



DDR and DDR2 SDRAM High-Performance Controller User Guide



101 Innovation Drive
San Jose, CA 95134
www.altera.com

Software Version: 9.0
Document Date: March 2009

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

Chapter 1. About These MegaCore Functions

Release Information	1-1
Device Family Support	1-1
Features	1-2
General Description	1-2
MegaCore Verification	1-3
Performance and Resource Utilization	1-4
Installation and Licensing	1-7
OpenCore Plus Evaluation	1-7
OpenCore Plus Time-Out Behavior	1-8

Chapter 2. Getting Started

Design Flow	2-1
Select Flow	2-2
SOPC Builder Flow	2-2
Specify Parameters	2-2
Complete the SOPC Builder System	2-3
Simulate the System	2-4
MegaWizard Plug-In Manager Flow	2-4
Specify Parameters	2-5
Simulate the Example Design	2-8
Simulating Using NativeLink	2-8
IP Functional Simulations	2-9
Compile the Design	2-13
Program Device and Implement the Design	2-14

Chapter 3. Parameter Settings

Memory Settings	3-1
PHY Settings	3-1
Controller Settings	3-1

Chapter 4. Functional Description

Block Description	4-2
Command FIFO	4-3
Write Data FIFO	4-3
Write Data Tracking Logic	4-3
Main State Machine	4-3
Bank Management Logic	4-3
Timer Logic	4-3
Initialization State Machine	4-4
Address and Command Decode	4-4
PHY Interface Logic	4-4
ODT Generation Logic	4-4
Low Power Mode Logic	4-4
Control Logic	4-5
Latency	4-5
Error Correction Coding (ECC)	4-7
Interrupts	4-9

Partial Writes	4-9
Partial Bursts	4-10
ECC Latency	4-10
Example Design	4-11
Example Driver	4-12
Interfaces and Signals	4-14
Interface Description	4-14
Full Rate Write, Avalon-MM Interface Mode	4-14
Full Rate Write, Native Interface Mode—Non-Consecutive Write	4-17
Half Rate Write, Avalon-MM Interface Mode	4-20
Half Rate Write, Native Interface Mode	4-22
Full Rate Read, Avalon-MM Interface Mode	4-25
Half Rate Read, Native Interface Mode	4-27
Half Rate Read, Avalon-MM Interface Mode—Non-Consecutive Read	4-28
Full Rate, Native Interface Mode—Alternate Read-Write	4-31
User Refresh Control	4-34
Self-Refresh and Power-Down Commands	4-35
Auto-Precharge Commands	4-36
Signals	4-37
Chapter 5. Example Design Walkthrough	
Creating A Simulation Testbench Environment	5-1
Creating the Example Project	5-1
Configuring the DDR2 SDRAM High-Performance Controller	5-1
Understanding the Example Design and Testbench	5-2
Testbench Description	5-2
Running the Example Testbench from Your Simulator	5-3
The Testbench Stages	5-4
Memory Device Initialization	5-4
Functional Memory Use	5-7
Appendix A. ECC Register Description	
ECC Registers	A-1
Register Bits	A-3
Additional Information	
Revision History	Info-1
How to Contact Altera	Info-1
Typographic Conventions	Info-2

Release Information

Table 1–1 provides information about this release of the DDR and DDR2 SDRAM High-Performance Controller MegaCore® functions.

Table 1–1. DDR and DDR2 SDRAM High-Performance Controller Release Information

Item	Description
Version	9.0
Release Date	March 2009
Ordering Codes	IP-SDRAM/HPDDR (DDR SDRAM) IP-SDRAM/HPDDR2 (DDR2 SDRAM)
Product IDs	00BE (DDR SDRAM) 00BF (DDR2 SDRAM) 00CO (ALTMEMPHY Megafunction)
Vendor ID	6AF7

Altera® verifies that the current version of the Quartus® II software compiles the previous version of each MegaCore function. The *MegaCore IP Library Release Notes and Errata* report any exceptions to this verification. Altera does not verify compilation with MegaCore function versions older than one release.

Device Family Support

MegaCore functions provide either full or preliminary support for target Altera device families, as described below:

- *Full support* means the MegaCore function meets all functional and timing requirements for the device family and may be used in production designs
- *Preliminary support* means the MegaCore function meets all functional requirements, but may still be undergoing timing analysis for the device family; it may be used in production designs with caution

Table 1–2 shows the level of support offered by the DDR and DDR2 SDRAM high-performance controller to each of the Altera device families.

Table 1–2. Device Family Support (Part 1 of 2)

Device Family	Support
Arria® GX	Full
Arria II GX	Preliminary
Cyclone® III	Full
HardCopy® II	Full
HardCopy III	Preliminary
HardCopy IV E	Preliminary

Table 1-2. Device Family Support (Part 2 of 2)

Device Family	Support
Stratix® II	Full
Stratix II GX	Full
Stratix III	Full
Stratix IV	Preliminary
Other device families	No support

Features

- Integrated error correction coding (ECC) function
- Power-up calibrated on-chip termination (OCT) support for Cyclone III, Stratix III, and Stratix IV devices
- Full-rate and half-rate support
- SOPC Builder ready
- Support for ALTMEMPHY megafunction
- Support for industry-standard DDR and DDR2 SDRAM devices; and registered and unbuffered DIMMs
- Optional support for self-refresh and power-down commands
- Optional support for auto-precharge read and auto-precharge write commands
- Optional user-controller refresh
- Optional Avalon® Memory-Mapped (Avalon-MM) local interface
- Optional Altera PHY interface (AFI) Controller-PHY Interface
- Optional multiple controller clock sharing in SOPC Builder Flow
- Easy-to-use MegaWizard™ interface
- Support for OpenCore Plus evaluation
- Support for the Quartus II IP Advisor
- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators

General Description

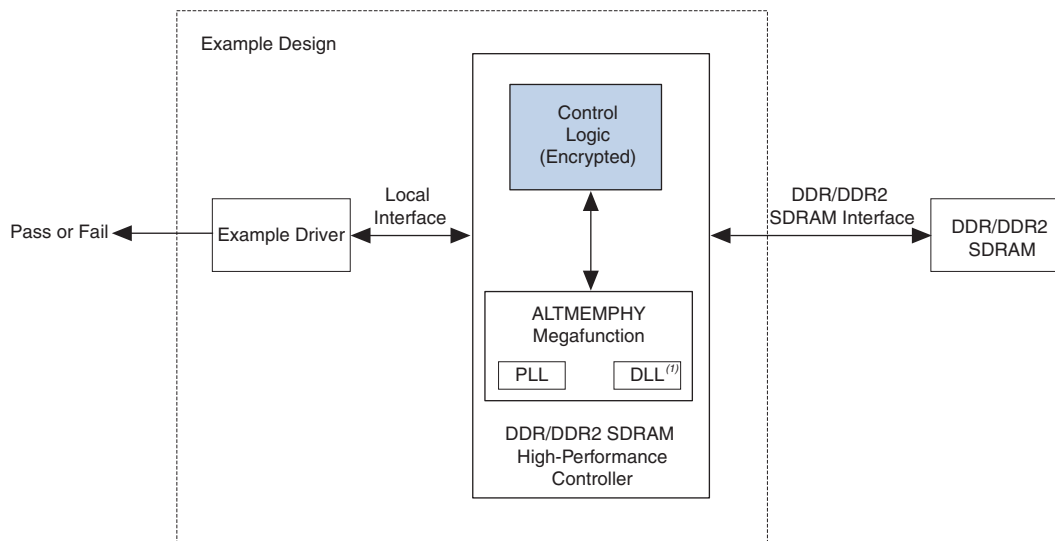
The Altera DDR and DDR2 SDRAM High-Performance Controller MegaCore functions provide simplified interfaces to industry-standard DDR SDRAM and DDR2 SDRAM. The MegaCore functions work in conjunction with the Altera ALTMEMPHY megafunction.



For more information on the ALTMEMPHY megafunction, refer to the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)*.

Figure 1-1 on page 1-3 shows a system-level diagram including the example design that the DDR or DDR2 SDRAM High-Performance Controller MegaCore functions create for you.

Figure 1-1. System-Level Diagram



Note for Figure 1-1:

(1) When you choose **Instantiate DLL Externally**, DLL is instantiated outside the controller.

The MegaWizard Plug-In Manager generates an example design, consisting of an example driver, and your DDR or DDR2 SDRAM high-performance controller custom variation. The controller instantiates an instance of the ALTMEMPHY megafunction which in turn instantiates a PLL and DLL. You can optionally instantiate the DLL outside the ALTMEMPHY megafunction to share the DLL between multiple instances of the ALTMEMPHY megafunction.

The example design is a fully-functional design that you can simulate, synthesize, and use in hardware. The example driver is a self-test module that issues read and write commands to the controller and checks the read data to produce the pass/fail and test complete signals.

MegaCore Verification

MegaCore verification involves simulation testing. Altera has carried out extensive random, directed tests with functional test coverage using industry-standard Denali models to ensure the functionality of the DDR and DDR2 SDRAM high-performance controller. In addition, Altera performs a wide variety of gate-level tests of the DDR and DDR2 SDRAM high-performance controllers to verify the post-compilation functionality of the controllers.

Performance and Resource Utilization

Table 1-3 shows maximum performance results for the DDR and DDR2 SDRAM high-performance controllers using the Quartus II software, version 9.0 with Arria GX, Cyclone III, HardCopy II, Stratix II, Stratix II GX, Stratix III, and Stratix IV devices.

Table 1-3. Maximum Performance for Half Rate and Full Rate Controllers

Device	System f_{MAX} (MHz)			
	DDR SDRAM		DDR2 SDRAM	
	Half Rate	Full Rate	Half Rate	Full Rate
Arria GX	200	167	233	167
Cyclone III	167	167	200	167
HardCopy II	200	200	267	267
Stratix II	200	200	333	267
Stratix II GX	200	200	333	267
Stratix III	200	200	400	267
Stratix IV	200	200	400	267

 For more information on device performance, refer to the relevant device handbook.

Table 1-4 shows typical sizes for the DDR or DDR2 SDRAM high-performance controller in AFI mode (including ALTMEMPHY) for Arria GX devices.

Table 1-4. Resource Utilization in Arria GX Devices

Controller Rate	Local Data Width (Bits)	Memory Width (Bits)	Combinational ALUTs	Dedicated Logic Registers	Memory	
					M512	M4K
Half	32	8	1,851	1,562	4	2
	64	16	1,904	1,738	4	4
	256	64	2,208	2,783	5	15
	288	72	2,289	2,958	4	17
Full	32	8	1,662	1,332	6	0
	64	16	1,666	1,421	3	3
	256	64	1,738	1,939	3	9
	288	72	1,758	2,026	4	9

Table 1-5 shows typical sizes for the DDR or DDR2 SDRAM high-performance controller in AFI mode (including ALTMEMPHY) for Cyclone III devices.

Table 1-5. Resource Utilization in Cyclone III Devices

Controller Rate	Local Data Width (Bits)	Memory Width (Bits)	Combinational ALUTs	Dedicated Logic Registers	Memory (M9K)
Half	32	8	2,683	1,563	3
	64	16	2,905	1,760	5
	256	64	4,224	2,938	17
	288	72	4,478	3,135	18
Full	32	8	2,386	1,276	3
	64	16	2,526	1,387	3
	256	64	3,257	2,037	9
	288	72	3,385	2,146	10

Table 1-6 shows typical sizes for the DDR or DDR2 SDRAM high-performance controller in AFI mode (including ALTMEMPHY) for Stratix II and Stratix II GX devices.

Table 1-6. Resource Utilization in Stratix II and Stratix II GX Devices

Controller Rate	Local Data Width (Bits)	Memory Width (Bits)	Combinational ALUTs	Dedicated Logic Registers	Memory	
					M512	M4K
Half	32	8	1,853	1,581	4	2
	64	16	1,901	1,757	4	4
	256	64	2,206	2,802	5	15
	288	72	2,281	2,978	4	17
Full	32	8	1,675	1,371	6	0
	64	16	1,675	1,456	3	3
	256	64	1,740	1,976	3	9
	288	72	1,743	2,062	4	9

Table 1-7 shows typical sizes for the DDR or DDR2 SDRAM high-performance controller in AFI mode (including ALTMEMPHY) for Stratix III devices.

Table 1-7. Resource Utilization in Stratix III Devices

Controller Rate	Local Data Width (Bits)	Memory Width (Bits)	Combinational ALUTs	Dedicated Logic Registers	Memory (M9K)
Half	32	8	1,752	1,432	2
	64	16	1,824	1,581	3
	256	64	2,210	2,465	9
	288	72	2,321	2,613	10
Full	32	8	1,622	1,351	2
	64	16	1,630	1,431	2
	256	64	1,736	1,897	5
	288	72	1,749	1,975	6

Table 1-8 shows typical sizes for the DDR or DDR2 SDRAM high-performance controller in AFI mode (including ALTMEMPHY) for Stratix IV devices.

Table 1-8. Resource Utilization in Stratix IV Devices

Controller Rate	Local Data Width (Bits)	Memory Width (Bits)	Combinational ALUTs	Dedicated Logic Registers	Memory (M9K)
Half	32	8	1,755	1,452	1
	64	16	1,820	1,597	2
	256	64	2,202	2,457	8
	288	72	2,289	2,601	9
Full	32	8	1,631	1,369	1
	64	16	1,630	1,448	1
	256	64	1,731	1,906	4
	288	72	1,743	1,983	5

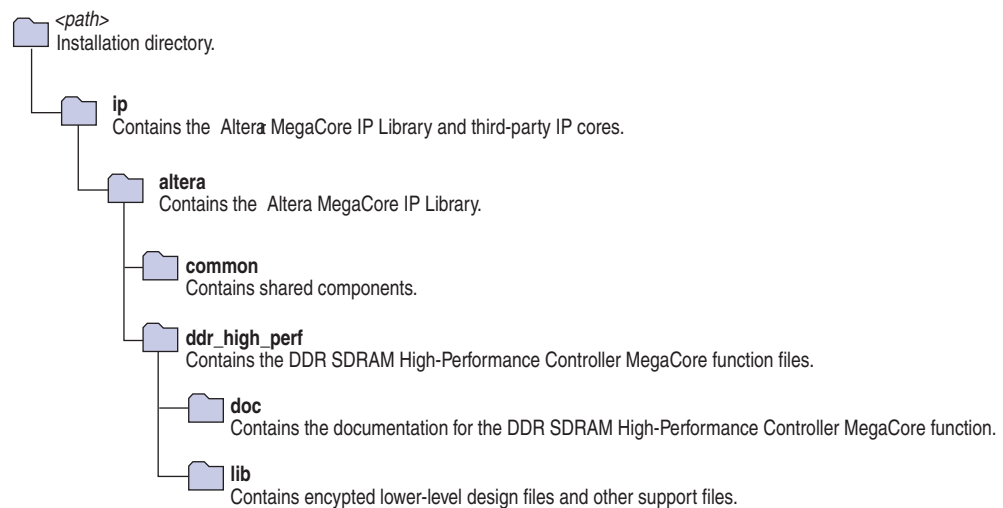
Installation and Licensing

The DDR and DDR2 SDRAM High-Performance Controller MegaCore functions are part of the MegaCore IP Library, which is distributed with the Quartus II software and downloadable from the Altera website, www.altera.com.

For system requirements and installation instructions, refer to *Quartus II Installation & Licensing for Windows and Linux Workstations*.

Figure 1-2 shows the directory structure after you install the DDR and DDR2 SDRAM High-Performance Controller MegaCore functions, where *<path>* is the installation directory. The default installation directory on Windows is `c:\altera\<version>`; on Linux it is `/opt/altera<version>`.

Figure 1-2. Directory Structure



You need a license for the MegaCore function only when you are completely satisfied with its functionality and performance, and want to take your design to production.

If you want to use the DDR or DDR2 SDRAM High-Performance Controller MegaCore function, you can request a license file from the Altera web site at www.altera.com/licensing and install it on your computer. When you request a license file, Altera emails you a **license.dat** file. If you do not have Internet access, contact your local Altera representative.

OpenCore Plus Evaluation

With Altera's free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPPSM megafunction) within your system
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily
- Generate time-limited device programming files for designs that include MegaCore functions

- Program a device and verify your design in hardware

You need to purchase a license for the megafunction only when you are completely satisfied with its functionality and performance, and want to take your design to production.



For more information on OpenCore Plus hardware evaluation using the DDR and DDR2 SDRAM high-performance controller, refer to *AN320: OpenCore Plus Evaluation of Megafunctions*.

OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation can support the following two modes of operation:

- Untethered—the design runs for a limited time
- Tethered—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely

All megafunctions in a device time out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior may be masked by the time-out behavior of the other megafunctions.



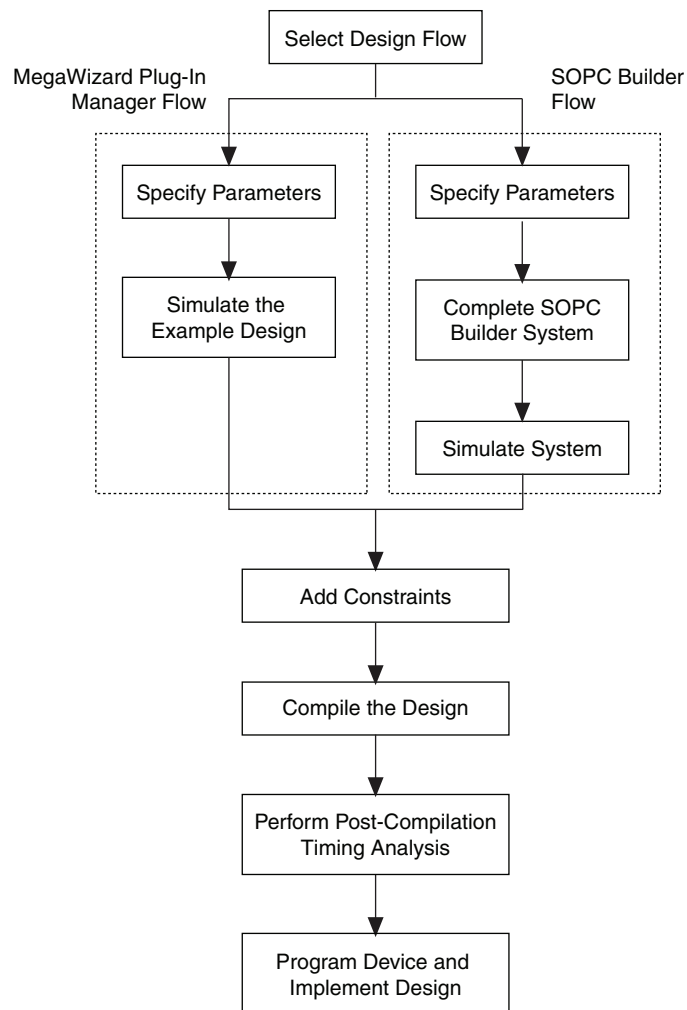
For MegaCore functions, the untethered time-out is 1 hour; the tethered time-out value is indefinite.

Your design stops working after the hardware evaluation time expires and the `local_ready` output goes low.

Design Flow

Figure 2–1 shows the stages for creating a system with the DDR and DDR2 SDRAM High-Performance Controller MegaCore function and the Quartus II software. The sections in this chapter describe each stage.

Figure 2–1. Design Flow



Select Flow

You can parameterize the DDR and DDR2 SDRAM High-Performance Controller MegaCore function using either one of the following flows:

- SOPC Builder flow
- MegaWizard Plug-In Manager flow

Table 2-1 summarizes the advantages offered by the different parameterization flows.

Table 2-1. Advantages of the Parameterization Flows

SOPC Builder Flow	MegaWizard Plug-In Manager Flow
<ul style="list-style-type: none"> ■ Automatically-generated simulation environment ■ Create custom components and integrate them via the component wizard ■ All components are automatically interconnected with the Avalon-MM interface 	<ul style="list-style-type: none"> ■ Design directly from the DDR or DDR2 SDRAM interface to peripheral device or devices ■ Achieves higher-frequency operation

SOPC Builder Flow

The SOPC Builder flow allows you to add the DDR and DDR2 SDRAM High-Performance Controller MegaCore function directly to a new or existing SOPC Builder system. You can also easily add other available components to quickly create an SOPC Builder system with a DDR and DDR2 SDRAM High-Performance Controller, such as the Nios II processor, external memory controllers, and scatter/gather DMA controllers. SOPC Builder automatically creates the system interconnect logic and system simulation environment.



For more information about SOPC Builder, refer to [volume 4](#) of the *Quartus II Handbook*. For more information about how to use controllers with SOPC Builder, refer to [AN 517: Using High-Performance DDR, DDR2, and DDR3 SDRAM With SOPC Builder](#). For more information on the Quartus II software, refer to the Quartus II Help.

Specify Parameters


To specify DDR and DDR2 SDRAM High-Performance Controller parameters using the SOPC Builder flow, follow these steps:

1. In the Quartus II software, create a new Quartus II project with the **New Project Wizard**.
2. On the Tools menu, click **SOPC Builder**.
3. For a new system, specify the system name and language.
4. Add **DDR or DDR2 SDRAM High-Performance Controller** to your system from the **System Contents** tab.



The **DDR or DDR2 SDRAM High-Performance Controller** is in the **SDRAM** folder under the **Memories and Memory Controllers** folder.

5. Specify the required parameters on all pages in the **Parameter Settings** tab.

 For detailed explanation of the parameters, refer to the “Parameter Settings” on page 3-1.

- Click **Finish** to complete the DDR and DDR2 SDRAM High-Performance Controller MegaCore function and add it to the system.

Complete the SOPC Builder System

To complete the SOPC Builder system, follow these steps:

- In the **System Contents** tab, select **Nios II Processor** and click **Add**.
- On the **Nios II Processor** page, in the **Core Nios II** tab, select **altmemddr** for **Reset Vector** and **Exception Vector**.
- Change the **Reset Vector Offset** and the **Exception Vector Offset** to an Avalon address that is not written to by the ALTMEMPHY megafunction during its calibration process.



