BFM Simulation in Platform Studio

This document describes the basics of Bus Functional Model simulation within Xilinx® Platform Studio. The following topics are included:

- "Introduction"
- "Bus Functional Simulation Basics"
- "Bus Functional Model Use Cases"
- "Bus Functional Simulation Methods"
- "Getting and Installing the Platform Studio BFM Package"
- "Using the Platform Studio BFM Package"

Introduction

Typically, there are two ways to verify the functionality of a hardware component that has a bus interface: (1) you can create a test bench or (2) you can create a larger system with other known-good components that create or respond to bus transactions. Both methods (described in more detail below) have drawbacks that Bus Functional Simulation allows you circumvent entirely.

Test Bench Verification

Creating a test bench is time-consuming. It involves describing the connections and test vectors for all combinations of bus transactions.

© Copyright 2002 - 2007 Xilinx, Inc. All Rights Reserved.

XILINX, the Xilinx logo, the Brand Window and other designated brands included herein are trademarks of Xilinx, Inc.

The PowerPC name and logo are registered trademarks of IBM Corp., and used under license. All other trademarks are the property of their respective owners.

Xilinx is disclosing this user guide, manual, release note, and/or specification (the "Documentation") to you solely for use in the development of designs to operate with Xilinx hardware devices. You may not reproduce, distribute, republish, download, display, post, or transmit the Documentation in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Xilinx expressly disclaims any liability arising out of your use of the Documentation. Xilinx reserves the right, at its sole discretion, to change the Documentation without notice at any time. Xilinx assumes no obligation to correct any errors contained in the Documentation, or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information. THE DOCUMENTATION IS DISCLOSED TO YOU "AS-IS" WITH NO WARRANTY OF ANY KIND. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DOCUMENTATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS. IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOSS OF DATA OR LOST PROFITS, ARISING FROM YOUR USE OF THE DOCUMENTATION.

Verification in the Context of a Larger System

Creating a system with added components to stimulate the bus is also time-consuming. It requires that you describe the established connections to the device under test and program the added components to do one or both of the following:

- Generate the desired bus transactions to which the device under test will respond.
- Respond to bus transactions that the device under test is generating.

Programming such a system usually involves creating some code, compiling it, storing it in memory for the components to read, and generating the correct bus transactions.

Verification through Bus Functional Simulation

Bus Functional Simulation simplifies the verification of hardware components that attach to a bus by providing the ability to generate bus stimulus.

Bus Functional Simulation Basics

Bus Functional Simulation usually involves the following components:

- A Bus Functional Model
- A Bus Functional Language
- A Bus Functional Compiler

Bus Functional Models (BFMs)

BFMs are hardware components that include and model a bus interface. There are different BFMs for different buses. For example, there are OPB BFM components and PLB BFM components. Each is used to connect to its own respective bus.

For each bus, there are different model types. For the OPB bus, for example, there are OPB Master, OPB Slave, and OPB Monitor BFM components. The same set of components and more could exist for other busses, or, in some cases, the functionality of BFM components could be combined into a single model.

Bus Functional Language (BFL)

The BFL describes the behavior of the BFM components. You may specify how to initiate or respond to bus transactions using commands in a BFL file.

Bus Functional Compiler (BFC)

The BFC translates a BFL file into the commands that actually program the selected Bus Functional Model.

Bus Functional Model Use Cases

There are two main use cases for Bus Functional Models:

- IP Verification
- Speed Up Simulation

IP Verification

When verifying a single piece of IP that includes a bus interface, you are mainly concerned with internal details of the IP design and the bus interactions. It is inefficient to attach the IP to a large system only to verify that it is functioning properly.

The following figure shows an example in which a master BFM generates bus transactions to which the device under test responds. The monitor BFM reports any errors regarding the bus compliance of the device under test.

