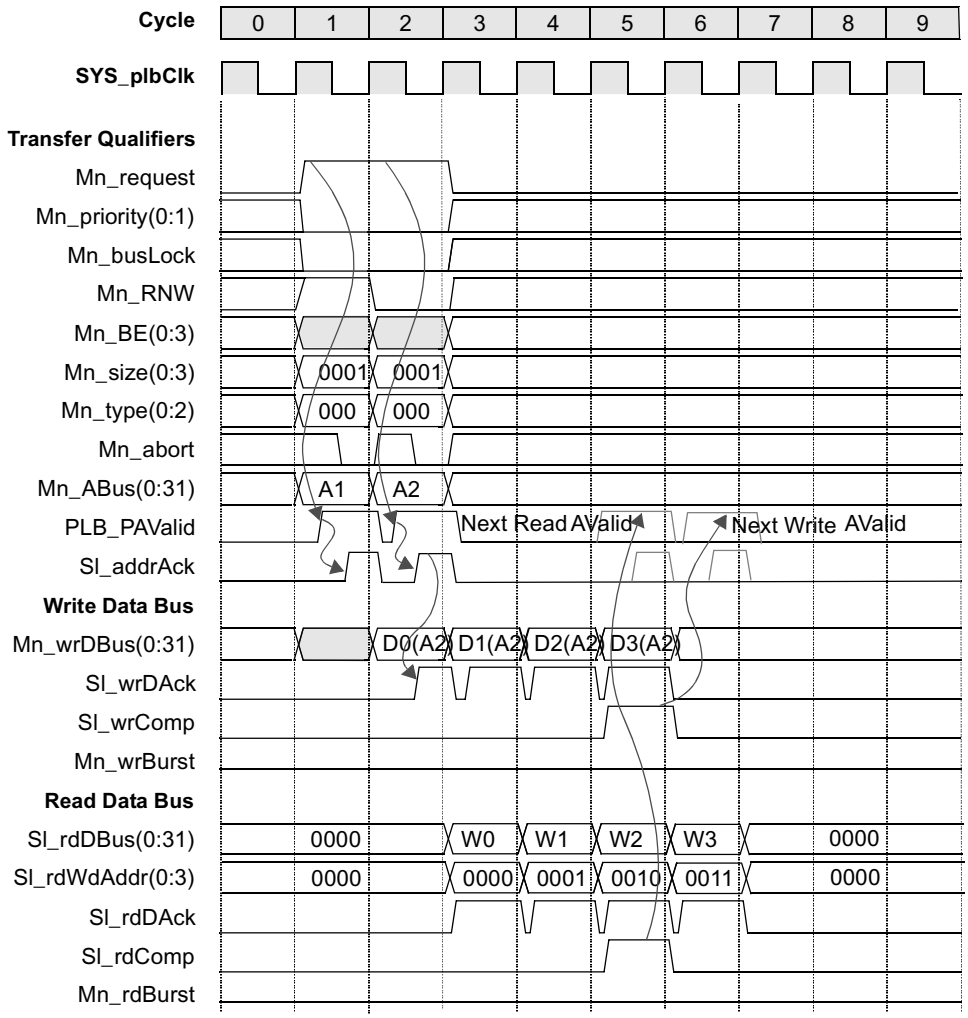


Figure 16. Four Word Line Read followed by Four Word Line Write

set_device (path=/plb_complex/plb_master0,device_type=plb_master)

mem_init(addr=00000000,data=00010203)

mem_init(addr=00000004,data=04050607)

mem_init(addr=00000008,data=08090a0b)

mem_init(addr=0000000c,data=0c0d0e0f)

mem_init(addr=00000010,data=10111213)

mem_init(addr=00000014,data=14151617)

mem_init(addr=00000018,data=18191a1b)

```
    mem_init(addr=0000001c,data=1c1d1e1f)
    read(addr=00000000,size=0001,be=1111)
    write(addr=00000010,size=0001,be=1111)
    send(level=0)
set_device (path=/plb_complex/plb_slave0,device_type=plb_slave)    -- Initialize device 0
    mem_init(addr=00000000,data=00010203)
    mem_init(addr=00000004,data=04050607)
    mem_init(addr=00000008,data=08090a0b)
    mem_init(addr=0000000c,data=0c0d0e0f)
    mem_init(addr=00000010,data=10111213)
    mem_init(addr=00000014,data=14151617)
    mem_init(addr=00000018,data=18191a1b)
    mem_init(addr=0000001c,data=1c1d1e1f)
    write_response(aack_delay=0,dack_delay=0)
    read_response(aack_delay=0,dack_delay=0)
    mem_check(level=0,addr=00000000,data=00010203)
    mem_check(level=0,addr=00000004,data=04050607)
    mem_check(level=0,addr=00000008,data=08090a0b)
    mem_check(level=0,addr=0000000c,data=0c0d0e0f)
    mem_check(level=0,addr=00000010,data=10111213)
    mem_check(level=0,addr=00000014,data=14151617)
    mem_check(level=0,addr=00000018,data=18191a1b)
    mem_check(level=0,addr=0000001c,data=1c1d1e1f)
```

Chapter 8. PLB Compliance Checks

8.1 Monitor Model

The PLB monitor model attaches to the PLB bus and generates informational architectural compliance messages during simulation. The model is passive in that it does not participate in any bus protocol signal assertions, but it samples PLB signals in every clock to keep track of the state of the PLB bus and inform the user of any detectable PLB compliance problems. Messages are generated to the simulator console, which also may be re-directed to file systems in most simulation environments.

