XAPP516 (v1.0) May 25, 2006

Bus Functional Model (BFM) Simulation of Processor Intellectual Property

Author: Lester Sanders

Summary

This note provides the flow for simulating Processor Intellectual Property (PIP) cores using Bus Functional Models (BFMs). The PIP cores used as examples include a new core, existing cores, and a sample system core. The simulation results of the OPB GPIO core are given to illustrate the capability of a BFM simulation. An introduction to writing stimuli for BFM simulation using a Bus Functional Language (BFL) in conjunction with a VHDL test bench is given.

The example designs are intended to aid in the understanding the functionality of PIP cores. The comments in the code indicate the function being verified. The user can learn techniques for writing test benches/BFLs by reading and re-using the example code. The references provide information for users interested in writing complex or exhaustive test bench/BFLs.

Included Files

This application note includes BFM simulation files:

www.xilinx.com/bvdocs/appnotes/xapp516.zip

Introduction

Xilinx manufactures FPGAs which contain IBM Power PC™ hard processors on Virtex™ -II Pro and Virtex-4 families. Xilinx also offers the MicroBlaze™ soft core processor, which can be used in either Spartan™ or Virtex families. The Power PC processor interfaces to PIP peripherals using the IBM CoreConnect buses, including the Processor Local Bus (PLB), On-chip Peripheral Bus (OPB), and the Device Control Register (DCR) bus. The Microblaze soft core processor uses the OPB. The BFMs discussed in this note model transactions on the PLB and OPB buses.

The Xilinx EDK Bus Functional Language (BFM) package is not included in EDK. The BFM package can be downloaded at no charge after obtaining a license for the IBM CoreConnect Bus Architecture. To obtain a license for CoreConnect, complete the form at http://www.xilinx.com/ipcenter/processor_central/register_coreconnect.htm. Once the request has been approved (typically within 24 hours), an e-mail granting access to the protected web site provides instructions on downloading the toolkit.

The traditional form of simulation, shown in Figure 1, uses a test bench to generate stimuli to,

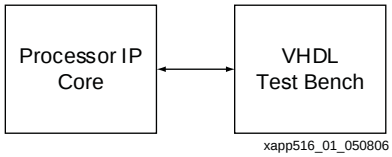


Figure 1: **Traditional Simulation (Not Using BFMs)**

and to check the results from, a device under test (DUT), the PIP core. In the BFM simulation shown in Figure 2, the PIP core interfaces to the microprocessor bus on one side and to external pins on the other side. The BFM is used to model the microprocessor bus side. As with

traditional simulation, a VHDL test bench generates stimuli to, and checks results from, the PIP core. A Bus Functional Language (BFL) provides a method to read and write the registers of the PIP core. The BFL configures the BFM to generate stimuli to and check results from the bus side.

Using a BFM provides an efficient means of including bus transactions in simulation. It is simpler to generate bus transactions using a BFM than an actual microprocessor model because there is no C code involved. A designer need know only the addresses of the PIP registers and the bus operation. Knowledge of the microprocessor architecture, instructions, registers, and ports is not required. The use of a BFM allows control over bus transactions, transaction spacing, and the ability to simulate abnormal transactions, such as aborts, retries, and errors.

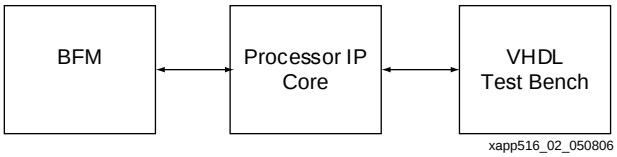


Figure 2: **Simulation (Using BFMs)**

Synchronizing Stimuli

Either the BFM or the VHDL test bench interfaces with the PIP core at a time. There are handshaking signals between the BFM and the test bench which ensure that the BFM-PIP core and VHDL test bench-PIP core interfaces are synchronized. In a typical simulation of the microprocessor writing to a peripheral, the BFM writes to the control registers of the PIP core, causing the PIP core to perform some function, then the VHDL test bench verifies the function by reading the PIP output. In a simulation of the PIP core generating data for the microprocessor to read, the test bench generates external stimuli to the PIP core, then the BFM does a read operation to verify that the PIP core produces the expected results.

Examples for BFM simulation are provided in the bfm_simulation.zip file. The first example, opb_myled_cntlr, is a BFM simulation of a new IP core using Create IP wizard. It is a simple example, providing limited BFL - VHDL test bench code. The second set of examples are existing PIP cores. The reason for simulating existing peripherals is that they provide good examples of BFL-VHDL test benches generating BFL - VHDL test bench stimuli.