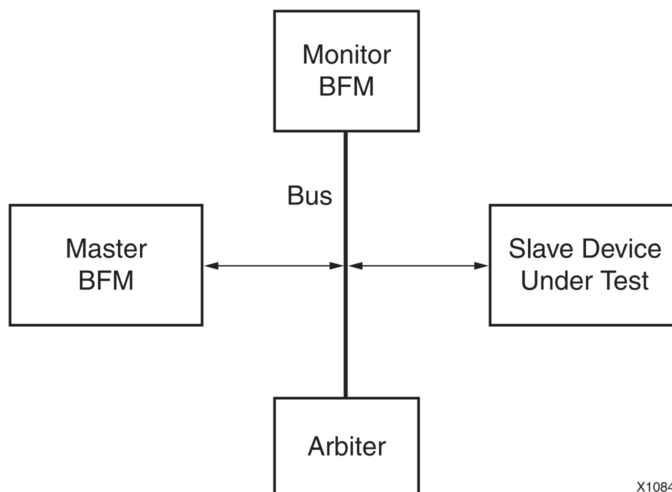

Figure 1: **Slave IP Verification Use Case**

The following figure shows an example in which a slave BFM responds to bus transactions that the device under test generates. The monitor BFM reports any errors regarding the bus compliance of the device under test.

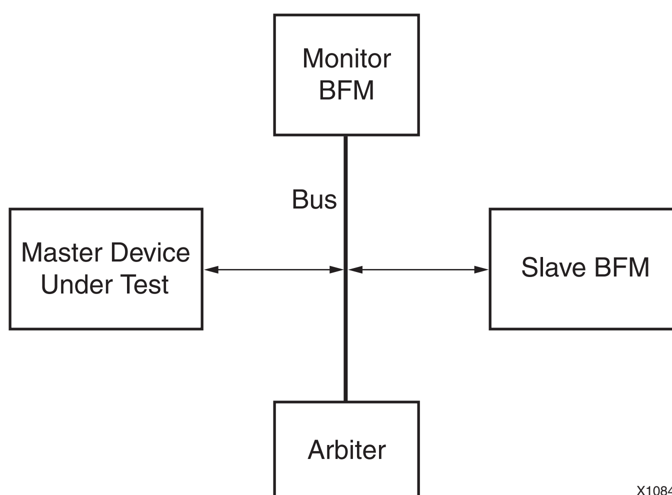

Figure 2: **Master IP Verification Use Case**

Speed-Up Simulation

When verifying a large system design, it is often time consuming to simulate the internal details of each IP component that attaches to a bus. There are certain complex pieces of IP that take a long time to simulate and could be easily replaced by a Bus Functional Model, especially when the internal details of the IP are not of interest. Some of these IP components are not easy to program to generate the desired bus transactions.

The following figure shows how two different IP components that are bus masters have been replaced by BFM master modules. These modules are simple to program and may provide a shorter simulation time because no internal details are modeled.

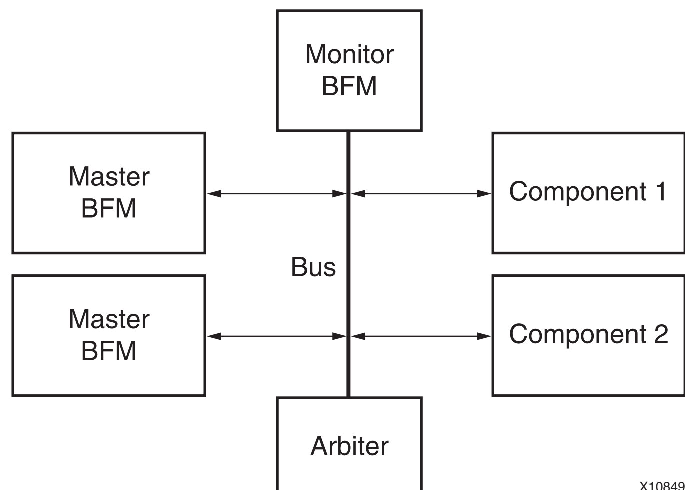

Figure 3: **Speed-Up Simulation Use Case**

Bus Functional Simulation Methods

There are two software packages that allow you to perform Bus Functional Simulation, and each applies its own methodology:

- IBM® CoreConnect™ Toolkit
- Xilinx® EDK BFM Package

Neither software package is included with EDK, but they are required if you intend to perform bus functional simulation. You can download them free of charge once you obtain a license for the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to a wealth of documentation, Bus Functional Models, and the Bus Functional Compiler.

