

ISim Hardware Co-Simulation Tutorial: Interacting with Spartan-6 Memory Controller and On-Board DDR2 Memory

UG818 (v 13.2) July 28, 2011



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.

© Copyright 2002-2011 Xilinx Inc. All Rights Reserved. XILINX, the Xilinx logo, the Brand Window and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners. 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.

Table of Contents

Chapter 1 Introduction	5
Prerequisites.....	6
Tutorial Files.....	6
Chapter 2 Tutorial	9
Step 1: Generating a Design Using the MIG Tool	9
Step 2: Creating a Test Bench	15
Step 3: Creating a Custom Constraints File.....	16
Step 4: Compiling the Design for Hardware Co-Simulation	19
Step 5: Running ISim Hardware Co-Simulation	22
Appendix Additional Resources	27

Introduction

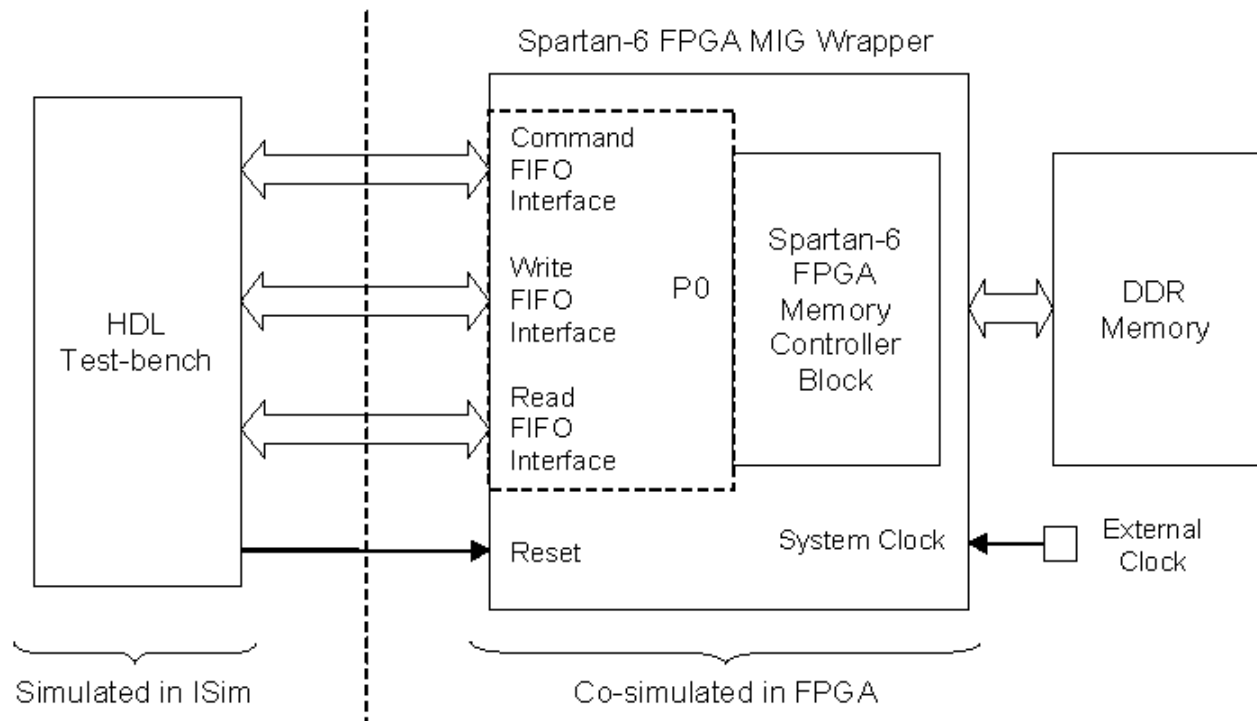
This tutorial describes how to use ISim hardware co-simulation to interact with the Spartan®-6 FPGA memory controller block (MCB) and drive external DDR2 memory from your HDL test bench at run time.

External memory is commonly used in embedded, image, and video processing applications that require a large amount of memory.

When developing an FPGA design that uses external memory, it is often challenging to verify the whole design including the memory controller and external memory module. Traditionally, we either simulate the whole design in software, or run the whole design in hardware. A full software simulation approach is useful in two aspects. It offers full visibility into the design and allows the test bench or design to be changed and re-verified in a rapid manner. The challenges, however, are getting an accurate simulation model of the external memory module and achieving a reasonable simulation speed. In contrast, running the design in hardware addresses these problems, but at the cost of reduced visibility into the design. It is also very complex to set up and change the test bench in hardware.

ISim hardware co-simulation is a third option in your toolbox. It gives you the flexibility to run a portion of your design in hardware while simulating the rest in software. The memory controller and external memory are, for example, good candidates to put in hardware so that they are modeled accurately and simulated quickly. The test bench and the application logic in your design, which are under development, should be simulated in software so you can change, verify and debug them easily and rapidly. The following figure shows how a design that uses a memory controller and external memory, can be partitioned to leverage the ISim hardware co-simulation features.

Partitioning a Memory Design for ISim Hardware Co-Simulation



Prerequisites

This tutorial requires the following software and hardware:

- Xilinx® ISE® Design Suite, version 13.2
- Spartan-6 FPGA SP601 Evaluation Kit
- Design File: [rdf0126_ddr2_mem_tutorial.zip](#)

Tutorial Files

File	Description
mig_dut.v	Wrapper that instantiates the MIG core and ties c3_p0_cmd_clk, c3_p0_wr_clk and c3_p0_rd_clk of the MIG core to a single input clock c3_clk0.
mig_hw_tb.v	Top-level test bench provides Verilog tasks that issues read and write transactions to the MCB.
mig_dut_hwcosim.ucf	Custom constraints file for hardware co-simulation that indicates which ports on the mig_dut module to be mapped to external I/Os and which ports are controlled from the test bench.
init.tcl	Custom simulation command file that tells ISim to load testmem.tcl and initialize the simulation.
testmem.tcl	Provides a testmem Tcl command that can be used in ISim console to invoke the test_memory Verilog task in the test bench.
mig_hw_tb.wcfg	Custom waveform configuration file.
mig_hw_tb.prj	ISim project file for the command line flow.

File	Description
hwcosim.bsp	Modified hardware co-simulation board support to use the 27MHz user clock, instead of the 200MHz differential clock, on the SP601 board as the hardware co-simulation interface clock.
full_compile.bat	Windows batch file to fully compile the design for hardware co-simulation with the Fuse command line.
full_compile.sh	Linux shell script to fully compile the design for hardware co-simulation with the Fuse command line.
incr_compile.bat	Windows batch file to incrementally compile the test bench for hardware co-simulation with the Fuse command line.
incr_compile.sh	Linux shell script to incrementally compile the test bench for hardware co-simulation with the Fuse command line.
run_isim.bat	Windows batch file to launch the ISim simulation.
run_isim.sh	Linux shell script to launch the ISim simulation.

Note Please note that when performing this tutorial, all data files must be copied to your current working directory.

Tutorial

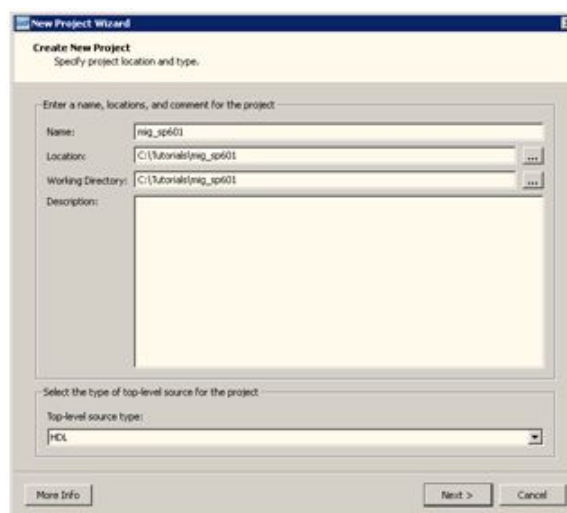
The following section describes the five steps for running a memory design through ISim hardware co-simulation.

