DDR3 SDRAM High-Performance

Controller User Guide

I.S. EN ISO 9001

Contents

Chapter 1. About This MegaCore Function

Chapter 2. Getting Started

Chapter 3. Parameter Settings

Chapter 4. Functional Description

Additional Information

Appendix A. ECC Register Description

Appendix B. Latency

1. About This MegaCore Function

Release Information

Table 1–1 provides information about this release of the DDR3 SDRAM High-Performance Controller MegaCore® functions.

Table 1–1. DDR3 SDRAM High-Performance Controller Release Information	
Item	**Description**
Version	8.0
Release Date	May 2008
Ordering Codes	IP-SDRAM/DDR3
Product IDs	00C2 00CO (altmemphy Megafunction)
Vendor ID	6AF7

Altera verfies 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 DDR3 SDRAM High-Performance controller to each of the Altera device families.

Table 1–2. Device Family Support	
Device Family	**Support**
Stratix® III	Preliminary
Stratix IV	Preliminary
Other device families	No support

Features

- Integrated error correction coding (ECC) function
- Power-up calibrated on-chip termination (OCT) support
- Half-rate support for Stratix III and IV devices
- SOPC Builder ready
- Automatically generated memory simulation model simplifies simulation flow
- Support for altmemphy megafunction
- Support for industry-standard DDR3 SDRAM devices and modules
- 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
- Easy-to-use MegaWizard® interface
- Support for OpenCore Plus evaluation
- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators

General Description

The Altera DDR3 SDRAM High-Performance Controller MegaCore functions provide simplified interfaces to industry-standard DDR3 SDRAM. The MegaCore functions work in conjunction with the Altera altmemphy megafunction.

For more information on the altmemphy megafunction, refer to the *External DDR 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 DDR3 SDRAM High-Performance Controller MegaCore functions create for you.

Figure 1–1. System-Level Diagram
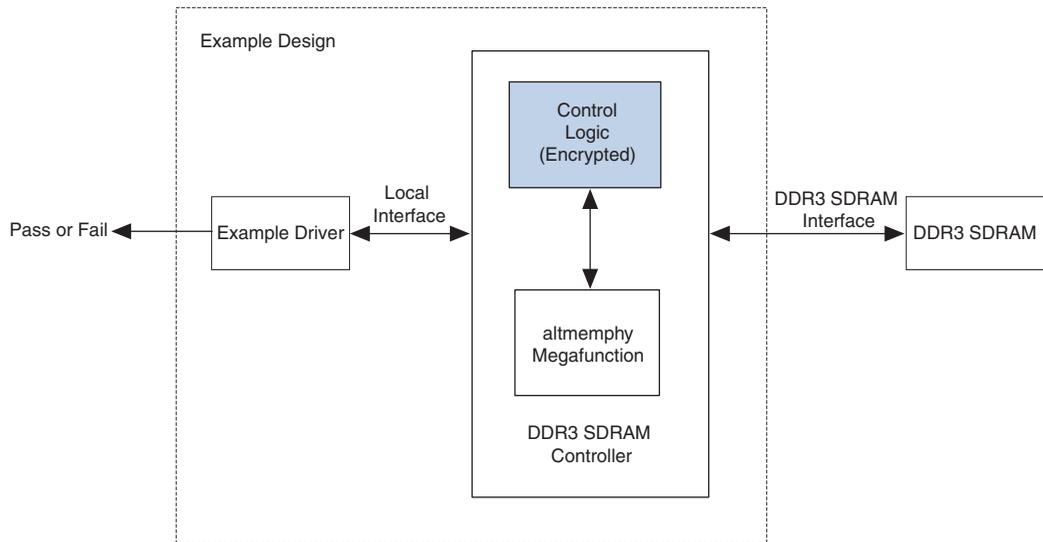
The MegaWizard Plug-In Manager generates an example design that instantiates an example driver and your DDR3 SDRAM high-performance controller custom variation. The controller instantiates an instance of the altmemphy megafunction which in turn instantiates a PLL and DLL. You can optionally instantiate the DLL outside the altmemphy megafunction in order 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 performs extensive random, directed tests with functional test coverage using industry-standard Denali models to ensure the functionality of the DDR3 SDRAM high-performance controller. In addition, Altera has carried out a wide variety of gate-level tests of the DDR3 SDRAM high-performance controller to verify the post-compilation functionality of the controller.

Performance and Resource Utilization

Table 1–3 shows typical performance results for the DDR3 SDRAM high-performance controller using the Quartus® II software, version 8.0.

☞ The performance of the MegaCore function in Stratix IV devices are similar to the performance in Stratix III devices..

Table 1–3. Typical Performance

Device	Rate	System f$_{MAX}$ (MHz)
Stratix III	Half Rate	400 *(1)*

Note to Table 1–3:
(1) Pending device characterization.

👣 For more information on device performance, see the relevant device handbook.

Table 1–4 shows typical sizes for the DDR3 SDRAM high-performance controller on Stratix III devices.

Table 1–4. Typical Size—Stratix III & IV Devices

Local Data Width (Bits)	Memory Width (Bits)	Combinational ALUTs	Logic Registers	Memory	
				M9K	MLAB
32	8	1,746	1,375	1	0
64	16	1,946	1,611	2	0
256	64	3,045	3,015	8	0

Installation and Licensing

The DDR3 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* or *Quartus II Installation & Licensing for UNIX & Linux Workstations*.

Figure 1–2 shows the directory structure after you install the DDR3 SDRAM High-Performance Controller MegaCore functions, where *<path>* is the installation directory. The default installation directory on Windows is **c:\altera\80**; on UNIX and Solaris it is **/opt/altera/80**.

Figure 1–2. Directory Structure
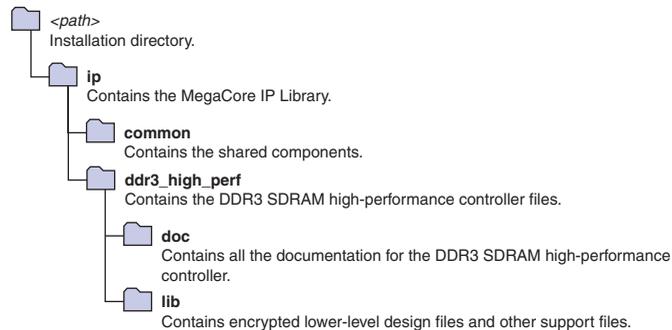
You need to purchase a license for the MegaCore function only when you are completely satisfied with its functionality and performance, and want to take your design to production.

After you purchase a license for DDR3 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 AMPP℠ 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 DDR3 SDRAM high-performance controller, refer to *AN 320: 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 timeout is 1 hour; the tethered timeout 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 DDR3 SDRAM High-Performance Controller MegaCore® function and the Quartus® II software. The sections in this chapter describe each stage.