Xilinx provides a Web-based licensing mechanism that enables you to obtain the CoreConnect from the Xilinx web site. To license CoreConnect, use an internet browser to access: http://www.xilinx.com/coreconnect. Once the request has been approved (typically within 24 hours), you will receive an E-mail granting you access to the protected web site from which to download the toolkit.

For further documentation on the CoreConnect Bus Architecture, refer to the IBM CoreConnect web site: http://www.ibm.com/chips/products/coreconnect

Note: There are some differences between IBM CoreConnect and the Xilinx implementation of CoreConnect. These are described in the *Processor IP Reference Guide*, available in your `$XILINX_EDK/doc/usenglish` directory. Refer to the following sections: "On-Chip Peripheral Bus V2.0 with OPB Arbiter" for differences in the OPB bus, "Processor Local Bus (PLB) V3.4" for differences in the PLB bus, and "Device Control Register Bus (DCR) V2.9" for differences in the DCR bus.

IBM CoreConnect Toolkit

The IBM CoreConnect Toolkit is actually a collection of three toolkits:

- OPB Toolkit
- PLB Toolkit
- DCR Toolkit

Each toolkit includes a collection of HDL files that represents predefined systems, including a bus, bus masters, bus slaves, and bus monitors.

The predefined systems included in the toolkits may be modified manually to connect the hardware components to be tested. This usually involves a fair amount of work to describe all the connections to the bus and to make sure there are no errors in setting up the test environment.

Refer to the CoreConnect Toolkit documentation for more information on how to use it to verify your hardware module.

Platform Studio BFM Package

The Platform Studio BFM package includes a set of CoreConnect BFMs, the Bus Functional Compiler, and CoreConnect documents tailored for use within Platform Studio. The BFM package enables you to specify bus connections from a high-level description, such as an MHS file. By allowing the Platform Studio tools to write the HDL files that describe the connections, the time and effort required to set up the test environment are reduced.

The following sections describe how to perform BFM simulation using the Platform Studio BFM Package.

Getting and Installing the Platform Studio BFM Package

The use of the CoreConnect BFM components requires the acceptance of a license agreement. For this reason, the BFM components are not installed along with EDK. Xilinx provides a separate installer for these called the "Xilinx EDK BFM Package."

To obtain the Xilinx EDK BFM Package, you must register and obtain a license to use the IBM CoreConnect Toolkit at:

http://www.xilinx.com/coreconnect

After you register, you receive instructions and a link to download the CoreConnect Toolkit files. You can then install the files using the registration key provided.

After running the installer, you may verify that the files were installed by typing the following at the shell command prompt:

```
xilbfc -check
```

If you get a **Success!** message, you are ready to continue. If not, you receive instructions on the error.

Using the Platform Studio BFM Package

After successfully downloading and installing the Platform Studio BFM Package, you may launch Platform Studio. Notice that in the **Add/Edit Cores** dialog box you have the following components:

- OPB Device BFM (`opb_device_bfm`)

  The OPB device model may act as a master, slave, or both. The master contains logic to initiate transactions on the bus automatically. The slave contains logic to respond to bus transactions based on an address decode operation. The model maintains an internal memory that can be initialized through the Bus Functional Language and may be dynamically checked during simulation or when all bus transactions have completed.

- OPB Monitor BFM (`opb_monitor_bfm`)

  The OPB monitor is a model that connects to the OPB and continuously samples the bus signals. It checks for bus compliance or violations of the OPB architectural specifications and reports warnings and errors.