1. Generate a Spartan®-6 memory reference design using the Memory Interface Generator (MIG) tool in CORE Generator™.
2. Create a test bench to exercise the memory reference design.
3. Create a custom constraints file to specify which ports on the design are controlled by ISim and which are mapped to external I/Os.
4. Compile the test bench for ISim simulation with the design targeted for hardware co-simulation.
5. Connect the target FPGA board to your computer and run the ISim simulation.

Step 1: Generating a Design Using the MIG Tool

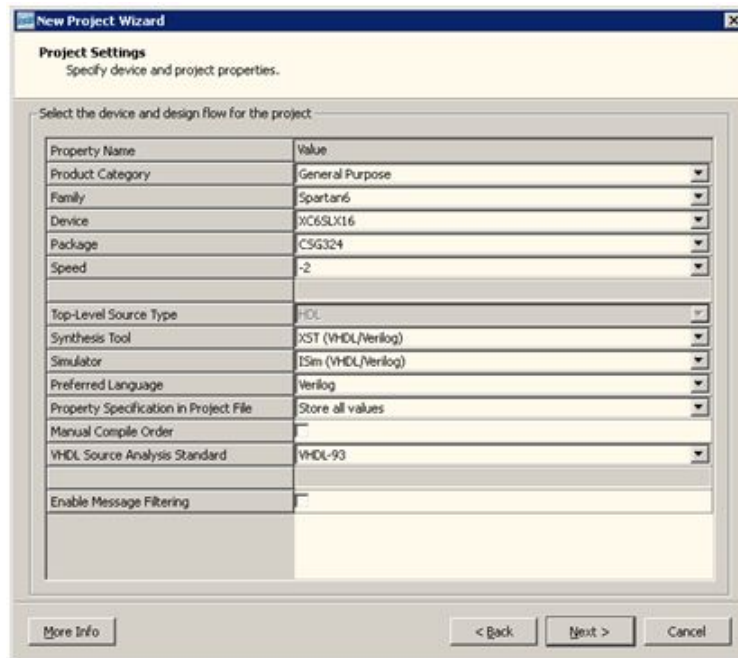
The Spartan-6 FPGA has an embedded multi-port memory controller block (MCB), which provides a simple and reliable way to interface with external DDR memory. The Memory Interface Generator (MIG) tool in CORE Generator simplifies the design process for interfacing with the MCB. In this tutorial, we are going to use the reference design generated by the MIG tool and create an ISim hardware co-simulation test bench that runs on the Spartan-6 FPGA SP601 Evaluation Kit.

1. Launch ISE® Project Navigator.
2. Choose **File > New Project** to open the New Project wizard. Enter a project name (**mig_sp601**) and location. Click **Next**.

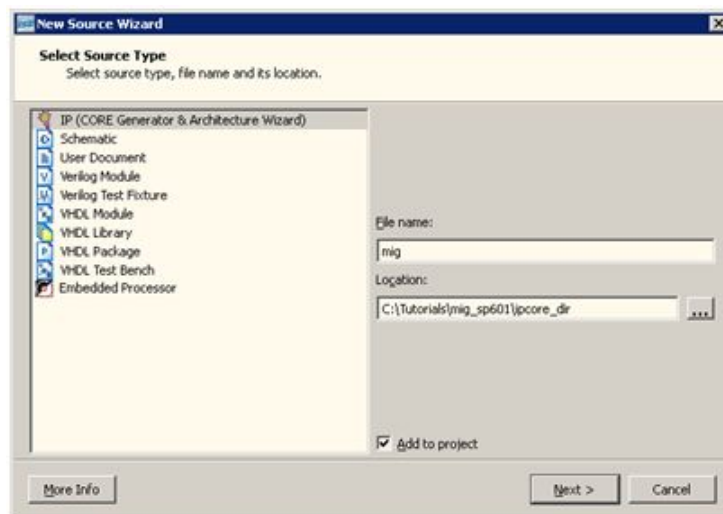


3. On the **Project Settings** page, choose the part for the **SP601 board**, which is Spartan-6 device **XC6SLX16**, package **CSG324**, and **speed -2**. Select **ISim** as the

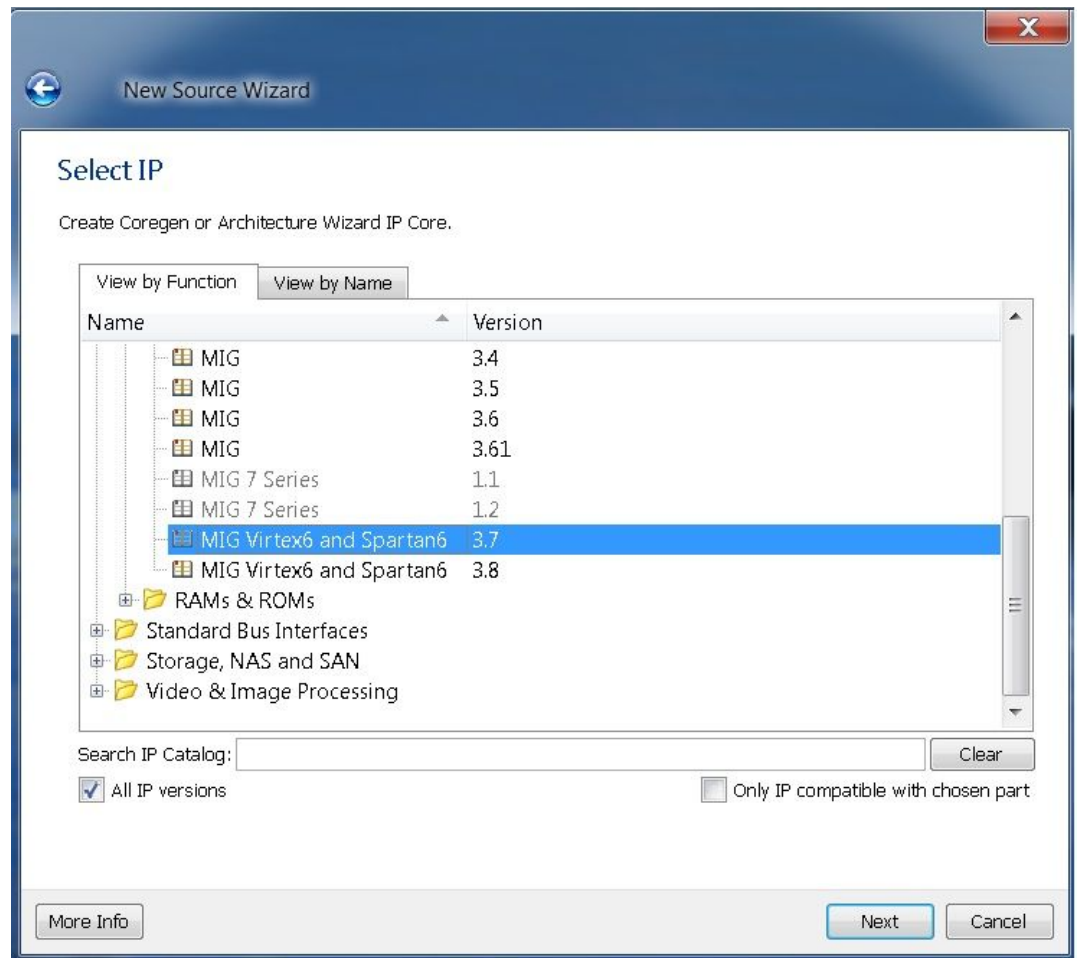
Simulator and **Verilog** as the **Preferred Language**. Click **Next** and then **Finish** to complete the project creation.