☞ You can alternatively use the IP Advisor to start your DDR3 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 DDR3 SDRAM High-Performance Controller MegaCore function into your design. It then guides you through a complete Quartus II compilation of your project.

Figure 2–1. Design Flow

```
                    ┌──────────────────┐
                    │ Select Design Flow│
                    └──────────────────┘
MegaWizard Plug-In                        SOPC Builder
  Manager Flow                               Flow

  ┌─────────────────────┐         ┌─────────────────────┐
  │ Specify Parameters  │         │ Specify Parameters  │
  └─────────────────────┘         └─────────────────────┘
            │                               │
  ┌─────────────────────┐         ┌─────────────────────┐
  │    Simulate the     │         │  Complete SOPC      │
  │   Example Design    │         │  Builder System     │
  └─────────────────────┘         └─────────────────────┘
                                            │
                                  ┌─────────────────────┐
                                  │   Simulate System   │
                                  └─────────────────────┘

                    ┌──────────────────┐
                    │  Add Constraints │
                    └──────────────────┘

                    ┌──────────────────┐
                    │   Compile the    │
                    │  Example Design  │
                    └──────────────────┘

                    ┌──────────────────────┐
                    │ Perform Post-Compilation│
                    │    Timing Analysis    │
                    └──────────────────────┘

                    ┌──────────────────┐
                    │ Program Device and│
                    │ Implement Design  │
                    └──────────────────┘
```
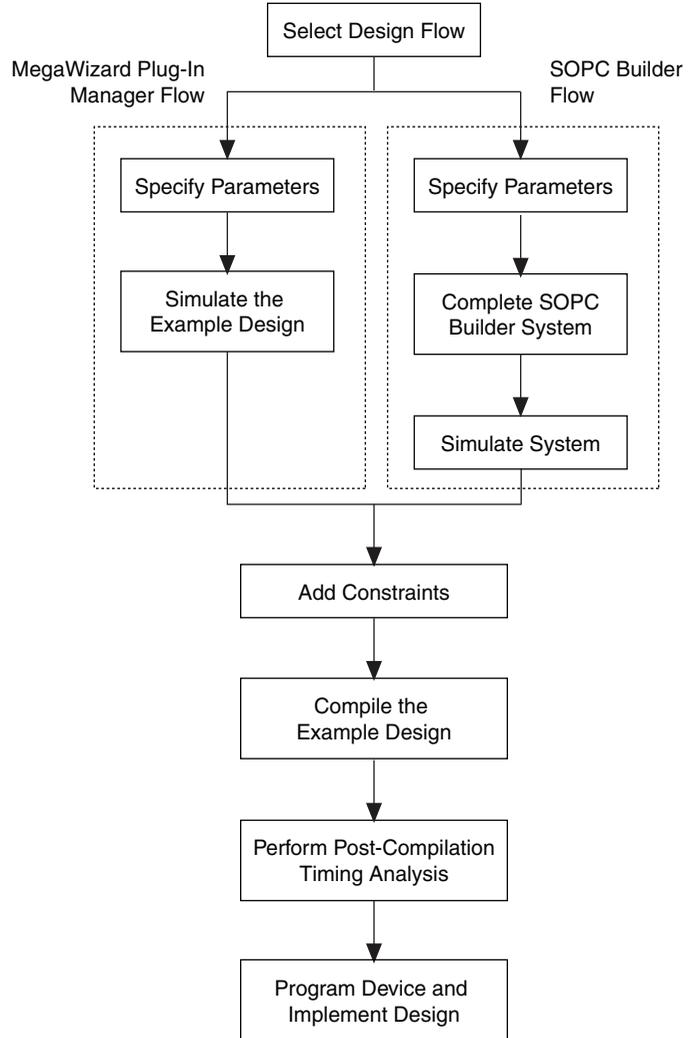
Select Flow

You can parameterize the DDR3 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**
● Automatically-generated simulation environment ● Create custom components and integrate them via the component wizard ● All components are automatically interconnected with the Avalon-MM interface	● Design directly from the DDR3 SDRAM interface to peripheral device(s) ● Achieves higher-frequency operation

SOPC Builder Flow

The SOPC Builder flow allows you to add the DDR3 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 DDR3 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 Information About	**Refer To**
SOPC Builder	Volume 4 of the *Quartus II Handbook*
How to use controllers with SOPC Builder	*AN517: Using High-Performance DDR, DDR2 and DDR3 SDRAM With SOPC Builder*
The Quartus II software	Quartus II Help

Specify Parameters

To specify DDR3 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 **DDR3 SDRAM High-Performance Controller** to your system from the **System Contents** tab.

    ☞ The **DDR3 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.

6. Click **Finish** to complete the DDR3 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:

7. In SOPC Builder, select **Nios II Processor** and click **Add**.

8. On the **Nios II Processor** page, in the **Core Nios II** tab, select **altmemddr** for **Reset Vector** and **Exception Vector** .

9. 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.

CAUTION

The altmemphy megafunction performs memory interface calibration every time it is reset and in doing so writes to addresses `0x0` to `0x47`. If you want your memory contents to remain intact through a system reset, you should avoid using the memory addresses below `0x48`. 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 `0x0` to `0x47`, you should 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, then the **Reset Vector Offset** should be `0x60` and the **Exception Vector Offset** should be `0x80`.

External Memory Interface Width	Reset Vector Offset	Exception Vector Offset
8	`0x60`	`0x80`
16	`0xA0`	`0xC0`
32	`0x120`	`0x140`
64	`0x240`	`0x260`

10. Click **Finish**.

11. In SOPC Builder expand **Interface Protocols** and expand **Serial**.

12. Select **JTAG UART** and click **Add**.

13. 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**.

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

15. Click **Generate**.

16. To ensure the automatically-generated constraints function correctly, you must ensure the pin names and pin group assignments match, otherwise the design may not fit when you compile your design. You can either create your own top-level design file or edit the **altmemddr_example_top.v** file to replace the example driver and the DDR3 SDRAM High-Performance controller and instantiate your SOPC Builder-generated system.

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 Information About	Refer To
Simulating SOPC Builder systems	Volume 4 of the *Quartus II Handbook*
	AN 351 :Simulating Nios II Systems

MegaWizard Plug-In Manager Flow

The MegaWizard Plug-In Manager flow allows you to customize the DDR3 SDRAM High-Performance Controller MegaCore function, and manually integrate the function into your design.

For Information About	Refer To
MegaWizard Plug-In Manager	Quartus II Help
Quartus II software	

Specify Parameters

To specify DDR3 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**.

    ☞ Ensure you select **Yes** for **Do you want to assign a specific device?** to choose a specific device.

2. On the Tools menu click **MegaWizard Plug-In Manager** and follow the steps to start the MegaWizard Plug-In Manager.

    ☞ The DDR3 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.

    ⚠ CAUTION 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**.