8.2 Terminology

The following terminology is used in the specification of the PLB monitor compliance checks:

- Active: refers to the situation when a signal is at its true state (a logical 1).
- Asserted: refers to the situation when a signal transitions from inactive to its active state.
- Inactive: refers to the situation when a signal is at its false state. (a logical 0)
- Deasserted: refers to the situation when a signal transitions from active to its inactive state.
- Time-out: refers to the event which occurs when there have been 16 cycles of non-response from a slave (or Mn_abort) since the assertion of PLB_PAValid. The 16 cycle count may be interrupted by the slave response of Sln_wait active. And the count will remain interrupted for as long as the wait is active. The count will be terminated by slave asserting Sln_rearbitrate or Sn_addrAck. The count may also be terminated by the master asserting Mn_abort (given no Sn_addrAck has been received). If the count reaches 16 and none of the terminating responses (from master or slave) have been asserted then a time-out has occurred.

8.3 Address Map Set-up Error

An error message is issued when PLB_reset is de-asserted and the PLB monitor slave address map has not been initialized.

8.4 Slave Interface/PLB Core OR Logic Error

An error message is issued when the OR'ed value of all the PLB slave control outputs does not equal the output of the system OR gates which are inputs to the PLB arbiter.

8.5 Signal Summary Table

Table 1 provides a summary of all PLB input/output signals in alphabetical order which includes the source, a brief description and page reference for bus functional compliance checks. See PLB architecture specifications for detailed signal description.

Table 1. Summary of PLB Signals

Signal Name	Source	Description	Page
Mn_abort	Master n	Master n abort bus request indicator	76
Mn_ABus(0:31)	Master n	Master n address bus	76
Mn_BE(0:3)	Master n	Master n byte enables	73
Mn_busLock[1]	Master n	Master n bus lock	72
Mn_TAttribute	Master n	Master n storage attributes	75
Mn_guarded	Master n	Master n guarded transfer indicator	75
Mn_lockErr	Master n	Master n lock error indicator	75
Mn_priority(0:1)	Master n	Master n bus request priority	72
Mn_rdBurst	Master n	Master n burst read transfer indicator	77
Mn_request	Master n	Master n bus request	72
Mn_RNW	Master n	Master n read/write	73
Mn_size(0:3)	Master n	Master n transfer size	74
Mn_type(0:2)	Master n	Master n transfer type	75
Mn_wrBurst	Master n	Master n burst write transfer indicator	77
Mn_wrDBus(0:31)	Master n	Master n write data bus	76
PLB_abort	Arbiter	PLB abort bus request indicator	91
PLB_ABus(0:31)	Arbiter	PLB address bus	89
PLB_BE(0:3)	Arbiter	PLB byte enables	87
PLB_busLock	Arbiter	PLB bus lock	89
PLB_guarded	Arbiter	PLB guarded transfer indicator	88
PLB_lockErr	Arbiter	PLB lock error indicator	88
PLB_masterID(0:1)	Arbiter	PLB current master identifier	82
PLB_Mn_Busy	Master n	PLB master n slave busy indicator	82
PLB_Mn_Err	Master n	PLB master n slave error indicator	92
PLB_Mn_WrBTerm	Master n	PLB master n terminate write burst indicator	91
PLB_Mn_WrDAck	Master n	PLB master n write data acknowledge	78
PLB_MnAddrAck	Master n	PLB master n address acknowledge	78

Table 1. Summary of PLB Signals (Continued)

Signal Name	Source	Description	Page
PLB_MnRdBTerm	Master n	PLB master n terminate read burst indicator	92
PLB_MnRdDAck	Master n	PLB master n read data acknowledge	79
PLB_MnRdDBus(0:31)	Master n	PLB master n read data bus	79
PLB_MnRdWdAddr(0:3)	Master n	PLB master n read word address	79
PLB_MnRearbitrate	Master n	PLB master n bus rearbitrate indicator	78
PLB_PAValid	Arbiter	PLB primary address valid indicator	80
PLB_pendPri(0:1)	Arbiter	PLB pending request priority	83
PLB_pendReq	Arbiter	PLB pending bus request indicator	82
PLB_rdBurst	Arbiter	PLB burst read transfer indicator	90
PLB_rdPrim	Arbiter	PLB secondary to primary read request indicator	90
PLB_reqPri(0:1)	Arbiter	PLB current request priority	83
PLB_RNW	Arbiter	PLB read not write	87
PLB_SAValid	Arbiter	PLB secondary address valid indicator	80
PLB_size(0:3)	Arbiter	PLB transfer size	87
PLB_type(0:2)	Arbiter	PLB transfer type	88
PLB_wrBurst	Arbiter	PLB burst write transfer indicator	90
PLB_wrDBus(0:31)	Arbiter	PLB write data bus	83
PLB_wrPrim	Arbiter	PLB secondary to primary write request indicator	91
Sl_addrAck	Slave	Slave address acknowledge	81
Sl_MBusy(0:3)	Slave	Slave busy indicator	86
Sl_MErr(0:3)	Slave	Slave error indicator	86
Sl_rdBTerm	Slave	Slave terminate read burst transfer	86
Sl_rdComp	Slave	Slave read transfer complete indicator	85
Sl_rdDAck	Slave	Slave read data acknowledge	85
Sl_rdDBus(0:31)	Slave	Slave read data bus	84
Sl_rdWdAddr(0:3)	Slave	Slave read word address	85
Sl_rearbitrate	Slave	Slave rearbitrate bus indicator	82
Sl_wait	Slave	Slave wait indicator	82
Sl_wrBTerm	Slave	Slave terminate write burst transfer	84
Sl_wrComp	Slave	Slave write transfer complete indicator	84
Sl_wrDAck	Slave	Slave write data acknowledge	83

Table 1. Summary of PLB Signals (Continued)

Signal Name	Source	Description	Page
Synch_in			
Synch_out			

8.6 Master Interface Checks

This section describes the PLB master interface checks performed during simulation.