- PLB Master BFM (`plb_master_bfm`)

  The PLB master model contains logic to initiate transactions on the bus automatically. The model maintains an internal memory that can be initialized through the Bus Functional Language and may be dynamically checked during simulation or when all bus transactions have completed.

- PLB Slave BFM (`plb_slave_bfm`)

  The PLB slave contains logic to respond to bus transactions, based on an address decode operation. The model maintains an internal memory that can be initialized through the Bus Functional Language and may be dynamically checked during simulation or when all bus transactions have completed.

- PLB Monitor (`plb_monitor_bfm`)

  The PLB monitor is a model that connects to the PLB and continuously samples the bus signals. It checks for bus compliance or violations of the PLB architectural specifications and reports warnings and errors.

- PLB v4.6 Master BFM (`plbv46_master_bfm`)

  The PLB v4.6 master model contains logic to initiate bus transactions on the PLB v4.6 bus automatically. The model maintains an internal memory that can be initialized through the Bus Functional Language and may be dynamically checked during simulation or when all bus transactions have completed.

- PLB v4.6 Slave BFM (`plbv46_slave_bfm`)

  The PLB v4.6 slave contains logic to respond to PLB v4.6 bus transactions based on an address decode operation. The model maintains an internal memory that can be initialized through the Bus Functional Language and may be dynamically checked during simulation or when all bus transactions have completed.

- PLB v4.6 Monitor (`plbv46_monitor_bfm`)

  The PLB v4.6 monitor is a model that connects to the PLB v4.6 and continuously samples the bus signals. It checks for bus compliance or violations of the PLB v4.6 architectural specifications and reports warnings and errors.

- BFM Synchronization Bus (`bfm_synch`)

  The BFM Synchronization Bus is not a bus BFM but a simple bus that connects BFMs in a design and allows communication between them. The BFM Synchronization Bus is required whenever BFM devices are used.

These components may be instantiated in an MHS design file for the Platform Studio tools to create the simulation HDL files.

Note: Xilinx has written an adaptation layer to connect the IBM CoreConnect Bus Functional Models to the Xilinx implementation of CoreConnect. Some of these BFM devices have different data/instruction bus widths.

OPB BFM Component Instantiation

The following is an example MHS file that instantiates OPB BFM components and the BFM synchronization bus.

```
# Parameters
PARAMETER VERSION = 2.1.0

# Ports
PORT rx = rx, DIR = IN
PORT tx = tx, DIR = OUT
PORT leds = leds, VEC = [0:7], DIR = INOUT
PORT sys_reset = sys_reset, DIR = IN
PORT sys_clk = sys_clk, DIR = IN, SIGIS = CLK

# Components
BEGIN opb_device_bfm
 PARAMETER INSTANCE = my_device
 PARAMETER HW_VER = 1.00.a
 PARAMETER SLAVE_ADDR_LO_0 = 0x00000000
 PARAMETER SLAVE_ADDR_HI_0 = 0x0000ffff
 PORT SYNCH_IN = synch
 PORT SYNCH_OUT = synch0
 BUS_INTERFACE MSOPB = opb_bus
END

BEGIN opb_monitor_bfm
 PARAMETER INSTANCE = my_monitor
 PARAMETER HW_VER = 1.00.a
 PORT OPB_Clk = sys_clk
 PORT SYNCH_IN = synch
 PORT SYNCH_OUT = synch1
 BUS_INTERFACE MON_OPB = opb_bus
END

BEGIN bfm_synch
 PARAMETER INSTANCE = my_synch
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_NUM_SYNCH = 2
 PORT FROM_SYNCH_OUT = synch0 & synch1
```

```
 PORT TO_SYNCH_IN = synch
END

BEGIN opb_v20
 PARAMETER INSTANCE = opb_bus
 PARAMETER HW_VER = 1.10.b
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_reset
 PORT OPB_Clk = sys_clk
END

BEGIN opb_uartlite
 PARAMETER INSTANCE = my_uart
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_DATA_BITS = 8
 PARAMETER C_CLK_FREQ = 100000000
 PARAMETER C_BAUDRATE = 19200
 PARAMETER C_USE_PARITY = 0
 PARAMETER C_BASEADDR = 0xffff0200
 PARAMETER C_HIGHADDR = 0xffff02ff
 BUS_INTERFACE SOPB = opb_bus
 PORT RX = rx
 PORT TX = tx
END

BEGIN opb_gpio
 PARAMETER INSTANCE = my_gpio
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_GPIO_WIDTH = 8
 PARAMETER C_ALL_INPUTS = 0
 PARAMETER C_BASEADDR = 0xffff0100
 PARAMETER C_HIGHADDR = 0xffff01ff
 BUS_INTERFACE SOPB = opb_bus
 PORT GPIO_IO = leds
END
```