The ALTMEMPHY megafunction performs memory interface calibration every time it is reset, and in doing so, writes to a range of addresses. If you want your memory contents to remain intact through a system reset, you should avoid using these memory addresses. This step is not necessary, if you reload your SDRAM memory contents from flash every time you reset.

To calculate the Avalon-MM address equivalent of the memory address range 0×0 to $0 \times 1f$, multiply the memory address by the width of the memory interface data bus in bytes. For example, if your external memory data width is 8 bits in non-AFI mode, then the **Reset Vector Offset** should be 0×20 and the **Exception Vector Offset** should be 0×40 . Refer to [Table 2-2](#) for more Avalon-MM addresses for AFI and non-AFI modes.

Table 2-2. Avalon-MM Addresses for AFI and Non-AFI mode

External Memory Interface Width	Reset Vector Offset		Exception Vector Offset	
	AFI	Non-AFI	AFI	Non-AFI
8	0×40	0×20	0×60	0×40
16	0×80	0×40	$0 \times A0$	0×60
32	0×100	0×80	0×120	$0 \times A0$
64	0×200	0×100	0×220	0×120


- Click **Finish**.
- On the **System Contents** tab, expand **Interface Protocols** and expand **Serial**.
- Select **JTAG UART** and click **Add**.
- Click **Finish**.



If there are warnings about overlapping addresses, on the System menu, click **Auto Assign Base Addresses**.

If you enable ECC and there are warnings about overlapping IRQs, on the System menu click **Auto Assign IRQs**.

8. For this example system, ensure all the other modules are clocked on the `altmemddr_sysclk`, to avoid any unnecessary clock-domain crossing logic.
9. Click **Generate**.

 Among the files generated by SOPC Builder is the Quartus II IP File (**.qip**). This file contains information about a generated IP core or system. In most cases, the **.qip** file contains all of the necessary assignments and information required to process the MegaCore function or system in the Quartus II compiler. Generally, a single **.qip** file is generated for each SOPC Builder system. However, some more complex SOPC Builder components generate a separate **.qip** file. In that case, the system **.qip** file references the component **.qip** file.

10. Compile your design, refer to “[Compile the Design](#)” on page 2-13.



If you are upgrading your Nios system design from version 8.1 or previous, ensure that you change the **Reset Vector Offset** and the **Exception Vector Offset** to **AFI** mode.

Simulate the System

During system generation, SOPC Builder optionally generates a simulation model and testbench for the entire system, which you can use to easily simulate your system in any of Altera's supported simulation tools. SOPC Builder also generates a set of ModelSim® Tcl scripts and macros that you can use to compile the testbench, IP functional simulation models, and plain-text RTL design files that describe your system in the ModelSim simulation software.



For more information about simulating SOPC Builder systems, refer to [volume 4](#) of the *Quartus II Handbook* and [AN 351: Simulating Nios II Systems](#).

MegaWizard Plug-In Manager Flow

The MegaWizard Plug-In Manager flow allows you to customize the DDR and DDR2 SDRAM High-Performance Controller MegaCore function, and manually integrate the function into your design.



You can alternatively use the IP Advisor to help you start your DDR and DDR2 SDRAM High-Performance Controller MegaCore design. On the Quartus II Tools menu, point to **Advisors**, and then click **IP Advisor**. The IP Advisor guides you through a series of recommendations for selecting, parameterizing, evaluating, and instantiating a DDR and DDR2 SDRAM High-Performance Controller MegaCore function into your design. It then guides you through a complete Quartus II compilation of your project.



For more information about the MegaWizard Plug-In Manager and the IP Advisor, refer to the Quartus II Help.

Specify Parameters

To specify DDR or DDR2 SDRAM High-Performance Controller parameters using the MegaWizard Plug-In Manager flow, follow these steps:

1. In the Quartus II software, create a new Quartus II project with the **New Project Wizard**.
2. On the Tools menu, click **MegaWizard Plug-In Manager** and follow the steps to start the MegaWizard Plug-In Manager.



The DDR or DDR2 SDRAM High-Performance Controller MegaCore function is in the **Interfaces** folder under the **Memory Controllers** folder.

3. Specify the parameters on all pages in the **Parameter Settings** tab.



For detailed explanation of the parameters, refer to the “[Parameter Settings](#)” on page 3-1.

4. On the **EDA** tab, turn on **Generate simulation model** to generate an IP functional simulation model for the MegaCore function in the selected language.

An IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus II software.



Use the simulation models only for simulation and not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.



Some third-party synthesis tools can use a netlist that contains only the structure of the MegaCore function, but not detailed logic, to optimize performance of the design that contains the MegaCore function. If your synthesis tool supports this feature, turn on **Generate netlist**.

5. On the **Summary** tab, select the files you want to generate. A gray checkmark indicates a file that is automatically generated. All other files are optional.



For more information about the files generated in your project directory, refer to [Table 2-3](#).

6. Click **Finish** to generate the MegaCore function and supporting files.
7. If you generate the MegaCore function instance in a Quartus II project, you are prompted to add the **.qip** files to the current Quartus II project. When prompted to add the **.qip** files to your project, click **Yes**. The addition of the **.qip** files enables their visibility to Nativelink. Nativelink requires the **.qip** files to include libraries for simulation.



The **.qip** file is generated by the MegaWizard interface, and contains information about the generated IP core. In most cases, the **.qip** file contains all of the necessary assignments and information required to process the MegaCore function or system in the Quartus II compiler. The MegaWizard interface generates a single **.qip** file for each MegaCore function.

8. After you review the generation report, click **Exit** to close the MegaWizard Plug-In Manager.

Table 2-3 describes the generated files and other files (AFI mode) that may be in your project directory. The names and types of files specified in the MegaWizard Plug-In Manager report vary based on whether you created your design with VHDL or Verilog HDL.

Table 2-3. Generated Files (Part 1 of 2)

Filename	Description
<code><variation name>.bsf</code>	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<code><variation name>.html</code>	MegaCore function report file.
<code><variation name>.v</code> or <code>.vhd</code>	A MegaCore function variation file, which defines a VHDL or Verilog HDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
<code><variation name>.qip</code>	Contains Quartus II project information for your MegaCore function variations.
<code><variation name>.ppf</code>	This XML file describes the MegaCore pin attributes to the Quartus II Pin Planner. MegaCore pin attributes include pin direction, location, I/O standard assignments, and drive strength. If you launch IP Toolbench outside of the Pin Planner application, you must explicitly load this file to use Pin Planner.
<code><variation name>_auk_ddr_hp_controller_wrapper.vo</code> or <code>.vho</code>	VHDL or Verilog HDL IP functional simulation model.
<code><variation name>_example_driver.v</code> or <code>.vhd</code>	Example self-checking test generator that matches your variation.
<code><variation name>_example_top.v</code> or <code>.vhd</code>	Example top-level design file that you should set as your Quartus II project top level. Instantiates the example driver and the controller.
<code>alt_mem_phy_defines.v</code>	Contains constants used in the interface. This file is always in Verilog HDL regardless of the language you chose in the MegaWizard Plug-In Manager.
<code><variation_name>_phy.html</code>	Lists the top-level files created and ports used in the megafunction.
<code><variation_name>_phy.v</code> or <code>.vhd</code>	Top-level file of your ALTMEMPHY variation, generated based on the language you chose in the MegaWizard Plug-In Manager.
<code><variation_name>_phy.vho</code>	Contains functional simulation model for VHDL only.
<code><variation_name>_phy_alt_mem_phy_delay.vhd</code>	Includes a delay module for simulation. This file is only generated if you choose VHDL as the language of your MegaWizard Plug-In Manager output files.
<code><variation_name>_phy_alt_mem_phy_dq_dqs.vhd</code> or <code>.v</code>	Generated file that contains DQ/DQS I/O atoms interconnects and instance. Arria II GX devices only.

Table 2-3. Generated Files (Part 2 of 2)

Filename	Description
<variation_name>_phy_alt_mem_phy_dq_dqs_clearbox.txt	Specification file that generates the <variation_name>_alt_mem_phy_dq_dqs file using the clearbox flow. Arria II GX devices only.
<variation_name>_phy_alt_mem_phy_pll.qip	Quartus II IP file for the PLL that your ALTMEMPHY variation uses that contains the files associated with this megafunction.
<variation_name>_phy_alt_mem_phy_pll.v/.vhd	The PLL megafunction file for your ALTMEMPHY variation, generated based on the language you chose in the MegaWizard Plug-In Manager.
<variation_name>_phy_alt_mem_phy_pll_bb.v/.cmp	Black box file for the PLL used in your ALTMEMPHY variation. Typically unused.
<variation_name>_phy_alt_mem_phy_reconfig.qip	Quartus II IP file for the PLL reconfiguration block. Only generated when targeting Arria GX, Arria II GX, HardCopy II, Stratix II, and Stratix II GX devices.
<variation_name>_phy_alt_mem_phy_reconfig.v/.vhd	PLL reconfiguration block module. Only generated when targeting Arria GX, Arria II GX, HardCopy II, Stratix II, and Stratix II GX devices.
<variation_name>_phy_alt_mem_phy_reconfig_bb.v/.cmp	Blackbox file for the PLL reconfiguration block. Only generated when targeting Arria GX, Arria II GX, HardCopy II, Stratix II, and Stratix II GX devices.
<variation_name>_phy_alt_mem_phy_seq.vhd	Contains the sequencer used during calibration. This file is encrypted and is always in VHDL language regardless of the language you chose in the MegaWizard Plug-In Manager.
<variation_name>_phy_alt_mem_phy_seq_wrapper.v/.vhd	A wrapper file, for compilation only, that calls the sequencer file, created based on the language you chose in the MegaWizard Plug-In Manager.
<variation_name>_phy_alt_mem_phy_seq_wrapper.vo/.vho	A wrapper file, for simulation only, that calls the sequencer file, created based on the language you chose in the MegaWizard Plug-In Manager.
<variation_name>_phy_alt_mem_phy.v	Contains all modules of the ALTMEMPHY variation except for the sequencer. This file is always in Verilog HDL language regardless of the language you chose in the MegaWizard Plug-In Manager.
<variation_name>_phy_bb.v/.cmp	Black box file for your ALTMEMPHY variation, depending whether you are using Verilog HDL or VHDL language.
<variation_name>_phy_ddr_pins.tcl	Contains procedures used in the <variation_name>_report_timing.tcl file.
<variation_name>_phy_ddr_timing.sdc	Contains timing constraints for your ALTMEMPHY variation.
<variation_name>_phy_report_timing.tcl	Script that reports timing for your ALTMEMPHY variation during compilation.
<variation_name>_pin_assignments.tcl	Contains I/O standard, drive strength, output enable grouping, and termination assignments for your ALTMEMPHY variation. If your top-level design pin names do not match the default pin names or a prefixed version, edit the assignments in this file.

9. Set the `<variation name>_example_top.v` or `.vhd` file to be the project top-level design file.
 - a. On the File menu, click **Open**.
 - b. Browse to `<variation name>_example_top` and click **Open**.
 - c. On the Project menu, click **Set as Top-Level Entity**.
10. Simulate the example design (refer to “[Simulate the Example Design](#)” on page 2-8) and compile (refer to “[Compile the Design](#)” on page 2-13).

Simulate the Example Design

You can simulate the example design with the MegaWizard Plug-In Manager-generated IP functional simulation models. The MegaWizard Plug-In Manager generates a VHDL or Verilog HDL testbench for your example design, which is in the **testbench** directory in your project directory.

You can use the IP functional simulation model with any Altera-supported VHDL or Verilog HDL simulator. You can perform a simulation in a third-party simulation tool from within the Quartus II software, using NativeLink.



For more information on the testbench, refer to “[Example Design](#)” on page 4-11.

For more information on NativeLink, refer to the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

Simulating Using NativeLink

To set up simulation in the Quartus II software using NativeLink, follow these steps:

1. Create a custom variation with an IP functional simulation model, refer to step 4 in the “[Specify Parameters](#)” section on page 2-5.
2. Set the top-level entity to the example project.
 - a. On the File menu, click **Open**.
 - b. Browse to `<variation name>_example_top` and click **Open**.
 - c. On the Project menu, click **Set as Top-Level Entity**.
3. Set up the Quartus II NativeLink.
 - a. On the Assignments menu, click **Settings**. In the **Category** list, expand **EDA Tool Settings** and click **Simulation**.
 - b. From the **Tool name** list, click on your preferred simulator.



Check that the absolute path to your third-party simulator executable is set. On the Tools menu, click **Options** and select **EDA Tools Options**.

- c. In **NativeLink settings**, select **Compile test bench** and click **Test Benches**.
- d. Click **New** at the **Test Benches** page to create a testbench.

4. On the **New Test Bench Settings** dialog box, do the following:
 - a. Type a name for the **Test bench name**.
 - b. In **Top level module in test bench**, type the name of the automatically generated testbench, `<variation name>_example_top_tb`.
 - c. In **Design instance in test bench**, type the name of the top-level instance, `dut`.
 - d. Under **Simulation period**, set **End simulation at** to 600 μ s.
 - e. Add the testbench files and automatically-generated memory model files. In the **File name** field, browse to the location of the memory model and the testbench, click **Open** and then click **Add**. The testbench is `<variation name>_example_top_tb.v`; memory model is `<variation name>_mem_model.v`.
5. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration** to start analysis.
6. On the Tools menu, point to **Run EDA Simulation Tool** and click **EDA RTL Simulation**.



The auto generated generic SDRAM model may be used as a placeholder for a specific memory vendor supplied model. For information on how to replace the generic model with a vendor specific model, refer to “Perform RTL/Functional Simulation (Optional)” in *AN 328: Interfacing DDR2 SDRAM with Stratix II, Stratix II GX, and Arria GX Devices*.



Ensure that the Quartus II **EDA Tool Options** are configured correctly for your simulation environment. On the Tools menu, click **Options**. In the **Category** list, click **EDA Tool Options** and verify the locations of the executable files.



If your Quartus II project appears to be configured correctly but the example testbench still fails, check the known issues on the *Knowledge Database* page before filing a service request.

For a complete MegaWizard Plug-In Manager system design example containing the DDR and DDR2 SDRAM high-performance controller MegaCore function, refer to *Chapter 5, Example Design Walkthrough*.

IP Functional Simulations

For VHDL simulations with IP functional simulation models, perform the following steps:

1. Create a directory in the `<project directory>\testbench` directory.

2. Launch your simulation tool from this directory and create the following libraries:
 - altera_mf
 - lpm
 - sgate
 - *<device name>*
 - altera
 - ALTGXB
 - *<device name>*_hssi
 - auk_ddr_hp_user_lib
3. Compile the files into the appropriate library (AFI mode) as shown in [Table 2-4](#). The files are in VHDL93 format.

Table 2-4. Files to Compile—VHDL IP Functional Simulation Models (Part 1 of 2)

Library	File Name
altera_mf	<i><QUARTUS_ROOTDIR>/eda/sim_lib/altera_mf_components.vhd</i> <i><QUARTUS_ROOTDIR>/eda/sim_lib/altera_mf.vhd</i>
lpm	<i>/eda/sim_lib/220pack.vhd</i> <i>/eda/sim_lib/220model.vhd</i>
sgate	<i>eda/sim_lib/sgate_pack.vhd</i> <i>eda/sim_lib/sgate.vhd</i>
<i><device name></i>	<i>eda/sim_lib/<device name>_atoms.vhd</i> <i>eda/sim_lib/<device name>_components.vhd</i> <i>eda/sim_lib/<device name>_hssi_atoms.vhd (1)</i>
altera	<i>eda/sim_lib/altera_primitives_components.vhd</i> <i>eda/sim_lib/altera_syn_attributes.vhd</i> <i>eda/sim_lib/altera_primitives.vhd</i>
ALTGXB (1)	<i><device name>_mf.vhd</i> <i><device name>_mf_components.vhd</i>
<i><device name></i> _hssi (1)	<i><device name>_hssi_components.vhd</i> <i><device name>_hssi_atoms.vhd</i>

Table 2-4. Files to Compile—VHDL IP Functional Simulation Models (Part 2 of 2)

Library	File Name
auk_ddr_hp_user_lib	<QUARTUS_ROOTDIR>/ libraries/vhdl/altera/altera_europa_support_lib.vhd
	<project directory>/<variation name>_phy_alt_mem_phy_seq_wrapper.vho
	<project directory>/<variation name>_auk_ddr_hp_controller_wrapper.vho
	<project directory>/<variation name>_phy.vho
	<project directory>/<variation name>.vhd
	<project directory>/<variation name>_example_top.vhd
	<project directory>/<variation name>_controller_phy.vhd
	<project directory>/<variation name>_phy_alt_mem_phy_reconfig.vhd (2)
	<project directory>/<variation name>_phy_alt_mem_phy_pll.vhd
	<project directory>/<variation name>_phy_alt_mem_phy_seq.vhd
	<project directory>/<variation name>_example_driver.vhd
	<project directory>/<variation name>_ex_lfsr8.vhd
	testbench/<variation name>_example_top_tb.vhd
	testbench/<variation name>_mem_model.vhd

Note for Table 2-4:

- (1) Applicable only for Arria GX, Arria II GX, Stratix GX, Stratix II GX and Stratix IV devices.
- (2) Applicable only for Arria GX, Hardcopy II, Stratix II and Stratix II GX devices.



If you are targeting Stratix IV devices, you need both the Stratix IV and Stratix III files (**stratixiv_atoms** and **stratixiii_atoms**) to simulate in your simulator, unless you are using NativeLink.

4. Load the testbench in your simulator with the timestep set to picoseconds.

For Verilog HDL simulations with IP functional simulation models, follow these steps:

1. Create a directory in the <project directory>\testbench directory.
2. Launch your simulation tool from this directory and create the following libraries:
 - altera_mf_ver
 - lpm_ver
 - sgate_ver
 - <device name>_ver
 - altera_ver
 - ALTGXB_ver
 - <device name>_hssi_ver
 - auk_ddr_hp_user_lib
3. Compile the files into the appropriate library as shown in [Table 2-5 on page 2-12](#).

Table 2-5. Files to Compile—Verilog HDL IP Functional Simulation Models

Library	File Name
altera_mf_ver	<QUARTUS_ROOTDIR>/eda/sim_lib/altera_mf.v
lpm_ver	/eda/sim_lib/220model.v
sgate_ver	eda/sim_lib/sgate.v
<device name>_ver	eda/sim_lib/<device name>_atoms.v eda/sim_lib/<device name>_hssi_atoms.v (1)
altera_ver	eda/sim_lib/altera_primitives.v
ALTGXB_ver (1)	<device name>_mf.v
<device name>_hssi_ver (1)	<device name>_hssi_atoms.v
auk_ddr_hp_user_lib	<QUARTUS_ROOTDIR>/ libraries/vhdl/altera/altera_europa_support_lib.v
	alt_mem_phy_defines.v
	<project directory>/<variation name>_phy_alt_mem_phy_seq_wrapper.vo
	<project directory>/<variation name>_auk_ddr_hp_controller_wrapper.vo
	<project directory>/<variation name>.v
	<project directory>/<variation name>_example_top.v
	<project directory>/<variation name>_phy.v
	<project directory>/<variation name>_controller_phy.v
	<project directory>/<variation name>_phy_alt_mem_phy_reconfig.v (2)
	<project directory>/<variation name>_phy_alt_mem_phy_pll.v
	<project directory>/<variation name>_phy_alt_mem_phy.v
	<project directory>/<variation name>_example_driver.v
	<project directory>/<variation name>_ex_lfsr8.v
	testbench/<variation name>_example_top_tb.v
testbench/<variation name>_mem_model.v	

Notes for Table 2-5:

- (1) Applicable only for Arria GX, Arria II GX, Stratix GX, Stratix II GX and Stratix IV devices.
(2) Applicable only for Arria GX, Hardcopy II, Stratix II and Stratix II GX devices.



If you are targeting Stratix IV devices, you need both the Stratix IV and Stratix III files (**stratixiv_atoms** and **stratixiii_atoms**) to simulate in your simulator, unless you are using NativeLink

- Configure your simulator to use transport delays, a timestep of picoseconds, and to include all the libraries in [Table 2-5](#).

Compile the Design

To use the Quartus II software to compile the example design and perform post-compilation timing analysis, follow these steps:

1. Set up the TimeQuest timing analyzer:
 - a. On the Assignments menu, click **Timing Analysis Settings**, select **Use TimeQuest Timing Analyzer during compilation**, and click **OK**.
 - b. Add the Synopsys Design Constraints (.sdc) file, `<variation name>_phy_ddr_timing.sdc`, to your project. On the Project menu, click **Add/Remove Files in Project** and browse to select the file.
 - c. Add the .sdc file for the example top-level design, `<variation name>_example_top.sdc`, to your project. This file is only required if you are using the example as the top-level design.
2. Use one of the following procedures to specify I/O standard assignments for pins:
 - If you have a single DDR or DDR2 SDRAM interface, and your top-level pins have default naming shown in the example design, run `<variation name>_pin_assignments.tcl`.
 - If your design contains pin names that do not match the design, edit the `<variation name>_pin_assignments.tcl` file before you run the script. Follow these steps:
 - a. Open `<variation name>_pin_assignments.tcl` file.
 - b. Based on the flow you are using, set the `sopc_mode` value to Yes or No.
 - SOPC Builder System flow:

```
if {[info exists socp_mode]} {set socp_mode YES}
```
 - MegaWizard Plug-In Manager flow:

```
if {[info exists socp_mode]} {set socp_mode NO}
```
 - c. Type your preferred prefix in the `pin_prefix` variable. For example, to add the prefix `my_mem`, do the following:

```
if {[info exists set_prefix]}{set pin_prefix "my_mem "}
```

After setting the prefix, the pin names are expanded as shown in the following:
 - SOPC Builder System flow:

```
my_mem_cs_n_from_the_<your instance name>
```
 - MegaWizard Plug-In Manager flow:

```
my_mem_cs_n[0]
```
3. Set the top-level entity to the top-level design.
 - a. On the File menu, click **Open**.
 - b. Browse to your SOPC Builder system top-level design or `<variation name>_example_top` if you are using MegaWizard Plug-In Manager, and click **Open**.
 - c. On the Project menu, click **Set as Top-Level Entity**.