⚠ CAUTION The memory model generated by the wizard cannot be used if you have chosen **Full Calibration (long simulation time)**. You must use a memory-vendor provided memory model.

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

👣 For more information about the files generated in your project directory, see Table 2–2.

6. Click **Finish** to generate the MegaCore function and supporting files.

☞ The Quartus II IP File (**.qip**) is a file generated by the MegaWizard interface or SOPC Builder that contains information about a generated IP core. You are prompted to add this **.qip** file to the current Quartus II project at the time of file generation. In most cases, the **.qip** file contains all of the necessary assignments and information required to process the core or system in the Quartus II compiler. Generally, a single **.qip** file is generated for each MegaCore function and for each SOPC Builder system. However, some more complex SOPC Builder components generate a separate **.qip** file, so the system **.qip** file references the component **.qip** file.

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

☞ When prompted to add the **.qip** files to your project, click **Yes**. The addition of the **.qip** files enables their visibility to the Nativelink. The Nativelink needs the **.qip** files to include libraries for simulation.

Table 2–2 describes the generated files and other files 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 2–2. Generated Files *Note (1)*

Filename	Description
<variation name>.**bsf**	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<variation name>.**html**	MegaCore function report file.
<variation name>.**ppf**	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.
<variation name>.**vo or .vho**	VHDL or Verilog HDL gate-level simulation model.
<variation name>.**vhd**, or **.v**	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.
*<variation name>*_**auk_ddr3_hp_controller_wrapper.vo or .vho**	VHDL or Verilog HDL IP functional simulation model.
*<variation name>*_**sequencer_wrapper.vo or .vho**	VHDL or Verilog HDL IP functional simulation model.
*<variation name>*_**bb.v**	Verilog HDL black-box file for the MegaCore function variation. Use this file when using a third-party EDA tool to synthesize your design.
*<variation name>*_**example_driver.vhd**, or **.v**	Example driver.
*<variation name>*_**example_top.vhd**, or **.v**	Example design.
*<variation name>*_**example_top_tb.vhd**, or **.v**	Example testbench.
*<variation name>*_**mem_model.v**	Memory model.
*<variation name>*_**pin_assignments.tcl**	The pin assignments constraints script.
<variation name>.**qip**	Contains Quartus II project information for your MegaCore function variations.

Notes to Table 2–2:
(1)   *<variation name>* is the name you give to the controller you create with the Megawizard.

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

9. Set the *<variation name>*_**example_top.v** or **.vhd** file to be the project top-level design file.

Now, simulate the example design (see "Simulate the Example Design" on page 2–10) and compile (see "Compile the Example Design" on page 2–11).

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 and a simulation model of the memory you are targeting, which are 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 NativeLink, refer to the *Simulating Altera IP Using NativeLink* chapter in volume 3 of the *Quartus II Handbook*.

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.

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

3. 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**.

4. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**.

5. On the Assignments menu click **Settings**, expand **EDA Tool Settings** and select **Simulation**.

    a.   Select a simulator under **Tool name**.

    b.   In **NativeLink settings**, select **Compile test bench** and click **Test Benches**.

6.   Click **New**.

7.   Enter a name for the **Test bench name**.

8.   Enter the name of the automatically generated testbench, *<variation name>*`_example_top_tb`, in **Top level module in test bench**.

9.   Enter the name of the top-level instance in **Design instance name in test bench**.

10.   Under **Simulation period**, set **End simulation at** to **500 μs**.

11.   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 **OK** and then click **Add**. The testbench is *<variation name>*_**example_top_tb.v**; memory model is *<variation name>*_**mem_model.v**.

12.   In the **New Testbench Settings** dialog box, click **OK**.

13.   Click **OK**.

14.   On the Tools menu point to **EDA Simulation Tool** and click **Run EDA RTL Simulation**.

Compile the Example Design

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

1.   Enable 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 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 top level.

2. Use one of the following procedures to specify I/O standard assignments for pins:

   a. If you have a single DDR3 SDRAM interface, and your top-level pins have default naming shown in the example design, run *<variation name>*_**pin_assignments.tcl**. The script assigns the memory interface pins, the correct I/O standard, and avoids the Quartus II fitter failing.

   or

   b. If your design contains pin names that do not match the design example, follow these steps:

   ● On the Assignments menu, click **Pins**. Right-click in the **Groups** or **All Pins** window and click **Create/Import Megafunction**.
   ● Select **Import an existing custom megafunction** and navigate to *<variation name>*.**ppf**.
   ● The *<variation name>*_**pin_assignments.tcl** and *<variation name>*_**dq_groups.tcl** scripts run automatically.

3. 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**..

4. On the Processing menu, point to **Start** and click **Start Analysis & Synthesis**.

5. Assign the pin locations to the pins in your design.

   a. 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.

   or

   b. Manually specify all DQ and DQS pins to align your project with your PCB requirements.

   or

    c.   Manually specify all project pin locations to align your project with your PCB requirements.

☞   When you are assigning pins, ensure that you set an appropriate I/O standard for the non-memory interfaces, like the clock source and the reset inputs. For example, for DDR3 SDRAM, select **1.5 V**. Also select in which bank or side of the device you want the Quartus II software to place them.

6.   Set the output pin loading for all memory interface pins.

7.   Select your required I/O driver strength (derived from simulation) to ensure that you correctly drive each signal or ODT setting and do not suffer from overshoot or undershoot.

8.   To compile the design, on the Processing menu, click **Start Compilation**.

9.   If you want to generate a detailed DDR3 SDRAM interface timing analysis report, run the DDR Report. On the Tools menu click **TimeQuest Timing Analyzer** and on the **Tasks** pane, double-click **Report DDR**.

Program a Device and Implement the Design

After you have compiled the example design, you can perform gate-level simulation (see "Simulate the Example Design" on page 2–10) 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 DDR 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 DDR Memory PHY Interface Megafunction User Guide (ALTMEMPHY)*.