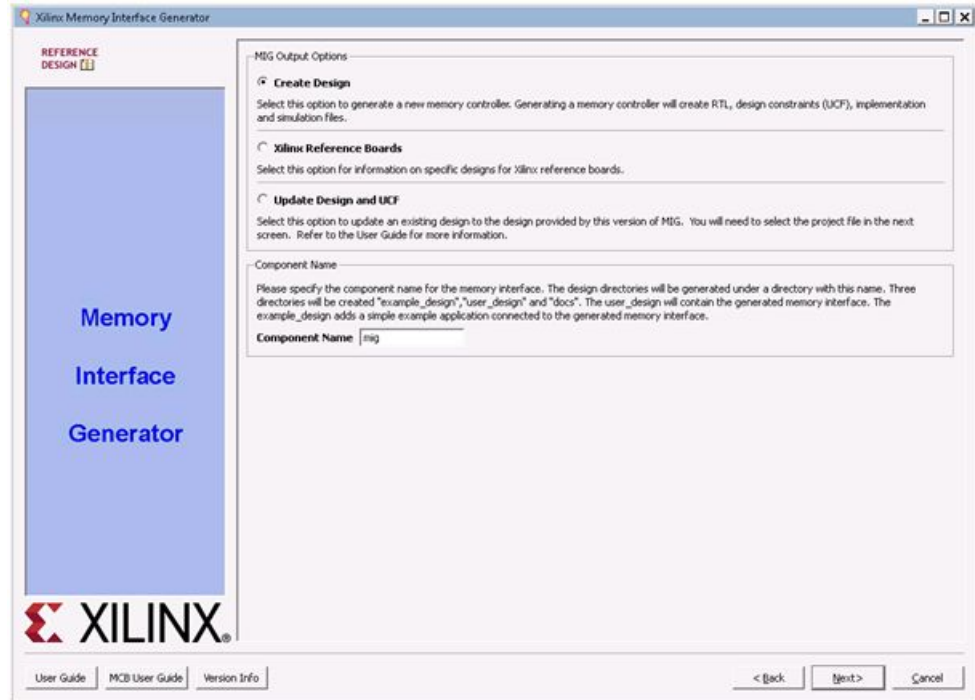
4. Choose **Project > New Source** to open the New Source Wizard. Select **IP (CORE Generator & Architecture Wizard)** and name the IP `mig`. Click **Next**.



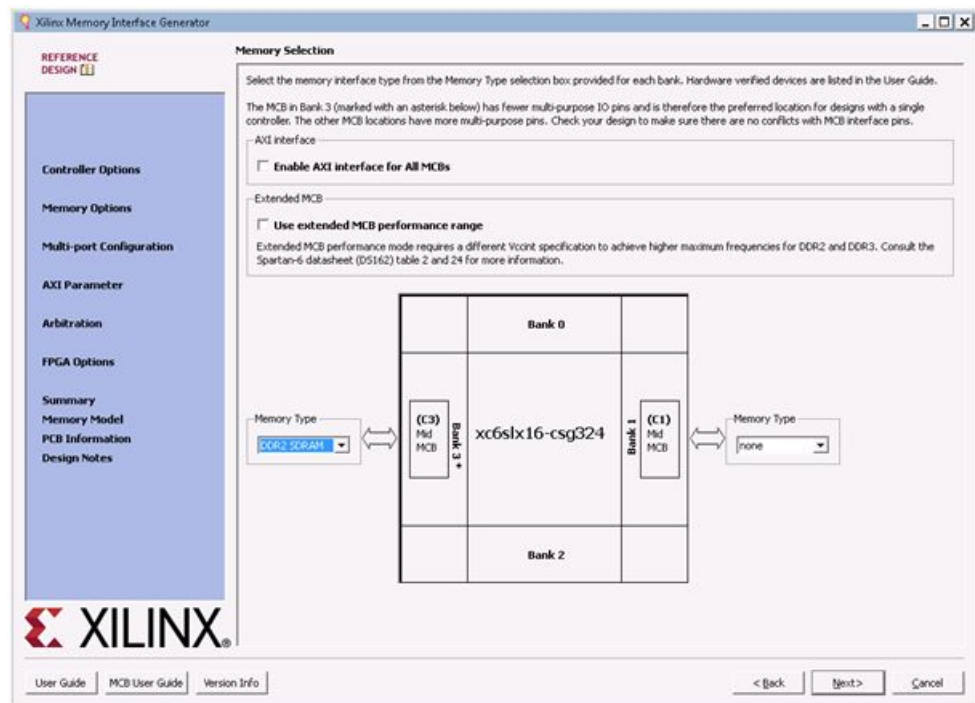
5. Select **MIG version 3.7** from the IP list. Click **Next** and then **Finish**.



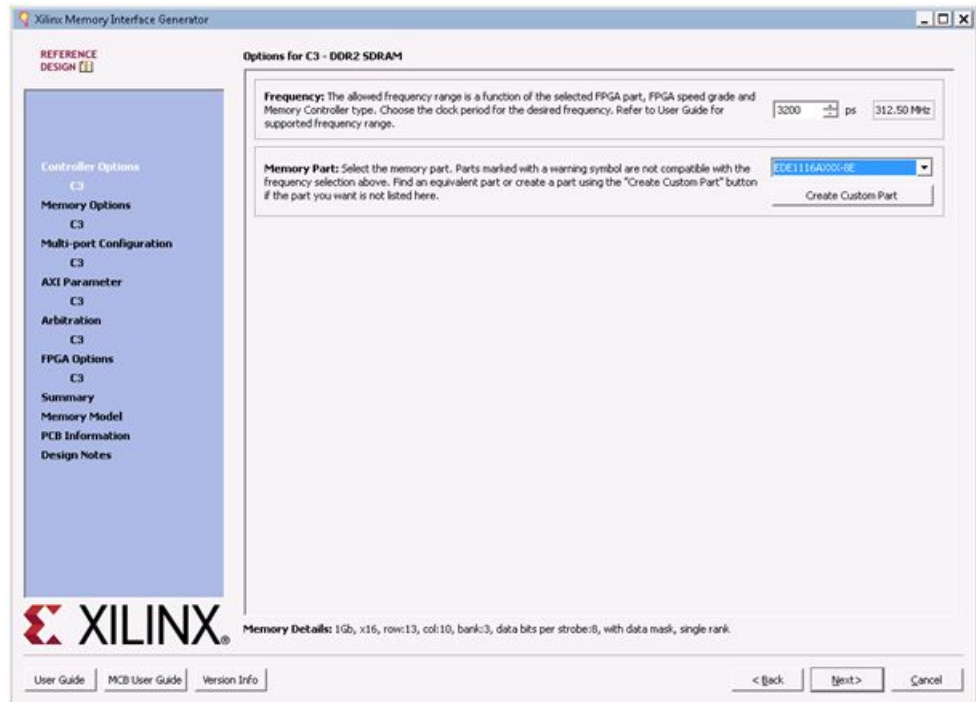
6. When the MIG GUI is launched, select **Create Design** to create a new MCB based memory interface. Type `mig` in the **Component Name** field. Click **Next**.



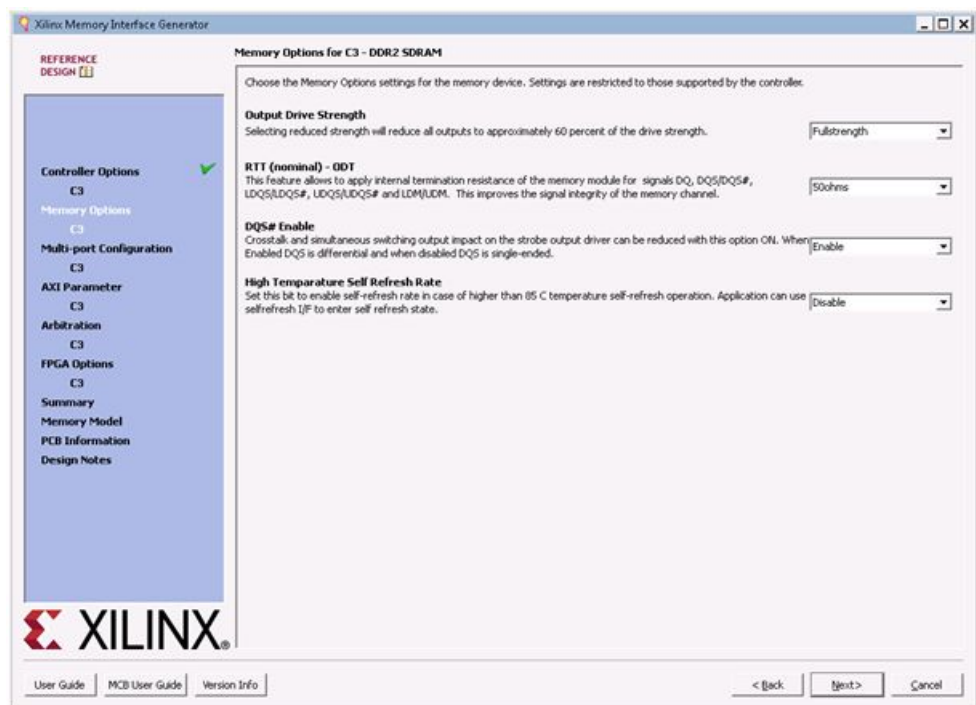
7. Select **DDR2 SDRAM** as the Memory Type for the MCB (C3) on bank 3. Click **Next**.