4. Assign the DQ and DQS pin locations.
 - a. You should assign pin locations to the pins in your design, so the Quartus II software can perform fitting and timing analysis correctly.
 - b. Use either the Pin Planner or Assignment Editor to assign the clock source pin manually. Also choose which DQS pin groups should be used by assigning each DQS pin to the required pin. The Quartus II Fitter then automatically places the respective DQ signals onto suitable DQ pins within each group.



When assigning pins in your design, ensure that you set an appropriate I/O standard for the non-memory interfaces, such as the clock source and the reset inputs. For example, for DDR SDRAM select **2.5 V** and for DDR2 SDRAM select **1.8 V**. Also select in which bank or side of the device you want the Quartus II software to place them.

5. For Stratix III and Stratix IV designs, if you are using advanced I/O timing, specify board trace models in the **Device & Pin Options** dialog box. If you are using any other device and not using advanced I/O timing, specify the output pin loading for all memory interface pins.
6. Select your required I/O driver strength (derived from your board simulation) to ensure that you correctly drive each signal or ODT setting and do not suffer from overshoot or undershoot.
7. To compile the design, on the Processing menu, click **Start Compilation**.



To attach the SignalTap® II logic analyzer to your design, refer to *AN 380: Test DDR or DDR2 SDRAM Interfaces on Hardware Using the Example Driver*.


Program Device and Implement the Design

After you have compiled the example design, you can perform RTL simulation (refer to “[Simulate the Example Design](#)” on page 2-8) or program your targeted Altera device to verify the example design in hardware.

To implement your design based on the example design, replace the example driver in the example design with your own logic.


Memory Settings

The **Memory Settings** page provides the same options as the ALTMEMPHY megafunction **Memory Settings** page.

 For more information on the memory settings, refer to the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)*.

PHY Settings

Board skew is the skew across all the memory interface signals, which includes clock, address, command, data, mask, and strobe signals.

 For more information on the PHY settings, refer to the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)*.

Controller Settings

Table 3–1 shows the options provided in the **Controller Settings** page.

Table 3–1. Controller Settings

Parameter	Range	Description
Enable error detection and correction logic	On or off	Turn on to add the optional error correction coding (ECC) to the design, refer to “ Error Correction Coding (ECC) ” on page 4–7.
Enable user auto-refresh controls	On or off	Turn on for user control of the refreshes, refer to “ User Refresh Control ” on page 4–34.
Enable auto-precharge control	On or off	Turn on if you need fast random access, refer to “ Auto-Precharge Commands ” on page 4–36
Enable power down controls	On or off	Turn on to enable the controller to allow you to place the external memory device in a power-down mode, refer to “ Self-Refresh and Power-Down Commands ” on page 4–35
Enable self-refresh controls	On or off	Turn on to enable the controller to allow you to place the external memory device in a self-refresh mode, refer to “ Self-Refresh and Power-Down Commands ” on page 4–35
Local Interface Protocol	Native or Avalon Memory-Mapped	Specifies the local side interface between the user logic and the memory controller. The Avalon Memory-Mapped (MM) interface allows you to easily connect to other Avalon-MM peripherals.
Controller/Phy Interface Protocol	AFI or non-AFI	Specifies the controller/PHY interface. Refer to the <i>External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)</i> for more information.
Multiple Controller Clock Sharing	On or off	This option is only in SOPC Builder Flow. Turn on if you want to improve your system efficiency when your system has multiple controllers. Refer to the <i>External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)</i> for more information.

The DDR and DDR2 SDRAM high-performance controllers instantiate encrypted control logic and the ALTMEMPHY megafunction. The controller accepts read and write requests from the user on its local interface, using either the Avalon-MM interface protocol or the native interface protocol. It converts these requests into the necessary SDRAM commands, including any required bank management commands. Each read or write request on the Avalon-MM or native interface maps to one SDRAM read or write command. Since the controller uses a memory burst length of 4, read and write requests are always of length 1 on the local interface if the controller is in half-rate mode. In full-rate mode, the controller accepts requests of size 1 or 2 on the local interface. Requests of size 2 on the local interface produce better throughput as whole memory burst is used.

The bank management logic in the controller keeps a row open in every bank in the memory system. For example, a controller configured for a double-sided, 4-bank DDR or DDR2 SDRAM DIMM keeps an open row in each of the 8 banks. The controller allows you to request an auto-precharge read or auto-precharge write, allowing control over whether to keep that row open after the request. You can achieve maximum efficiency when you issue reads and writes to the same bank, with the last access to that bank being an auto-precharge read or write. The controller does not do any access reordering.

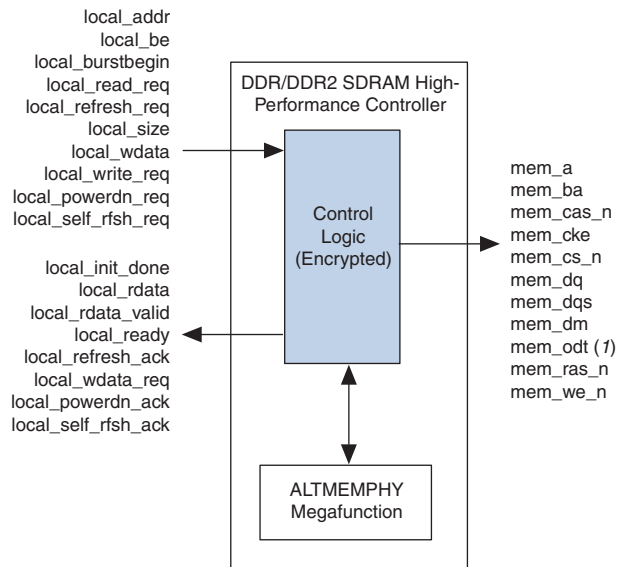


For more information on the ALTMEMPHY megafunction, refer to the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)*.

Block Description

Figure 4-1 shows a block diagram of the DDR or DDR2 SDRAM high-performance controller in non-AFI mode.

Figure 4-1. DDR and DDR2 SDRAM High-Performance Controller (Non-AFI) Block Diagram

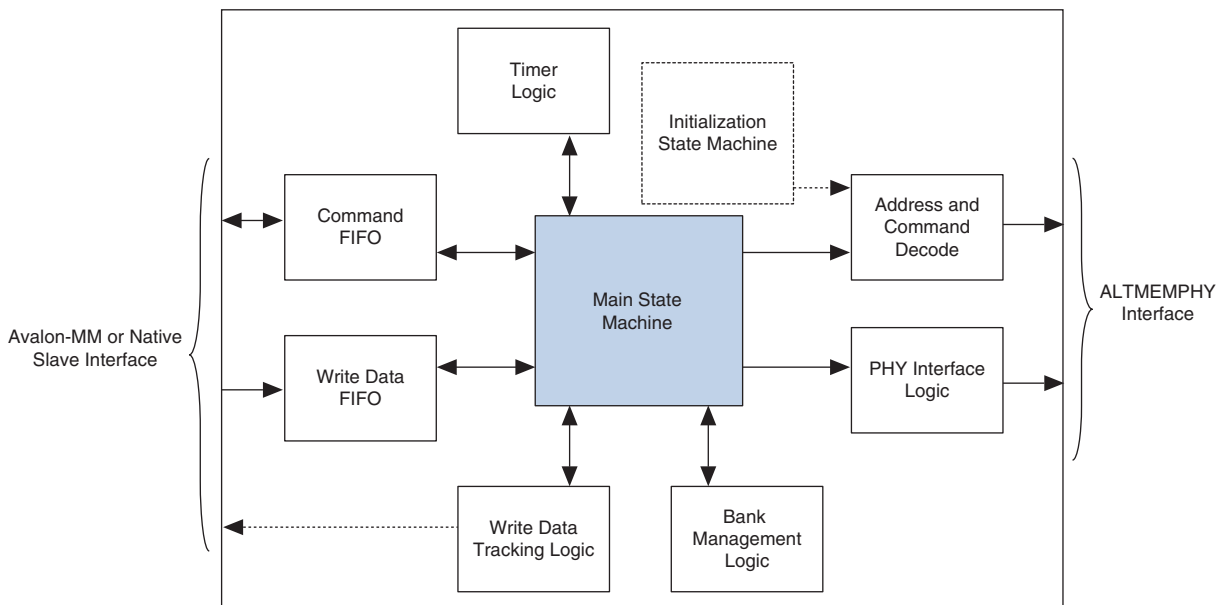


Note to Figure 4-1:

- (1) DDR2 SDRAM high-performance controller only.

Figure 4-2 shows a block diagram of the DDR or DDR2 SDRAM high-performance controller architecture.

Figure 4-2. DDR and DDR2 SDRAM High-Performance Controller Architecture Block Diagram



The blocks in [Figure 4–2 on page 4–2](#) are described in the following sections.

Command FIFO

This FIFO allows the controller to buffer up to four consecutive read or write commands. It is built from logic elements, and stores the address, read or write flag, and burst count information. If this FIFO fills up, the `local_ready` signal to the user is deasserted until the main state machine takes a command from the FIFO.

Write Data FIFO

The write data FIFO holds the write data from the user until the main state machine can send it to the ALTMEMPHY megafunction (which does not have a write data buffer). In Avalon-MM interface mode, the user logic presents a write request, address, burst count, and one or more beats of data at the same time. The write data beats are placed into the FIFO until they are needed. In native interface mode, the user logic presents a write request, address, and burst count. The controller then requests the correct number of write data beats from the user via the `local_wdata_req` signal, and the user logic must return the write data in the clock cycle after the write data request signal.

This FIFO is sized to be deeper than the command FIFO to prevent it from filling up and interrupting streaming writes.

Write Data Tracking Logic

This logic keeps track of how many beats of write data are in the FIFO. In native interface mode, this logic manages how much more data to request from the user logic and issues the `local_wdata_req` signal.

Main State Machine

This state machine decides what DDR commands to issue based on inputs from the command FIFO, the bank management logic, and the timer logic.

Bank Management Logic

The bank management logic keeps track the current state of each bank. It can keep a row open in every bank in your memory system. The state machine uses the information provided by this logic to decide whether it needs to issue bank management commands before it reads or writes to the bank. The controller always leaves the bank open unless the user requests an auto-precharge read or write. The periodic refresh process also causes all the banks to be closed.

Timer Logic

The timer logic tracks whether the required minimum number of clock cycles has passed since the last relevant command was issued. For example, the timer logic records how many cycles have elapsed since the last activate command so that the state machine knows it is safe to issue a read or write command (t_{RCD}). The timer logic also counts the number of clock cycles since the last periodic refresh command and sends a high priority alert to the state machine if the number of clock cycles has expired.

Initialization State Machine

The initialization state machine issues the appropriate sequence of command to initialize the memory devices. It is specific to DDR and DDR2 as each memory type requires a different sequence of initialization commands.

If you select the AFI mode, then the ALTMEMPHY megafunction is responsible for initializing the memory. If you select the non-AFI mode, then the controller is responsible for initializing the memory.

Address and Command Decode

When the state machine wants to issue a command to the memory, it asserts a set of internal signals. The address and command decode logic turns these into the DDR-specific RAS/CAS/WE commands.

PHY Interface Logic

When the main state machine issues a write command to the memory, the write data for that write burst has to be fetched from the write data FIFO. The relationship between write command and write data depends on the memory type, ALTMEMPHY megafunction interface type, CAS latency, and the full-rate or half-rate setting. The PHY interface logic adjusts the timing of the write data FIFO read request signal so that the data arrives on the external memory interface DQ pins at the correct time.

ODT Generation Logic

The ODT generation logic (not shown) calculates when and for how long to enable the ODT outputs. It also decides which ODT bit to enable, based on the number of chip selects in the system.

- 1 DIMM (1 or 2 Chip Selects)

In the case of a single DIMM, the ODT signal is only asserted during writes. The ODT signal on the DIMM at `mem_cs [0]` is always used, even if the write command on the bus is to `mem_cs [1]`. In other words, `mem_odt [0]` is always asserted even if there are two ODT signals.

- 2 or more DIMMs

In the multiple DIMM case, the appropriate ODT bit is asserted for both read and writes. The ODT signal on the adjacent DIMM is enabled as shown.

If a write/read is happening to:

`mem_cs [0] or cs [1]`
`mem_cs [2] or cs [3]`
`mem_cs [4] or cs [5]`
`mem_cs [6] or cs [7]`

ODT enabled:

`mem_odt [2]`
`mem_odt [0]`
`mem_odt [6]`
`mem_odt [4]`

Low Power Mode Logic

The low power mode logic (not shown) monitors the `local_powerdn_req` and `local_self_rfsh_req` request signals. This logic also informs the user of the current low power state via the `local_powerdn_ack` and `local_self_rfsh_ack` acknowledge signals.

Control Logic

Bus commands control SDRAM devices using combinations of the `mem_ras_n`, `mem_cas_n`, and `mem_we_n` signals. For example, on a clock cycle where all three signals are high, the associated command is a no operation (NOP). A NOP command is also indicated when the chip select signal is not asserted. Table 4-1 shows the standard SDRAM bus commands.

Table 4-1. Bus Commands

Command	Acronym	ras_n	cas_n	we_n
No operation	NOP	High	High	High
Active	ACT	Low	High	High
Read	RD	High	Low	High
Write	WR	High	Low	Low
Burst terminate	BT	High	High	Low
Precharge	PCH	Low	High	Low
Auto refresh	ARF	Low	Low	High
Load mode register	LMR	Low	Low	Low

The DDR and DDR2 SDRAM high-performance controllers must open SDRAM banks before they access addresses in that bank. The row and bank to be opened are registered at the same time as the active (ACT) command. The DDR and DDR2 SDRAM high-performance controllers close the bank and open it again if they need to access a different row. The precharge (PCH) command closes only a bank.

The primary commands used to access SDRAM are read (RD) and write (WR). When the WR command is issued, the initial column address and data word is registered. When a RD command is issued, the initial address is registered. The initial data appears on the data bus 2 to 3 clock cycles later (3 to 5 for DDR2 SDRAM). This delay is the column address strobe (CAS) latency and is due to the time required to read the internal DRAM core and register the data on the bus. The CAS latency depends on the speed of the SDRAM and the frequency of the memory clock. In general, the faster the clock, the more cycles of CAS latency are required. After the initial RD or WR command, sequential reads and writes continue until the burst length is reached or a burst terminate (BT) command is issued. DDR and DDR2 SDRAM devices support burst lengths of 2, 4, or 8 data cycles. The auto-refresh command (ARF) is issued periodically to ensure data retention. This function is performed by the DDR or DDR2 SDRAM high-performance controller.

The load mode register command (LMR) configures the SDRAM mode register. This register stores the CAS latency, burst length, and burst type.



For more information, refer to the specification of the SDRAM that you are using.

Latency


There are two types of latency that you must consider for memory controller designs—read and write latencies. We define the read and write latencies as follows.

- Read latency is the time it takes for the read data to appear at the local interface after you assert the read request signal to the controller.

- Write latency is the time it takes for the write data to appear at the memory interface after you assert the write request signal to the controller.

Latency calculations are made with the following assumptions:

- Reading and writing to the rows that are already open
- The `local_ready` signal is asserted high (no wait states)
- No refresh cycles occur before transaction
- The latency is defined using the local side frequency and absolute time (ns)


 For the half rate controller, the local side frequency is half the memory interface frequency; for the full rate controller, it is equal to the memory interface frequency.

Altera defines the read and write latencies in terms of the local interface clock frequency and by the absolute time for the memory controllers.

Table 4–2 shows the read and write latency derived from the write and read latency definitions for half and full rate DDR2 SDRAM high-performance controller and for Arria GX, Cyclone III, Stratix II, Stratix III, and Stratix IV devices.

Table 4–2. Typical Latency

Device	Controller Rate	Frequency (MHz)	Controller Latency	Latency Type	Total Latency	
					Local Clock Cycles	Time (ns)
Arria GX	Half	233	5	Read	18	151
				Write	11	91
	Full	167	4	Read	20	120
				Write	10	60
Cyclone III	Half	200	5	Read	18	175
				Write	11	105
	Full	167	4	Read	20	120
				Write	10	60
Stratix II	Half	333	5	Read	18	105
				Write	11	63
	Full	200	4	Read	20	100
				Write	10	50
Stratix III	Half	400	5	Read	21	111
				Write	13	65
	Full	267	4	Read	21	85
				Write	12	44
Stratix IV	Half	400	5	Read	21	111
				Write	13	65
	Full	267	4	Read	21	85
				Write	12	44

 The exact latency depends on your precise configuration. You should obtain precise latency from simulation, but this figure may vary in hardware because of the automatic calibration process.

 Refer to the Latency section in chapter 1 of the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)* for more detailed information.

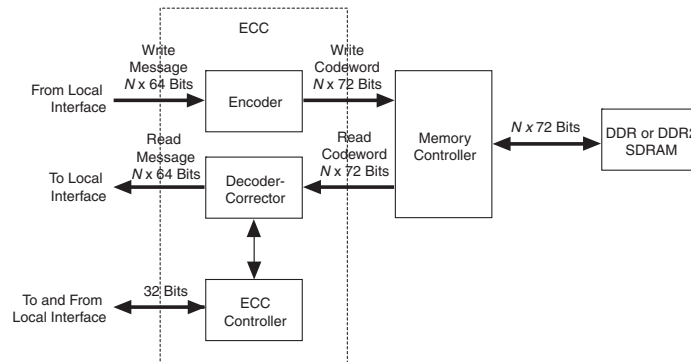
Error Correction Coding (ECC)

The optional ECC comprises an encoder and a decoder-corrector, which can detect and correct single-bit errors and detect double-bit errors. The ECC uses an 8-bit ECC for each 64-bit message. The ECC has the following features:

- Hamming code ECC that encodes every 64-bits of data into 72-bits of codeword with 8-bits of Hamming code parity bits
- Latency:
 - Maximum of 1 or 2 clock delay during writes
 - Minimum 1 or 3 clock delay during reads
- Detects and corrects all single-bit errors. Also the ECC sends an interrupt when the user-defined threshold for a single-bit error is reached.
- Detects all double-bit errors. Also, the ECC counts the number of double-bit errors and sends an interrupt when the user-define threshold for double-bit error is reached.
- Accepts partial writes
- Creates forced errors to check the functioning of the ECC
- Powers up in a sensible state

Figure 4-3 shows the ECC block diagram.

Figure 4-3. ECC Block Diagram



The ECC comprises the following blocks:

- The encoder—encodes the 64-bit message to a 72-bit codeword
- The decoder-corrector—decodes and corrects the 72-bit codeword if possible
- The ECC controller—controls multiple encoder and decoder-correctors, so that the ECC can handle different bus widths. Also, it controls the following functions of the encoder and decoder-corrector:
 - Interrupts:
 - Detected and corrected single-bit error
 - Detected double-bit error
 - Single-bit error counter threshold exceeded
 - Double-bit error counter threshold exceeded
 - Configuration registers:
 - Single-bit error detection counter threshold
 - Double-bit error detection counter threshold
 - Capture status for first encountered error or most recent error
 - Enable deliberate corruption of ECC for test purposes
 - Status registers:
 - Error address
 - Error type: single-bit error or double-bit error
 - Respective byte error ECC syndrome
 - Error signal—an error signal corresponding to the data word is provided with the data and goes high if a double-bit error that cannot be corrected occurs in the return data word.

- Counters:
 - Detected and/or corrected single-bit errors
 - Detected double-bit errors

 For more information on the ECC registers, refer to [Appendix A, ECC Register Description](#).

The ECC can instantiate multiple encoders, each running in parallel, to encode any width of data words assuming they are integer multiples of 64.

The ECC operates between the local (native or Avalon-MM interface) and the memory controller.

The ECC has an $N \times 64$ -bit (where N is an integer) wide interface, between the local interface and the ECC, for receiving and returning data from the local interface. This interface can be a native interface or an Avalon-MM slave interface, you select the type of interface in the MegaWizard interface.

The ECC has a second interface between the local interface and the ECC, which is a 32-bit wide Avalon-MM slave to control and report the status of the operation of the ECC controller.

The encoded data from the ECC is sent to the memory controller using a $N \times 72$ -bit wide Avalon-MM master interface, which is between the ECC and the memory controller.

When testing the DDR SDRAM high-performance controller, you can turn off the ECC.

Interrupts

The ECC issues an interrupt signal when one of the following scenarios occurs:

- The single-bit error counter reaches the set maximum single-bit error threshold value.
- The double-bit error counter reaches the set maximum double-bit error threshold value.

The error counters increment every time the respective event occurs for all N parts of the return data word. This incremented value is compared with the maximum threshold and an interrupt signal is sent when the value is equal to the maximum threshold. The ECC clears the interrupts when you write a 1 to the respective status register. You can mask the interrupts from either of the counters using the control word.

Partial Writes

The ECC supports partial writes. Along with the address, data, and burst signals, the Avalon-MM interface also supports a signal vector that is responsible for byte-enable. Every bit of this signal vector represents a byte on the data-bus. Thus, a 0 on any of these bits is a signal for the controller not to write to that particular location—a partial write.

For partial writes, the ECC performs the following steps:

- Stalls further read or write commands from the Avalon-MM interface when it receives a partial write condition.
- Simultaneously sends a self-generated read command, for the partial write address, to the memory controller.
- Upon receiving a return data from the memory controller for the particular address, the ECC decodes the data, checks for errors, and then sends it to the ECC controller.
- The ECC controller merges the corrected or correct dataword with the incoming information.
- Sends the updated dataword to the encoder for encoding and then sends to the memory controller with a write command.
- Releases the stall of commands from the Avalon-MM interface, which allows it to receive new commands.