Controller Settings

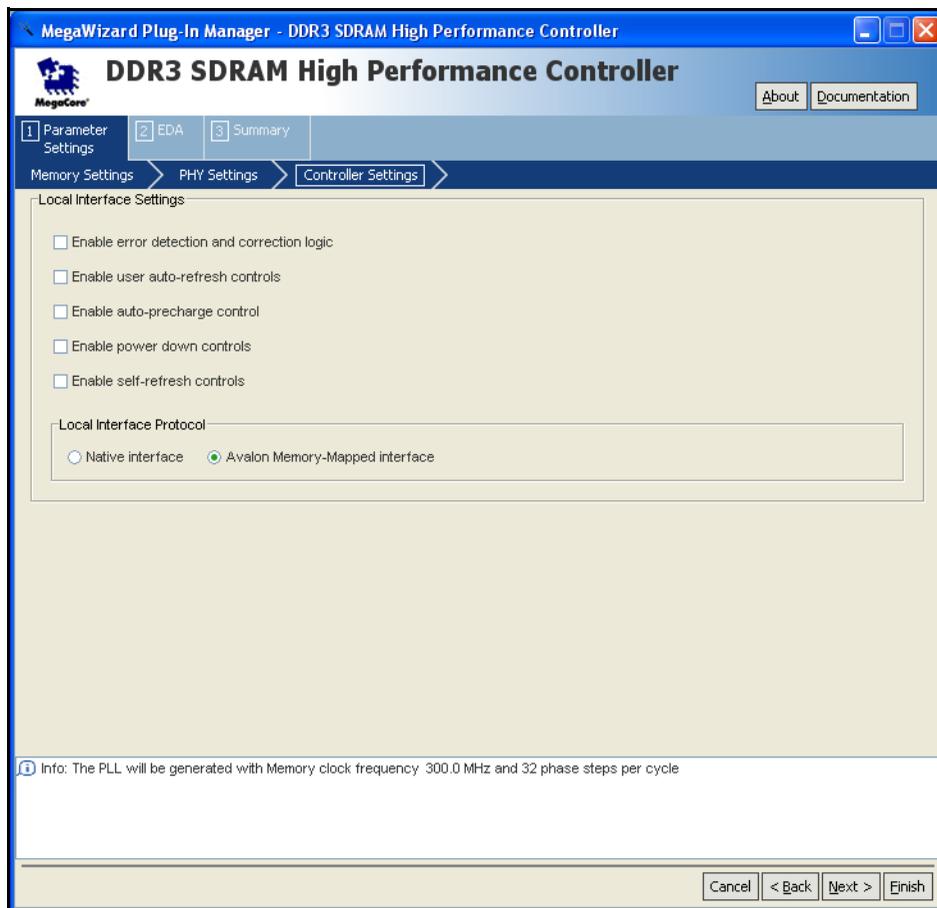Figure 3–1 shows the controller settings page.

Figure 3–1. Controller Settings

Table 3–1 shows the controller settings.

Table 3–1. Controller Settings

Parameter	Range	Description
Enable error correction and detection logic	On or off	Turn on to add the optional error correction coding (ECC) to the design, see "ECC" on page 4–4.
Enable user auto-refresh controls	On or off	Turn on for user control of the refreshes, see "User Refresh Control" on page 4–14.
Enable auto-precharge control	On or off	Turn on if you need fast random access, see "Auto-Precharge Commands" on page 4–17

Table 3–1. Controller Settings		
Parameter	**Range**	**Description**
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, see "Self-Refresh and Power-Down Commands" on page 4–16
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, see "Self-Refresh and Power-Down Commands" on page 4–16
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.

The DDR3 SDRAM High-Performance Controller MegaCore function instantiates 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® Memory-Mapped (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. In full rate, the controller accepts requests of size 1 or 2 on the local interface. Requests of size 2 on the local interface produce better through-put 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 DDR3 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. Maximum efficiency can be achieved by issuing 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 re-ordering.
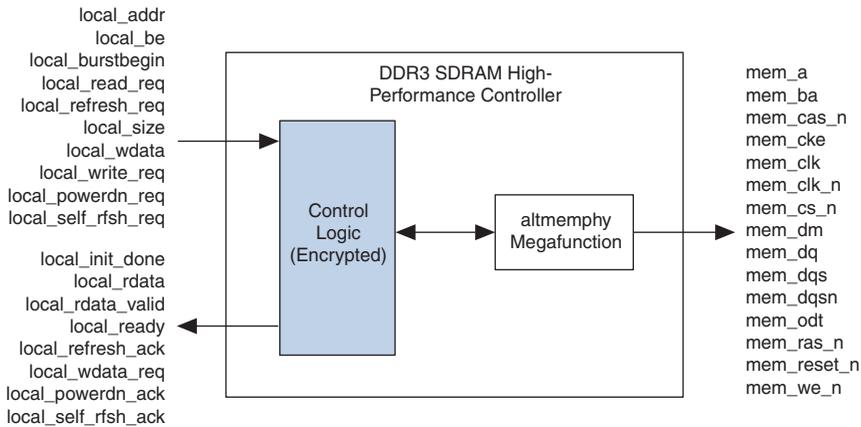
For more information on the altmemphy megafunction, refer to the *External DDR Memory PHY Interface Megafunction User Guide (ALTMEMPHY)*.

Block Description

Figure 4–1 on page 4–2 shows a block diagram of the DDR3 SDRAM high-performance controller.

Figure 4–1. DDR3 SDRAM High-Performance Controller Block Diagram

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
Precharge	PCH	Low	High	Low
Auto refresh	ARF	Low	Low	High
Load mode register	LMR	Low	Low	Low

The DDR3 SDRAM high-performance controller 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

DDR3 SDRAM high-performance controller closes the bank and opens the bank again if it needs 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 5 to 11 clock cycles later. 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 (of 6) 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. DDR3 SDRAM devices support fixed burst lengths of 4 or 8 data cycles or a the on-the-fly mode where the controller can request a burst of 4 or 8 for each read or write command. This on-the-fly mode is the only mode supported. The auto-refresh command (ARF) is issued periodically to ensure data retention. This function is performed by the DDR3 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 initiate 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 initiate 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.

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 read and write latency derived from the write and read latency definitions for half-rate controllers and for Stratix III and Stratix IV devices.

Table 4–2. Typical Latency				
Controller Rate	**Frequency (MHz)**	**Latency Type**	**Latency (Cycles)**	**Latency (ns)**
Half	400	Read	23	115
		Write	13.5	67.5

☞ 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 Appendix B, Latency for more detailed information.

ECC

The optional error correction coding (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 4–2 shows the ECC block diagram.

Figure 4–2. ECC Block Diagram
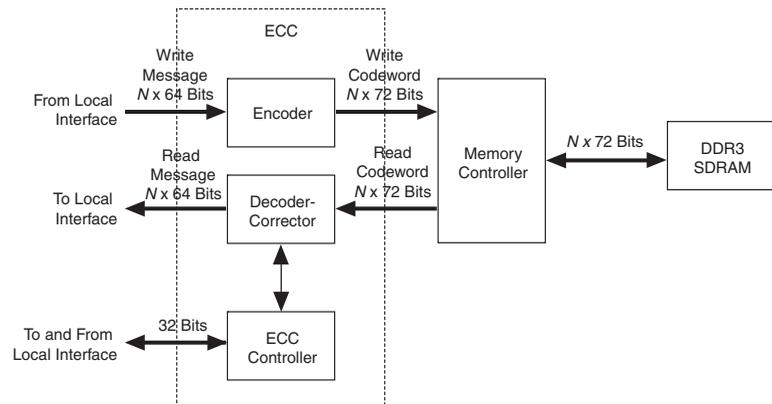
The ECC comprises the following blocks:

■ 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, see 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 DDR3 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 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.