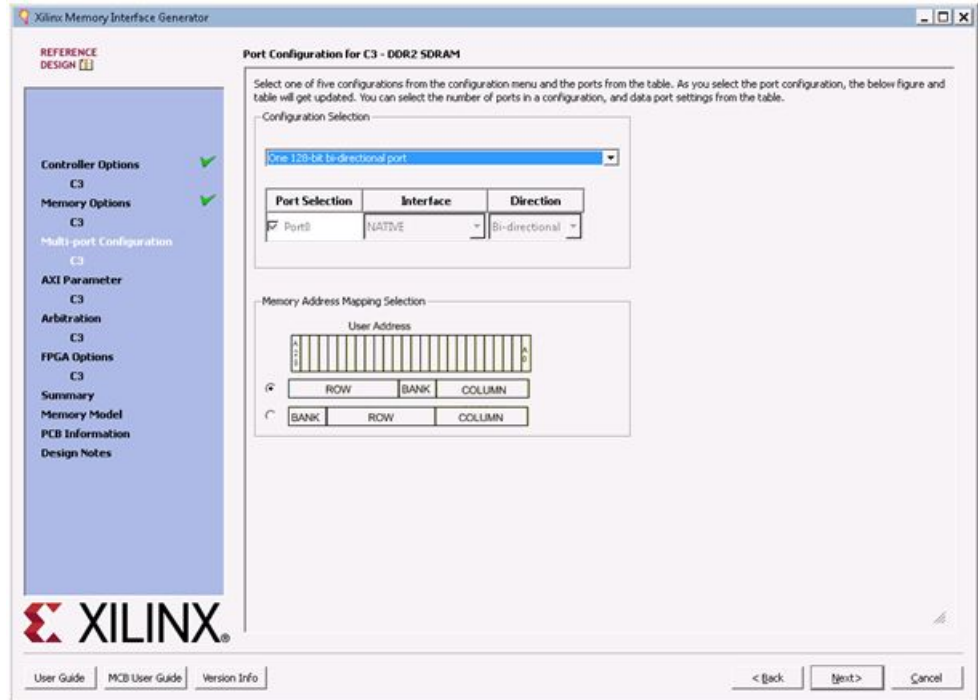
8. Select **3200ps (312.50MHz)** for the frequency. Select **EDE1116AXXX-8E** as the Memory Part. Click **Next**.



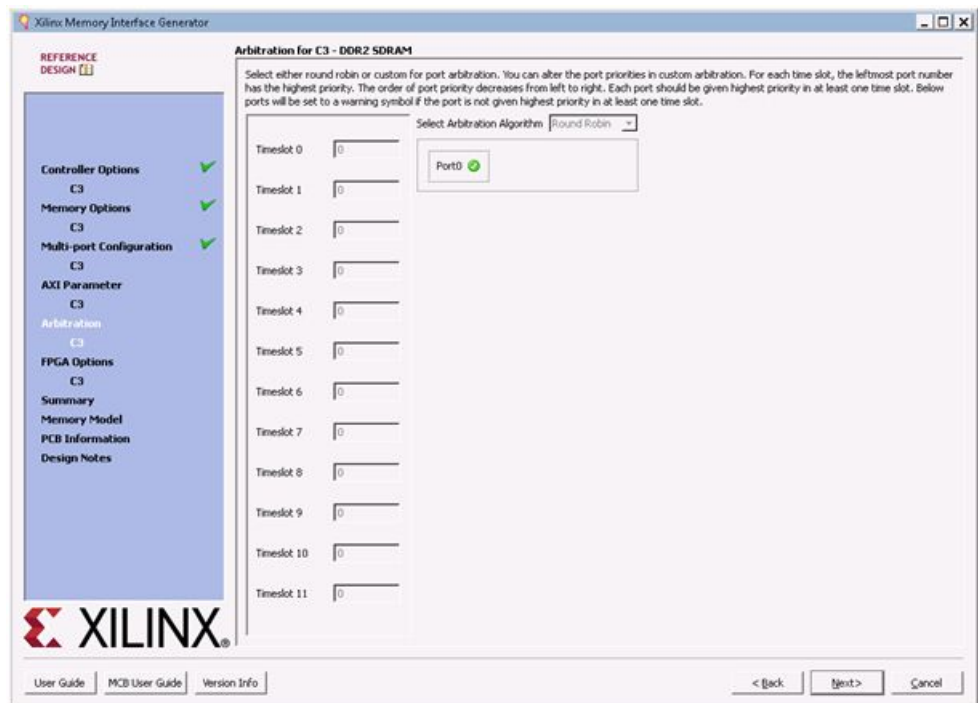
9. Use the default settings on the Memory Options page. Click Next.



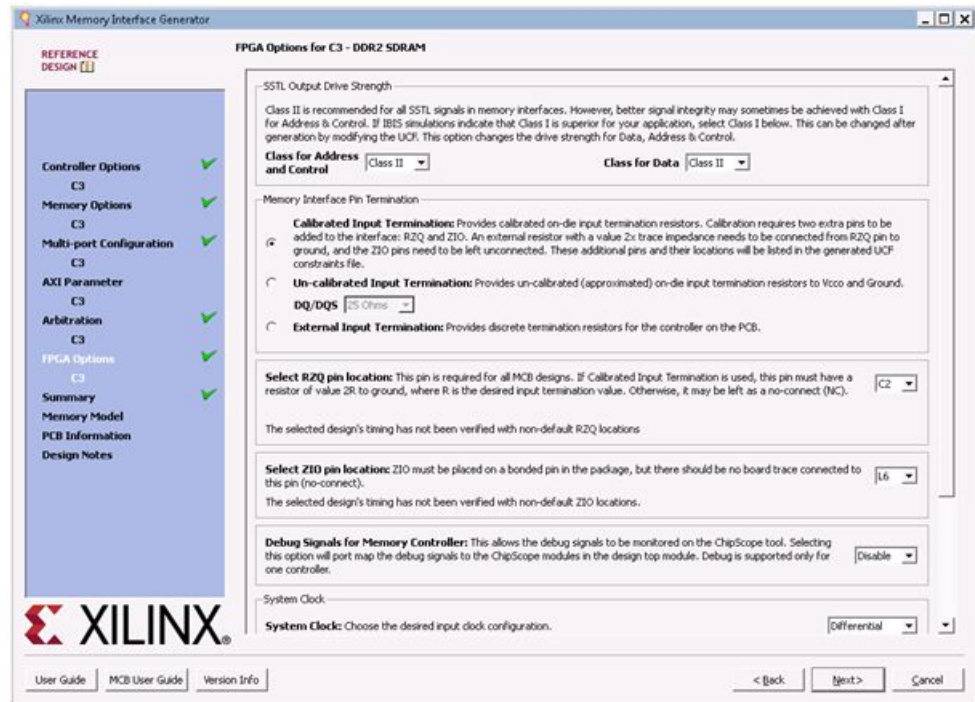
10. Select One 128-bit bi-directional port as the Port Configuration. Click Next.



- Use the default settings on the Arbitration page. Click Next.



- On the FPGA Options page, set RZQ pin location to C2 and ZIO pin location to L6. Click Next.



13. Click **Next** to go through the rest of the pages, and click **Generate** on the last page to generate the MIG core.
14. After the MIG core is generated, modify the PLL settings in the generated `ipcore_dir/mig/users_design/rtl/mig.v` in order to use the 200 MHz differential clock on the SP601 board as the system clock to generate the 625 MHz clock for the MIG core. Use the following parameter values in the `mig.v`.


```
localparam C3_CLKFBOUT_MULT = 25; // 200 * (25/8) = 625 MHz
localparam C3_DIVCLK_DIVIDE = 8;
```
15. Add a Verilog module `mig_dut.v` that instantiates the MIG core and ties the clocks (`c3_p0_cmd_clk`, `c3_p0_rd_clk`, `c3_p0_wr_clk`) for command, read, and write FIFOs to a single clock (`c3_clk0`). You can use the completed `mig_dut.v` file provided in this tutorial.