PLB BFM Component Instantiation

The following is an example MHS file that instantiates PLB BFM components and the BFM synchronization bus.

```
# Parameters
PARAMETER VERSION = 2.1.0

# Ports
PORT sys_clk = sys_clk, DIR = I, SIGIS = CLK
PORT sys_reset = sys_reset, DIR = IN

# Components
BEGIN plb_v34
 PARAMETER INSTANCE = myplb
 PARAMETER HW_VER = 1.01.a
 PARAMETER C_DCR_INTFCE = 0
 PORT PLB_Clk = sys_clk
 PORT SYS_Rst = sys_reset
END
```

```
BEGIN plb_bram_if_cntlr
 PARAMETER INSTANCE = myplbbram_cntlr
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xFFFF8000
 PARAMETER C_HIGHADDR = 0xFFFFFFFF
 BUS_INTERFACE PORTA = porta
 BUS_INTERFACE SPLB = myplb
END

BEGIN bram_block
 PARAMETER INSTANCE = bram1
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = porta
END

BEGIN plb_master_bfm
 PARAMETER INSTANCE = my_master
 PARAMETER HW_VER = 1.00.a
 PARAMETER PLB_MASTER_ADDR_LO_0 = 0xFFFF0000
 PARAMETER PLB_MASTER_ADDR_HI_0 = 0xFFFFFFFF
 BUS_INTERFACE MPLB = myplb
 PORT SYNCH_OUT = synch0
 PORT SYNCH_IN = synch
END

BEGIN plb_slave_bfm
 PARAMETER INSTANCE = my_slave
 PARAMETER HW_VER = 1.00.a
 PARAMETER PLB_SLAVE_ADDR_LO_0 = 0xFFFF0000
 PARAMETER PLB_SLAVE_ADDR_HI_0 = 0xFFFF7FFF
 BUS_INTERFACE SPLB = myplb
 PORT SYNCH_OUT = synch1
 PORT SYNCH_IN = synch
END

BEGIN plb_monitor_bfm
 PARAMETER INSTANCE = my_monitor
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE MON_PLB = myplb
 PORT SYNCH_OUT = synch2
 PORT SYNCH_IN = synch
END

BEGIN bfm_synch
 PARAMETER INSTANCE = my_synch
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_NUM_SYNCH = 3
 PORT FROM_SYNCH_OUT = synch0 & synch1 & synch2
 PORT TO_SYNCH_IN = synch
END
```