Running BFM Simulation

To run BFM simulation on an existing core, unzip bfm_simulation.zip and change to a core's simulation/behavioral directory. Invoke Modelsim, edit the path to the test bench in bfm_system.do if necessary, and run

do ../../scripts/run.do

This provides a complete functional simulation of the core. The <core>_tb.vhd and <core>.m4 files provide stimuli. Comments in these files indicate the test performed. The wave.do file adds the signals and generics needed to understand the operation of the core.

Developing a BFM Simulation Using Create IP Wizard

This section provides the steps used in developing a BFM simulation of the OPB GPIO.

1. Create a directory named designs/opb_gpio_bfm. From XPS, do the following.

   Invoke Hardware -> **Create/Import User Peripheral** and select **Create templates for a new peripheral**

   Repository or Project panel : Browse to /designs/opb_gpio_bfm

   Core Name and Version : Enter opb_gpio for the Name, and 3_01_b for the Version name

   Bus Interfaces panel : Select OPB.

IPIF Services panel : Do not select any services.

Peripheral Simulation Support panel : Check Generate BFM Simulation.

Peripheral Implementation Support panel : Check Generate ISE and XST projects.

In the following, <sim_project> is used as shorthand for the path generated by Create IP Wizard, which is <project>/MyProcessorIPLib/pcores/<core_version>/devl/bfmsim/.

2. Edit the pao file. Change to the <sim_project>/pcores/<core>/data directory. If a new IP core is used, Create IP Wizard generates template files which assist in the development of the new core simulation. If an existing core is used, the VHDL source files and the pao file generated by Create IP wizard should be replaced with

   <EDK>/hw/XilinxProcessorIPLib/pcores/opb_gpio_v3_01_b/data/opb_gpio_v2_1_0.pao, renamed to opb_gpio_tb_v2_1_0.pao, with the following line added to the bottom.

   simlib **opb_gpio_tb_v3_01_b opb_gpio_tb**

3. Edit the VHDL Testbench template. Change to <sim_project>/pcores/<core>/simhdl/vhdl. To edit the template created by Create IP wizard, add the library declaration for xil_bfm and the xil_bfm_pkg.vhd package. Add the PIP core generics which are not passed from a higher level. Add external ports as signals. Provide generic and port maps. Finally, add a test process for generating stimuli and checking results. The edits of the <core>_tb.vhd files are given in the examples.

4. Create m4/BFL files

   The BFM stimuli can be written in BFL directly, or written in m4 and translated into BFL. In the examples, the scripts directory contains the **m4** sub-directory which provides the BFM stimuli. The m4 directory contains a definition file and a stimulus file, <core>_defs.m4 and <core>.m4. The <core>_defs.m4 file provides address locations of the PIP registers and the values of the generics for the simulation being run. These must match the address locations and generics provided in the VHDL test bench located in <sim_project>/pcores/<core_tb>/simhdl/vhdl.

   To produce <core>.bfl in the <sim_project>/scripts directory, run **gen_bfl** from the <sim_project>/scripts/m4 directory. Change to the <sim_project>/scripts directory and run **xilbfc** <core>.bfl to create the simulator stimuli file.

5. In <sim_project>/scripts/run.do, change sample to opb_gpio, and comment the 2nd half of the file (all statements from quit -f).