The following corner cases can occur:

- A single-bit error during the read phase of the read-modify-write process. In this case, the single-bit error is corrected first, the single-bit error counter is incremented and then a partial write is performed to this corrected decoded data word.
- A double-bit error during the read phase of the read-modify-write process. In this case, the double-bit error counter is incremented and an interrupt is sent through the Avalon-MM interface. The new write word is not written to its location. A separate field in the interrupt status register highlights this condition.

Partial Bursts

Some DIMMs do not have the DM pins and so do not support partial bursts. A minimum of four words must be written to the memory at the same time. In cases of partial burst write, the ECC offers a mechanism similar to the partial write.

In cases of partial bursts, the write data from the native interface is stored in a 64-bit wide FIFO buffer of maximum burst size depth, while in parallel a read command of the corresponding addresses is sent to the DIMM. Further commands from the native interface are stalled until the current burst is read, modified, and written back to the memory controller.

ECC Latency

Using the ECC results in the following latency changes:

- Local Burst Length 1
- Local Burst Length 2

Local Burst Length 1

For a local burst length of 1, the write latency increases by one clock cycle; the read latency increases by one clock cycle (including checking and correction).

A partial write results in a read followed by write in the ECC controller, so latency depends on the time the controller takes to fetch the data from the particular address.

Table 4-3 shows the relationship between burst lengths and rate.

Table 4-3. Burst Lengths and Rates

Local Burst Length	Rate	Memory Burst Length
1	Half	4
2	Full	4

Local Burst Length 2

For a local burst length of 2, the write latency increases by two clock cycles; the read latency increases by one clock cycle (including checking and correction).

A partial write results in a read followed by write in the ECC controller, so latency depends on the time the controller takes to fetch the data from the particular address.

For a single-bit error, the automatic correction of memory takes place without stalling the read cycle (if enabled), which stalls further commands to the ECC controller, while the correction takes place.

Example Design

The MegaWizard Plug-In Manager helps you create an example design that shows you how to instantiate and connect the DDR or DDR2 SDRAM high-performance controller. The example design consists of the DDR or DDR2 SDRAM high-performance controller, some driver logic to issue read and write requests to the controller, a PLL to create the necessary clocks, and a DLL (Stratix series only). The example design is a working system that you can compile and use for both static timing checks and board tests.

Figure 4-4 shows the testbench and the example design.

Figure 4-4. Testbench and Example Design

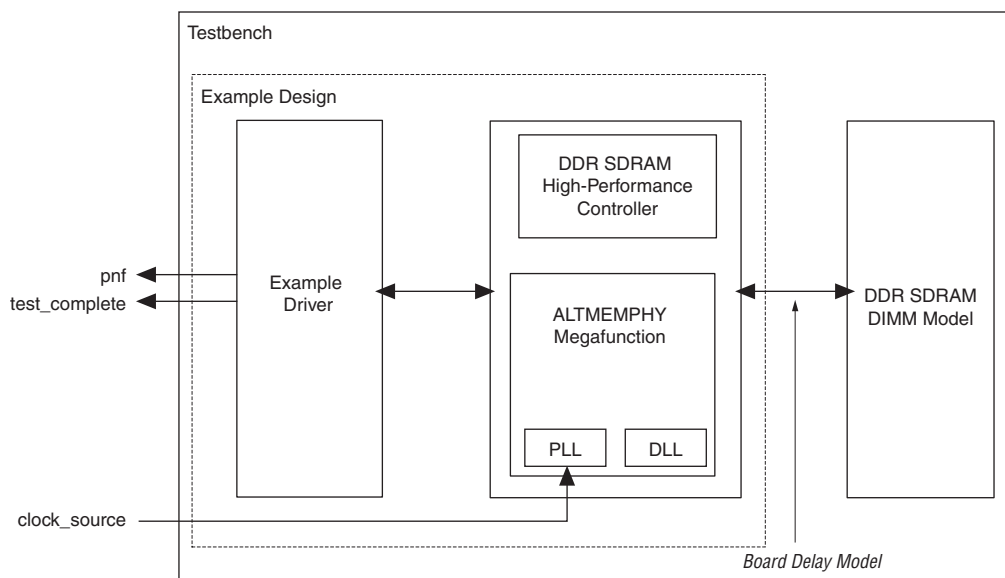


Table 4-4 describes the files that are associated with the example design and the testbench.

Table 4-4. Example Design and Testbench Files

Filename	Description
<variation name>_example_top_tb.v or .vhd	Testbench for the example design.
<variation name>_example_top.v or .vhd	Example design.
<variation name>_mem_model.v or .vhd	Memory model.
<variation name>_example_driver.v or .vhd	Example driver.
<variation name>.v or .vhd	Top-level description of the custom MegaCore function.
<variation name>.qip	Contains Quartus II project information for your MegaCore function variations.

There are two Altera-generated memory models available—associative-array memory model and full-array memory model.

The associative-array memory model (<variation name>_mem_model.v) allocates reduced set of memory addresses with a default depth of 2,048 or 2K address spaces. This allocation allows for a larger memory array compilation and simulation which enables you to easily reconfigure the depth of the associate array.

The full-array memory model (<variation name>_mem_model_full.v) allocates memory for all addresses accessible by the DDR cores. This allocation makes it impossible to simulate large memory (more than 2K address spaces) designs, because simulators need more memory than what is available on a typical system.



The memory model, <variation name>_test_component.v/vhd, used in SOPC Builder designs, is actually a variation of the full-array memory model. To ensure your simulation works in SOPC Builder, use memory model with less than 512-Mbit capacity.

Example Driver

The example driver is a self-checking test pattern generator for the memory interface. It uses a state machine to write and read from the memory to verify that the interface is operating correctly.

It performs the following tests and loops back the tests indefinitely:

- Sequential addressing writes and reads

The state machine writes pseudo-random data generated by a linear feedback shift register (LFSR) to a set of incrementing row, bank, and column addresses. The state machine then resets the LFSR, reads back the same set of addresses, and compares the data it receives against the expected data. You can adjust the length and pattern of the bursts that are written by changing the MAX_ROW, MAX_BANK, and MAX_COL constants in the example driver source code, and the entire memory space can be tested by adjusting these values. You can skip this test by setting the test_seq_addr_on signal to logic zero.

- Incomplete write operation

The state machine issues a series of write requests that are less than the maximum burst size supported by your controller variation. The addresses are then read back to ensure that the controller has issued the correct signals to the memory. This test is only applicable in full-rate mode, when the local burst size is two. You can skip this test by setting the `test_incomplete_writes_on` signal to logic zero.

- Byte enable/data mask pin operation

The state machine issues two sets of write commands, the first of which clears a range of addresses. The second set of write commands has only one byte enable bit asserted. The state machine then issues a read request to the same addresses and the data is verified. This test checks if the data mask pins are operating correctly. You can skip this test by setting the `test_dm_pin_on` signal to logic zero.

- Address pin operation

The example driver generates a series of write and read requests starting with an all-zeros pattern, a walking-one pattern, a walking-zero pattern, and ending with an all-zeros pattern. This test checks to make sure that all the individual address bits are operating correctly. You can skip this test by setting the `test_addr_pin_on` signal to logic zero.

- Low-power mode operation

The example driver requests the controller to place the memory into power-down and self-refresh states, and hold it in those states for the amount of time specified by the `COUNTER_VALUE` signal. You can vary this value to adjust the duration the memory is kept in the low-power states. This test is only available if your controller variation enables the low-power mode option.

The example driver has four outputs that allow you to observe which tests are currently running and if the tests are passing. The pass not fail (`pnf`) signal goes low once one or more errors occur and remains low. The pass not fail per byte (`pnf_per_byte`) signal goes low when there is incorrect data in a byte but goes back high again once correct data is observed in the following byte. The `test_status` signal indicates the test that is currently running, allowing you to determine which test has failed. The `test_complete` signal goes high for a single clock cycle at the end of the set of tests.

Table 4-5 shows the bit mapping for each test status.

Table 4-5. Test Status[] Bit Mapping

Bit	Test
0	Sequential address test
1	Incomplete write test
2	Data mask pin test
3	Address pin test
4	Power-down test
5	Self-refresh test
6	Auto precharge test

Interfaces and Signals

This section describes the following topics:

- “Interface Description”
- “Signals” on page 4-37

Interface Description

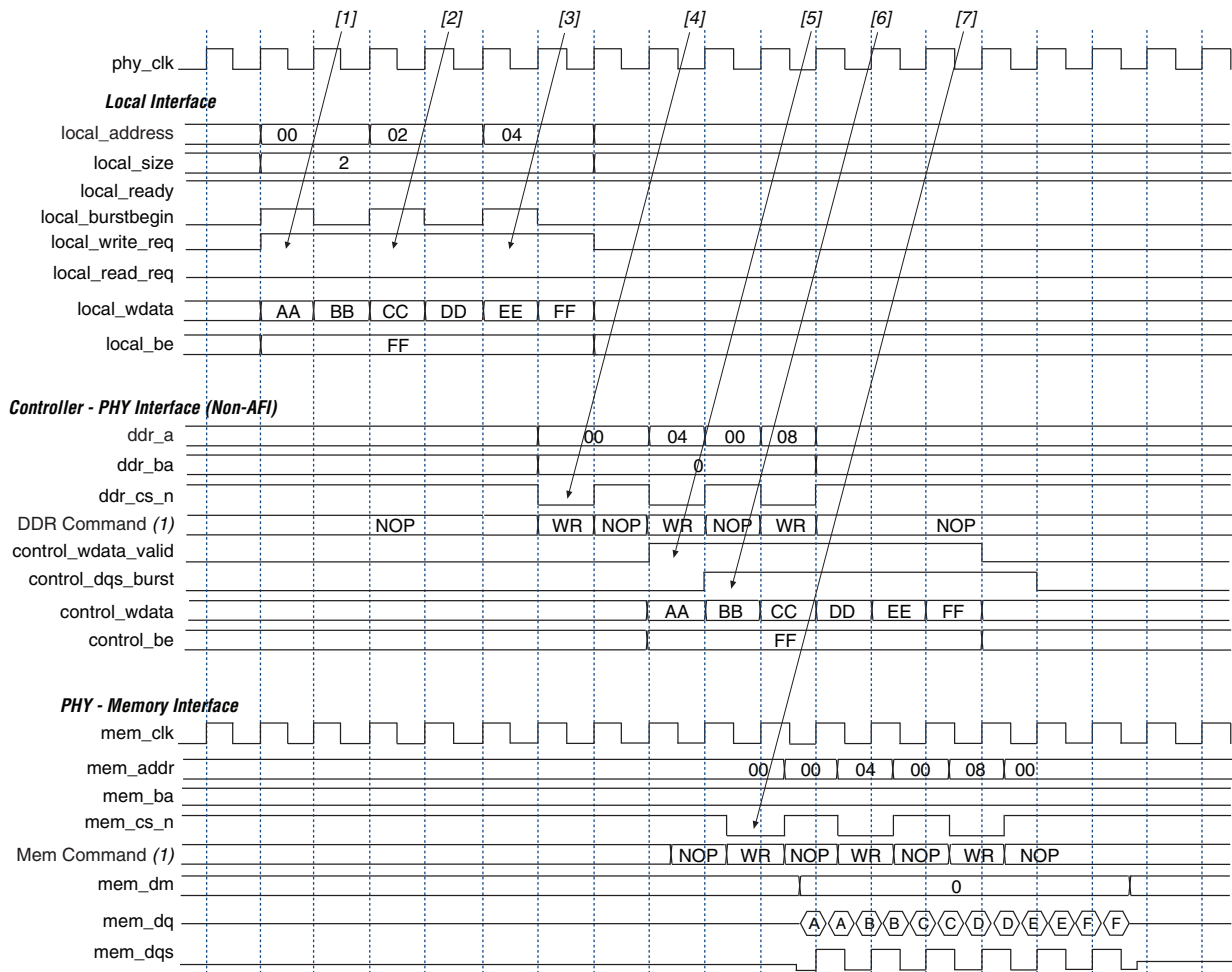
This section describes the following local-side interface requests. You can use the half-rate and full-rate modes with both Avalon-MM and native interface modes.

- “Full Rate Write, Avalon-MM Interface Mode” on page 4-14
- “Full Rate Write, Native Interface Mode—Non-Consecutive Write” on page 4-17
- “Half Rate Write, Avalon-MM Interface Mode” on page 4-20
- “Half Rate Write, Native Interface Mode” on page 4-22
- “Full Rate Read, Avalon-MM Interface Mode” on page 4-25
- “Half Rate Read, Native Interface Mode” on page 4-27
- “Half Rate Read, Avalon-MM Interface Mode—Non-Consecutive Read” on page 4-28
- “Full Rate, Native Interface Mode—Alternate Read-Write” on page 4-31
- “User Refresh Control” on page 4-34
- “Self-Refresh and Power-Down Commands” on page 4-35
- “Auto-Precharge Commands” on page 4-36

Full Rate Write, Avalon-MM Interface Mode

Figure 4-5 on page 4-15 shows write accesses with a controller in full-rate mode and using the **Local Interface Protocol** option set to **Avalon Memory-Mapped interface**. The figure shows three back-to-back write requests, each of burst size 2 to sequential addresses. In full-rate mode, the controller allows you to use burst size 1 or 2. To achieve the highest throughput, you should use bursts of size 2, which correspond to a complete memory burst of 4. Bursts of size 1 on the local interface are only half as efficient because each request still corresponds to a memory burst of size 4 but only of half of the data is used.

Figure 4-5. Full Rate Write, Avalon-MM Interface



Note to Figure 4-5:

(1) DDR Command and Mem Command show the command that the command signals are issuing.

The following sequence corresponds with the numbered items in [Figure 4–5](#) on [page 4–15](#).

1. The user logic requests the first write, by asserting the `local_write_req` signal, and the size and address for this write. In this example, the request is a burst of length 2 (4 on the memory side) to chip select 1. The `local_ready` signal is asserted, which indicates that the controller has accepted this request, and the user logic can request another read or write in the following clock cycle. If the `local_ready` signal was not asserted, the user logic must keep the write request, size, and address signals asserted until the `local_ready` signal is registered high.

 Refer to [Avalon Interface Specifications](#) for more details.

 `local_be` is active high while `mem_dm` is active low.


To map `local_wdata` and `local_be` to `mem_dq` and `mem_dm`, consider the following full rate example with 32-bit wide `local_wdata` and 16-bit wide `mem_dq`.

```
local_wdata =          <22334455>          <667788AA>          <BBCCDDEE>
local_be   =           <1100>             <0110>             <1010>
```

These values map to:

```
mem_dq =    <4455>  <2233>  <88AA>  <6677>  <DDEE>  <BBCC>
mem_dm =    <1 1>   <0 0>   <0 1>   <1 0>   <0 1>   <0 1>
```

2. The user logic requests a second write to a sequential address of size 2 (4 on the memory side). The `local_ready` signal remains asserted, which indicates that the controller has accepted the request. The address increments by the local burst size.
3. The user logic requests a third write to a sequential address, again of size 2. The controller is able to buffer up to four requests so the `local_ready` signal stays high and the request is accepted.
4. The controller issues the necessary memory command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
5. The controller asserts the `control_wdata_valid` signal to indicate to the ALTMEMPHY megafunction that valid write data and write data masks are present on the inputs to the ALTMEMPHY megafunction.
6. The controller asserts the `control_dqs_burst` signals to control the timing of the DQS signal that the ALTMEMPHY megafunction issues to the memory.

 Refer to the “Handshake Mechanism Between Write Commands and Write Data” section of the [External Memory PHY Interface Megafunction User Guide \(ALTMEMPHY\)](#) for more details of this interface.

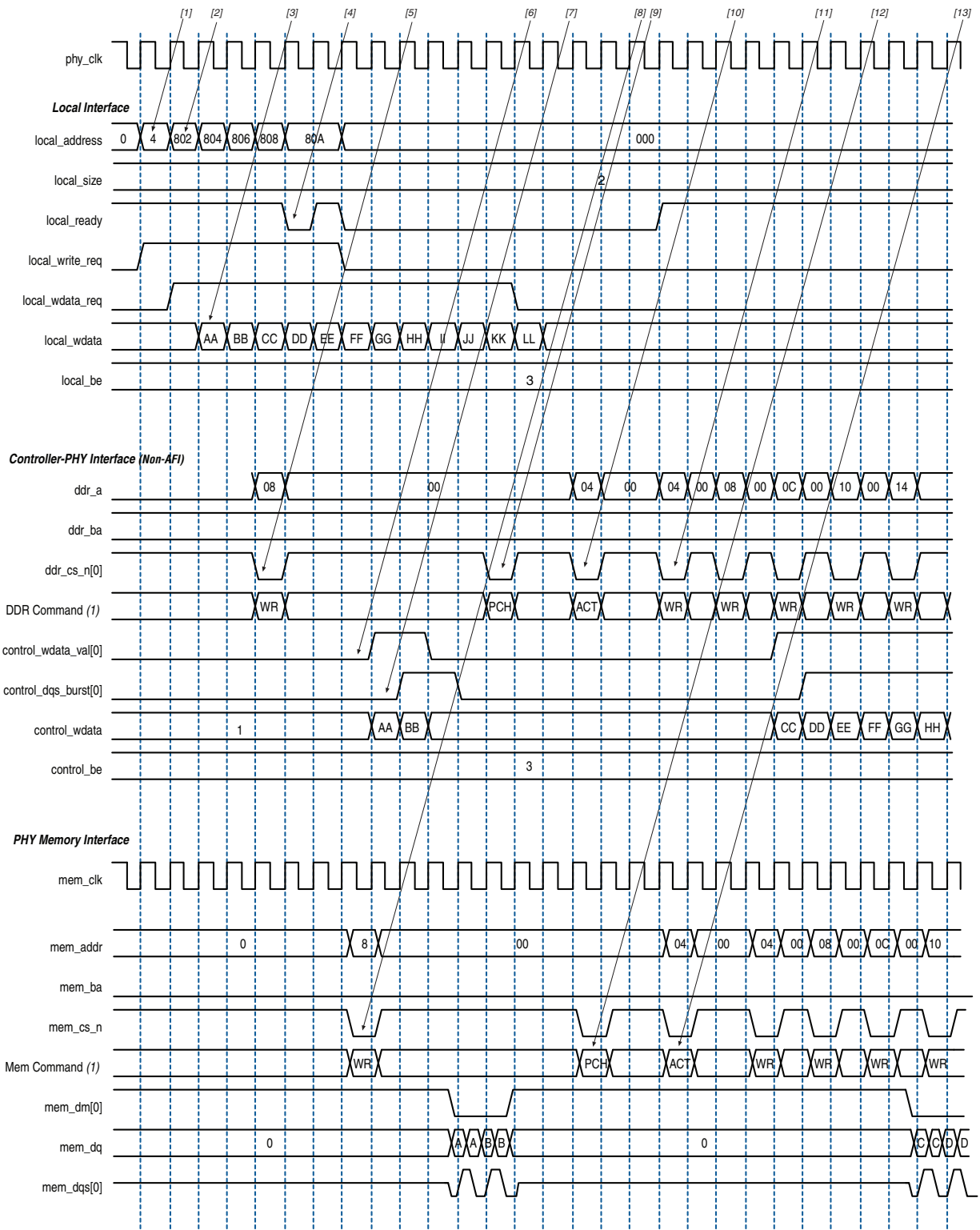
7. The ALTMEMPHY megafunction issues the write command and sends the write data and write DQS to the memory.

Full Rate Write, Native Interface Mode—Non-Consecutive Write

Figure 4-6 on page 4-18 shows write accesses with a controller in full-rate mode and using the **Local Interface Protocol** setting set to **Native interface**. The figure shows non-consecutive write-to-write requests, each of burst size 2 to sequential addresses.

In full-rate mode, the controller allows you to use burst size 1 or 2. To achieve the highest throughput, you should use bursts of size 2, which correspond to a complete memory burst of 4. Bursts of size 1 on the local interface are only half as efficient because each request still corresponds to a memory burst of size 4 but only half of the data is used.

Figure 4-6. Full Rate Write, Native Interface Mode—Non-Consecutive Write



Note to Figure 4-6:

(1) DDR Command and Mem Command show the command that the command signals are issuing.

The following sequence corresponds with the numbered items [Figure 4-6 on page 4-18](#).

1. The user logic initiates the first write by asserting `local_write_req` signal, and the size and address for this write. In this example, the request is a burst length of 2 to local address `0x000004`. This local address is mapped to the following memory address in full-rate mode.

```
mem_row_address = 0x0000
mem_col_address = 0x0004<<1 = 0x0008
mem_bank_address = 0x00
```

The `local_ready` signal is asserted, which indicates that the controller has accepted this request, and the user logic can request another read or write in the following clock cycle. If the `local_ready` signal is not asserted, the user logic must keep the write request, size, and address signals asserted until the `local_ready` signal is registered high.

2. The user logic initiates a second write to a different memory row within the same bank. The request for the second write is a burst length is 2 to local address `0x000004`. In this example, the user logic continues to request subsequent writes to addresses `0x000804`, `0x000806`, `0x000808` and `0x00080A`. The starting address, `0x000802` is mapped to the following memory address in full-rate mode.

```
mem_row_address = 0x0004
mem_col_address = 0x0002<<1 = 0x0004
mem_bank_address = 0x00
```

3. In native mode, the controller requests write data and byte enables by asserting `local_wdata_req` signal. The `local_wdata` and `local_be` signals must be asserted within one clock cycle after the `local_wdata_req` signal. In this example, the controller also continues to request write data for the subsequent writes. The user logic must be able to supply the write data for the entire burst when it requests a write.

```
First write local_wdata = <AA> <BB> to local_address = 0x000004
Second write local_wdata = <CC> <DD> to local_address = 0x000802
```

4. The controller continues to accept commands until the command queue is full. When the command queue is full, the controller deasserts the `local_ready` signal indicating that it has not accepted the command.
5. As the `local_ready` is deasserted for one clock cycle, the user logic keeps the `write_req`, `local_address`, `local_size`, and `local_wdata` signals for two clock cycles until the `local_ready` signal is asserted again.
6. The controller issues the first write memory command and column address (`0x0008`) to the `ALTMEMPHY` megafunction for it to send to the memory device.
7. The controller asserts the `control_wdata_valid` signal to indicate to the `ALTMEMPHY` megafunction that valid write data and write data masks are present on the inputs to the `ALTMEMPHY` megafunction.
8. The controller asserts the `control_dqs_burst` signals to control the timing of the `DQS` signal that the `ALTMEMPHY` megafunction issues to the memory.