PLB v4.6 BFM Component Instantiation

The following is an example MHS file that instantiates PLB v4.6 BFM components and the BFM synchronization bus.

```
# Parameters
PARAMETER VERSION = 2.1.0

# Ports
PORT sys_clk = sys_clk, DIR = I, SIGIS = CLK
PORT sys_reset = sys_reset, DIR = IN

# Components
BEGIN plb_v46
PARAMETER INSTANCE = myplb
PARAMETER HW_VER = 1.01.a
PARAMETER C_DCR_INTFCE = 0
PORT PLB_Clk = sys_clk
PORT SYS_Rst = sys_reset
END

BEGIN plb_bram_if_cntlr
PARAMETER INSTANCE = myplbbram_cntlr
PARAMETER HW_VER = 1.00.a
PARAMETER C_BASEADDR = 0xFFFF8000
PARAMETER C_HIGHADDR = 0xFFFFFFFF
BUS_INTERFACE PORTA = porta
BUS_INTERFACE SPLB = myplb
END

BEGIN bram_block
PARAMETER INSTANCE = bram1
PARAMETER HW_VER = 1.00.a
BUS_INTERFACE PORTA = porta
END

BEGIN plbv46_master_bfm
PARAMETER INSTANCE = my_master
PARAMETER HW_VER = 1.00.a
PARAMETER PLB_MASTER_ADDR_LO_0 = 0xFFFF0000
PARAMETER PLB_MASTER_ADDR_HI_0 = 0xFFFFFFFF
BUS_INTERFACE MPLB = myplb
PORT SYNCH_OUT = synch0
PORT SYNCH_IN = synch
END

BEGIN plbv46_slave_bfm
PARAMETER INSTANCE = my_slave
PARAMETER HW_VER = 1.00.a
PARAMETER PLB_SLAVE_ADDR_LO_0 = 0xFFFF0000
PARAMETER PLB_SLAVE_ADDR_HI_0 = 0xFFFF7FFF
BUS_INTERFACE SPLB = myplb
PORT SYNCH_OUT = synch1
PORT SYNCH_IN = synch
END

BEGIN plbv46_monitor_bfm
PARAMETER INSTANCE = my_monitor
```

```
PARAMETER HW_VER = 1.00.a
BUS_INTERFACE MON_PLB = myplb
PORT SYNCH_OUT = synch2
PORT SYNCH_IN = synch
END

BEGIN bfm_synch
PARAMETER INSTANCE = my_synch
PARAMETER HW_VER = 1.00.a
PARAMETER C_NUM_SYNCH = 3
PORT FROM_SYNCH_OUT = synch0 & synch1 & synch2
PORT TO_SYNCH_IN = synch
END
```

BFM Synchronization Bus Usage

The BFM synchronization bus collects the SYNCH_OUT outputs of each BFM component in the design. The bus output is then connected to the SYNCH_IN of each BFM component. The following figure depicts an example for three BFMs, and the MHS examples above show their instantiation for OPB, PLB, and PLB v4.6 BFMs.