6. Run

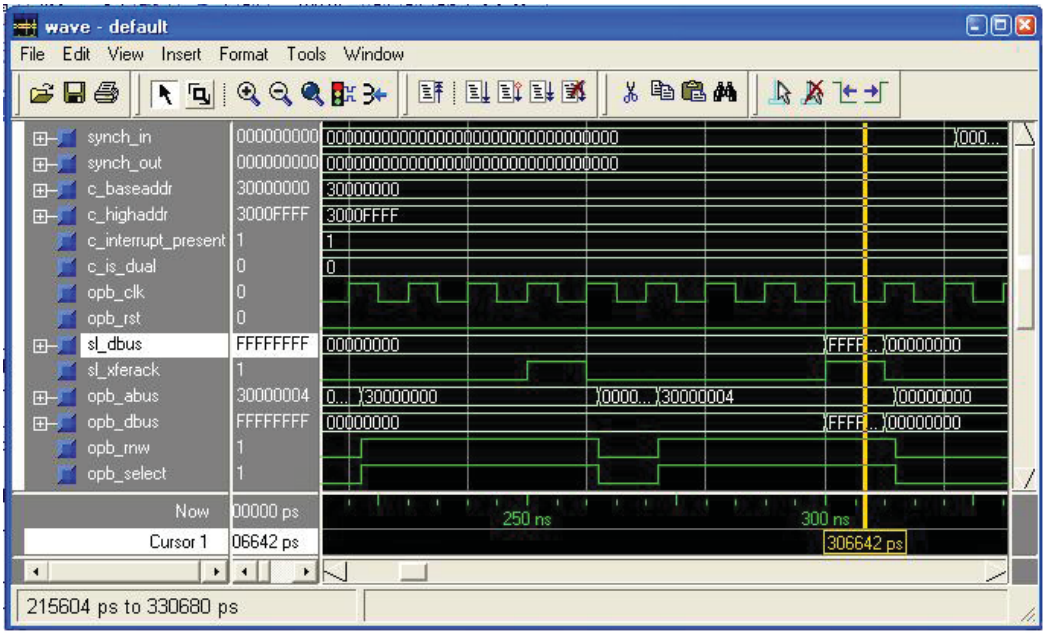
   make -f bfm_sim_cmd.make sim

Understanding BFM Simulation Results

This section describes the simulation results of the OPB GPIO core. A common use of the OPB GPIO core is to provide output register interfaces to LEDs and input register interfaces to switches. This example shows external stimuli generated by the VHDL test bench and checked by the BFM. It then shows stimuli generated by the BFM and checked by the VHDL test bench. It also shows the synchronization signals used in the simulation.

The GPIO is configured with C_IS_DUAL = 0, C_INTERRUPT_PRESENT = 1, and C_GPIO_WIDTH = 32, the generic values in the opb_gpio_defs.m4, and opb_gpio_tb_v3_01_b.vhd files. The signal waveforms below are the result of the stimuli from the test code in the opb_gpio.m4 and opb_gpio_tb_v3_01_b.vhd files.

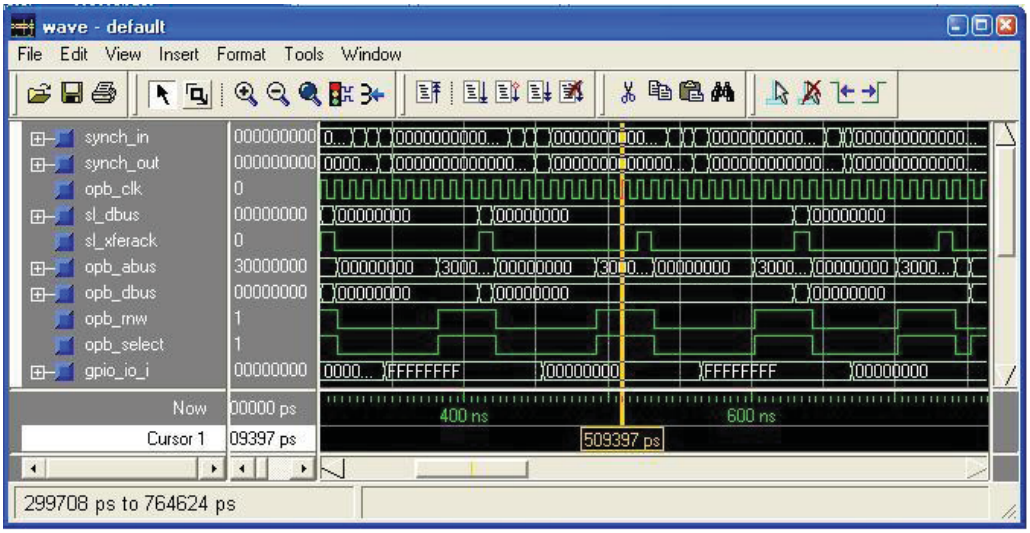
As shown in Figure 3, at 250 and 300 ns, the BFM (lines 11:12 in opb_gpio.m4) reads 0x00000000 and 0xFFFFFFFF from the DATA and TRI registers. The test bench writes a status message to the simulator transcript window indicating the operation performed.



XAPP516_03_050806

Figure 3: **Checking Reset Values**

As shown in Figure 4, from 350 ns to 650 ns, the opb_gpio.m4 (20:31) prompts the VHDL test bench to generate input stimuli into GPIO_IO_I. The synchronization pulses occur at 350, 450, 550, and 650 ns. The 0xFFFFFFFF, 0x00000000, 0xFFFFFFFF, 0x00000000 input stimuli is generated by the test bench at 360, 460, 560, 660 ns. Control is transferred to the BFM, and the BFM verifies that the data register contains the data received from the GPIO_IO_I inputs at 410, 510, 610, and 710 ns.