Step 2: Creating a Test Bench

Add a Verilog Test Bench module `mig_hw_tb.v` that drives the `mig_dut` instance. You can use the completed `mig_hw_tb.v` file provided in this tutorial.

This tutorial provides a test bench (`mig_hw_tb.v`) and a Tcl command (`testmem.tcl`) to allow interactions with the MCB and external memory through the ISim Tcl console.

The test bench contains two arrays, `input_data` and `output_data`, for buffering the data to be written into and read from the external memory. The MCB supports a write/read burst up to 64 128-bit words. In the previous steps, you configured the MCB to expose one 128-bit bi-directional port. Therefore, you set the size of `input_data` and `output_data` to 64x128-bits each. The test bench also defines a `test_parameters` module (instantiated as `params`), which holds parameters for the test bench. The usage of those parameters is described below.

The following Verilog tasks are defined in the test bench:

- `clear_input_output_data` - Fills the `input_data` and `output_data` array with zeros.
- `compare_input_output_data(input nwords)` - Compares `nwords` words of data in the `input_data` and `output_data` array, and reports any mismatches found.
- `use_walking_pattern(input b)` Fills the `input_data` array with walking zeros pattern if `b = 0` or with walking ones pattern if `b = 1`.
- `write_data(input start_addr, input burst_size)` - Writes `burst_size` words of data from the `input_data` array to the external memory starting at address `start_addr`. It first pushes data to the write FIFO interface (`c3_p0_wr_*`) on the MCB and then pushes a write command to the command FIFO interface (`c3_p0_cmd_*`).
- `read_data(input start_addr, input burst_size)` - Reads `burst_size` words of data from the external memory starting at address `start_addr` into the `output_data` array. It first pushes a read command to the command FIFO interface (`c3_p0_cmd_*`) on the PCB and then pulls data from the read FIFO interface (`c3_p0_rd_*`).
- `test_memory` - Writes data from the `input_data` array into the external memory of a specified region and then read the data back from the same region to the `output_data` array. The memory region is specified by `params.StartAddress` and `params.EndAddress`. The data pattern used to fill the `input_data` array is specified by `params.DataPattern` (0 – use the current data in `input_data`, 1 – use walking zeros, 2 – use walking ones).

The `testmem Tcl` command sets the value of `StartAddress`, `EndAddress`, and `DataPattern` in the `params` module. It then toggles the `run_test_trigger` signal in the test bench. Upon a rising edge of the `run_test_trigger` signal, the `test_read_write` task is called to exercise the write and read transaction on the external memory and check to make sure the data are written correctly to the external memory by comparing against the readback data.

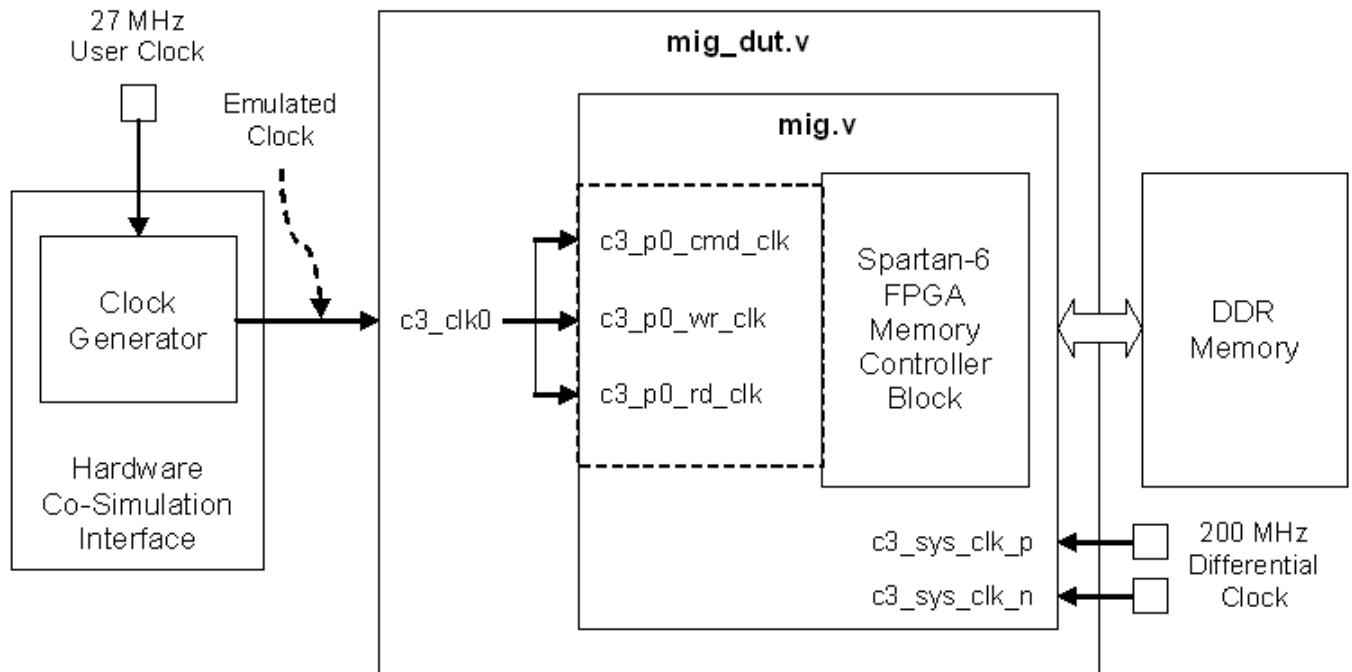
Step 3: Creating a Custom Constraints File

Partitioning the Design into Lock-step and Free-running Portion

The key concept of this tutorial is to partition the design into two portions:

- A free-running portion that interfaces with the external memory through the Spartan-6 MCB. It connects to external I/Os and clocks, and runs at full memory clock speed.
- A lock-step portion that is driven by the HDL test bench through ISim. It is synchronized to the ISim simulation, and receives stimuli and clock events virtually over the hardware co-simulation interface. As a result, it runs at a much lower speed.

The following figure shows how the MIG design is clocked under hardware co-simulation. The hardware co-simulation interface is inserted automatically during the compilation. It generates an emulated clock based on the 27 MHz user clock on the SP601 board. The emulated clock corresponds to the clock event on the `c3_clk0` signal in the test bench and drives the `c3_clk0` port of `mig_dut` running in hardware. The system clock for the MIG core is derived from the 200 MHz differential clock on the SP601 board.



Mapping Ports to External I/Os and Clocks

You can provide a custom constraints file, in Xilinx UCF format, to instruct the ISim compiler about which ports of the instance under hardware co-simulation are to be mapped to FPGA IOBs, and which ports are controlled by the HDL test bench. The ISim compiler looks for LOC constraints in the provided UCF file. A port with a LOC constraint is mapped to the corresponding FPGA IOB. A port without a LOC constraint is mapped to the hardware co-simulation interface and is accessible from the HDL test bench.

The partitioning of a design into a free-running portion and a lock-step portion happens implicitly based on how clock ports are mapped. If a clock port is mapped to an FPGA IOB through a LOC constraint, the logic driven by this clock belongs to the free-running portion. If a clock port has no LOC constraint assigned, the hardware co-simulation interface toggles the value on this port when a corresponding clock event occurs in the test bench. The logic driven by this clock, therefore, belongs to the lock-step portion.

Since the free-running and lock-step portion run at different speeds with separate clocks, the design should handle clock domain crossing between the two portions. The ISim hardware co-simulation compilation does not modify the internal of the design, and so it assumes the design can handle the speed difference and synchronization between the two portions.

The following table lists the ports on the `mig_dut` module that are mapped to external I/Os, and the ports are controlled by the test bench.