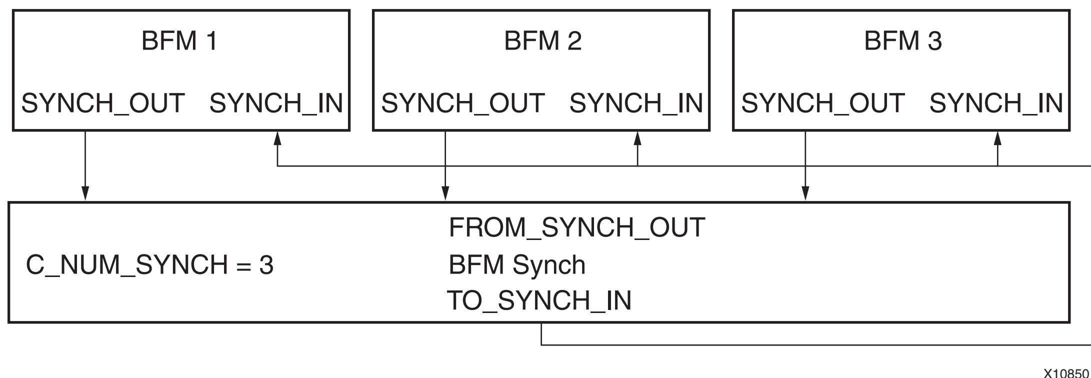

Figure 4: **BFM Synchronization Bus Usage**

OPB Bus Functional Language Usage

The following is a sample BFL file written for the OPB MHS file example above, which instantiates OPB BFM components.

```
-- FILE: sample.bfl
-- This example initializes an OPB master
--
-- Initialize my_device
-- Note: The instance name for my_device is duplicated in the
-- path due to the wrapper level inserted by the tools set_device
(path=/system/my_device/my_device/device,device_type=opb_device)

-- Write and read 32-bit data using byte-enable architecture
-- Note: The CoreConnect opb_device is a 64-bit device, hence the 8-bit byte enables
--       are aligned accordingly (shifted by 0's).
write (addr=ffff0100,be=11110000,data=00112233)
```

```
write (addr=ffff0104,be=00001111,data=44556677)
read  (addr=ffff0100,be=11110000,data=00000033)
read  (addr=ffff0104,be=00001111,data=00000000)

-- Write and read 16-bit data using byte-enable architecture
write (addr=ffff0100,be=11000000,data=00040004)
read  (addr=ffff0100,be=11000000,data=00040004)
write (addr=ffff0102,be=00110000,data=00400040)
read  (addr=ffff0102,be=00110000,data=00400040)
write (addr=ffff0104,be=00001100,data=04000400)
read  (addr=ffff0104,be=00001100,data=04000400)
write (addr=ffff0106,be=00000011,data=40004000)
read  (addr=ffff0106,be=00000011,data=40004000)

-- Write and read 8-bit data using byte-enable architecture
write (addr=ffff0100,be=10000000,data=01010101)
read  (addr=ffff0100,be=10000000,data=01010101)
write (addr=ffff0101,be=01000000,data=02020202)
read  (addr=ffff0101,be=01000000,data=02020202)
write (addr=ffff0102,be=00100000,data=03030303)
read  (addr=ffff0102,be=00100000,data=03030303)
write (addr=ffff0103,be=00010000,data=04040404)
read  (addr=ffff0103,be=00010000,data=04040404)
write (addr=ffff0104,be=00001000,data=05050505)
read  (addr=ffff0104,be=00001000,data=05050505)
write (addr=ffff0105,be=00000100,data=06060606)
read  (addr=ffff0105,be=00000100,data=06060606)
write (addr=ffff0106,be=00000010,data=07070707)
read  (addr=ffff0106,be=00000010,data=07070707)
write (addr=ffff0107,be=00000001,data=08080808)
read  (addr=ffff0107,be=00000001,data=08080808)
```

Note: The CoreConnect BFM devices differ in certain bus widths. In this example, the `opb_device` is a 64-bit device, so the byte enable signals are shifted accordingly to match the address alignment. Also, the data mirroring required by the CoreConnect bus specifications are performed on the data field. The adaptation layer translates this into a 32-bit bus width.

More information about the OPB Bus Functional Language may be found in the `OpbToolkit.pdf` document in the `$XILINX_EDK/third_party/doc` directory.