XAPP516_04_050906

Figure 4: **Reading Input Data**

As shown in Figure 5, from 750 ns to 1000 ns, the simulation verifies that the GPIO writes data correctly to the GPIO_IO_O outputs. The opb_gpio.m4 writes 0x00000000, 0xFFFFFFFF, and 0x00000000 to the data register at 770, 860, and 940 ns. The opb_gpio_tb_v3_01_b.vhd verifies this data at the GPIO_IO_O output pins, and writes a status message.

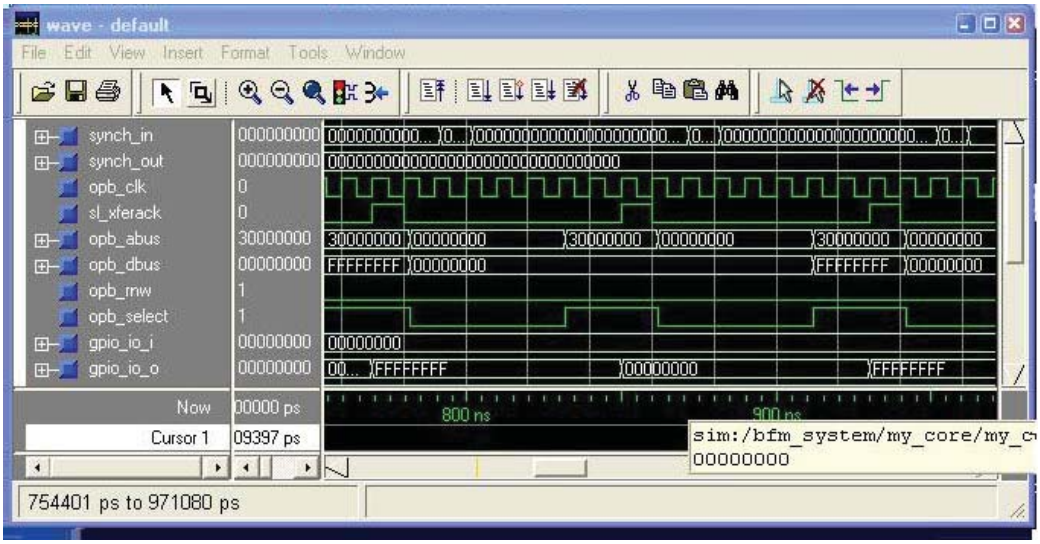


Figure 5: **Writing Output Data**

As shown in Figure 6, from 1660 to 2000 ns, the simulation verifies that interrupts function correctly. For the GPIO, an interrupt is generated when there is a change in input value on GPIO_IO_I. The Interrupt Service Register (ISR) is reset at 1580 ns. The BFM enables interrupts by writing to the IER and GIE at 1630 and 1710 ns. The BFM receives an interrupt at 1840ns. The BFM reads the ISR at 1940 ns.