8.6.1 Mn_request

The following error messages are issued for this signal:

- Error 1.1.1

  An error message is issued when a master deasserts its Mn_request without any of the following events: (that is, if any one of the following events occur then no error is flagged)

  PLB_MnAddrAck is asserted

  Mn_abort is asserted

  PLB_MnRearbitrate is asserted

  MnTimeout is asserted

- Error 1.1.2

  An error message is issued when Mn_request is active and SYS_plbReset is active.

- Warning1.1.3

  A warning message of possible deadlock is issued when Mn_request is not inactive for 2 clock cycles after PLB_MnRearbitrate, Mn_request and Mn_busLock are sampled active with PLB_Mnaddrack inactive.

- Error 1.1.4

  Requesting master which has the bus locked did not deassert Mn_Request for two clocks after a rearbitrated request.

8.6.2 Mn_priority(0:1)

No protocol checks.

8.6.3 Mn_busLock

The following error messages are issued for this signal:

- Error 1.3.1

  An error message is issued when a master's Mn_busLock is active without its Mn_request being active unless the bus is already locked by the requesting master.

- Error 1.3.2

A warning message of possible deadlock is issued when Master n has the bus locked and Mn_busLock is not inactive for 2 clock cycles after PLB_MnRearbitrate, Mn_Request and Mn_busLock are sampled active with PLB_MnAddrAck inactive.

- Error 1.3.3

  An error message is issued when Mn_busLock is active and SYS_plbReset is active.

- Warning1.3.4

  A warning message is issued when Mn_busLock is de-asserted during a master request phase before PLB_MnaddrAck, Mn_abort, MnTimeout or PLB_MnRearbitrate is asserted.

- Error 1.3.5

  Requesting master did not deassert Mn_busLock for two cycles after a rearbitrated request.

- ERROR 1.3.6

  An error message is issued when Mn_busLock is asserted for a requesting master after a new request was started and before PLB_MnAddrAck, Mn_abort, PLB_MnRearbitrate, or MnTimeout.

8.6.4 Mn_RNW

The following error messages are issued for this signal:

- Error 1.4.1

  An error message is issued when a master changes this transfer qualifier from when a request to transfer is made until PLB_MnAddrAck, Mn_abort, PLB_MnRearbitrate, or Mn_Timeout is asserted.

8.6.5 Mn_BE(0:3)

The following error messages are issued for this signal:

- Error 1.5.1

  An error message is issued when a master changes this transfer qualifier from when a request to transfer is made until PLB_MnAddrAck, Mn_abort, PLB_MnRearbitrate, or Mn_Timeout is asserted.

- Error 1.5.2

  An error message is issued if there are no enabled bytes on Mn_BE when Mn_request is active and Mn_size indicate a non-line, non-burst transfer.

- Error 1.5.3

  An error message is issued if Mn_BE contains non-contiguous enabled bytes when Mn_request is active and Mn_size indicate a non-line, non-burst transfer memory transfer.

- Error 1.5.4

  An error message is issued if the Mn_BE bit corresponding to the LSB bits of Mn_ABus encode is not active for a non-line, non-burst transfer. For example: On a non-line, non-burst transfer request with Mn_ABus(30:31)=”11”, Mn_BE(0:3) must be “0001”.

- Error 1.5.5

An error message is issued if the Mn_BE signal is not properly “replicated” as indicated in the architecture specification for smaller masters connected to larger width PLB Arbiters.

8.6.6 Mn_size(0:3)

The following error messages are issued for this signal:

- Error 1.6.1

  An error message is issued when a master changes this transfer qualifier from when a request to transfer is made until the addressed slave asserts its SIn_addrAck, or the transfer is terminated via Mn_abort, PLB_MnRearbitrate, or Mn_Timeout.

- Error 1.6.2

  When Mn_request is active an error message is issued for the following combination of Mn_type values (as shown in Table 2) and Mn_size values: (as shown in Table 3)

Table 2. Mn_type Values

Mn_type	Mn_size
000	Any of the reserved size codes
001	Any of the reserved size codes or any line size code
010	Codes other than 0000
011	Codes other than 0000
100	Codes other than 0000
101	Codes other than 0000
110	Any of the reserved size codes

Table 3. Mn_size Values

Mn_size(0:3)	Definition
0000	Transfer one to four bytes of a word starting at the target address.
0001	Transfer the 4-word line containing the target word
0010	Transfer the 8-word line containing the target word.
0011	Transfer the 16-word line containing the target word
0100	Reserved
0101	Reserved
0110	Reserved
0111	Reserved
1000	Burst Transfer - Bytes - Length determined by master
1001	Burst Transfer - Halfwords - Length determined by master

Table 3. Mn_size Values (Continued)

Mn_size(0:3)	Definition
1010	Burst Transfer - Words - Length determined by master
1011	Burst Transfer - Double words - Length determined by master
1100	Burst Transfer - Quad words - Length determined by master
1101	Burst Transfer - Octal words - Length determined by master
1110	Reserved
1111	Reserved

- Error 1.6.3

  An error message is issued

  – When master has a double word burst size and Mn_MSize indicates a 32-bit master.