PLB Bus Functional Language Usage

The following is a sample BFL file written for the PLB and PLB v4.6 MHS examples above, which instantiate the PLB BFM or PLB v4.6 BFM components.

```
-- FILE: sample.bfl
-- This test case initializes a PLB master

-- Initialize my_master
-- Note: The instance name for plb_master is duplicated in the
-- path due to the wrapper level inserted by the tools
   set_device(path=/system/my_master/my_master/master,device_type=plb_master)

-- Configure as 64-bit master
configure(msize=01)

-- Write and read 64-bit data using byte-enable architecture
```

```
mem_update(addr=ffff8000,data=00112233_44556677)
mem_update(addr=ffff8008,data=8899aabb_ccddeeff)
write     (addr=ffff8000,size=0000,be=11111111)
write     (addr=ffff8008,size=0000,be=11111111)
read      (addr=ffff8000,size=0000,be=11111111)
read      (addr=ffff8008,size=0000,be=11111111)

-- Write and read 32-bit data using byte-enable architecture
mem_update(addr=ffff8010,data=11111111_22222222)
write     (addr=ffff8010,size=0000,be=11110000)
write     (addr=ffff8014,size=0000,be=00001111)
read      (addr=ffff8010,size=0000,be=11110000)
read      (addr=ffff8014,size=0000,be=00001111)

-- Write and read 16-bit data using byte-enable architecture
mem_update(addr=ffff8020,data=33334444_55556666)
write     (addr=ffff8020,be=1100_0000)
write     (addr=ffff8022,be=0011_0000)
write     (addr=ffff8024,be=0000_1100)
write     (addr=ffff8026,be=0000_0011)
read      (addr=ffff8020,be=1100_0000)
read      (addr=ffff8022,be=0011_0000)
read      (addr=ffff8024,be=0000_1100)
read      (addr=ffff8026,be=0000_0011)

-- Write and read 8-bit data using byte-enable architecture
mem_update(addr=ffff8030,data=778899aa_bbccddee)
write     (addr=ffff8030,be=1000_0000)
write     (addr=ffff8031,be=0100_0000)
write     (addr=ffff8032,be=0010_0000)
write     (addr=ffff8033,be=0001_0000)
write     (addr=ffff8034,be=0000_1000)
write     (addr=ffff8035,be=0000_0100)
write     (addr=ffff8036,be=0000_0010)
write     (addr=ffff8037,be=0000_0001)
read      (addr=ffff8030,be=1000_0000)
read      (addr=ffff8031,be=0100_0000)
read      (addr=ffff8032,be=0010_0000)
read      (addr=ffff8033,be=0001_0000)
read      (addr=ffff8034,be=0000_1000)
read      (addr=ffff8035,be=0000_0100)
read      (addr=ffff8036,be=0000_0010)
read      (addr=ffff8037,be=0000_0001)

-- Write and read a 16-word line
mem_update(addr=ffff8080,data=01010101_01010101)
mem_update(addr=ffff8088,data=02020202_02020202)
mem_update(addr=ffff8090,data=03030303_03030303)
mem_update(addr=ffff8098,data=04040404_04040404)
mem_update(addr=ffff80a0,data=05050505_05050505)
mem_update(addr=ffff80a8,data=06060606_06060606)
mem_update(addr=ffff80b0,data=07070707_07070707)
mem_update(addr=ffff80b8,data=08080808_08080808)
write     (addr=ffff8080,size=0011,be=1111_1111)
read      (addr=ffff8080,size=0011,be=1111_1111)
```

More information about the PLB Bus Functional Language may be found in the `PlbToolkit.pdf` document in the `$XILINX_EDK/third_party/doc` directory.

Bus Functional Compiler Usage

The Bus Functional Compiler provided with the CoreConnect toolkit is a Perl script called BFC. The script uses a configuration file called `.bfcrc`, which tells the script which simulator is used and the paths to the BFMs. Xilinx EDK includes a helper executable called `xilbfc`, which enables this configuration for you. The helper application has been previously used to verify the correct installation on the BFM Package.

To compile a BFL file you can type the following at a command prompt:

```
xilbfc sample.bfl
```

This creates a script targeted for the selected simulator that initializes the BFM devices. In the case of ModelSim, it creates a file called `sample.do`.

Running BFM Simulations

To run the BFM simulation, you must:

1. Compile the simulation HDL files.
2. Load the system into the simulator.
3. Initialize the Bus Functional Models.
4. (Optional) Create a waveform list or load a previously created one.
5. Provide the clock and reset stimulus to the system.
6. Run the simulation.

The following is an example script called `run.do`, which you can write to perform the steps listed above:

```
do system.do
vsim system
do sample.do
do wave.do
force -freeze sim:/system/sys_clk 1 0, 0 {10 ns} -r 20 ns
force -freeze sim:/system/sys_reset 0, 1 {200 ns}
run 2 us
```

Note: If your design has an input reset that is active high, replace the reset line with:
`force -freeze sim:/system/sys_reset 1 , 0 {200 ns}`

At the ModelSim prompt you then type: `do run.do`.