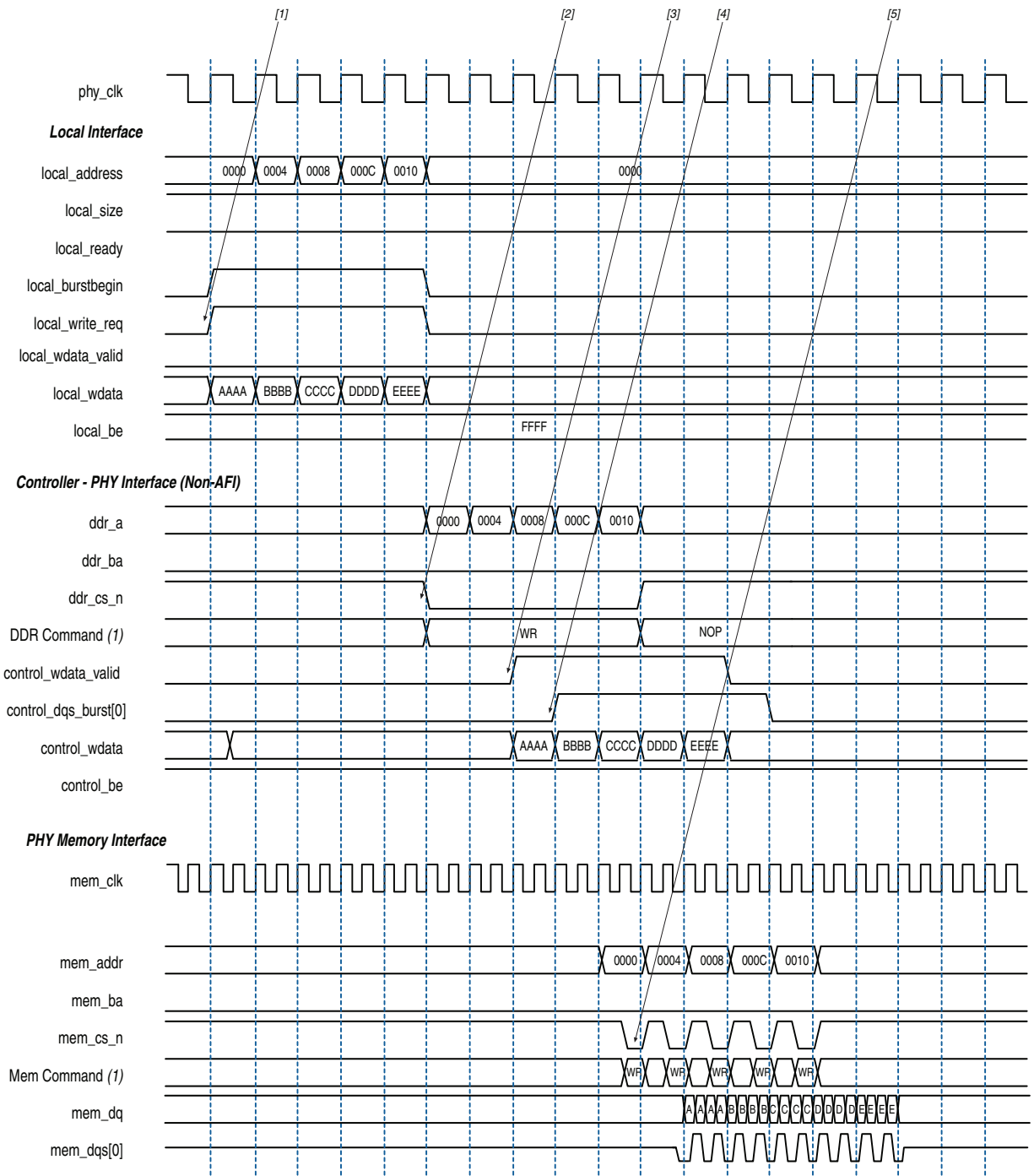
Refer to the "Handshake Mechanism Between Write Commands and Write Data" section of the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)* for more details of this interface.

9. The ALTMEMPHY megafunction issues the write command and sends the write data and write DQS to the memory.
10. The controller issues a PCH command to close current memory row (0x0000) and allow the second write to a different memory row (0x0004).
11. The controller, then issues an ACT command to open next memory row (0x0004).
12. The controller also issues the next write memory command and column address (0x0004) to the ALTMEMPHY megafunction for it to send to the memory device.
13. The ALTMEMPHY megafunction issues the PCH commands to the memory.
14. The ALTMEMPHY megafunction issues the ACT commands to the memory.

Half Rate Write, Avalon-MM Interface Mode

Figure 4-7 on page 4-21 shows write accesses with a controller in half-rate mode and using the **Local Interface Protocol** option set to **Avalon Memory-Mapped interface**. The figure shows three back-to-back write requests of the same burst size. In half-rate mode, the controller allows you to use burst size 1, which corresponds to a complete memory burst of 4.

Figure 4-7. Half Rate Write, Avalon-MM Interface Mode




Note to Figure 4-7:

(1) DDR Command and Mem Command show the command that the command signals are issuing.


The following sequence corresponds with the numbered items in [Figure 4-7 on page 4-21](#).

1. The user logic requests the first write, by asserting the `local_write_req` signal, and the size and address for this write. In this example, the request is a burst of length 1 (4 on the memory side) to chip select 1. The `local_ready` signal is asserted, which indicates that the controller has accepted this request, and the user logic can request another read or write in the following clock cycle. If the `local_ready` signal was not asserted, the user logic must keep the write request, size, and address signals asserted until the `local_ready` signal is registered high.

 Refer to *Avalon Interface Specifications* for more details.

 `local_be` is active high while `mem_dm` is active low.

2. The controller issues the necessary memory command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
3. The controller asserts the `control_wdata_valid` signal to indicate to the ALTMEMPHY megafunction that valid write data and write data masks are present on the inputs to the ALTMEMPHY megafunction.
4. The controller asserts the `control_dqs_burst` signals to control the timing of the DQS signal that the ALTMEMPHY megafunction issues to the memory.

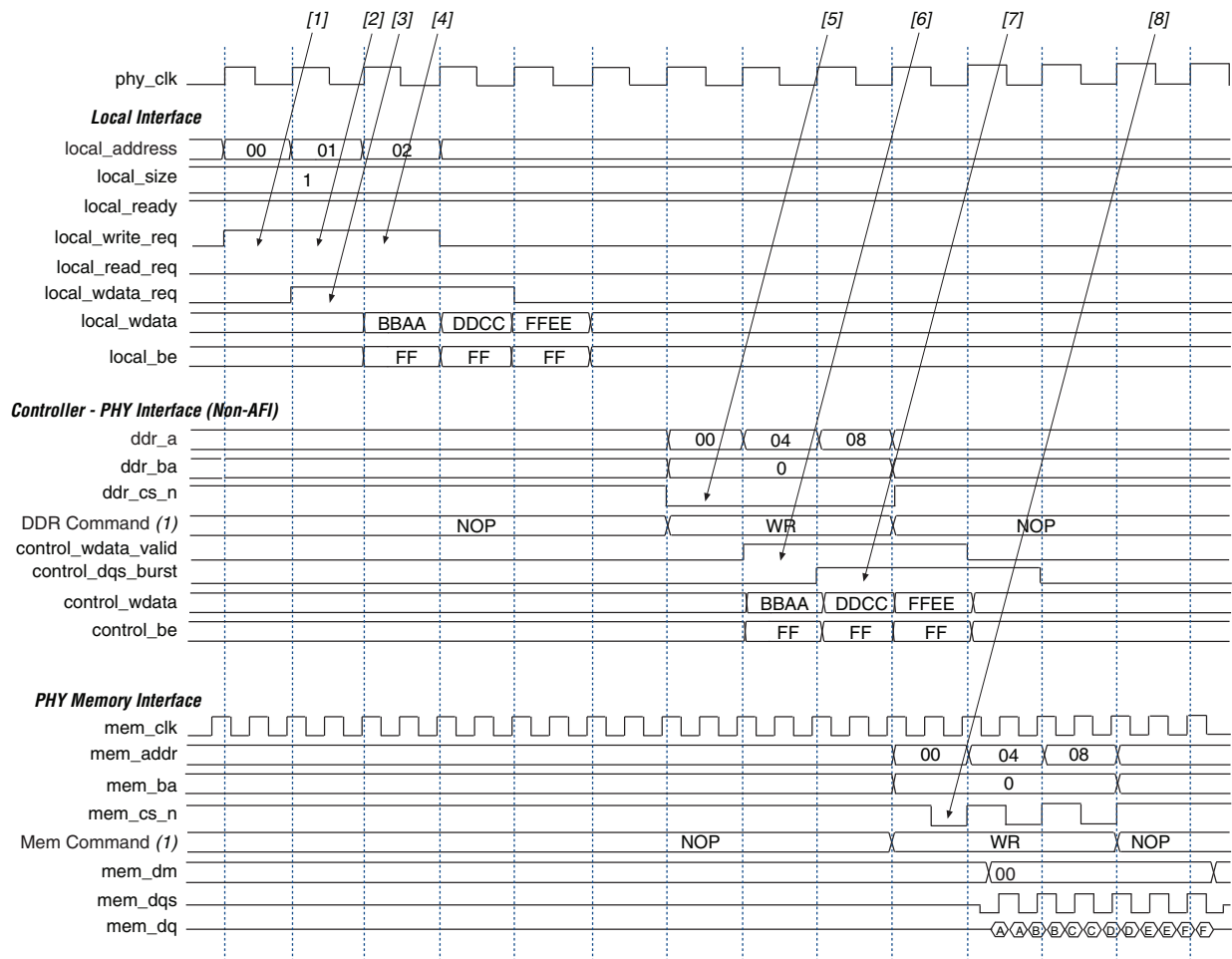
 Refer to the “Handshake Mechanism Between Write Commands and Write Data” section of the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)* for more details of this interface.

5. The ALTMEMPHY megafunction issues the write command and sends the write data and write DQS to the memory.

Half Rate Write, Native Interface Mode

[Figure 4-8 on page 4-23](#) shows write accesses with a controller in half-rate mode and using the **Local Interface Protocol** setting set to **Native interface**. The figure shows three back-to-back write requests, each of burst length 1 to sequential addresses. Each request on the native interface maps directly to a single write burst of the length of 4 on the memory side because the controller is in half-rate mode. In half-rate, the ratio between the width of the local interface write data bus and the memory data bus is 4:1.

Figure 4-8. Half Rate Write, Native Interface Mode



Note to Figure 4-8:

(1) DDR Command and Mem Command show the command that the command signals are issuing.

The following sequence corresponds with the numbered items in [Figure 4–8](#) on [page 4–23](#).

1. The user logic requests the first write by asserting the `local_write_req` signal, and the size and address for this write. In this example, the request is a burst of length 1 (4 on the memory side) address 0. The `local_ready` signal is asserted, which indicates that the controller has accepted this request, and the user logic can request another read or write in the following clock cycle. If the `local_ready` signal was not asserted, the user logic must keep the write request, size, and address signals asserted until the `local_ready` signal is registered high.

```
local_wdata =      <22334455>      <667788AA>      <BBCCDDEE>
local_be  =      <1100>          <0110>          <1010>
```

These values map to::

```
mem_dq = <55> <44> <33> <22> <AA> <88> <77> <66> <EE> <DD> <CC> <BB>
mem_dm = <1> <1> <0> <0> <1> <0> <0> <1> <1> <0> <1> <1>
```

2. The user logic requests a second write to a sequential address of size 1 (4 on the memory side). The `local_ready` signal remains asserted, which indicates that the controller has accepted the request. The address increments by the local burst size.
3. The controller requests the write data and byte enables for the write from the user logic. The write data and byte enables must be presented in the clock cycle after the request. In this example, the controller also continues to request write data for the subsequent writes. The user logic must be able to supply the write data for the entire burst when it requests a write.
4. The user logic to a sequential address, again of size 1. The controller is able to buffer up to four requests so the `local_ready` signal stays high and the request is accepted.
5. The controller issues the necessary memory command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
6. The controller asserts the `control_wdata_valid` signal to indicate to the ALTMEMPHY megafunction that valid write data and write data masks are present on the inputs to the ALTMEMPHY megafunction.
7. The controller asserts the `control_dqs_burst` signals to control the timing of the DQS signal that the ALTMEMPHY megafunction issues to the memory.



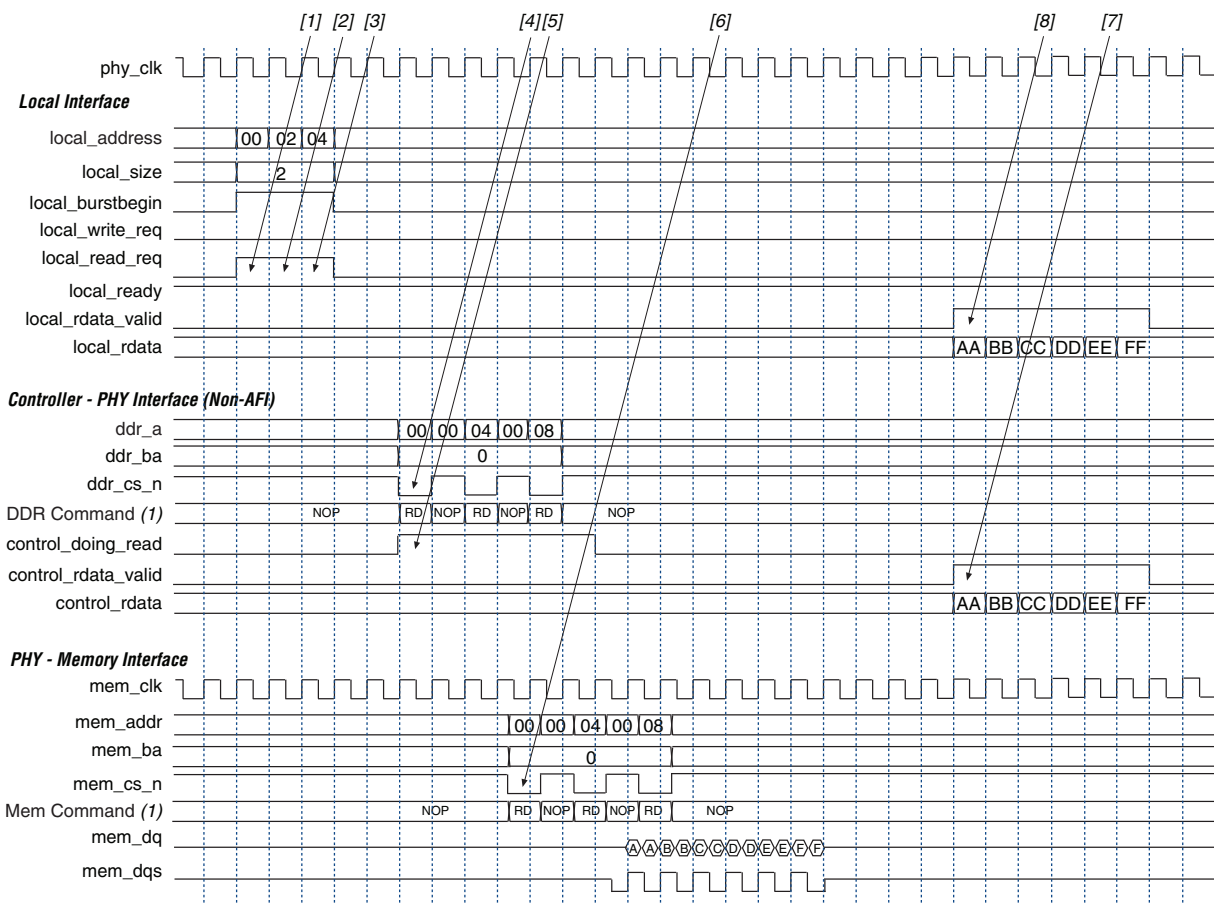
Refer to the “Handshake Mechanism Between Write Commands and Write Data” section of the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)* for more details of this interface.

8. The ALTMEMPHY megafunction issues the write command and sends the write data and write DQS to the memory.

Full Rate Read, Avalon-MM Interface Mode

Figure 4-9 shows three consecutive read requests of the same burst size. In full-rate mode, the controller allows you to use burst size 1 or 2. To achieve the highest throughput, you use bursts of 2, which correspond to a complete memory burst of 4. Bursts of size 1 on the local interface are only half as efficient because each request still corresponds to a memory burst of size 4 but only half of the data is used.

Figure 4-9. Full Rate Read, Avalon-MM Interface Mode



Note to Figure 4-9:


(1) DDR Command and Mem Command show the command that the command signals are issuing.

The following sequence corresponds with the numbered items in Figure 4-9.

1. The user logic requests the first read by asserting the read request signal, the burst begin signal, the burst size and address for this read. In this example, the request is a burst of length 2 (4 on the memory side). The local_ready signal is asserted, which indicates that the controller has accepted this request, and the user logic can request another read or write in the following clock cycle. If the local_ready signal was not asserted, the user logic must keep the read request, size, and address signals asserted. The burst begin signal does not need to be held asserted if the ready signal is not asserted.

 Refer to *Avalon Interface Specifications* for more details.

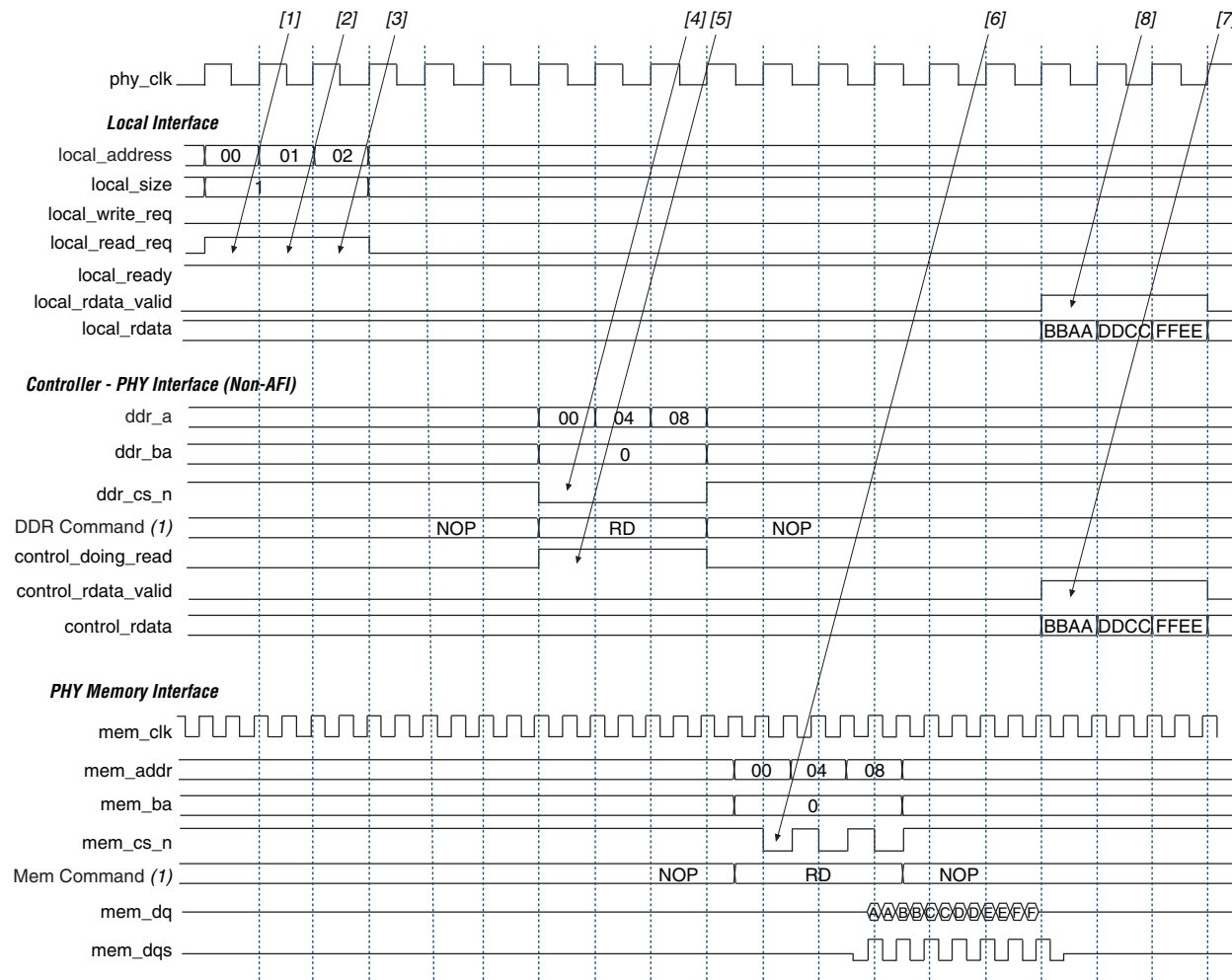
2. The user logic requests a second read to a different address, again of size 2 (4 on the memory side). The controller is able to buffer up to four requests so the `local_ready` signal stays high and the request is accepted.
3. The user logic requests a third read to a different address, of size 2 (4 on the memory side). The `local_ready` signal remains asserted, which indicates that the controller has accepted the request.
4. The controller issues the necessary memory command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
5. The controller asserts the `control_doing_rd` signal to indicate to the ALTMEMPHY megafunction how many clock cycles of read data it should expect. The ALTMEMPHY megafunction uses the `control_doing_rd` signal to enable its capture registers for the expected duration of the memory burst.