  – When master has a quad word burst size and Mn_MSize indicates a 32-bit master.

  – When master has a quad word burst size and Mn_MSize indicates a 64-bit master.

  – When master has a octal word burst size and Mn_MSize indicates a 32-bit master.

  – Master has a octal word burst size and Mn_MSize indicates a 64-bit master.

  – Master has a octal word burst size and Mn_MSize indicates a 128-bit master.

8.6.7 Mn_type(0:2)

The following error messages are issued for this signal:

- Error 1.7.1

  An error message is issued when a master changes this transfer qualifier from when a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated via Mn_abort, PLB_MnRearbitrate, or Mn_Timeout.

8.6.8 Mn_TAttribute

The following error messages are issued for this signal:

- Error 1.10.1

  An error message is issued when a master changes this transfer qualifier from when a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated via Mn_abort, PLB_MnRearbitrate, or Mn_Timeout.

8.6.9 Mn_lockErr

The following error messages are issued for this signal:

- Error 1.11.1

An error message is issued when a master changes this transfer qualifier from when a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated via Mn_abort, PLB_MnRearbitrate, or Mn_Timeout.

8.6.10 Mn_ABus/Mn_UABus

The following error messages are issued for this signal:

- Error 1.12.1

  An error message is issued when a master changes these transfer qualifiers from when a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated via Mn_abort, PLB_MnRearbitrate, or Mn_Timeout.

- Error 1.12.2

  An error message is issued when the Mn_Abus word address is non-zero during a line write transfer.

- Error 1.12.3

  An error message is issued:

  – when the master has a non-aligned half word burst address for half word burst transfers.

  – when the master has a non-aligned word burst address for word burst transfers.

  – when the master has a non-aligned double word burst address for double word burst transfers.

  – when the master has a non-aligned quad word burst address for quad word burst transfers.

- Note1.12.4

  An note message is issued when the Mn_Abus word address is non-zero during a line write transfer.

8.6.11 Mn_abort

The following error messages are issued for this signal:

- Error 1.13.1

  A warning message is issued if Mn_abort is asserted without a Mn_Request.

8.6.12 Mn_wrDBus(0: PLB DATA BUS WIDTH)

The following error messages are issued for this signal:

- Error 1.14.1

  An error message is issued when a master changes this transfer qualifier from when a write request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated via Mn_abort, PLB_MnRearbitrate, or Mn_Timeout.

- Error 1.14.2

An error message is issued when the first word of data on Mn_wrDBus does not remain constant from the assertion of PLB_MnaddrAck to the first PLB_MnWrDAck (in the case when wrDAck is delayed). This check is also performed on words following the first of multi-word transfers in which case the word of data on Mn_wrDBus must remain constant from PLB_MnWrDAck through the next PLB_MnWrDAck.

- Error 1.14.3

  An warning message is issued when a master changes this transfer qualifier from when a **read** request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated via Mn_abort or PLB_MnRearbitrate.

- Error 1.14.4

  An error message is issued if the Mn_wrDbus is not properly “mirrored” as indicated in the architecture specification for smaller masters connected to larger width PLB Arbiters.

8.6.13 Mn_rdBurst

The following error messages are issued for this signal:

- Error 1.15.1

  Mn_RdBurst and PLB_RESET are active at the same time.

- ERROR 1.15.2

  Master asserted Mn_rdBurst before AddrAck was received.

- Error 1.15.3

  Master failed to deassert its Mn_rdBurst after receiving PLB_MnRdBTerm.

- Error 1.15.4

  Master reasserted Mn_rdBurst before the current burst transfer received it last DAck.

- Error 1.15.5

  An error message is issued if a master terminates a read burst transfer (by deasserting its Mn_rdBurst) and then reasserts its Mn_rdBurst signal before the last rdDAck for a multi-burst read transfer is terminated.

8.6.14 Mn_wrBurst

The following error messages are issued for this signal:

- Error 1.16.1

  An error message is issued when a master terminates a write burst transfer (by deasserting its Mn_wrBurst) and re-asserts its Mn_wrBurst before receiving the last wrDAck.

- Error 1.16.2

  An error message is issued if Mn_WrBurst is active in the cycle after PLB_MWrBTerm is asserted.

- Error 1.16.3

  An error message is issued if Mn_WrBurst is asserted without a valid burst request as indicated by Mn_size.

- Error 1.16.4

  An error message is issued when Mn_WrBurst is active and SYS_plbReset is active.

8.6.15 PLB_MnAddrAck

The following error messages are issued for this signal:

- Error 1.17.1

  An error message is issued when this signal is active and the corresponding Mn_request is inactive.

- Error 1.17.2

  An error message is issued when this signal is active and the PLB_masterID identifies a different master as granted.

- Error 1.17.3

  An error message is issued if arbiter does not assert this signal when a time-out occurs. (This check is not performed for PLB 4.x)

- Error 1.17.4

  An error message is issued if PLB asserts PLB_MnAddrAck when a different master has locked the PLB.

- Error 1.17.5

  An error message is issued if PLB asserts more than one PLB_MnAddrAck in the same clock.

8.6.16 PLB_MnRearbitrate

The following error messages are issued for this signal:

- Error 1.18.1

  An error message is issued when this signal is active and the PLB_masterID identifies a different master as granted.

- Error 1.18.2

  An error message is issued when this signal is active and the corresponding Mn_request is inactive.