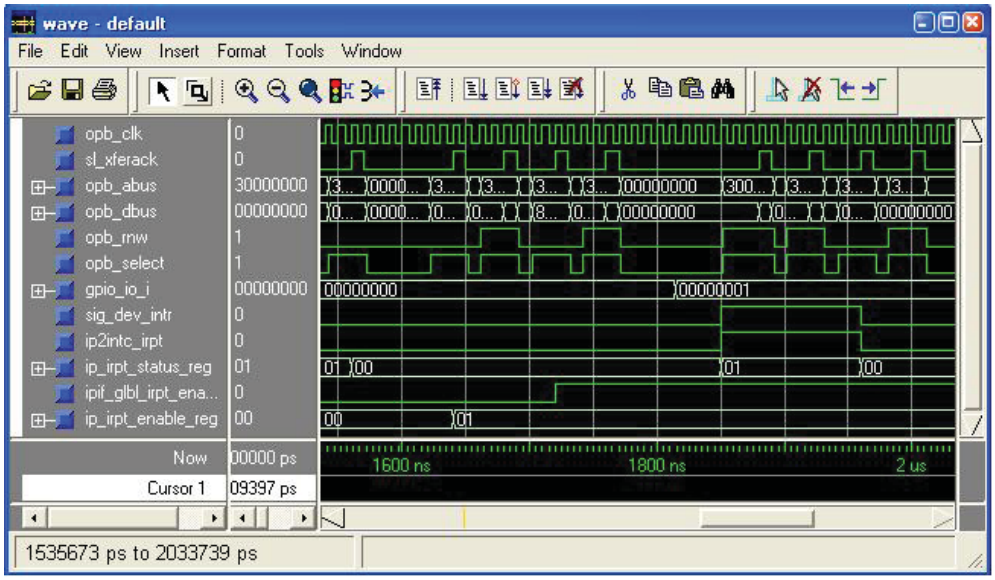


Figure 6: **Testing Interrupts**

Tools/Methods for Writing BFL

Figure 7 shows the tools used to develop BFM test code. Generating the BFM test code is done by writing either m4 or BFL code. If m4 is used, the GNU m4 processor reads a <core>.m4 and writes <core>.bfl. The Bus Functional Compiler reads the BFL and writes the simulator specific simulation commands. The simulator used is specified in the BFC control file (.bfcrc). The simulator command files are **<core>.do** and **system.sh** for ModelSim and NcSim, respectively. The m4 and BFL files are easier to read and write than the simulator specific command files.

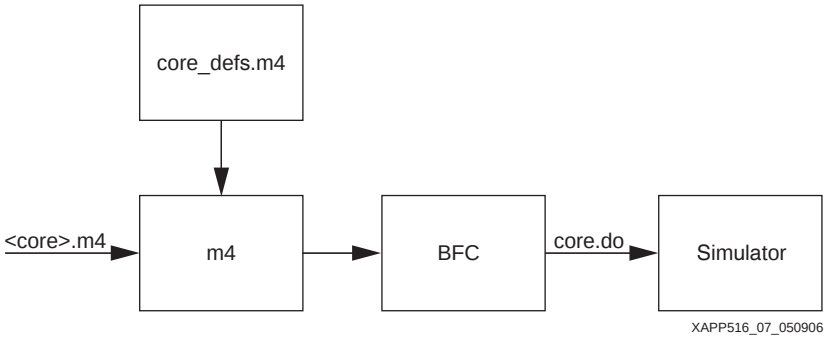


Figure 7: **Development of BFM Test Code**

The BFM read/write commands generate stimuli and test results. The example below shows the read/write commands in m4 and the corresponding BFL commands. The commands writes 0x0000FFFF to the DATA register and verifies the contents with a read operation. The location of the DATA register is defined as 0x30000400 in the opb_gpio_defs.m4 file.

m4:

```
write_word(DATA, 0x0000FFFF)
read_word (DATA, 0x000FFFF)
```

BFL:

```
write(addr=30000400, be=11110000, data=0000FFFF)
read(addr=30000400, be=11110000, data=0000FFFF)
```

Synchronizing BFL and VHDL Testbench Code

This section discusses the synchronization of BFL and VHDL code using the OPB GPIO simulation model shown in Figure 8.

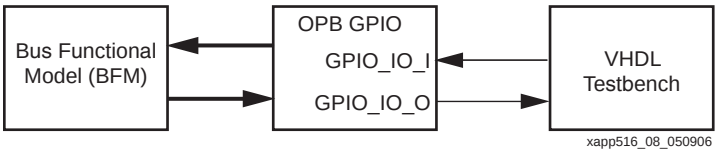


Figure 8: **OPB GPIO as PIP Core in BFM Simulation.**

The **<core>.m4** read/write commands consists of the following sections:

Setup
Check reset value of registers
PLB/OPB Write
    Generate stimuli to registers.
    Request VHDL Testbench to verify PIP core results.
PLB/OPB Read
    Request VHDL Testbench to generate external stimuli.
    Verify PIP core results

For synchronization, both the BFL and VHDL test bench generate and detect synchronization pulses. These pulses occur on the synch_in[0:31] and synch_out[0:31] buses. Common BFL synchronization commands are **start**, **assign**, **assert_in**, **wait_for_intr**, and **stop**.

The VHDL test bench has similar commands to generate and detect synchronization pulses to/from the BFL. Some of the VHDL procedures, included in xil_bfm_pkg.vhd in the xil_bfm library, which simplify the synchronization code for the VHDL test bench are listed below. The following procedures are called from <core_tb>.vhd:

assert_in(1, opb_clk, synch_out(5 to 5)) - VHDL waits for sync pulse from BFL. When received, the VHDL displays a status message to simulator transcript window.

assert_out(1, opb_clk, synch_out(6 to 6)) - VHDL generates a sync pulse to BFL indicating that message has been displayed

assign_in(1, opb_clk, synch_in(7 to 7)) - VHDL waits for sync pulse from BFL. When received, the VHDL generates external stimuli to PIP core.

assign_out(1, opb_clk(synch_out(8 to 8)) - VHDL generates a sync pulse to BFL indicating external stimuli to PIP core has been generated.