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.

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 DDR3 SDRAM high-performance controller. The example design consists of the DDR3 SDRAM high-performance controller and some driver logic to issue read and write requests to the controller. The example design is a working system that you can compile and use for both static timing checks and board tests.

Figure 4–3 shows the testbench and the example design.

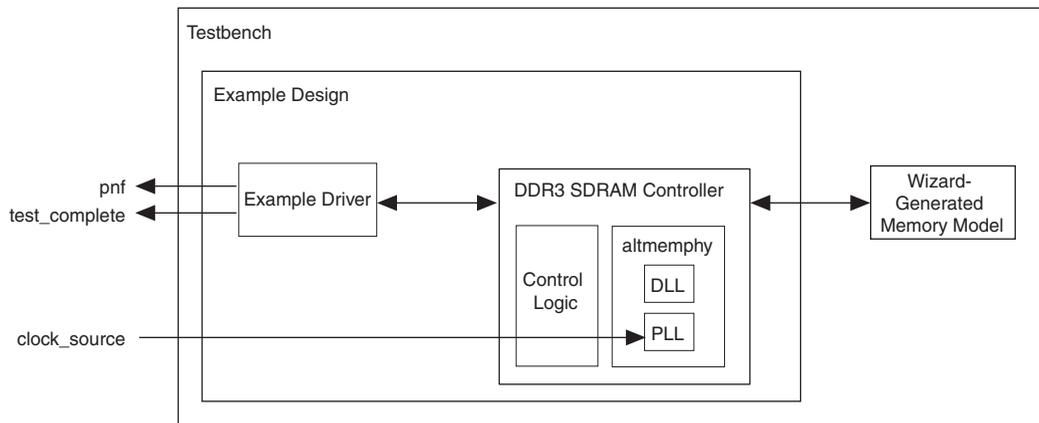*Figure 4–3. Testbench & Example Design*

Table 4–3 describes the files that are associated with the example design and the testbench.

Table 4–3. Example Design & 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*>_**example_driver.v** or **.vhd**	Example driver.
<*variation name*>_**mem_model.v** or **.vhd**	Wizard-generated memory model.
<*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.

The example driver is a self-checking test generator for the DDR3 SDRAM high-performance controller. It uses a state machine to write data patterns to a range of column addresses, within a range of row addresses in all memory banks. It then reads back the data from the same locations, and checks that the data matches. The pass not fail (pnf) output transitions low if any read data fails the comparison. There is also a pnf_per_byte output, which shows the comparison on a per byte basis. The test_complete output transitions high for a clock cycle at the end of the write or read test sequence. After this transition the test restarts from the beginning.

The data patterns used are generated using an 8-bit LFSR per byte, with each LFSR having a different initialization seed.

When test_complete is detected high, a test finished message is printed out, which shows whether the test has passed.

For more details on how to run the simulation script, see "Simulate the Example Design" on page 2–10.

Interfaces & Signals

This section describes the following topics:

- "Interface Description" on page 4–9
- "Signals" on page 4–18

Interface Description

This section describes the following local-side interface requests:

- "Writes" on page 4–10

■ "Reads" on page 4–13
■ "User Refresh Control" on page 4–14
■ "Self-Refresh and Power-Down Commands" on page 4–16
■ "Auto-Precharge Commands" on page 4–17
■ "Initialization Timing" on page 4–18

☞ These interface requests are for the native interface. For the Avalon™ Memory-Mapped (Avalon-MM) interface see the *Avalon Memory-Mapped Interface Specification*.

Writes

Figure 4–4 on page 4–11 shows three back-to-back write requests of size 2, to sequential addresses. The DDR3 SDRAM controller supports the on-the-fly burst mode. This mode allows you to request bursts of length 1 or 2 on the local side interface (equivalent to 4 or 8 on the DDR3 SDRAM side interface).

Figure 4–4. Writes
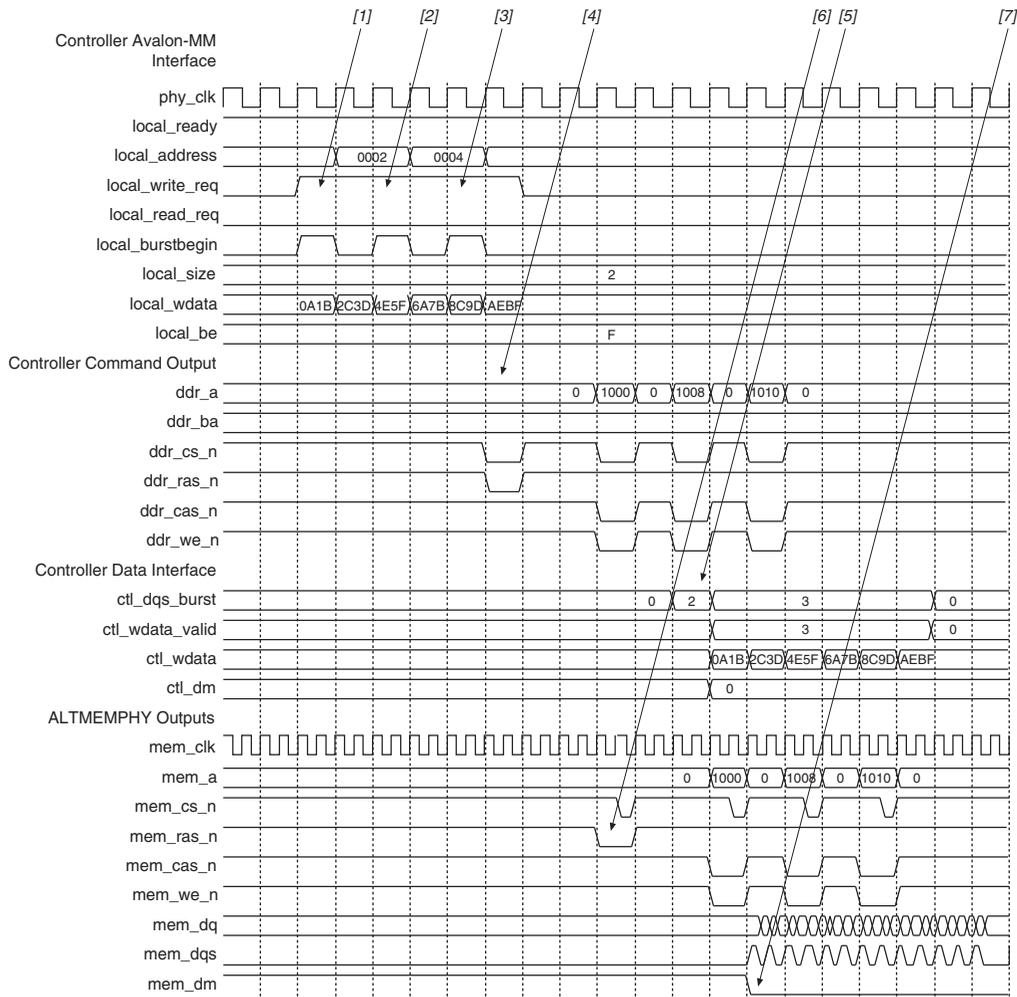
The following sequence corresponds with the numbered items in Figure 4–4.