 Refer to the “Handshake Mechanism Between Read Commands and Read Data” section of the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)* for more details of this interface.
6. The ALTMEMPHY megafunction issues the read commands to the memory and captures the read data from the memory.
7. The ALTMEMPHY megafunction returns data to the controller after resynchronizing it to the `phy_clk` domain by asserting the `control_rdata_valid` signal when there is valid read data on the `control_rdata` bus.
8. The controller returns the read data to the user by asserting the `local_rdata_valid` signal when there is valid read data on the `local_rdata` bus. If **Enable error correction and detection logic** is disabled, there is no delay between the `control_rdata` and the `local_rdata` buses. If there is ECC logic in the controller, there is one or three clock cycles of delay between the `control_rdata` and `local_rdata` buses.

Half Rate Read, Native Interface Mode

Figure 4-10 on page 4-27 shows three consecutive read requests of the same burst size. In half-rate mode, the controller allows you to use burst size 1, which corresponds to a complete memory burst of 4.

Figure 4-10. Half Rate Read, Native Interface Mode




Note to Figure 4-10:

(1) DDR Command and Mem Command show the command that the command signals are issuing.

The following sequence corresponds with the numbered items in Figure 4-10.

1. The user logic requests the first read by asserting the read request signal, the burst size and address for this read. In this example, the request is a burst of length 1 (4 on the memory side). The local_ready signal is asserted, which indicates that the controller has accepted this request, and the user logic can request another read or write in the following clock cycle. If the local_ready signal was not asserted, the user logic must keep the read request, size, and address signals asserted.
2. The user logic requests a second read to a different address, again of size 1 (4 on the memory side). The controller is able to buffer up to four requests so the local_ready signal stays high and the request is accepted.

3. The user logic requests a third read to a different address, of size 1 (4 on the memory side). The `local_ready` signal remains asserted, which indicates that the controller has accepted the request.
4. The controller issues the necessary memory command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
5. The controller asserts the `control_doing_rd` signal to indicate to the ALTMEMPHY megafunction how many clock cycles of read data it should expect. The ALTMEMPHY megafunction uses the `control_doing_rd` signal to enable its capture registers for the expected duration of the memory burst.

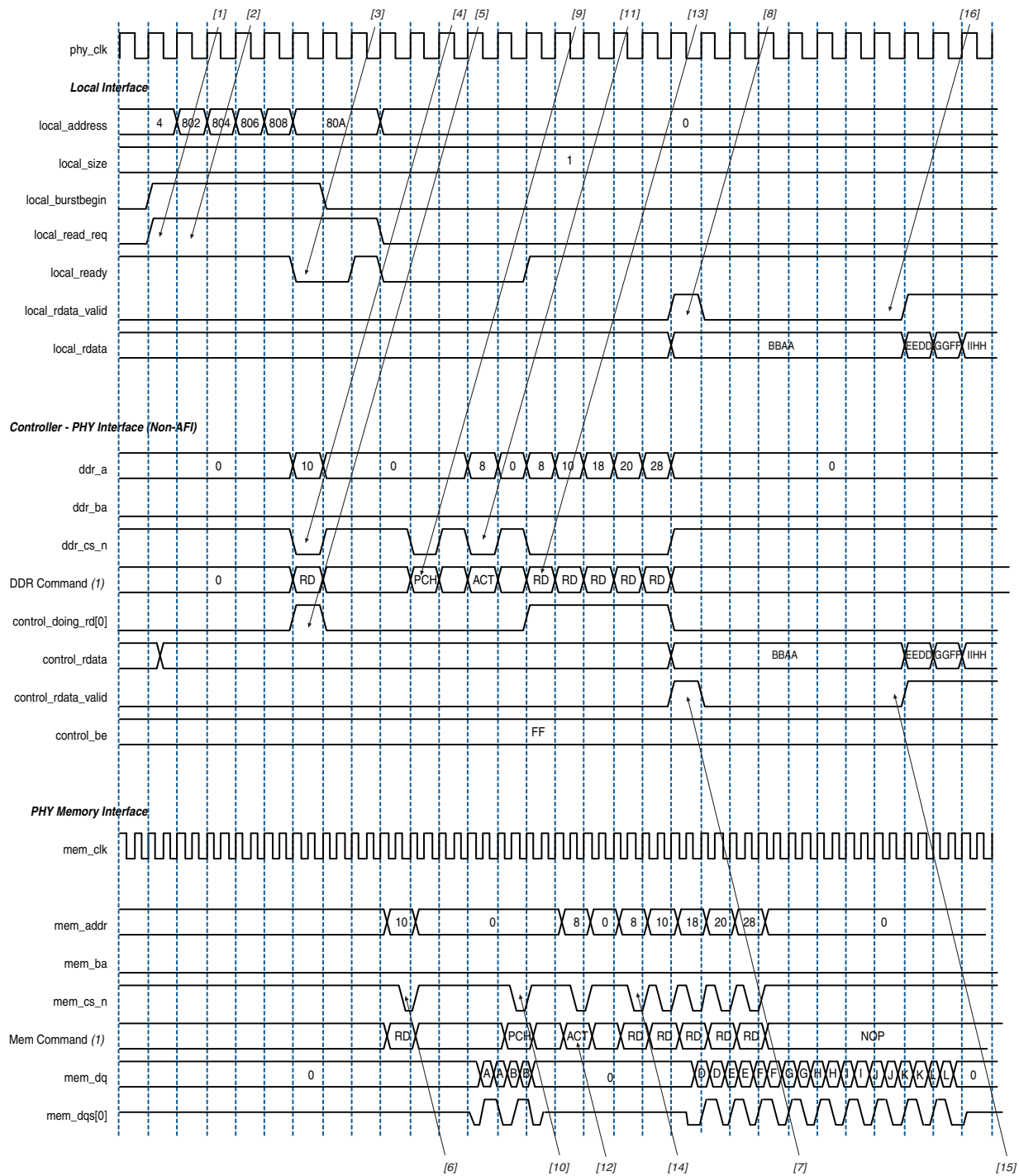
 Refer to the “Handshake Mechanism Between Read Commands and Read Data” section of the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)* for more details of this interface.

6. The ALTMEMPHY megafunction issues the read commands to the memory and captures the read data from the memory.
7. The ALTMEMPHY megafunction returns data to the controller after resynchronizing it to the `phy_clk` domain by asserting the `control_rdata_valid` signal when there is valid read data on the `control_rdata` bus.
8. The controller returns the read data to the user by asserting the `local_rdata_valid` signal when there is valid read data on the `local_rdata` bus. If **Enable error correction and detection logic** is disabled, there is no delay between the `control_rdata` and the `local_rdata` buses. If there is ECC logic in the controller, there is one or three clock cycles of delay between the `control_rdata` and `local_rdata` buses.

Half Rate Read, Avalon-MM Interface Mode—Non-Consecutive Read

Figure 4-11 on page 4-29 shows three consecutive read requests of the same burst size. In half-rate mode, the controller allows you to use burst size 1, which corresponds to a complete memory burst of 4.

Figure 4-11. Half Rate Read, Avalon-MM Interface Mode—Non-Consecutive Read



Note to Figure 4-11:

(1) DDR Command and Mem Command show the command that the command signals are issuing.

The following sequence corresponds with the numbered items in [Figure 4–11](#).

1. The user logic requests the first read by asserting `local_read_req` signal, and the size and address for this read. In this example, the request is a burst length of 1 to local address `0x000004`. This local address is mapped to the following memory address in half-rate mode:

```
mem_row_address = 0x0000
mem_col_address = 0x0004<<2 = 0x0010
mem_bank_address = 0x00
```

2. The user logic initiates a second read to a different memory row within the same bank. The request for the second write is a burst length of 1. In this example, the user logic continues to request subsequent reads to addresses `0x000804`, `0x000806`, `0x000808`, and `0x00080A`. The controller continues to accept commands until the command queue is full. When the command queue is full, the controller deasserts the `local_ready` signal. The starting address `0x000804` is mapped to the following memory address in half-rate mode:

```
mem_row_address = 0x0008
mem_col_address = 0x0002<<2 = 0x0008
mem_bank_address = 0x00
```

3. When the command queue is full, the controller deasserts the `local_ready` signal to indicate that the controller has not accepted the command. The user logic must keep the read request, size, and address signal until the `local_ready` signal is asserted again.
4. The controller issues the first read memory command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
5. The controller asserts the `control_doing_rd` signal to indicate to the ALTMEMPHY megafunction the number of clock cycles of read data it must expect for the first read. The ALTMEMPHY megafunction uses the `control_doing_rd` signal to enable its capture registers for the expected duration of memory burst.



Refer to the "Handshake Mechanism Between Read Commands and Read Data" section of the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)* for more details of this interface.

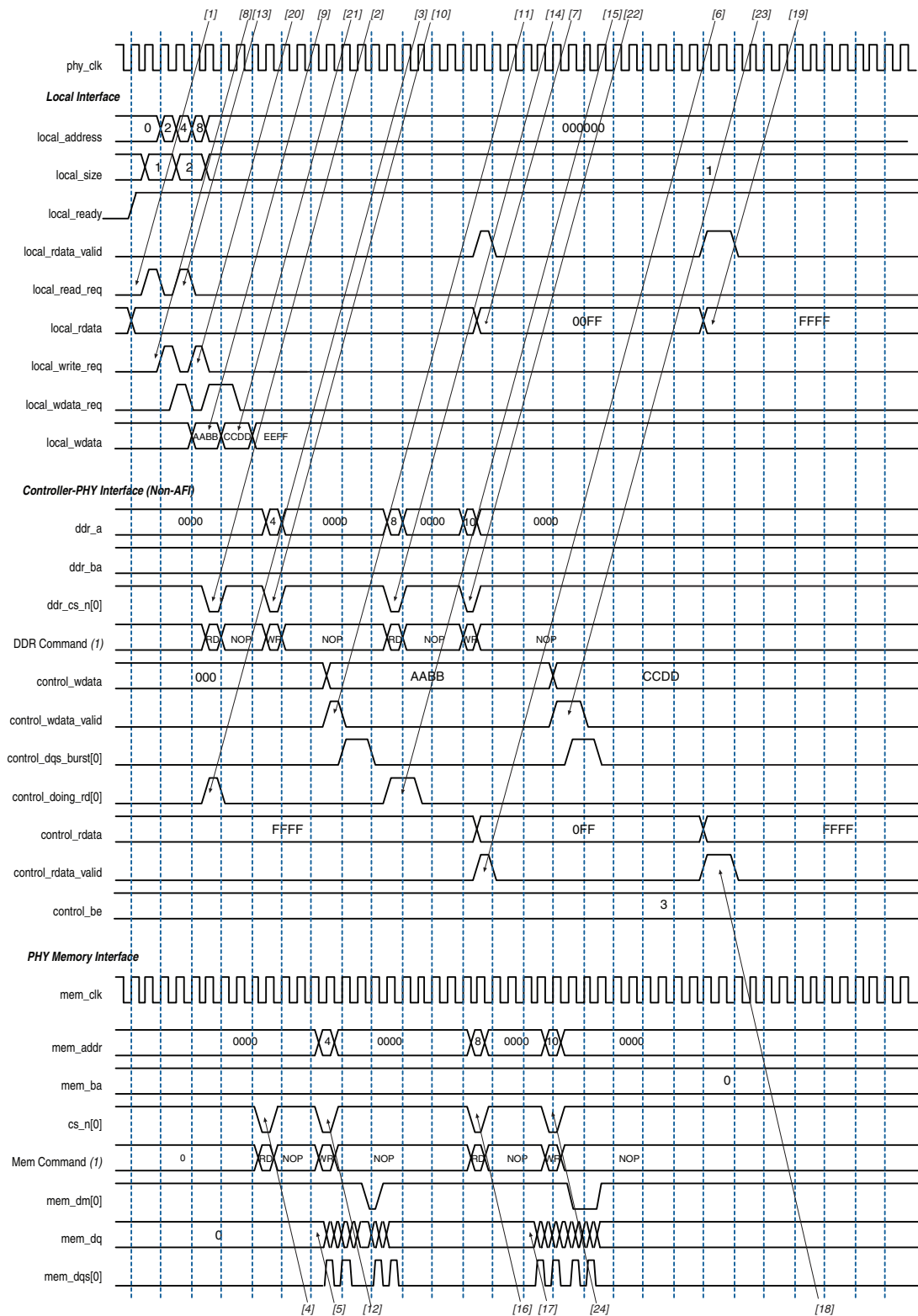
6. The ALTMEMPHY megafunction issues the first read command to the memory and captures the read data from the memory.
7. The ALTMEMPHY megafunction returns the first data read to the controller after resynchronizing the data to the `phy_clk` domain, by asserting the `control_rdata_valid` signal when there is valid read data on the `control_rdata` bus.

8. The controller returns the first read data to the user by asserting the `local_rdata_valid` signal when there is valid read data on the `local_rdata` bus. If **Enable error correction and detection logic** is disabled, there is no delay between the `control_rdata` and the `local_rdata` buses. If there is ECC logic in the controller, there is one or three clock cycles of delay between the `control_rdata` and `local_rdata` buses.
9. The controller issues a PCH command to close current memory row (0x0000) and allow the second read to a different memory row (0x0008).
10. The ALTMEMPHY megafunction issues the PCH commands to the memory.
11. The controller issues an ACT command to open the next memory row (0x0008).
12. The ALTMEMPHY megafunction issues the ACT commands to the memory.
13. The controller issues the second read memory command and column address (0x0008) to the ALTMEMPHY megafunction for it to send to the memory device.
14. The ALTMEMPHY megafunction issues the read commands to the memory.
15. The controller returns the second read data to the user by asserting the `local_rdata_valid` signal when there is valid read data on the `local_rdata` bus.

Full Rate, Native Interface Mode—Alternate Read-Write

Figure 4-12 on page 4-32 shows read, write, read, write operation in full-rate mode and using the **Local Interface Protocol** setting set to **Native interface**.

Figure 4-12. Full Rate, Read-Write (Size 1), Read-Write (Size 2) Native Interface Mode



Note to Figure 4-12:

(1) DDR Command and Mem Command show the command that the command signals are issuing.

The following sequence corresponds with the numbered items in [Figure 4–12 on page 4–32](#).

1. The user logic requests the first read by asserting the read request signal. In this example, the request is a burst length of 1. The `local_ready` signal is asserted, which indicates that the controller has accepted this request, and the user logic can request another read or write in the following clock cycle.
2. The controller issues the first memory read command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
3. The controller asserts the `control_doing_rd` signal to indicate to the ALTMEMPHY megafunction how many clock cycles of read data it should expect. The ALTMEMPHY megafunction uses the `control_doing_rd` signal to enable its capture registers for the expected duration of the memory burst.



Refer to the “Handshake Mechanism Between Read Command and Read Data” section of the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)* for more details of this interface.

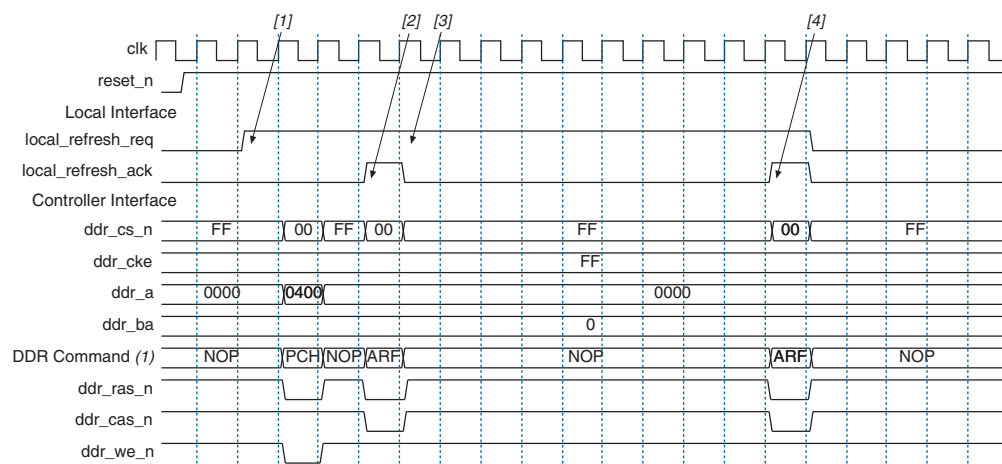
4. The ALTMEMPHY megafunction issues the first read commands to the memory and captures the read data from the memory.
5. The memory returns the first read data to the ALTMEMPHY megafunction (0xFF00).
6. The ALTMEMPHY megafunction returns the data to the controller by asserting `control_rdata_valid`.
7. The controller returns the first read data to the user logic by asserting the `local_rdata_valid` signal when there is a valid read data on the `local_rdata` bus.
8. The user logic requests the first write by asserting the write request signal. In this example, the request is a burst length of 1.
9. In native interface mode, the controller requests write data and byte enables from the user logic by asserting `local_wdata_req`. The `local_wdata` (0xAABB) and `local_be` signals must be presented within 1 clock cycle after the `local_wdata_req` is asserted.
10. The controller issues the first memory write command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
11. The controller asserts the `control_wdata_valid` signal to indicate to the ALTMEMPHY megafunction that valid write data and write data masks are present on the inputs to the ALTMEMPHY megafunction.
12. The ALTMEMPHY megafunction issues the write command and sends the first write data and write DQS to the memory.
13. The user logic requests the second read by asserting the read request signal. In this example, the request is a burst length of 2.
14. The controller issues the second memory read command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
15. The controller asserts the `control_doing_rd` signal to indicate to the ALTMEMPHY megafunction how many clock cycles of read data it should expect.

16. The ALTMEMPHY megafunction issues the second read command to the memory and captures the read data from the memory.
17. The memory returns the second read data to the ALTMEMPHY megafunction.
18. The ALTMEMPHY megafunction returns data to the controller by asserting the `control_rdata_valid` signal.
19. The controller returns the second read data to user by asserting the `local_rdata_valid` signal when there is a valid read data on the `local_rdata` bus.
20. The user logic requests the second write by asserting the write request signal. In this example, the request is a burst length of 2.
21. In native interface mode, the controller requests write data and byte enables from the user logic by asserting `local_wdata_req`. The `local_wdata` (0xCCDD) and `local_be` signals must be presented one clock cycle after `local_wdata_req` is asserted.
22. The controller issues the second memory write command and address signals to the ALTMEMPHY megafunction for it to send to the memory device.
23. The controller asserts the `control_wdata_valid` signal to indicate to the ALTMEMPHY megafunction that valid write data and write data masks are present on the inputs to the ALTMEMPHY megafunction.
24. The ALTMEMPHY megafunction issues the second write command and sends the second write data and write DQS to the memory.

User Refresh Control

Figure 4-13 shows the user refresh control interface. This feature allows you to control when the controller issues refreshes to the memory. This feature allows better control of worst case latency and allows refreshes to be issued in bursts to take advantage of idle periods.

Figure 4-13. User Refresh Control



Note to Figure 4-13:

- (1) DDR Command shows the command that the command signals are issuing.

The following sequence corresponds with the numbered items in [Figure 4-13](#).

1. The user logic asserts the refresh request signal to indicate to the controller that it should perform a refresh. The read and write requests signal do not need to be interrupted or paused in any way. If the user logic asserts `refresh_req`, the controller stops taking commands from its internal queue and services the refresh first (although the controller may have to wait a few cycles until it is legal to do the precharge-all command that comes before the refresh).



Refresh requests are higher priority requests that go straight past the command queue. If the read and write queue is not yet full, the controller accepts more commands and holds them until it starts to read or write again. As soon as the refresh operation is completed, the controller continues processing the commands in the queue.

2. The controller asserts the refresh acknowledge signal to indicate that it has sent a refresh command to the ALTMEMPHY megafunction. The exact time that the refresh command occurs on the memory interface depends on the ALTMEMPHY megafunction command output latency. This signal is still available even if the **Enable user auto-refresh controls** option is not turned on, allowing the user logic to track when the controller issues refreshes.
3. The user logic keeps the refresh request signal asserted to indicate that it wishes to perform another refresh request.

The controller again asserts the refresh acknowledge signal to indicate that it has issued a refresh. At this point the user logic deasserts the refresh request signal and the controller continues with the reads and writes in its buffers.

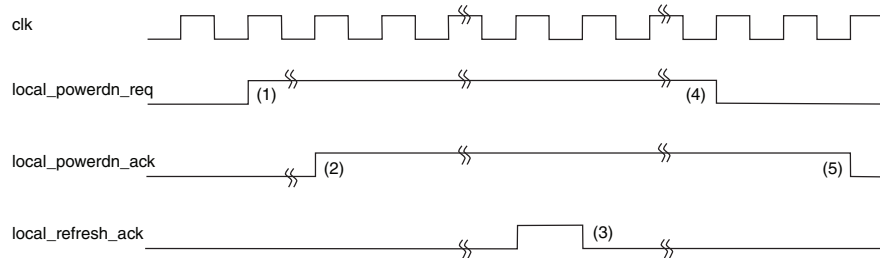
Self-Refresh and Power-Down Commands

This feature allows you to direct the controller to put the external memory device into a low-power state. There are two possible low-power states: self-refresh and power down. The controller supports both and manages the necessary memory timings to ensure that the data in the memory is maintained at all times.

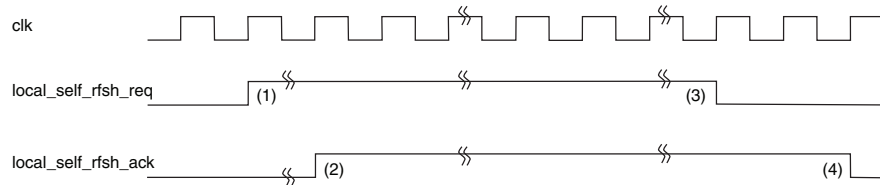
The local interface input pins (`local_powerdn_req`, and `local_self_rfsh_req`) allow you to direct the controller to place the memory device in power-down or self-refresh mode, respectively. The local interface output pins (`local_powerdn_ack`, and `local_self_rfsh_ack`) allow the controller to acknowledge the request and also indicate the current state of the memory.