- Error 1.18.3

  An error message is issued when this signal is asserted by PLB during a secondary request. (This check is not performed for PLB 4.x)

8.6.17 PLB_MnWrDAck

The following error messages are issued for this signal:

- Error 1.19.1

  An error message is issued if PLB_MnWrDAck is asserted outside of window from assertion of PLB_MnAddrAck to assertion of Sln_wrComp.

8.6.18 PLB_MnRdDAck

The following error messages are issued for this signal:

- Error 1.20.1

  An error message is issued if PLB_MnRdDAck is asserted outside of the window defined from 2 clocks after the assertion of PLB_MnAddrAck to the clock after Sln_rdComp.

- Error 1.20.2

  An error message is issued if PLB_MnRdDack is asserted during a DMA transfer which requires no rdDAcks.

8.6.19 PLB_MnRdWdAddr(0:3)

The following error messages are issued for this signal:

- Error 1.21.1

  An error message is issued when master does not own the read data bus and PLB_MnRdWdAddr(0:3) contains non-zero data.

- Error 1.21.2

  An error message is issued when PLB_MnRdWdAddr(0:3) does not match Sl_rdWdAddr(0:3).

- Error 1.21.3

  An error message is issued when PLB_MnRdWdAddr(0:3) does not cover all the necessary read word addresses in a line transfer.

8.6.20 PLB_MnBusy

The following error messages are issued for this signal:

- Error 1.22.1

  An error message is issued when PLB_MnBusy is not active during a read transaction with slave.

- Error 1.22.2

  An error message is issued when PLB_MnBusy is not active during a write transaction with slave.

- Warning 1.22.3

  A warning message is issued when PLB_MnBusy is active for more than busy_cycle_max cycles.

8.6.21 PLB_MRdErr/PLB_MWrErr

- Error 1.23.1

  An error message is issued when these signals are asserted without a PLB_Mrd/wrDack or without a busy signal.

8.6.22 PLB_MSSize

- Error 1.24.1

An error message is issued when PLB Arbiter core did not propagate Sl_Ssize to master when Sl_addrAck was asserted.

8.6.23 PLB_MnTimeout

The following error messages are issued for this signal:

- Error 1.25.1

  An error message is issued when this signal is active and the PLB_masterID identifies a different master as granted.

- Error 1.25.2

  An error message is issued when this signal is active and the corresponding Mn_request is inactive.

- Error 1.25.3

  An error message is issued when this signal is asserted by PLB during a secondary request.

- Error 1.25.4

  An error message is issued when this signal is asserted in the same clock as PLB_MnAddrAck.

- Error 1.25.5

  An error message is generated if the PLB Arbiter asserts address acknowledge during a secondary request.

- Error 1.25.6

  If “enable_timeout_report” is true then an error message will be issued whenever a timeout is received. This error can be turned on through monitor bfl, but it defaults to being turned off.

8.7 Slave Interface Checks

This section describes the PLB slave interface checks performed during simulation.

8.7.1 PLB_PAValid

The following error messages are issued for this signal:

- Error 2.1.1

  An error message will be issued if PLB_PAValid is deasserted without a valid termination of the request by either Sln_addrAck, Sln_rearbitrate, Mn_abort or PLB_MnTimeout.

- Error 2.1.2

  An error message is issued if PLB_PAValid is active and PLB_rd/wrpendReq is inactive.

- Error 2.1.3

  An error message is issued when PLB_PAValid is active and SYS_plbReset is active.

8.7.2 PLB_SAValid

The following error messages are issued for this signal:

- Error 2.2.1

  An error message will be issued if PLB_SAValid is deasserted for a secondary **read** cycle without a valid termination of the request by either Sln_addrAck or Mn_abort, Sl_rdComp, or Sl_rearbitrate.

- Error 2.2.2

  An error message will be issued if PLB_SAValid is deasserted for a secondary **write** cycle without a valid termination of the request by either Sln_addrAck or Mn_abort, Sl_wrComp or Sl_rearbitrate.

- Error 2.2.3

  An error message will be issued when a secondary request is not converted into a primary request. This occurs when PLB_SAValid is deasserted because of a Sln_rd/wrComp assertion (without overlapped Sl_addrAck/PLB_abort) and PLB_PAValid is not active in the next clock.

- Error 2.2.4

  An error message is issued if PLB_SAValid and PLB_PAValid are active in the same clock.

- Error 2.2.5

  An error message is issued if PLB_SAValid is active and PLB_rd/wrpendReq is inactive.

- Error 2.2.6

  An error message is issued when PLB_SAValid is active and SYS_plbReset is active.

8.7.3 Sln_addrAck

The following error messages are issued for this signal:

- Error 2.3.1

  An error message is issued when Sln_addrAck is asserted and PLB_PAValid or PLB_SAValid is inactive.

- Error 2.3.2

  An error message is issued when Sln_addrAck is active while the slave is not selected. This check depends on user defined address map.

- Error 2.3.3

  An error message is issued when more than one slave asserts Sln_addrAck in the same clock.

- Warning 2.3.4

  A warning message is issued if Sln_addrAck is asserted at the first assertion of PLB_PAValid or PLB_SAValid for the requested transaction. Sln_addrAck should be asserted at least one cycle after PLB_PLB_PAValid or PLB_SAValid or significant frequency degradation of the stated operational frequency is a possible outcome. The slave in the toolkit is programmable and by default will assert Sln_addrAck with zero delay, or with the assertion of PLB_PAValid or PLB_SAValid. To program the toolkit slave to assert its Sln_addrAck one cycle after PLB_PAValid or PLB_SAValid, put the following in the configuration statement for each slave: configure(aack_delay=1). This will delay the Sln_addAck signal until one cycle after PLB_PAValid or PLB_SAValid.