1. The user logic requests the first write, by asserting the `local_write_req`, `local_burstbegin`, size and address signals for this write. In this example, the request is a burst of length 2 (8 on the DDR3 SDRAM side) to 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 is not asserted, the user logic must keep the write request, size, and address signals asserted. For this burst of length 2, you must present the second beat of write data in the next clock cycle.

☞　　`local_be` is active high; `mem_dm` is active low. To map `local_wdata` and and `local_be` to `mem_dq` and `mem_dm`, consider the following full rate example with 32-bit local_wdata and 16-bit 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>
```
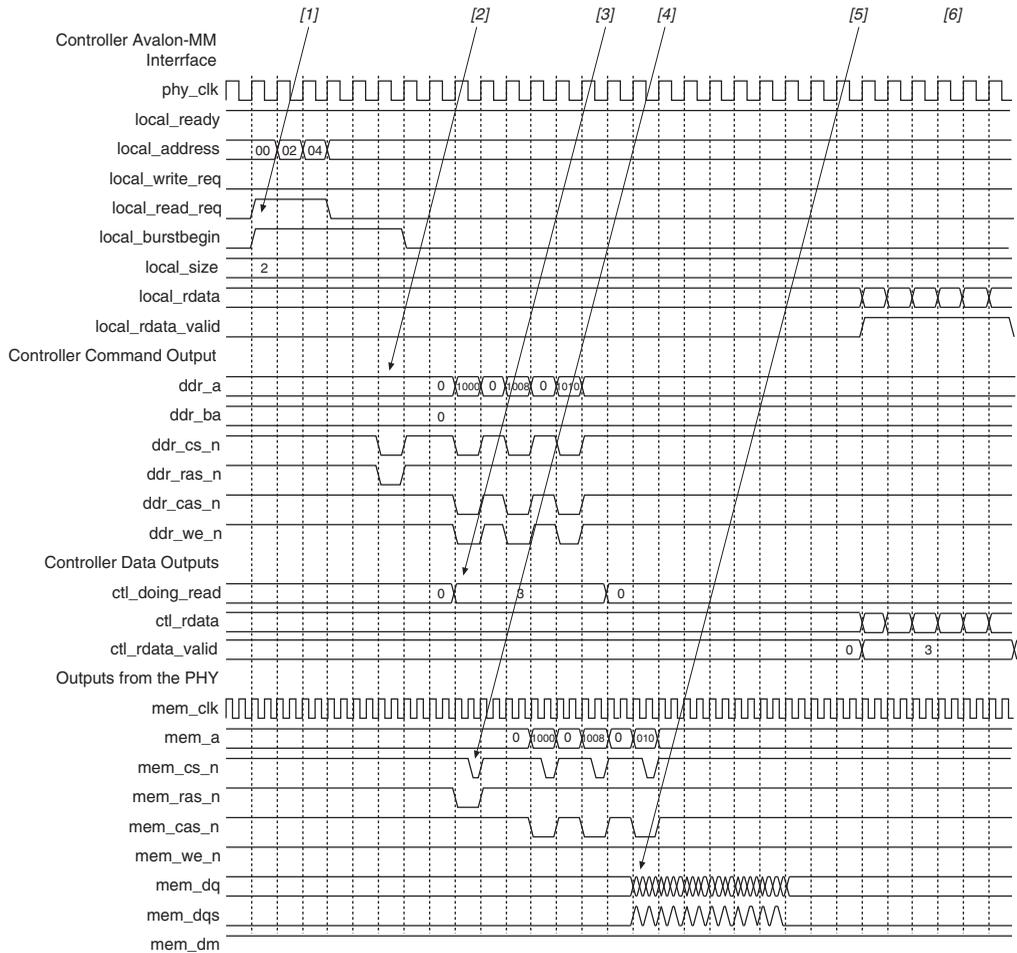
2. The user logic requests a second write to a sequential address, of size 2 (8 on the DDR3 SDRAM side). The `local_ready` signal remains asserted, which indicates that the controller has accepted the request.

3. The user logic requests the third write. 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 bank activation command and the three write commands sequentially to the altmemphy megafunction, which converts these commands from half-rate to full-rate and issues them to the memory device.

5. The controller asserts the signals that control how long the DQS (`ctl_dqs_burst`) and DQ (`ctl_wdata_valid`) outputs are enabled for. The `ctl_dqs_burst` and `ctl_wdata_valid` signals are two bits wide so that the controller can control how many full-rate `mem_clk` cycles the DQS and DQ signals are enabled for, even though the controller is operating on the half-rate clock. In this example, the DQS outputs are enabled for 13 full-rate clock cycles (to account for the DQS preamble) and the DQ is enabled for 12 full-rate clock cycles. The write data (`ctl_wdata`) and mask (`ctl_dm`) are issued at the same time as the `ctl_wdata_valid`.

6. The altmemphy megafunction issues the bank activation and write commands to the memory device.

7. The altmemphy megafunction issues the DQS, DQ and DM signals to write the data to the memory device.

Reads

Figure 4–5 shows shows three read requests of size 2. The DDR3 SDRAM controller supports the on-the-fly burst mode. This mode allows you to request bursts of length 1 or 2 on the local side interface (equivalent to 4 or 8 on the DDR3 SDRAM side interface).

Figure 4–5. Reads

The following sequence corresponds with the numbered items in Figure 4–5 on page 4–13.

1. The user logic requests the three back-to-back reads of size 2 (8 on the DDR3 SDRAM side) by asserting the `local_read_req`, `local_burstbegin`, `local_size` and `local_address` signals for each read. The `local_ready` signal is asserted, which indicates that the controller has accepted each 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 read request, size, and address signals asserted.

2. The controller issues the necessary bank activation command and the three read commands sequentially to the altmemphy megafunction, which converts these commands from half-rate to full-rate and issues them to the memory device.

3. The controller asserts the `ctl_doing_read` signals to indicate to the altmemphy megafunction when and for how long to enable to the capture registers.

4. The altmemphy megafunction issues the bank activation and read commands to the memory device.

5. The memory device returns the read data for the addresses requested after the CAS latency along with the DQS strobe signal that the altmemphy megafunction uses to capture the read data.

6. The controller issues the read data to the user logic, marking it valid with the `local_rdata_valid` signal. The for the subsequent read requests.The exact number of clock cycles between the controller accepting the request and returning the data depends on the number of other requests pending in the controller, the state the memory is in, and the timing requirements of the memory (for example, the CAS latency).

User Refresh Control

Figure 4–6 on page 4–15 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 4–6. User Refresh Control
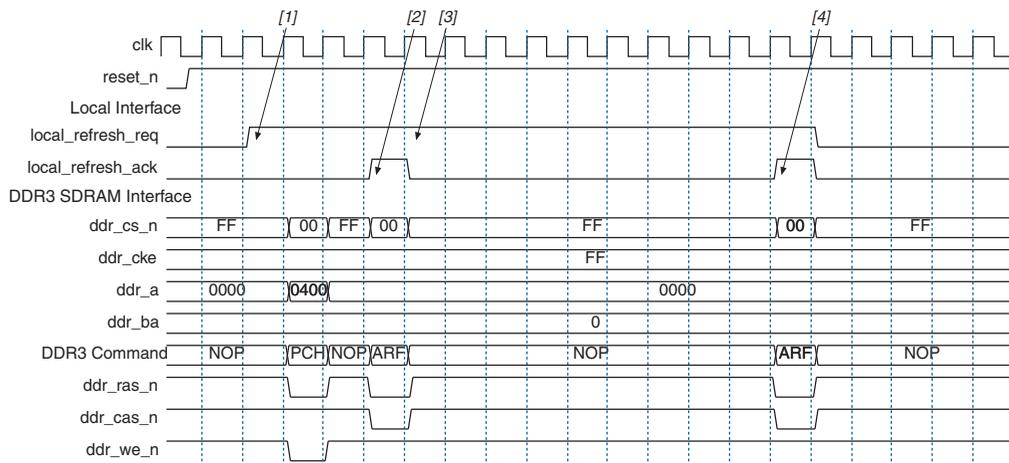
Note to Figure 4–6:
(1)   DDR3 Command shows the command that the command signals are issuing.

The following sequence corresponds with the numbered items in Figure 4–6.

1.   The user logic asserts the refresh request signal to indicate to the controller that it should perform a refresh. The state of the read and write requests signal does not matter as the controller gives priority to the refresh request (although it completes any currently active reads or writes).

2.   The controller asserts the refresh acknowledge signal to indicate that it has issued a refresh command to the altmemphy megafunction. This signal is still available even if the **Enable user auto-refresh controls** option is not switched 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 Figures 4–7 and 4–8 on page 4–17. 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.
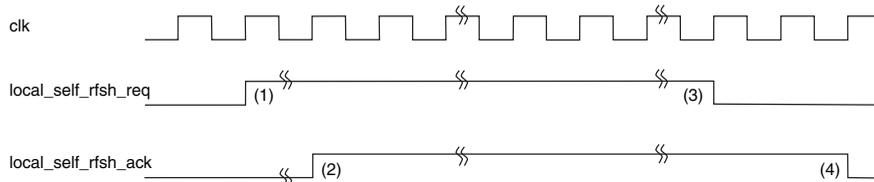
Figure 4–7. Power-Down Mode .
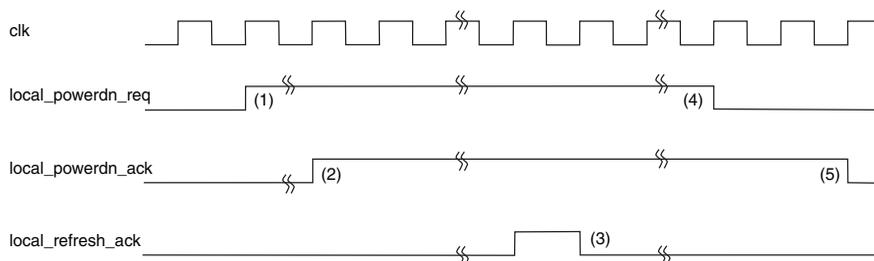
Notes to Figure 4–7
(1) The user synchronously asserts the request signal to indicate that the controller should put the memory into the power-down state as soon as possible.
(2) Once the controller is able to issue the correct commands to put the memory into the power-down state, it responds by asserting 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 the user has 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 4–8. Self-Refresh Mode