Partitioning ports on the mig_dut module

Ports mapped to external I/Os	Ports controlled by the test bench
c3_sys_clk_p	c3_sys_rst_n
c3_sys_clk_n	c3_clk0
mcb3_dram_dq	c3_rst0
mcb3_dram_a	c3_calib_done
mcb3_dram_ba	c3_p0_cmd_en
mcb3_dram_ras_n	c3_p0_cmd_instr
mcb3_dram_cas_n	c3_p0_cmd_bl
mcb3_dram_we_n	c3_p0_cmd_byte_addr
mcb3_dram_odt	c3_p0_cmd_empty
mcb3_dram_cke	c3_p0_cmd_full
mcb3_dram_ck	c3_p0_wr_en
mcb3_dram_ck_n	c3_p0_wr_mask
mcb3_dram_dqs	c3_p0_wr_data
mcb3_dram_dqs_n	c3_p0_wr_full
mcb3_dram_udqs	c3_p0_wr_empty
mcb3_dram_udqs_n	c3_p0_wr_count
mcb3_dram_udm	c3_p0_wr_underrun
mcb3_dram_dm	c3_p0_wr_error
rzq3_zio3	c3_p0_rd_en
	c3_p0_rd_data
	c3_p0_rd_full
	c3_p0_rd_empty
	c3_p0_rd_count
	c3_p0_rd_overflow
	c3_p0_rd_error

The MIG tool creates an example UCF file. The following procedure describes using the UCF file as a template to create the custom constraints file for hardware co-simulation.

1. Copy `ipcore_dir/mig/user_design/par/mig.ucf` to the ISim project directory where `mig_dut.v` is located. Name the copied file as `mig_dut_hwcosim.ucf`.
2. Modify the `mig_dut_hwcosim.ucf` file as follows for the SP601 board. Change the period constraint of `TS_SYS_CLK3` to 5 ns as we use the 200 MHz differential clock input on the SP601 as the system clock.

```
TIMESPEC "TS_SYS_CLK3" = PERIOD "SYS_CLK3" 5 ns HIGH 50 %;
```

Change the LOC constraint for `c3_sys_clk_n` to K16, and `c3_sys_clk_p` to K15 to match the pin assignments on SP601.

```
NET "c3_sys_clk_n" LOC = "K16";
NET "c3_sys_clk_p" LOC = "K15";
```

3. Modify the `mig_dut_hwcosim.ucf` file for ISim hardware co-simulation requirements.

Add a wildcard character `*` at the beginning of the hierarchical path for the following constraints. This is required because the `mig_dut` will be wrapped as a submodule when it is compiled for hardware co-simulation.

```
NET "*memc?_wrapper_inst/mcb_ui_top_inst/mcb_raw_wrapper_inst/selfrefresh_mcb_mode" TIG;
NET "*c?_pll_lock" TIG;
NET "*memc?_wrapper_inst/mcb_ui_top_inst/mcb_raw_wrapper_inst/gen_term_calib.mcb_soft_calibration_top_inst/
mcb_soft_calibration_inst/CKE_Train" TIG; ##This path exists for DDR2
NET "*memc3_infrastructure_inst/sys_clk_ibufg" TNM_NET = "SYS_CLK3";
```

Comment out the constraints for error, `calib_done`, and `c3_sys_rst_n`, especially the LOC constraints, as it will be controlled from the test bench.

```
#NET "error" IOSTANDARD = LVCMOS18 ;
```

```

#NET "calib_done"      IOSTANDARD = LVCMOS18 ;
#NET "calib_done"      LOC = "B2" ;
#NET "error"           LOC = "A2" ;

#NET "c3_sys_rst_n"    IOSTANDARD = LVCMOS18;
#NET "c3_sys_rst_n"    LOC = "M8" ;

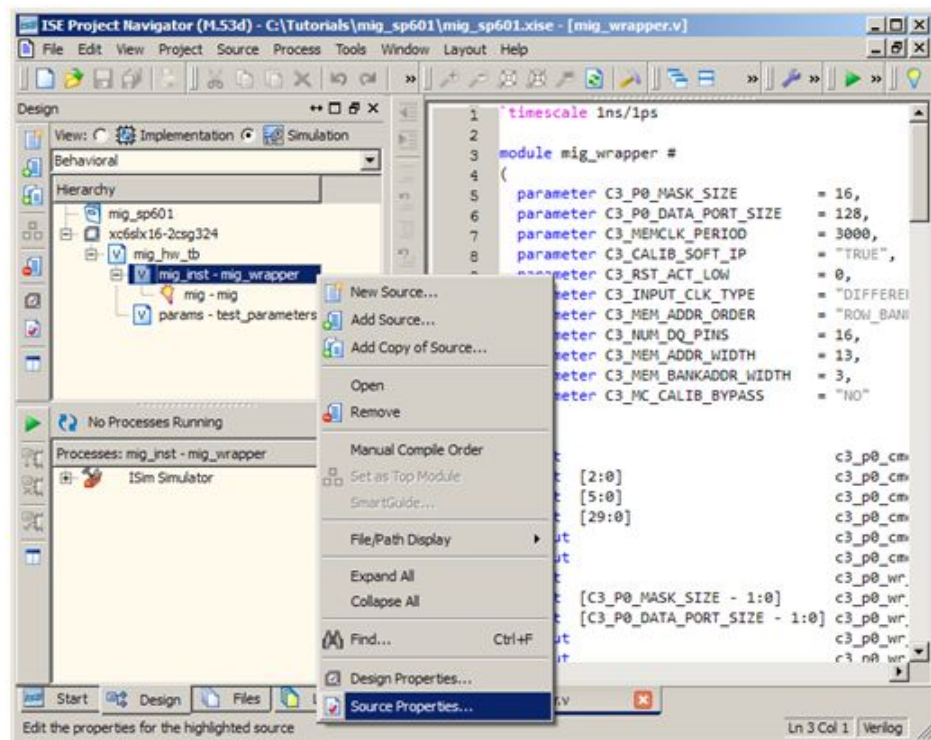
```

Note The `c3_clk0`, `c3_rst0`, `c3_calib_done`, and `c3_p0_*` ports on the `mig_dut` are not constrained because they are also controlled from the test bench.

Step 4: Compiling the Design for Hardware Co-Simulation

After you create the test bench and the custom constraints file, you can compile the design for hardware co-simulation using the ISim compiler. You can do this in Project Navigator by enabling hardware co-simulation on a selected instance in your design. The selected instance, including its submodules, will be co-simulated in hardware during the ISim simulation. Other modules will be simulated in software.

1. Switch to the **Simulation View** in Project Navigator. Right-click the `mig_inst - mig_wrapper` instance from the **Pane** of the Design panel and click **Source Properties**.



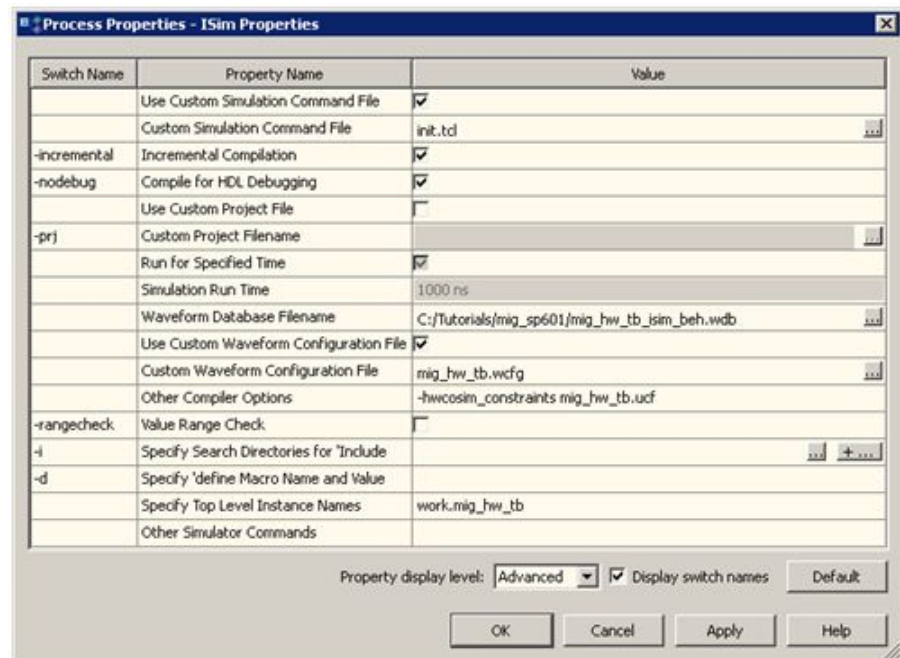
2. Select the **Hardware Co-Simulation** category. Check the **Enable Hardware Co-Simulation** check box. Set the Clock Port to `c3_clk0`. Select **SP601 (JTAG)** as the Target Board for Hardware Co-Simulation.