8.7.4 Sln_wait

The following error messages are issued for this signal:

- Error 2.4.1

  An error message is issued when Sln_wait is active while the slave is not selected. This check depends on user defined address map.

- Error 2.4.2

  An error message is issued when more than one slave asserts Sln_wait in the same clock.

8.7.5 Sln_rearbitrate

The following error messages are issued for this signal:

- Error 2.5.1

  An error message is issued when Sln_rearbitrate is asserted and both PLB_PAValid and PLB_SAValid are inactive.

- Error 2.5.2

  An error message is issued when Sln_rearbitrate is active while the slave is not selected. This check depends on user defined address map.

- Error 2.5.3

  A warning message is issued when Sln_rearbitrate is asserted and PLB_SAValid is active. (Not applicable to PLB 4.x)

- Error 2.5.4

  An error message is issued when more than one slave asserts Sln_rearbitrate in the same clock.

8.7.6 PLB_rd/wrpendReq

The following error messages are issued for this signal:

- Error 2.6.1/3

  An error message is issued when the PLB changes this transfer qualifier from the time when a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated through Mn_abort, PLB_MnRearbitrate, or PLB_MnTimeout.

- Error 2.6.2/4

  An error message is issued if this signal is active when there is no Mn_request.

8.7.7 PLB_MasterID(0:3)

The following error messages are issued for this signal:

- Error 2.7.1

  An error message is issued when the PLB changes this transfer qualifier from the time when a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated through Mn_abort, PLB_MnRearbitrate or PLB_MnTimeout.

- Error 2.7.2

  An error message is issued when the PLB is locked (Mn_busLock, Mn_Request, and PLB_MnaddrAck were sampled active) and PLB_masterID does not equal the locked master's id.

- Error 2.7.3

  An error message is issued if in the cycle following a rearbitrate with PLB_busLock inactive, PLB_PAValid is active and the current PLB_masterID is the same as in the previous cycle when rearbitrate was asserted.

8.7.8 PLB_rd/wrpendPri(0:1)

There are currently no protocol checks for this signal.

8.7.9 PLB_reqPri(0:1)

The following error messages are issued for this signal:

- Error 2.9.2

  An error message is issued when this bus does not equal the Mn_priority winner of primary and secondary requests when PLB_MnAddrAck is asserted.

8.7.10 PLB_wrDBus

The following error messages are issued for this signal:

- 2.10.1

  An error message is issued when the PLB_wrDBus does not remain constant from the slaves assertion of Sln_addrAck to Sln_wrDAck, and also between Sln_wrDAck assertions in multi-data transfers.

- 2.10.2

  An error message is issued when the PLB changes the write data bus from the time when a primary request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated through Mn_abort, PLB_MnRearbitrate or PLB_MnTimeout.

8.7.11 Sln_wrDAck

The following error messages are issued for this signal:

- Error 2.11.1

  An error message is issued when a slave asserts its Sln_wrDAck before asserting its Sln_addrAck or after asserting its Sln_wrComp.

- Error 2.11.2

  An error message is issued when on a non-burst write transfer the slave fails to give expected number of wrDAcks.

- Error 2.11.3

  An error message is issued when this signal is asserted during a DMA transfer type1,4,5,7 from the assertion of Sln_addrack to the assertion of Sln_wrComp.

- Error 2.11.4

  An error message is issued when a slave asserts is Sln_wrDAck after it has already been asserted for the last write data transfer of a burst write.

- Error 2.11.5

  An error message is issued if more than one Sl_wrDAck is asserted during the same clock cycle.

- Error 2.11.6

  An error message is asserted when this signal is asserted for a transfer that is outside of the slave's assigned address space.

- Error 2.11.7

  An error message is asserted when wrcomp is asserted for a burst transfer before the last wrdack of the burst is received.

8.7.12 Sln_wrComp

The following error messages are issued for this signal:

- Error 2.12.1

  An error message is issued when Sln_wrComp is asserted and PLB_wrBurst is active during the data phase of a burst write transfer.

- Error 2.12.2

  An error message is issued when Sln_wrComp is active outside of allowed window i.e. from the assertion of Sln_addrAck to the assertion of Sln_wrComp.

- Error 2.12.3

  An error message is issued when more than one slave asserts Sln_wrComp in the same clock.

8.7.13 Sln_wrBTerm

The following error messages are issued for this signal:

- Error 2.13.1

  An error message is issued when Sln_wrBTerm is active outside of allowed window from the assertion of Sln_addrAck to the assertion of Sln_wrComp.

- Warning 2.13.2

  A warning message is issued when Sln_wrBTerm is not asserted during a guarded burst transfer that crosses a 1k address boundary.

- Note 2.13.3

  A note message is issued if this signal is asserted pre-maturely during a fixed length burst transfer. This means the wrbterm is asserted when there is more than one dack required to meet the number of expected dacks.

8.7.14 Sln_rdDBus

The following error messages are issued for this signal:

- Error 2.14.1

  An error message is issued when a slave drives data other than zeroes on its Sln_rdDBus(0:31) when not selected for a read transfer.

8.7.15 Sln_rdWdAddr(0:3)

The following error messages are issued for this signal:

- Error 2.15.1

  An error message is issued when a slave drives data other than zeroes on its Sln_rdWdAddr(0:3) when not selected for a read transfer.