Notes to Figure 4–8
(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 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 for a complete burst. It is not possible to auto-precharge a partial burst to the memory.

Initialization Timing

The DDR3 SDRAM high-performance controller relies on the altmemphy megafunction for initialization.

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

When altmemphy has finished calibrating, the memory controller asserts the `local_init_done` signal, which shows that it has initialized the memory devices.

Signals

Table 4–4 shows the clock and reset signals.

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

Name	Direction	Description
global_reset_n	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, including the PLL, in reset while low.
pll_ref_clk	Input	The reference clock input to PLL.

Table 4–4. Clock and Reset Signals (Part 2 of 2)		
Name	**Direction**	**Description**
phy_clk	Output	The system clock that the altmemphy provides to the user. All user inputs to and outputs from the DDR3 high-performance controller must be synchronous to this clock.
reset_phy_clk_n	Output	The reset signal that the altmemphy provides to the user. It is asserted asynchronously and deasserted synchronously to phy_clk clock domain.
dll_reference_clk	Output	Reference clock to feed to an externally instantiated DLL.
reset_request_n	Output	Reset request output that indicates when the PLL outputs are not locked. Use this 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.
soft_reset_n	Input	Soft reset input for PLL. This causes a complete reset of the whole system. **NOTE:** soft_reset_n **is edge detected.**
oct_ctl_rs_value	Input	altmemphy signal that specifies the serial termination value. Should be connected to the ALT_OCT MegaFunction output "Seriesterminationcontrol".
oct_ctl_rt_value	Input	altmemphy signal that specifies the parallel termination value. Should be connected to the ALT_OCT MegaFunction output "Parallelterminationcontrol".
dqs_delay_ctrl_import	Input	Allows the use of DLL in another altmemphy instance in this altmemphy instance. Connect the export port on the altmemphy instance with a DLL to the import port on the other altmemphy instance.

Table 4–5 shows the DDR3 SDRAM high-performance controller local interface signals.

Table 4–5. Local Interface Signals (Part 1 of 3)		
Signal Name	**Direction**	**Description**
local_addr[]	Input	Memory address at which the burst should start. The width of this bus is sized using the following equation: For one chip select: width = bank bits + row bits + column bits – 1 For multiple chip selects: width = chip bits + bank bits + row bits + column bits – 1 For half-rate controllers, two least significant bits (LSB) 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
local_be[]	Input	Byte enable signal, which you use to mask off individual bytes during writes.
local_burstbegin	Input	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. Unlike all other Avalon-MM signals, the burst begin signal does not stay asserted if local_ready is deasserted.
local_read_req	Input	Read request signal. You cannot assert read request and write request signal at the same time.
local_refresh_req	Input	User controlled refresh request. If **Enable user auto-refresh controls** is turned on, local_refresh_req 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.
local_size[]	Input	Controls the number of beats in the requested read or write access to memory, encoded as a binary number. The DDR3 SDRAM high-performance controller supports burst lengths of 1 and 2 on the local side interface.
local_wdata[]	Input	Write data bus. The width of local_wdata is four times the memory data bus for half rate controller.
local_write_req	Input	Write request signal. You cannot assert read request and write request signal at the same time.

Table 4–5. Local Interface Signals (Part 2 of 3)		
Signal Name	**Direction**	**Description**
`local_init_done`	Output	Memory initialization complete signal, which is asserted once the controller has completed its initialization of the memory. Read and write requests are still accepted before `local_init_done` is asserted, however they are not issued to the memory until it is safe to do so.
`local_rdata[]`	Output	Read data bus. The width of `local_rdata` is four times that of the memory data bus.
`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 `local_rdata_valid` signal indicates that valid data is present on the read data bus. The timing of `local_rdata_valid` is automatically adjusted to cope with your choice of resynchronization and pipelining options.
`local_ready`	Output	The `local_ready` signal indicates that the DDR3 SDRAM high-performance controller is ready to accept request signals. If `local_ready` is asserted in the clock cycle that a read or write request is asserted, that request has been accepted. The `local_ready` signal is deasserted to indicate that the DDR3 SDRAM high-performance controller cannot accept any more 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, `local_refresh_ack` 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.
`local_autopch_req`	Input	User control of precharge. If **Enable auto precharge control** is turned on, `local_autopch_req` 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 4–5. Local Interface Signals (Part 3 of 3)