If either `local_powerdn_ack` or `local_self_rfsh_ack` signal is asserted, the memory is in the relevant low-power mode. Both pairs of signals follow the same basic protocol as shown in [Figure 4-14](#) and [Figure 4-15](#) on [page 4-36](#). The self-refresh pair of signals follows the same timing and behavior as the power-down pair. The only difference is that the `local_refresh_ack` signal is not asserted in self-refresh mode as the controller does not refresh the memory when the memory is in self-refresh mode.

You must not assert both request signals at the same time. Undefined behavior occurs if both `local_powerdn_req` and `local_self_rfsh_req` are asserted simultaneously.

Figure 4-14. Power-Down Mode**Notes to Figure 4-14:**

- (1) The user synchronously asserts the request signal to indicate that the controller should put the memory into the power-down state as soon as possible.
- (2) Once the controller is able to issue the correct commands to put the memory into the power-down state, it responds by asserting the acknowledge signal.
- (3) If you direct the controller to hold the memory in power-down mode for longer than a refresh cycle, the controller wakes the memory briefly to issue a refresh command at the required time. The `local_refresh_ack` signal indicates that this has happened - it is asserted for one clock cycle at approximately the same time as the refresh command is issued. If **Enable user auto-refresh controls** is turned on, you must issue refresh requests via the `local_refresh_req` input at the appropriate time, even if you have also requested power-down mode.
- (4) The controller holds the memory in power-down mode until you deassert the request signal.
- (5) The controller deasserts the acknowledge signal once it has released the memory from the power-down state and once the required timing parameters are met.


Figure 4-15. Self-Refresh Mode**Notes to Figure 4-15:**

- (1) You synchronously assert the request signal to indicate that the controller should put the memory into the self-refresh state as soon as possible.
- (2) Once the controller is able to issue the correct commands to put the memory into the self-refresh state, it responds by asserting the acknowledge signal.
- (3) The controller holds the memory in self-refresh mode until you deassert the request signal.
- (4) The controller deasserts the acknowledge signal once it has released the memory from the self-refresh state and once the required timing parameters are met.

Auto-Precharge Commands

The auto-precharge read and auto-precharge write commands allow you to indicate to the memory device that this read or write command is the last access to the currently open row. The memory device automatically closes (auto-precharges) the page it is currently accessing so that the next access to the same bank is quicker. This command is particularly useful for applications that require fast random accesses.

Request an auto-precharge by asserting the `local_autopch` input at the same time you assert the `local_read_req` or `local_write_req` signal. The timing and rules of the `local_autopch` input follow the basic Avalon-MM interface specifications (refer to *Avalon Interface Specifications*). You can assert it anytime, but once you have asserted it, the signal must stay asserted until the `local_ready` signal is high, which indicates that the current request has been accepted.

 If your MegaCore variation is configured to support local burst sizes greater than one, note that `local_autopch` is ignored unless you request a complete burst. It is not possible to auto-precharge a partial burst to the memory.

Signals

Table 4-6 shows the clock and reset signals.

Table 4-6. Clock and Reset Signals (Part 1 of 2)

Name	Direction	Description
<code>global_reset_n</code>	Input	The asynchronous reset input to the controller. All other reset signals are derived from resynchronized versions of this signal. This signal holds the complete ALTMEMPHY megafunction, including the PLL, in reset while low.
<code>pll_ref_clk</code>	Input	The reference clock input to PLL.
<code>phy_clk</code>	Output	The system clock that the ALTMEMPHY megafunction provides to the user. All user inputs to and outputs from the DDR high-performance controller must be synchronous to this clock.
<code>reset_phy_clk_n</code>	Output	The reset signal that the ALTMEMPHY megafunction provides to the user. It is asserted asynchronously and deasserted synchronously to <code>phy_clk</code> clock domain.
<code>aux_full_rate_clk</code>	Output	An alternative clock that the ALTMEMPHY megafunction provides to the user. This clock always runs at the same frequency as the external memory interface. In half-rate mode, this clock is twice the frequency of the <code>phy_clk</code> and can be used whenever a 2x clock is required. In full-rate mode, this clock is driven by the same PLL output as the <code>phy_clk</code> signal.
<code>aux_half_rate_clk</code>	Output	An alternative clock that the ALTMEMPHY megafunction provides to the user. This clock always runs at half the frequency as the external memory interface. In full-rate mode, this clock is half the frequency of the <code>phy_clk</code> and can be used, for example to clock the user side of a half-rate bridge. In half-rate mode, this clock is driven by the same PLL output as the <code>phy_clk</code> signal.
<code>dll_reference_clk</code>	Output	Reference clock to feed to an externally instantiated DLL.
<code>reset_request_n</code>	Output	Reset request output that indicates when the PLL outputs are not locked. Use this signal as a reset request input to any system-level reset controller you may have. This signal is always low while the PLL is locking, and so any reset logic using it is advised to detect a reset request on a falling edge rather than by level detection.
<code>soft_reset_n</code>	Input	Edge detect reset input intended for SOPC Builder use or to be controlled by other system reset logic. It is asserted to cause a complete reset to the PHY, but not to the PLL used in the PHY.
<code>oct_ctl_rs_value</code>	Input	ALTMEMPHY signal that specifies the serial termination value. Should be connected to the ALT_OCT megafunction output <code>seriesterminationcontrol</code> .

Table 4-6. Clock and Reset Signals (Part 2 of 2)

Name	Direction	Description
oct_ctl_rt_value	Input	ALTMEMPHY signal that specifies the parallel termination value. Should be connected to the ALT_OCT megafunction output <code>parallelterminationcontrol</code> .
dqs_delay_ctrl_import	Input	Allows the use of DLL in another ALTMEMPHY instance in this ALTMEMPHY instance. Connect the <code>export</code> port on the ALTMEMPHY instance with a DLL to the <code>import</code> port on the other ALTMEMPHY instance.

Table 4-7 on page 4-39 shows the DDR and DDR2 SDRAM high-performance controller local interface signals.

Table 4-7. Local Interface Signals (Part 1 of 4)


Signal Name	Direction	Description
<code>local_address []</code>	Input	<p>Memory address at which the burst should start.</p> <ul style="list-style-type: none"> ■ Full rate controllers <p>The width of this bus is sized using the following equation: For one chip select: $\text{width} = \text{bank bits} + \text{row bits} + \text{column bits} - 1$ For multiple chip selects: $\text{width} = \text{chip bits} + \text{bank bits} + \text{row bits} + \text{column bits} - 1$</p> <p>If the bank address is 2 bits wide, row is 13 bits wide and column is 10 bits wide, then the local address is 24 bits wide. To map <code>local_address</code> to bank, row and column address :</p> <pre>local_address [23:22] = bank address [1:0] local_address [21:9] = row address [13:0] local_address [8:0] = col_address [9:1]</pre> <p>The least significant bit (LSB) of the column address (multiples of four) on the memory side is ignored, because the local data width is twice that of the memory data bus width.</p> <ul style="list-style-type: none"> ■ Half rate controllers <p>The width of this bus is sized using the following equation: For one chip select: $\text{width} = \text{bank bits} + \text{row bits} + \text{column bits} - 2$ For multiple chip selects: $\text{width} = \text{chip bits} + \text{bank bits} + \text{row bits} + \text{column bits} - 2$</p> <p>If the bank address is 2 bits wide, row is 13 bits wide and column is 10 bits wide, then the local address is 23 bits wide. To map <code>local_address</code> to bank, row and column address :</p> <pre>local_address is 23 bits wide local_address [22:21] = bank address local_address [20:8] = row address [13:0] local_address [7:0] = col_address [9:2]</pre> <p>Two LSBs of the column address on the memory side are ignored, because the local data width is four times that of the memory data bus width.</p> <p> You can get the information on address mapping from the <code><variation_name>_example_top.v</code> or <code>vhd</code> file.</p>

Table 4-7. Local Interface Signals (Part 2 of 4)

Signal Name	Direction	Description
<code>local_be[]</code>	Input	<p>Byte enable signal, which you use to mask off individual bytes during writes. <code>local_be</code> is active high; <code>mem_dm</code> is active low.</p> <p>To map <code>local_wdata</code> and <code>local_be</code> to <code>mem_dq</code> and <code>mem_dm</code>, consider a full-rate design with 32-bit <code>local_wdata</code> and 16-bit <code>mem_dq</code>.</p> <pre>Local_wdata = < 22334455 >< 667788AA >< BBCCDDEE > Local_be = < 1100 >< 0110 >< 1010 ></pre> <p>These values map to:</p> <pre>Mem_dq = <4455><2233><88AA><6677><DDEE><BBCC> Mem_dm = <1 1 ><0 0 ><0 1 ><1 0 ><0 1 ><0 1 ></pre>
<code>local_burstbegin</code>	Input	<p>Avalon burst begin strobe, which indicates the beginning of an Avalon burst. This signal is only available when the local interface is an Avalon-MM interface and the memory burst length is greater than 2. Unlike all other Avalon-MM signals, the burst begin signal does not stay asserted if <code>local_ready</code> is deasserted.</p> <p>For write transactions, assert this signal at the beginning of each burst transfer and keep this signal high for one cycle per burst transfer, even if the slave has deasserted <code>local_ready</code>. After the slave deasserts <code>local_ready</code>, the master keeps all the write request signals asserted until <code>local_ready</code> becomes high again.</p> <p>For read transactions, assert this signal for one clock cycle when read request is asserted and the <code>local_address</code> from which the data should be read is given to the memory. After the slave deasserts <code>local_ready</code> (<code>waitrequest_n</code> in Avalon), the master keeps all the read request signals asserted until <code>local_ready</code> becomes high again.</p>
<code>local_read_req</code>	Input	<p>Read request signal.</p> <p>You cannot assert read request and write request signal at the same time.</p>
<code>local_refresh_req</code>	Input	<p>User controlled refresh request. If Enable user auto-refresh controls is turned on, <code>local_refresh_req</code> becomes available and you are responsible for issuing sufficient refresh requests to meet the memory requirements. This option allows complete control over when refreshes are issued to the memory including ganging together multiple refresh commands. Refresh requests take priority over read and write requests unless they are already being processed.</p>
<code>local_size[]</code>	Input	<p>Controls the number of beats in the requested read or write access to memory, encoded as a binary number. The range of values depend on the memory burst length and whether you select full or half rate in the wizard.</p> <p>If you select a memory burst length 4 and half rate, the local burst length is 1 and so <code>local_size</code> should always be driven with 1.</p> <p>If you select a memory burst length 4 and full rate, the local burst length is 2 and you should set the <code>local_size</code> to either 1 or 2 for each read or write request.</p>
<code>local_wdata[]</code>	Input	<p>Write data bus. The width of <code>local_wdata</code> is twice that of the memory data bus for a full rate controller; four times the memory data bus for a half rate controller.</p>

Table 4-7. Local Interface Signals (Part 3 of 4)

Signal Name	Direction	Description
local_write_req	Input	Write request signal. You cannot assert read request and write request signal at the same time.
local_init_done	Output	When the memory initialization, training, and calibration are complete, the ALTMEMPHY sequencer asserts the <code>ctrl_usr_mode_rdy</code> signal to the memory controller, which then asserts this signal to indicate that the memory interface is ready to be used. Read and write requests are still accepted before <code>local_init_done</code> is asserted, however they are not issued to the memory until it is safe to do so. This signal does not indicate that the calibration is successful. To find out if the calibration is successful, look for the calibration signal, <code>resynchronization_successful</code> or <code>postamble_successful</code> (for Stratix IV).
local_rdata[]	Output	Read data bus. The width of <code>local_rdata</code> is twice that of the memory data bus for a full rate controller; four times the memory data bus for a half rate controller.
local_rdata_error	Output	Asserted if the current read data has an error. This signal is only available if the Enable error detection and correction logic is turned on.
local_rdata_valid	Output	Read data valid signal. The <code>local_rdata_valid</code> signal indicates that valid data is present on the read data bus.
local_ready	Output	The <code>local_ready</code> signal indicates that the DDR or DDR2 SDRAM high-performance controller is ready to accept request signals. If <code>local_ready</code> is asserted in the clock cycle that a read or write request is asserted, that request has been accepted. The <code>local_ready</code> signal is deasserted to indicate that the DDR or DDR2 SDRAM high-performance controller cannot accept any more requests. The controller is able to buffer four read or write requests.
local_refresh_ack	Output	Refresh request acknowledge, which is asserted for one clock cycle every time a refresh is issued. Even if the Enable user auto-refresh controls option is not selected, <code>local_refresh_ack</code> still indicates to the local interface that the controller has just issued a refresh command.
local_wdata_req	Output	Write data request signal, which indicates to the local interface that it should present valid write data on the next clock edge. This signal is only required when the controller is operating in Native interface mode.
local_autopch_req	Input	User control of precharge. If Enable auto precharge control is turned on, <code>local_autopch_req</code> becomes available and you can request the controller to issue an auto-precharge write or auto-precharge read command. These commands cause the memory to issue a precharge command to the current bank at the appropriate time without an explicit precharge command from the controller. This is particularly useful if you know the current read or write is the last one you intend to issue to the currently open row. The next time you need to use that bank, the access could be quicker as the controller does not need to precharge the bank before activating the row you wish to access.

Table 4-7. Local Interface Signals (Part 4 of 4)

Signal Name	Direction	Description
local_powerdn_req	Input	User control of the power down feature. If Enable power down controls option is enabled, you can request that the controller place the memory devices into a power-down state as soon as it can without violating the relevant timing parameters and responds by asserting the local_powerdn_ack signal. You can hold the memory in the power-down state by keeping this signal asserted. The controller brings the memory out of the power-down state to issue periodic auto-refresh commands to the memory at the appropriate interval if you hold it in the power-down state. You can release the memory from the power-down state at any time by deasserting the local_powerdn_ack signal once it has successfully brought the memory out of the power-down state.
local_powerdn_ack	Output	Power-down request acknowledge signal. This signal is asserted and deasserted in response to the local_powerdn_req signal from the user.
local_self_rfsh_req	Input	User control of the self-refresh feature. If Enable self-refresh controls option is enabled, you can request that the controller place the memory devices into a self-refresh state by asserting this signal. The controller places the memory in the self-refresh state as soon as it can without violating the relevant timing parameters and responds by asserting the local_self_rfsh_ack signal. You can hold the memory in the self-refresh state by keeping this signal asserted. You can release the memory from the self-refresh state at any time by deasserting the local_self_rfsh_req signal and the controller responds by deasserting the local_self_rfsh_ack signal once it has successfully brought the memory out of the self-refresh state.
local_self_rfsh_ack	Output	Self refresh request acknowledge signal. This signal is asserted and deasserted in response to the local_self_rfsh_req signal from the user.

Table 4-8 shows the DDR and DDR2 SDRAM interface signals.

Table 4-8. DDR and DDR2 SDRAM Interface Signals (Part 1 of 2)

Signal Name	Direction	Description
mem_dq []	Bidirectional	Memory data bus. This bus is half the width of the local read and write data busses.
mem_dqs []	Bidirectional	Memory data strobe signal, which writes data into the DDR or DDR2 SDRAM and captures read data into the Altera device.
mem_clk (1)	Bidirectional	Clock for the memory device.
mem_clk_n (1)	Bidirectional	Inverted clock for the memory device.
mem_a []	Output	Memory address bus.
mem_ba []	Output	Memory bank address bus.
mem_cas_n	Output	Memory column address strobe signal.
mem_cke []	Output	Memory clock enable signals.
mem_cs_n []	Output	Memory chip select signals.
mem_dm []	Output	Memory data mask signal, which masks individual bytes during writes.
mem_odt []	Output	Memory on-die termination control signal (DDR2 SDRAM only).
mem_ras_n	Output	Memory row address strobe signal.

Table 4-8. DDR and DDR2 SDRAM Interface Signals (Part 2 of 2)

Signal Name	Direction	Description
mem_we_n	Output	Memory write enable signal.

Note to Table 4-8:

- (1) The mem_clk signals are output only signals from the FPGA. However, in the Quartus II software they must be defined as bidirectional (INOUT) I/Os to support the mimic path structure that the ALTMEMPHY megafunction uses.

Table 4-9 shows the ECC controller signals.


Table 4-9. ECC Controller Signals

Signal Name	Direction	Description
ecc_addr []	Input	Address for ECC controller.
ecc_be []	Input	ECC controller byte enable.
ecc_interrupt	Output	Interrupt from ECC controller.
ecc_rdata []	Output	Return data from ECC controller.
ecc_read_req	Input	Read request for ECC controller.
ecc_wdata []	Input	ECC controller write data.
ecc_write_req	Input	Write request for ECC controller.

The design example in this chapter shows you how to use a DDR2 SDRAM high-performance controller in non-AFI mode with a Cyclone III device, and half-rate implementation on a Windows-based system. The principles in this design example are the same for any other mode of the Altera DDR and DDR2 SDRAM high-performance ALTMEMPHY-based memory controllers.

Creating A Simulation Testbench Environment

The Megawizard Plug-In Manager automatically generates an example testbench. This flow is used as the simplest way to create a complete testbench, including an example driver, a memory controller, ALTMEMPHY megafunction, and a memory model.

 The DDR and DDR2 SDRAM High-Performance Controller MegaCore functions include the ability to generate an example testbench, whereas the megafunctions such as ALTMEMPHY do not.

Creating the Example Project

Follow the “[MegaWizard Plug-In Manager Flow](#)” on page 2–4 to create an example project targeting your chosen device family. This example uses the **EP3C40F48C6** device. However, as the example only uses the Quartus II software to generate the MegaCore variation and launch ModelSim-AE, the specific device is not important.

The example project is created in Verilog HDL although you can substitute with VHDL. Most memory vendors provide their memory models in Verilog HDL. ModelSim-AE only simulates a single HDL language at a time. The Altera “generic” memory model is more memory efficient in Verilog HDL.

Configuring the DDR2 SDRAM High-Performance Controller

Once you have created the example project, launch the Megawizard Plug-In Manager and follow these steps:

1. Expand the **Memory Controllers** folder under the **Interfaces** folder.
2. Click **DDR2 SDRAM High-Performance Controller**.
3. In the **Memory Settings** tab on the **Parameter Settings** page, under **General Settings** set the following values:
 - a. Set the **Device family** to **Cyclone III**. (This should already be default.)
 - b. Set the **Speed grade** to **6**.
 - c. Select **100 MHz** for **PLL reference clock frequency**.
 - d. Select **200 MHz** for **Memory clock frequency**.
 - e. Select **Half** for **Local interface clock frequency**.

4. Under **Show in 'Memory Presets' List**, set the following values:
 - a. Select **Micron** for **Memory Vendor**.
 - b. Select **Discrete Device** for **Memory format**.
 - c. Set **Maximum memory frequency** to **333.333 MHz**.
5. Under **Memory Presets**, select **Micron MT47H64M8CB-3**.
6. Click **Modify parameters** and in the **Preset Editor** page, select **1 pair** for the **Outlook clock pairs from FPGA**.



When specifying the **PLL reference clock frequency** and **Memory clock frequency**, it is important to set values that result in small M and N values within the PLL. For example, setting 133.33 MHz, 266.66 MHz, 333.33 MHz, or 166.67 MHz may result in smaller M and N values compared to setting 133.0 MHz, 267.0 MHz, 333.0 MHz, or 167.0 MHz

7. In the **Controller Settings** tab on the **Parameter Settings** page, under **Controller/Phy Interface Protocol**, select **non-AFI**.



Refer to the [ALTPLL Megafunction User Guide](#) for further information on PLL.

Understanding the Example Design and Testbench

The MegaWizard Plug-In Manager helps you create an example design that shows how to instantiate and connect both the DDR or DDR2 SDRAM high-performance controller, and the ALTMEMPHY megafunction. This example allows you to quickly create a working design.

The MegaCore function uses this example design in a testbench by connecting it to a generic memory model and providing the required `clock_source` and `global_reset_n` stimulus automatically.

Testbench Description

The example design consists of the following blocks or components:

- ALTMEMPHY megafunction
- memory controller
- example driver

The respective DDR and DDR2 SDRAM high-performance controllers provide a complete example of how to connect the ALTMEMPHY megafunction to a third party controller. Refer to the “*Integrating with Your Own Controller*” section of the [External Memory PHY Interface Megafunction User Guide \(ALTMEMPHY\)](#) for further information.

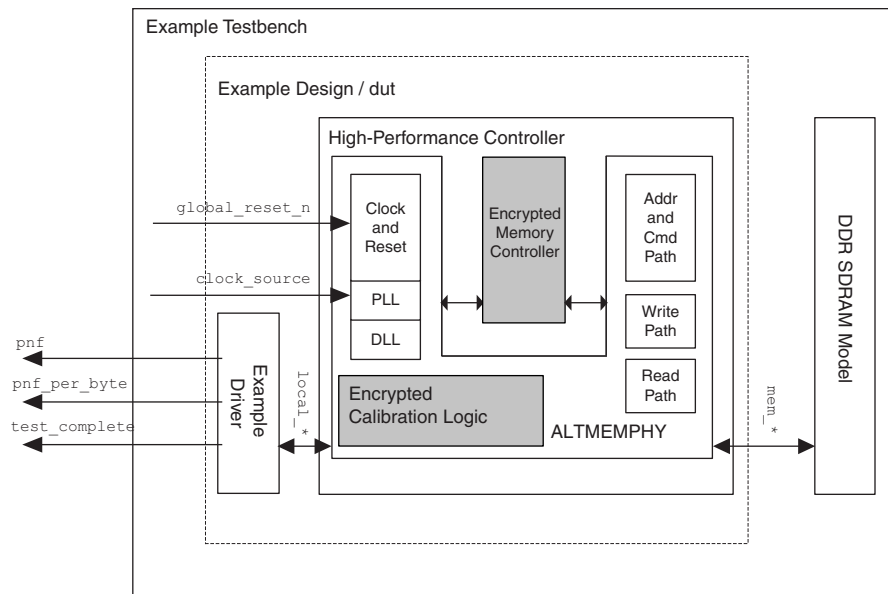
The generated example driver uses a simple LFSR structure to write data to the attached memory device (or model) and, read it back to perform a comparison between the read and write data. The example driver can be used as a placeholder for a customer specific design. It can also be used to check if your memory interface is working in hardware.