- Error 2.15.2

  An error message is issued when Sln_rdWdAddr(0:3) is not aligned for line transfer

8.7.16 Sln_rdDAck

The following error messages are issued for this signal:

- Error 2.16.1

  An error message is issued when Sln_rdDAck is asserted outside of the window from two clocks after the assertion of Sln_addrAck to one clock after the assertion of Sln_rdComp.

- Error 2.16.2

  An error message is issued when on a non-burst read transfer the slave fails to give the expected number of rdDAcks.

- Error 2.16.3

  An error message is issued when this signal is asserted during a DMA type=1,4,5,7 cycle from two clocks after Sln_addrack until one clock after Sln_rdComp is asserted.

- Error 2.16.4

  An error message is issued when this signal is asserted for a transfer that is outside of the slave's assigned address space.

- Error 2.16.5

  An error message is issued when this signal is asserted more than once after the rdburst signal has been deasserted for a read burst transfer.

8.7.17 Sln_rdComp

The following error messages are issued for this signal:

- Error 2.17.1

  An error message is issued when Sln_rdComp is active outside of allowed window from the cycle after slave asserts its Sln_addrAck to the cycle it asserts its Sln_rdComp.

- Error 2.17.4

  An error message is issued when more than one slave asserts Sln_rdComp in the same clock.

- Error 2.17.5

  An error message is issued when this signal is asserted before a variable length burst transaction is complete(last rddack received or rdbterm asserted).

- Error 2.17.6

  An error message is issued when this signal is asserted before the last rddack of the transaction is received.

8.7.18 Sln_rdBTerm

The following error messages are issued for this signal:

- Error 2.18.1

  An error message is issued if Sln_rdBTerm is active outside of allowed window, that is, from the cycle after slave asserts its Sln_addrAck(or arbiter asserts rdprim) to the cycle it asserts its Sln_rdComp.

- Note 2.18.2

  An note message is issued if Sln_rdBTerm is not asserted for a fixed length burst transfer, or the last Sl_rdDAck was not asserted with the Sl_rdBTerm.

- Warning 2.18.3

  A warning message is issued if Sln_rdBTerm is not asserted for a guarded burst transfer which crosses a 1k address boundary.

8.7.19 Sln_MBusy(0:15)

The following errors are issued to this signal.

- Error 2.19.1

  An error message is issued when a slave does not assert and hold active the corresponding bit of its Sln_MBusy bus from the time the slave asserts it Sln_addrAck until the read transfer is complete.

- Error 2.19.2

  An error message is issued when a slave does not assert and hold active the corresponding bit of its Sln_MBusy bus from the time the slave asserts it Sln_addrAck until the write transfer is complete.

8.7.20 Sln_Mrd/wrErr(0:15)

The following error messages are issued for this signal:

- Error 2.20.1

  An error message is issued when a slave asserts its Sl_Mrd/wrErr signal to master n without the corresponding Sl_MBusy, Sl_Mrddack, or Sl_Mwrdack signal asserted.

8.8 PLB Interface Checks

8.8.1 PLB_RNW

The following error messages are issued for this signal:

- Error 2.21.1

  An error message is issued when the PLB changes this transfer qualifier from the time a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated through Mn_abort or PLB_MnRearbitrate or timeout.

- Error 2.21.2

  An error message is issued if this transfer qualifier does not match Mn_RNW of the requesting master when PLB_MnAddrAck is active.

8.8.2 PLB_BE

The following error messages are issued for this signal:

- Error 2.22.1

  An error message is issued when the PLB changes this transfer qualifier from the time a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated via Mn_abort or PLB_MnRearbitrate or timeout.

- Error 2.22.2

  An error message is issued if this transfer qualifier has no enabled bytes(all zeroes) for a non-line, non-burst transfer.

- Error 2.22.3

  An error message is issued if the replication of the byte enables is not correct when a device is connected to a PLB of a larger size.

8.8.3 PLB_size(0:3)

The following error messages are issued for this signal:

- Error 2.23.1

  An error message is issued when the PLB changes this transfer qualifier from the time a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated through Mn_abort or PLB_MnRearbitrate or timeout.

- Error 2.23.2

  An error message is issued if this transfer qualifier does not match Mn_size when PLB_MnAddrAck is active.

- Error 2.23.3

When PLB_PAValid or PLB_SAValid is active an error message is issued for the following combination of PLB_type and PLB_size values as shown in Table 4:

Table 4. PLB_type Values

PLB_type	PLB_size
000	Any of the reserved size codes
001	Any of the reserved size codes or any line size code
010	Codes other than 0000
011	Codes other than 0000
100	Codes other than 0000
101	Codes other than 0000
110	Any of the reserved size codes
111	Codes other than 0000

8.8.4 PLB_type

The following error messages are issued for this signal:

- Error 2.24.1

  An error message is issued when the PLB changes this transfer qualifier from the time a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated through Mn_abort or PLB_MnRearbitrate or timeout.

- Error 2.24.2

  An error message is issued if this transfer qualifier does not match Mn_type when PLB_MnAddrAck is active.

8.8.5 PLB_TAttribute

The following error messages are issued for this signal:

- Error 2.27.1

  An error message is issued when the PLB changes this transfer qualifier from the time a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated via Mn_abort, PLB_MnRearbitrate or PLB_MnTimeout.

- Error 2.27.2

  An error message is issued if this transfer qualifier does not match Mn_TAttribute when PLB_MnAddrAck is active.

8.8.6 PLB_lockErr

The following error messages are issued for this signal:

- Error 2.28.1

An error message is issued when the PLB changes this transfer qualifier from the time a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated through Mn_abort or PLB_MnRearbitrate or Timeout.