- Select the **Use Custom Waveform Configuration File**
- Set **Custom Waveform Configuration File** to `mig_hw_tb.wcfg`
- Set **Other Compiler Options** to `-hwcosim_constraints mig_dut_hwcosim.ucf`

The `init.tcl` script runs when the ISim simulation starts. It loads a Tcl command `testmem` from the `testmem.tcl` script, which is used later in this tutorial to run the simulation.

The `mig_hw_tc.wcfg` file provides a customized waveform configuration view for this tutorial.

Note The custom constraints file for hardware co-simulation is provided to the ISim compiler through the `hwcosim_constraints` switch. Specify this property in the **Other Compiler Options**.



- Run the **Simulate Behavioral Model** process for the `mig_hw_tb` instance.

Using the Fuse Command Line Tool

You can invoke the ISim compiler through the Fuse command line tool. As in the pure software simulation flow, you need to provide Fuse a project file, the design top level module(s), and other optional arguments such as libraries to link in and library search paths. To compile the design for hardware co-simulation, you must provide the extra arguments listed below:

```

fuse -prj <project file> <top level modules>
     -hwcosim_instance <instance>
     -hwcosim_clock <clock>
     -hwcosim_board <board>
     -hwcosim_constraints <constraint file>
     -hwcosim_incremental <0|1>

```

- `hwcosim_instance` specifies the full hierarchical path of the instance to co-simulate in hardware
- `hwcosim_clock` specifies the port name of the clock input for the instance. This is the clock in the lock-step portion, which is to be controlled by the test bench.

For a design with multiple clocks, specify the fastest clock using this option so that ISim can optimize the simulation. Other clock ports are treated as regular data ports.

- `hwcosim_board` specifies the identifier of the hardware board to use for co-simulation. Two Spartan-6 boards are supported by default:
 - `sp601-jtag`: Xilinx® SP601 Evaluation Platform
 - `sp605-jtag`: Xilinx SP605 Evaluation Platform
- `hwcosim_constraints` (optional) specifies the custom constraints file that provides additional constraints for implementing the instance for hardware co-simulation. We also use the constraints file to specify which ports of the instance are mapped to external I/Os or clocks.
- `hwcosim_incremental` (optional) specifies whether Fuse should reuse the last generated hardware co-simulation bitstream and skip the implementation flow.

For example, to compile the EMAC design for this tutorial, you can run the Fuse command line as follows:


```

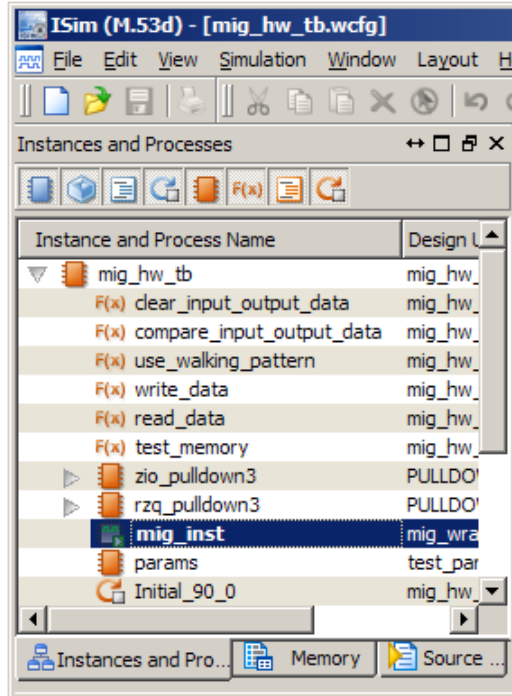
fuse -prj mig_hw_tb.prj mig_hw_tb glbl
     -L unisims_ver -L secureip
     -o mig_hw_tb.exe
     -hwcosim_instance /mig_hw_tb/mig_inst
     -hwcosim_clock c3_clk0
     -hwcosim_board sp601-jtag
     -hwcosim_constraints mig_dut_hwcosim.ucf

```

Step 5: Running ISim Hardware Co-Simulation

The simulation executable generated by the ISim compiler runs in the same way in both the pure software simulation and hardware co-simulation flow. Project Navigator automatically launches the simulation executable in GUI mode after the compilation finishes.

In the **Instances and Processes** view, the instance selected for hardware co-simulation is indicated with a special icon . As the instance runs in hardware, you cannot expand it to see its internal signals and submodules.

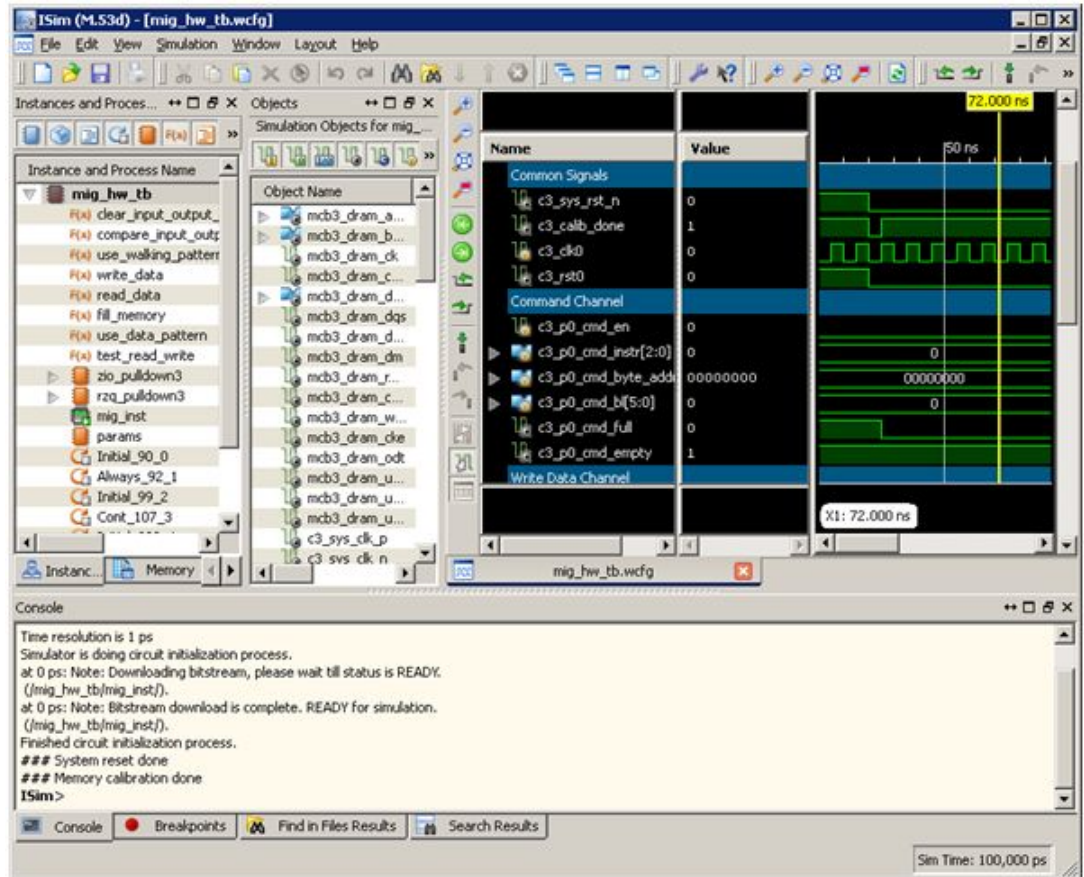


Before the simulation starts, ISim programs the FPGA with the bitstream file generated for hardware co-simulation. You might notice the message in the ISim console window: “Downloading bitstream, please wait till status is READY”. Once the FPGA is configured, the console shows “Bitstream download is complete. READY for simulation.” From this point, you can run the simulation and interact with the ISim GUI the same way you do in the software simulation flow.