For further information refer to *AN 380: Test DDR or DDR2 SDRAM Interfaces on Hardware Using the Example Driver*.

The auto-generated generic SDRAM model may be used as a placeholder for a specific memory vendor supplied model. For information on how to replace the generic model with a vendor specific model, refer to “Perform RTL/Functional Simulation (Optional)” in *AN 328: Interfacing DDR2 SDRAM with Stratix II, Stratix II GX, and Arria GX Devices*.

Figure 5-1 on page 5-3 shows the testbench and the example design for non-AFI mode.

Figure 5-1. Example Testbench Block Diagram for non-AFI mode.



Running the Example Testbench from Your Simulator


After you generate the testbench, you can run it directly from your simulation tool.

Before running a simulation directly from your simulation tool, you must run the simulation once from the Quartus II software to generate the *.do ModelSim file. To do this, click **Run EDA Simulation Tool** on the Tools menu and select **EDA RTL Simulation**.

You can follow these steps to run the simulation from your simulation tool:

1. Launch **ModelSim-Altera**.
2. On the File menu, click **Change Directory**.
3. Select <your project name>/**simulation/modelsim** and click **OK**.
4. On the Tools menu, click **Execute Macro**.
5. Select <your project name>_run_msim_rtl_verilog.do and click **OK**.

- ModelSim-AE includes all Altera device libraries; so a .do script for ModelSim-AE does not compile these libraries. NativeLink includes the relevant libraries for other simulators.

 Refer to *Simulation and Verification Support Resources* if you use other Altera-supported RTL simulation tools.


The Testbench Stages

Before the user logic (example driver) can read or write to the local interface, the external SDRAM must first be initialized and calibrated. Following power-up or a reset event, the following stages of operation take place. Table 5-1 indicates where each stage takes place, depending on the controller/PHY interface selected.

Table 5-1. Stages of Operation

Stages of Operation	AFI Mode	Non-AFI Mode
PLL initialization and lock	PHY	PHY
Memory device initialization	PHY	Controller
Interface training and calibration	PHY	PHY
Functional memory use	Controller	Controller

The following sections discuss the stages that take place in the controller.

 For more information for operations that take place in the PHY, refer to the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)*.

Memory Device Initialization

In non-AFI mode, memory devices must be initialized before functional use. The exact sequence is different for DDR2 and DDR. The memory controller sets the operating parameters of the memory based on the parameters you specify in the MegaWizard interface. This parameter is fixed at generation time and is not dynamically editable via the local interface.

Figure 5-2 on page 5-6 shows the memory initialization stage which is dominated by the NOP command where t_{INIT} is 200 μ s. The controller automatically skips t_{INIT} in simulation.

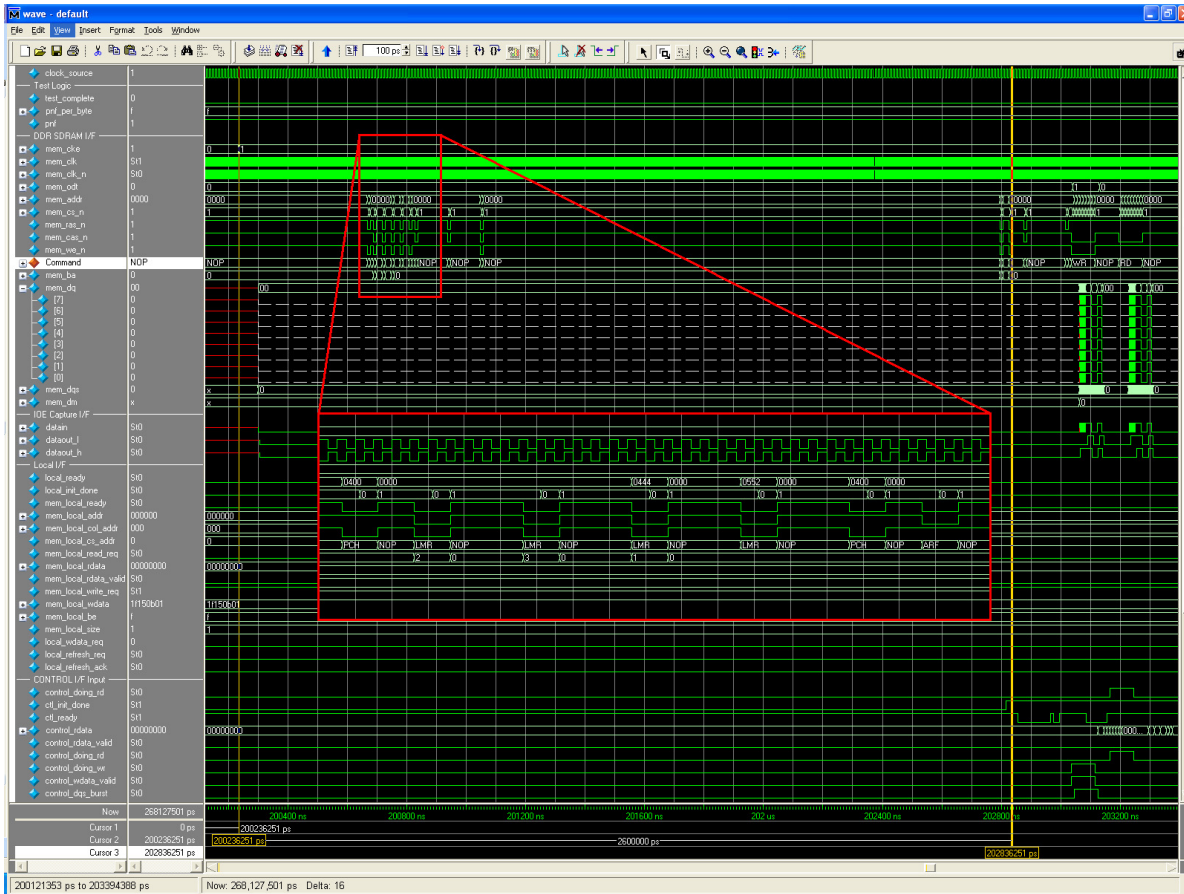
The exact sequence of commands differs between the various external memory families (refer to the respective the device datasheets for further information). For this DDR2 SDRAM example, the following sequence applies:

- Issue NOP commands for 200 μ s, programmable via t_{INIT} parameter.
- Assert mem_cke (high).
- Issue a PCH, then wait for 400 ns after t_{INIT} (400 ns is derived from dividing t_{INIT} counter by 500).
- Issue an LMR command to ELMR register 2 = 0.
- Issue an LMR command to ELMR register 3 = 0.

6. Issue an LMR command to ELMR register to enable the memory DLL and set Drive strength, AL, RTT, DQS#, RDQS, OE.
7. Issue an LMR command to MR register to reset DLL and set operating parameters.
8. Issue a PCH.
9. Issue an ARF.
10. Issue another ARF.
11. Issue an LMR command to MR register to set operating parameters.
12. Issue an LMR command to ELMR register to set default OCD and parameters. 200 clock cycles after DLL reset, the memory is initialized.

In [Figure 5-2](#), the expected waveform view of the initialization phase is directly following the NOP of 200 μ s. Steps 2 to 9 are expanded to increase detail. Initialization is complete by the second yellow cursor. Additional signals are added to simplify debugging.

Figure 5-2. Simulation Initialization Phase



Functional Memory Use

Once training and calibration are complete, the ALTMEMPHY sequencer asserts `seq_cal_complete` (AFI mode) or `ctrl_usr_mode_rdy` (non-AFI mode) to the memory controller, which is then copied to the local interface as the signal `local_init_done`. Local interface read and write transactions can now occur.

In the example testbench, the example driver now performs 16 writes followed by 16 reads to incremental address locations spanning column, row and bank locations using LFSR pattern based on the address being written.

Adding the following controller signals to your simulation provides you more information on the example driver operation:

```
clock_source
global_reset_n
test_complete
pnf
pnf_per_byte
mem_local_init_done
mem_local_ready
mem_local_addr
mem_local_col_addr
mem_local_cs_addr
mem_local_read_req
mem_local_rdata
mem_local_rdata_valid
mem_local_write_req
mem_local_wdata
mem_local_be
mem_local_size
mem_local_wdata_req (Native interface only)
mem_local_burstbegin (Avalon-MM interface only)
```



For external memory interface signals, refer to the *External Memory PHY Interface Megafunction User Guide (ALTMEMPHY)*.

You can use the example driver to test a custom controller and ALTMEMPHY megafunction combination. The driver performs a series of writes to the external memory, followed by a series of reads to the same locations, and compares the read and write data.

This comparison results in dynamic “pass not fail per byte” (`pnf_per_byte`) signals, and a latched combined pass not fail (`pnf`, 1=pass 0=fail) signal. Each completed series of writes and reads is signaled via the `test_complete` signal, and then the test repeats.

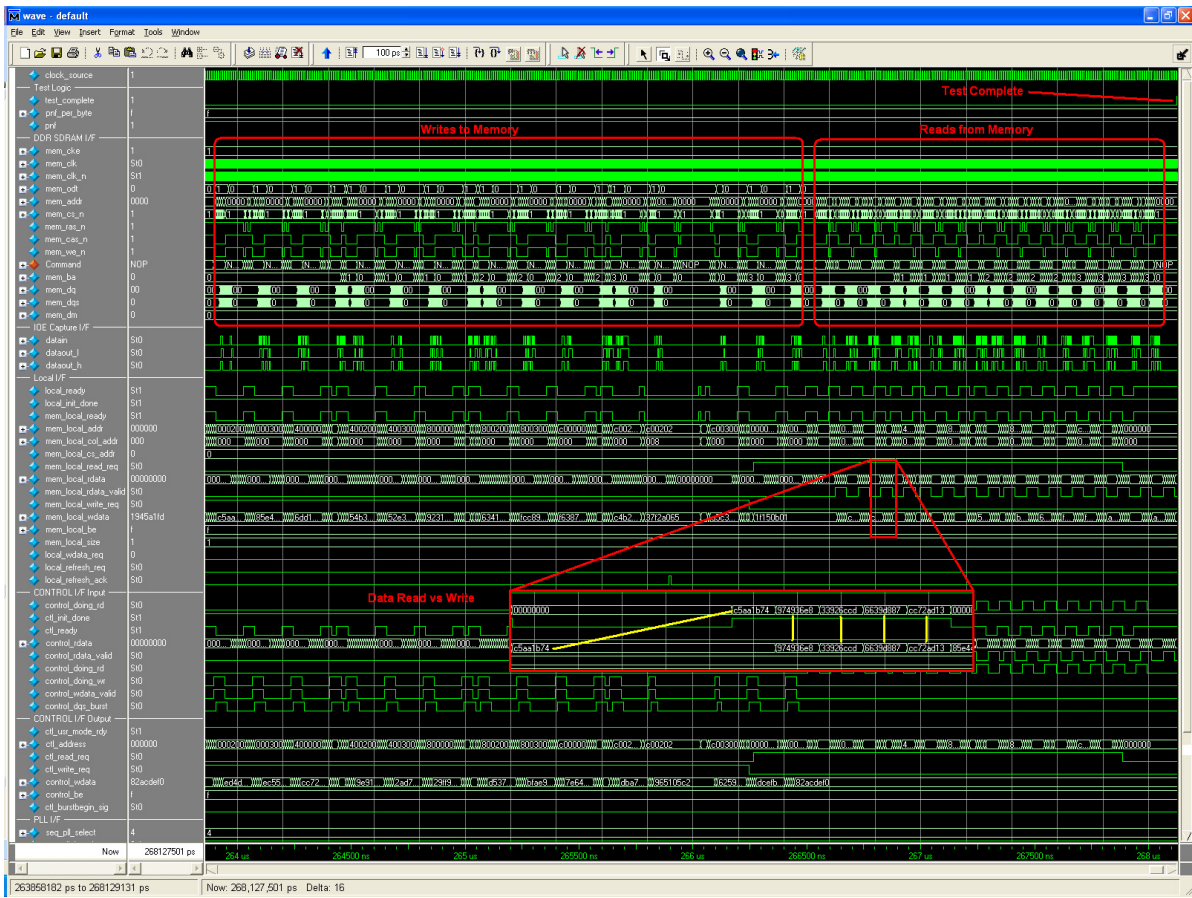


The example testbench stops when either `test_complete` is asserted or when 200,000 `mem_clk` cycles after the `tINIT` time.

Figure 5-3 on page 5-9 shows the series of writes followed by reads on both the local and memory interfaces, together with the test complete signals.

As the data written to the memory is simply an LFSR pattern, the example driver is able to generate expected read data from the memory to compare with that previously written to the same address. The data on the read data bus should match that on the write data bus during the read process.

Figure 5-3. Functional Memory Use Stage



This appendix describes the ECC registers and the register bits.

ECC Registers

Table A-1 shows the ECC registers.

Table A-1. ECC Registers (Part 1 of 2)

Name	Address	Size (Bits)	Attribute	Default	Description
Control word specifications	00	32	R/W	0000000F	This register contains all commands for the ECC functioning.
Maximum single-bit error counter threshold	01	32	R/W	00000001	The single-bit error counter increments (when a single-bit error occurs) until the maximum threshold, as defined by this register. When this threshold is crossed, the ECC generates an interrupt.
Maximum double-bit error counter threshold	02	32	R/W	00000001	The double-bit error counter increments (when a double-bit error occurs) until the maximum threshold, as defined by this register. When this threshold is crossed, the ECC generates an interrupt.
Current single-bit error count	03	32	RO	00000000	The single-bit error counter increments (when a single-bit error occurs) until the maximum threshold. You can find the value of the count by reading this status register.
Current double-bit error count	04	32	RO	00000000	The double-bit error counter increments (when a double-bit error occurs) until the maximum threshold. You can find the value of the count by reading this status register.
Last or first single-bit error error address	05	32	RO	00000000	This status register stores the last single-bit error error address. It can be cleared using the control word clear. If bit 10 of the control word is set high, the first occurred address is stored.
Last or first double-bit error error address	06	32	RO	00000000	This status register stores the last double-bit error error address. It can be cleared using the control word clear. If bit 10 of the control word is set high, the first occurred address is stored.

Table A-1. ECC Registers (Part 2 of 2)

Name	Address	Size (Bits)	Attribute	Default	Description
Last single-bit error error data	07	32	RO	00000000	This status register stores the last single-bit error error data word. As the data word is an M th multiple of 64, the data word is stored in a $2N$ -deep, 32-bit wide FIFO buffer with the least significant 32-bit sub word stored first. It can be cleared individually by using the control word clear.
Last single-bit error syndrome	08	32	RO	00000000	This status register stores the last single-bit error syndrome, which specifies the location of the error bit on a 64-bit data word. As the data word is an M th multiple of 64, the syndrome is stored in a N deep, 8-bit wide FIFO buffer where each syndrome represents errors in every 64-bit part of the data word. The register gets updated with the correct syndrome depending on which part of the data word is shown on the last single-bit error error data register. It can be cleared individually by using the control word clear.
Last double-bit error error data	09	32	RO	00000000	This status register stores the last double-bit error error data word. As the data word is an M th multiple of 64, the data word is stored in a $2N$ deep, 32-bit wide FIFO buffer with the least significant 32-bit sub word stored first. It can be cleared individually by using the control word clear.
Interrupt status register	0A	5	RO	00000000	This status register stores the interrupt status in four fields (refer to Table A-3). These status bits can be cleared by writing a 1 in the respective locations.
Interrupt mask register	0B	5	WO	00000001	This register stores the interrupt mask in four fields (refer to Table A-4).
Single-bit error location status register	0C	32	R/W	00000000	This status register stores the occurrence of single-bit error for each 64-bit part of the data word in every bit (refer to Table A-5). These status bits can be cleared by writing a 1 in the respective locations.
Double-bit error location status register	0D	32	R/W	00000000	This status register stores the occurrence of double-bit error for each 64-bit part of the data word in every bit (refer to Table A-6). These status bits can be cleared by writing a 1 in the respective locations.

Register Bits

Table A-2 shows the control word specification register.

Table A-2. Control Word Specification Register

Bit	Name	Direction	Description
0	Count single-bit error	Decoder-corrector	When 1, count single-bit errors.
1	Correct single-bit error	Decoder-corrector	When 1, correct single-bit errors.
2	Double-bit error enable	Decoder-corrector	When 1, detect all double-bit errors and increment double-bit error counter.
3	Reserved	N/A	Reserved for future use.
4	Clear all status registers	Controller	When 1, clear counters single-bit error and double-bit error status registers for first and last error address.
5	Reserved	N/A	Reserved for future use.
6	Reserved	N/A	Reserved for future use.
7	Counter clear on read	Controller	When 1, enables counters to clear on read feature.
8	Corrupt ECC enable	Controller	When 1, enables deliberate ECC corruption during encoding, to test the ECC.
9	ECC corruption type	Controller	When 0, creates single-bit errors in all ECC codewords; when 1, creates double-bit errors in all ECC codewords.
10	First or last error	Controller	When 1, stores the first error address rather than the last error address of single-bit error or double-bit error.
11	Clear interrupt	Controller	When 1, clears the interrupt.

Table A-3 shows the interrupt status register.

Table A-3. Interrupt Status Register

Bit	Name	Description
0	Single-bit error	When 1, single-bit error occurred.
1	Double-bit error	When 1, double-bit error occurred.
2	Maximum single-bit error	When 1, single-bit error maximum threshold exceeded.
3	Maximum double-bit error	When 1, double-bit error maximum threshold exceeded.
4	Double-bit error during read-modify-write	When 1, double-bit error occurred during a read modify write condition. (partial write).
Others	Reserved	Reserved.

Table A-4 shows the interrupt mask register.

Table A-4. Interrupt Mask Register

Bit	Name	Description
0	Single-bit error	When 1, masks single-bit error.
1	Double-bit error	When 1, masks double-bit error.
2	Maximum single-bit error	When 1, masks single-bit error maximum threshold exceeding condition.
3	Maximum double-bit error	When 1, masks double-bit error maximum threshold exceeding condition.
4	Double-bit error during read-modify-write	When 1, masks interrupt when double-bit error occurs during a read-modify-write condition. (partial write).
Others	Reserved	Reserved.

Table A-5 shows the single-bit error location status register.

Table A-5. Single-Bit Error Location Status Register

Bit	Name	Description
Bits $N-1$ down to 0	Interrupt	When 0, no single-bit error; when 1, single-bit error occurred in this 64-bit part.
Others	Reserved	Reserved.

Table A-6 shows the double-bit error location status register.

Table A-6. Double-Bit Error Location Status Register

Bit	Name	Description
Bits $N-1$ down to 0	Cause of Interrupt	When 0, no double-bit error; when 1, double-bit error occurred in this 64-bit part.
Others	Reserved	Reserved.

Revision History

The following table shows the revision history for this user guide.

Date	Version	Changes Made
March 2009	9.0	<ul style="list-style-type: none"> ■ Updated device support for Arria II GX, HardCopy III and HardCopy IV E. ■ Updated information on new features. ■ Updated new half rate write waveform for Avalon-MM Interface. ■ Added new chapter—Chapter 5, Example Design Walkthrough. ■ Removed Appendix B and Appendix C.
November 2008	8.1	<ul style="list-style-type: none"> ■ Updated new read and write waveforms. ■ Updated section on Example Driver. ■ Added new section on DDR/DDR2 SDRAM High-Performance Controller Architecture. ■ Added new section on Simulating With Other Simulators - VHDL/Verilog HDL IP Functional Simulation.
May 2008	8.0	<ul style="list-style-type: none"> ■ Updated new read and write waveforms. ■ Added new simulation appendix (Appendix B). ■ Added more detailed latency information (Appendix C). ■ Added Stratix IV support.
October 2007	7.2	<ul style="list-style-type: none"> ■ Corrected half rate and full rate information. ■ Corrected <code>local_wdata_req</code>, <code>mem_clk</code>, and <code>mem_clk_n</code> description. ■ Updated typical latency numbers.
May 2007	7.1	<ul style="list-style-type: none"> ■ Updated device support. ■ Added description of ECC. ■ Added Appendix A.
December 2006	7.0	Added Cyclone III support.
December 2006	6.1	First release.

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact <i>(Note 1)</i>	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com






Contact <i>(Note 1)</i> (Software Licensing)	Contact Method	Address
	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, Save As dialog box.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, <code>\qdesigns</code> directory, <code>d:</code> drive, and <code>chiptrip.gdf</code> file.
<i>Italic Type with Initial Capital Letters</i>	Indicates document titles. For example, <i>AN 519: Stratix IV Design Guidelines</i> .
<i>Italic type</i>	Indicates variables. For example, $n + 1$. Variable names are enclosed in angle brackets (<>). For example, <file name> and <project name>. <code>.pof</code> file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key and the Options menu.
“Subheading Title”	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, “Typographic Conventions.”
Courier type	Indicates signal, port, register, bit, block, and primitive names. For example, <code>data1</code> , <code>t_{di}</code> , and <code>input</code> . Active-low signals are denoted by suffix <code>n</code> . For example, <code>resetn</code> . Indicates command line commands and anything that must be typed exactly as it appears. For example, <code>c:\qdesigns\tutorial\chiptrip.gdf</code> . Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword <code>SUBDESIGN</code>), and logic function names (for example, <code>TRI</code>).
1., 2., 3., and a., b., c., and so on.	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The angled arrow instructs you to press Enter.
	The feet direct you to more information about a particular topic.