- Error 2.28.2

  An error message is issued if this transfer qualifier does not match Mn_lockError when PLB_MnAddrAck is active.

8.8.7 PLB_ABus/PLB_UAbus

The following error messages are issued for this signal:

- Error 2.29.1

  An error message is issued when the PLB changes this transfer qualifier from the time a request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated through Mn_abort or PLB_MnRearbitrate or timeout.

- Error 2.29.2

  An error message is issued if this transfer qualifier does not match Mn_ABus when PLB_MnAddrAck is active.

- Error 2.29.3

  An error message is issued when the PLB_Abus word address is non-zero during a line write transfer.

8.8.8 PLB_busLock

The following error messages are issued for this signal:

- Error 2.30.1

  An error message is issued if this arbitration qualifier does not match Mn_busLock when PLB_MnAddrAck is active.

- Error 2.30.2

  An error message is issued if PLB_busLock and PLB_SAValid are active at the same time and the bus is not actually locked. This may be due to the previous request being aborted.

- Error 2.30.3

  An error message is issued when PLB_busLock is active and SYS_plbReset is active.

- Error 2.30.4

  An error message is issued if this arbitration qualifier does not match Mn_busLock when the bus is already locked.

- Error 2.30.5

  An error message is issued if this signal is asserted with no PLB_PAValid or PLB_SAValid and the bus is not locked.

8.8.9 PLB_rdBurst

The following error messages are issued for this signal:

- Error 2.31.1

  PLB_rdBurst and PLB_RESET are active at the same time.

- Error 2.31.2

  PLB asserted PLB_rdBurst without a primary or secondary read burst transfer acknowledged.

- Error 2.31.3

  PLB_rdBurst does not match M-rdBurst during a read burst transfer.

- Error 2.31.4

  PLB reasserted PLB_rdBurst before the current burst transfer received its last dAck.

- Error 2.31.5

  PLB failed to deassert PLB_rdBurst after bTerm was received for the current burst transfer.

8.8.10 PLB_wrBurst

The following error messages are issued for this signal:

- Error 2.32.1

  An error message is issued if this transfer qualifier does not match Mn_wrBurst during a primary request with PLB_PAValid active.

- Error 2.32.3

  An error message is issued when PLB_wrBurst is re-asserted before the last Sln_wrDAck of a burst transfer is received.

- Error 2.32.4

  An error message is issued when PLB_wrBurst is active the cycle after Sln_wrBTerm.

- Error 2.32.5

  An error message is issued when PLB_wrBurst is active and SYS_plbReset is active.

- Error 2.32.6

  An error message is issued when PLB_wrBurst is active during a non-burst write cycle.

- Error 2.32.7

  An error message is issued when the PLB changes this transfer qualifier from when a primary request to transfer is made until the addressed slave asserts its Sln_addrAck, or the transfer is terminated via Mn_abort or PLB_MnRearbitrate or Timeout.

8.8.11 PLB_rdPrim

The following error messages are issued for this signal:

- Error 2.33.1

An error message is issued if this signal is asserted without the last read cycle SIn_AddrAck being a secondary acknowledge with PLB_SAValid.

8.8.12 PLB_wrPrim

The following error messages are issued for this signal:

- Error 2.34.1

  An error message is issued if this signal is asserted without SIn_wrComp.

- Error 2.34.2

  An error message is issued if this signal is asserted without a secondary/pipelined write transaction acknowledged.

8.8.13 PLB_Abort

The following error messages are issued for this signal:

- Error 2.35.1

  An error message is issued if this transfer qualifier does not match Mn_Abort when PLB_MnAddrAck is active.

8.8.14 PLB_SrdDbus

The following error messages are issued for this signal:

- Error 2.36.1

  An error message is issued if the PLB_sRdDbus is not properly “replicated” as indicated in the architecture specification for smaller slaves connected to larger width PLB Arbiters.

8.9 Time Out Handshake Checks (For 3.x PLB)

The following checks relate specifically to the event of time-out as described in the beginning of this document. Furthermore these checks are implemented based on two implementation assumptions:

1. The arbiter will respond with required handshaking to the master within two cycles of time-out.
2. All required handshaking to complete the timed-out request is done in a minimal number of cycles. For example in the event of a 4-word line read transfer that timed-out then 4 successive PLB_rdDAcks are to be issued (with accompanying PLB_MErr) with no wait cycles between these.

8.9.1 PLB_MnWrBTerm

The following error messages are issued for this signal:

- Error 3.1.1

  An error message is issued when a write burst transfer has timed out and the arbiter does not assert this signal to the corresponding master who’s request timed out.

8.9.2 PLB_MnRdBTerm

The following error messages are issued for this signal:

- Error 3.2.1

  An error message is issued when a read burst transfer has timed out and the arbiter does not assert this signal to the corresponding master who's request timed out.

8.9.3 PLB_MnErr

An error message is issued if a time-out occurs and the arbiter fails to assert the corresponding PLB_MnErr signal in any read/write DAck cycle for non-line, line or burst transfer requests.

Index

R

S

T

V

W

The information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in implantation or other life support applications where malfunction may result in injury or death to persons. The information contained in this document does not affect or change IBM's product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as illustration. The results obtained in other operating environments may vary.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable for any damages arising directly or indirectly from any use of the information contained in this document.

IBM Microelectronics Division
1580 Route 52, Bldg. 504
Hopewell Junction, NY
12533-6531

The IBM home page can be found at http://www.ibm.com

The IBM Microelectronics Division home page can be found at
http://www.chips.ibm.com

Document No. SA-14-2542-11