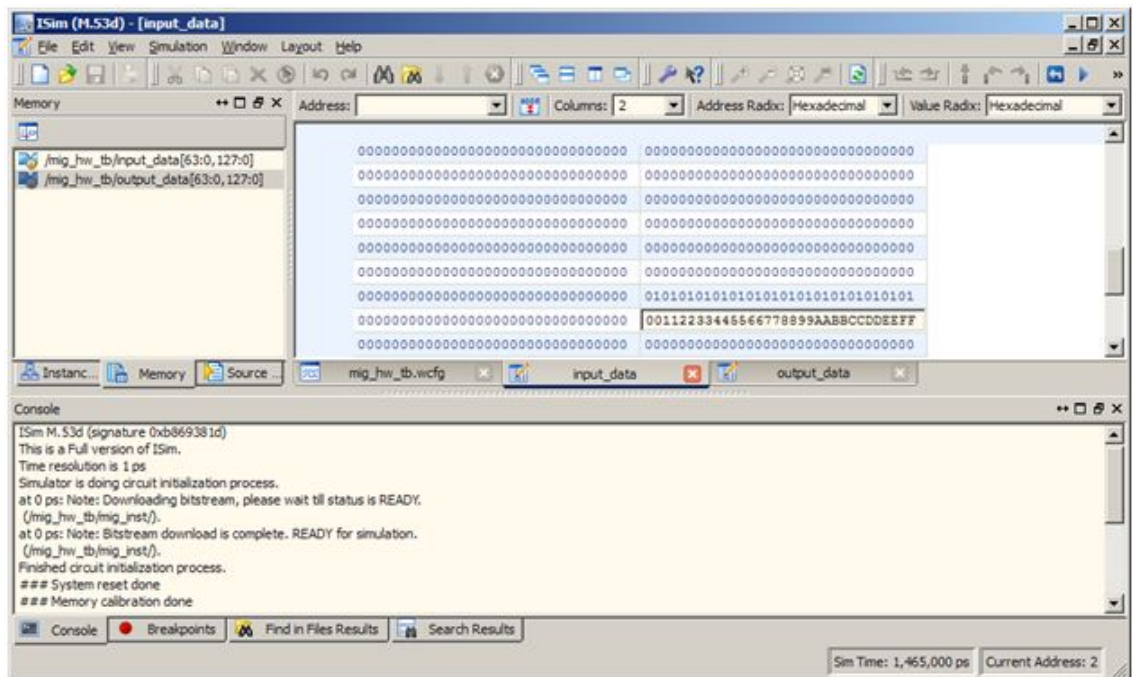
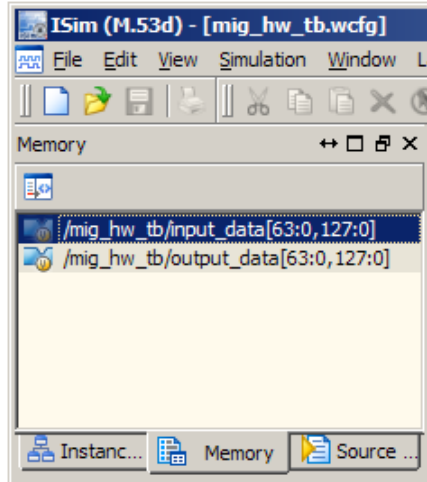
The test-bench initially resets the system by asserting the `c3_sys_rst_n` signal, which then triggers the memory calibration process. You should see the following messages in the ISim console:

```
### System reset done
### Memory calibration done
```

The `c3_calib_done` signal transitions from low to high after the reset is de-asserted. This is because the memory calibration process takes place in hardware at full speed. It takes a much longer time if the calibration process is simulated in software.



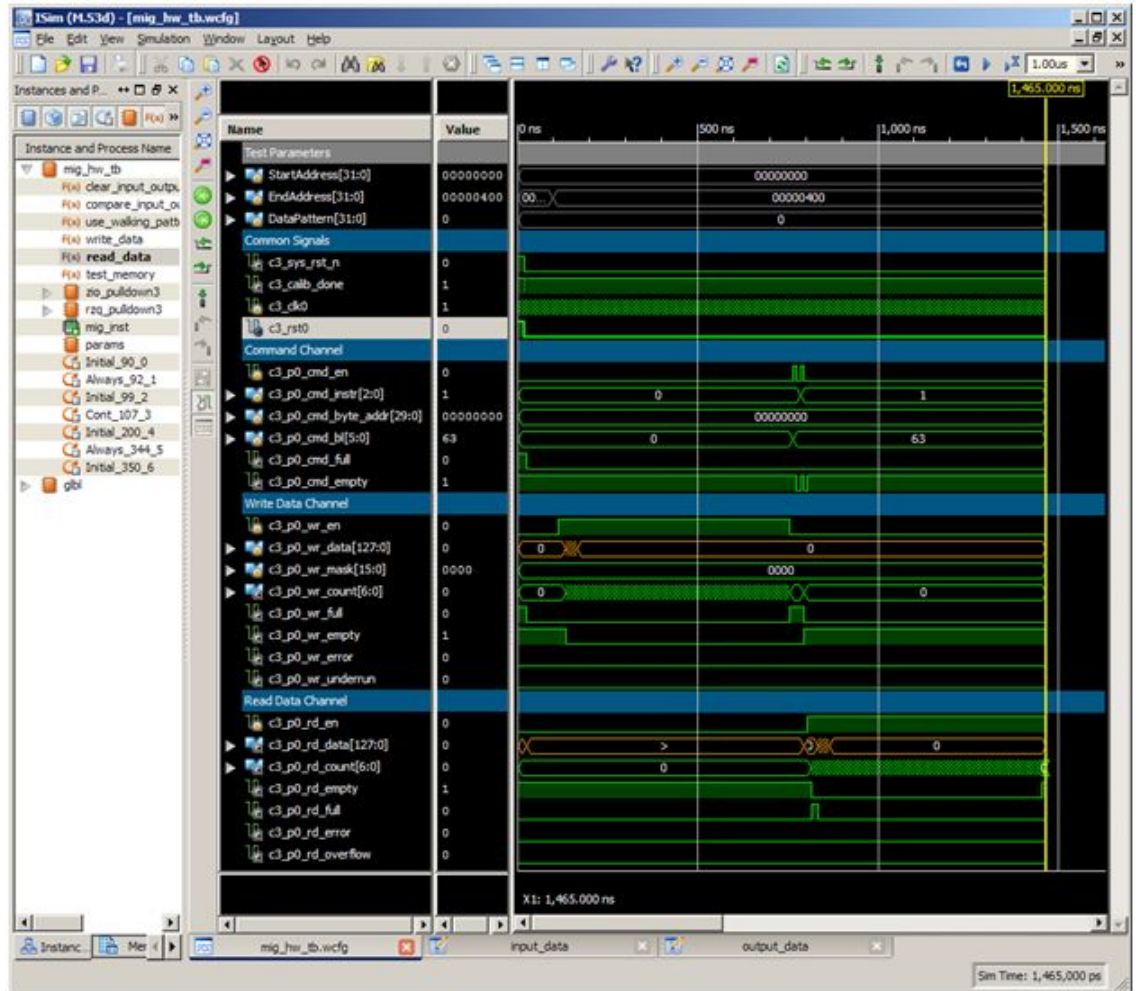
Select the **Memory** tab. Double-click on the `input_data` array to open the memory editor. Change the **Address Radix** and **Value Radix** to **Hexadecimal**. You can edit the content of the `input_data` and then use that to write to the external memory.



Run “testmem 0 1024 0” in the ISim console. This triggers the test_memory task in the test bench, which writes the data in input_data to the external memory starting from address 0 to 1024 and then reads the data back from the same memory region to output_data. You should see the following messages from the ISim console:

```
### Read/write test started
... Writing 00000000
... Reading 00000000
==> No data mismatches found.
### Read/write test done
```

Observe the waveform of c3_p0_cmd_*, c3_p0_wr_*, and c3_p0_rd_* to see how the test bench interacts with the MCB and how the MCB status signals (such as c3_p0_wr_count) changes during the memory write/read transaction.



Try different data patterns with various starting and ending addresses using the `testmem` command. For example:

- “`testmem 0 2048 1`” tests the memory region from address 0 to 2048 with a walking zeros pattern.
- “`testmem 1024 4096 2`” tests the memory region from address 1024 to 4096 with a walking ones pattern.

Additional Resources

- **Xilinx Glossary** - http://www.xilinx.com/support/documentation/sw_manuals/glossary.pdf
- **Xilinx Documentation** - <http://www.xilinx.com/support/documentation>
- **Xilinx Support** - <http://www.xilinx.com/support>