Simulating OPB Read Operations

The stimuli to GPIO_IO_I is generated by the VHDL test bench, not the BFL. The synchronization command used by the BFL to request that the VHDL test bench generate the stimuli, and provide a synchronization pulse back when it is done, is shown below. After the synchronization pulse is received, the BFM verifies that the DATA register contains 0xFFFFFFFF. This is repeated for the VHDL providing 0x00000000 into the GPIO_IO_I input.The GPIO_ALL_ONES and ALL_ZEROS constants are defined in opb_gpio_defs.m4.

```
assign
read_word(DATA, GPIO_ALL_ONES)
assign
read_word(DATA, ALL_ZEROS)
```

The corresponding VHDL code shown below displays a note indicating the test being done. The assign_in statement causes the VHDL test bench to wait for a sync pulse from the BFL. When the sync pulse is received, the VHDL test bench generates stimuli into GPIO_IO_I. The stimuli generated are all 1s (ones), and then all 0s (zeros). The assign_out(1, opb_clk,synch_out(8 to 8)) statement causes a synchronization pulse to be generated so that the BFL knows to test the values received.

```
assert FALSE report "Checking GPIO_IO_I gets 1's" severity NOTE;
assign_in(1,opb_clk,synch_in(7 to 7));
GPIO_IO_I <= (others => '1') after TB_OPB_CLK_PERIOD/2.0;
assign_out(1, opb_clk, synch_out(8 to 8));
assert FALSE report "Checking GPIO_IO_I gets 0's" severity NOTE;
assign_in(1,opb_clk,synch_in(7 to 7));
GPIO_IO_I <= (others => '0') after TB_OPB_CLK_PERIOD/2.0;
assign_out(1, opb_clk, synch_out(8 to 8));
```

Simulating OPB Write Operations

The BFL writes to the GPIO registers. Below, the DATA register is written with all 1s (ones) and then all 0s (zeros). After a write, the BFL generates a synchronization pulse to the VHDL test bench, using the **assert_in** command.

```
write_word(DATA, GPIO_ALL_ONES)
assert_in
write_word(DATA, ALL_ZEROS)
assert_in
```

When the VHDL test bench receives the **assert_in** synchronization pulse, it displays a status message and tests if the results are correct.

```
assert_in(1,opb_clk,synch_in(5 to 5));
   assert FALSE report "Checking that data register writes 1's" severity
NOTE;
   for abit in 0 to C_GPIO_WIDTH-1 loop
     assert GPIO_IO_O(abit) = '1' report "GPIO_IO_O did not write 1's"
severity ERROR;
    end loop;
assert_in(1,opb_clk,synch_in(5 to 5));
```

```
   assert FALSE report "Checking that data register writes 0's" severity
NOTE;
   for abit in 0 to C_GPIO_WIDTH-1 loop
     assert GPIO_IO_O(abit) = '0' report "GPIO_IO_O did not write 0's"
severity ERROR;
    end loop;
```

Simulating Interrupts in BFL-VHDL Testbench

The opb_gpio.m4 code listed below reads the interrupt registers:

```
write_word(IER_REG, 0x00000003)
read_word(IER_REG, 0x00000003)
write_word(GIE_REG, 0x80000000)
read_word(GIE_REG, 0x80000000)
assign -- Request VHDL to generate input on GPIO_IO_I to cause interrupt
read_word(DATA, 0x00000001)   -- verify data
read_word(ISR_REG, 0x00000001)  -- verify interrupt
write_word(ISR_REG, 0x00000001) -- reset interrupt
```

The VHDL test bench code for providing the stimuli to cause an interrupt is given below.

```
GPIO_IO_I(C_GPIO_WIDTH-1) <= '0';
GPIO_IO_I(0 to C_GPIO_WIDTH-2) <= (others => '0');
assign_out(1, opb_clk,synch_out(8 to 8));
assign_in(1, opb_clk, synch_in(7 to 7));
wait until (opb_clk'EVENT and opb_clk = '1');
GPIO_IO_I   <= (others => '1');
assign_out(1, opb_clk_synch_out(8 to 8));
```

References

XAPP515: Using Xilinx m4 Functions to Write BFL Stimuli for CoreConnect™ Buses

GNU m4, version 1.4

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/25/06	1.0	Initial Xilinx release.