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 will place 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 4–6 shows the DDR3 SDRAM interface signals.

Table 4–6. DDR3 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 DDR3 SDRAM and captures read data into the Altera device.
`mem_dqs_n[]`	Bidirectional	Memory data strobe signal, which writes data into the DDR3 SDRAM and captures read data into the Altera device.

Table 4–6. DDR3 SDRAM Interface Signals (Part 2 of 2)		
Signal Name	**Direction**	**Description**
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.
mem_ras_n	Output	Memory row address strobe signal.
mem_reset_n	Output	Memory reset signal.
mem_we_n	Output	Memory write enable signal.

Note to Table 4–6:

(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 4–7 shows the ECC controller signals.

Table 4–7. 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.

Additional Information

Revision History

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

Date	Version	Changes Made
May 2008	8.0	● Added a section on ECC ● Added more detailed ECC information (Appendix A) ● Added more detailed latency information (Appendix B) ● Added Stratix IV support
October 2007	7.2	First release.

How to Contact Altera

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

Contact *(1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	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.

Table Info–1. Typographic Conventions

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: **f$_{MAX}$**, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.

This appendix describes the ECC registers and then describes the register bits.

Table A–1 shows the ECC registers.

Table A–1. ECC Registers (Part 1 of 3)					
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.

Table A–1. ECC Registers (Part 2 of 3)					
Name	**Address**	**Size (Bits)**	**Attribute**	**Default**	**Description**
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.
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 Nth 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 Nth 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 Nth 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 (see 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 (see Table A–4).

Table A–1. ECC Registers (Part 3 of 3)					
Name	Address	Size (Bits)	Attribute	Default	Description
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 (see 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 (see Table A–6). These status bits can be cleared by writing a 1 in the respective locations.

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.

When designing for memory controllers, you must consider read and write latencies. Altera® defines read and write latencies as follows:

■ Read Latency—the amount of time it takes for the read data to appear at the user (local) interface after you initiate the read request.
■ Write Latency—the amount of time it takes for the write data to appear at the memory interface after you initiate the write request.

Altera assumes the following basic assumptions when calculating latency:

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

☞ For the half-rate controller, the user (local) side frequency is half of 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.

The latency for the high-performance controller comprises many different stages of the memory interface. Figure B–1 shows a typical memory interface read latency path showing the read latency from the time a `local_read_req` assertion is detected by the controller up to data available to be read from the DRAM module.
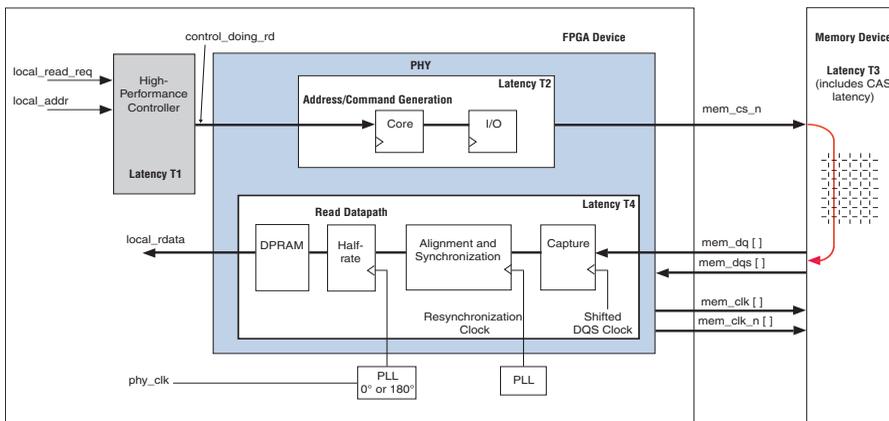
Figure B–1. Typical Read Latency Path

Table B–1 shows the different stages that make up the whole read latency, shown in Figure B–1.

Table B–1. High Performance Controller Latency Stages and Descriptions

Latency Number	Latency Stage	Description
T1	Controller	`local_read_req` or `local_write_req` signal assertion to `ddr_cs_n` signal assertion.
T2	Command Output	`ddr_cs_n` signal assertion to `mem_cs_n` signal assertion.
T3	CAS or WL	Read command to DQ data from the memory or write command to DQ data to the memory.
T4	altmemphy read data input	Read data appearing on the local interface.
T2 + T3	Write data latency	Write data appearing on the memory interface.

From Figure B–1, the read latency in the high-performance controllers is made up of four components:

Read latency = controller latency + command output latency +
        CAS latency + PHY read data input latency
      = T1 + T2 + T3 + T4

Similarly, the write latency in the high-performance controllers is made up of three components:

Write latency = controller latency + write data latency
          = T1 + T2 + T3

You can separate the controller and PHY read data input latency into latency that occurred in the input/output element (IOE) and latency that occurred in the FPGA fabric.

Tables B–2 and B–3 show a typical latency that can be achieved in Stratix III and Stratix IV devices.
The exact latency for your memory controller depends on your precise configuration. You should obtain precise latency from simulation, but this figure may vary slightly in hardware because of the automatic calibration process.

Table B–2. Typical Read Latency in Stratix III & IV High-Performance Controller *Note (1)*, *(2)*

Memory Standard	Frequency (MHz)	Interface Mode	Controller Latency *(3)*	Address and Command Latency		CAS Latency	Read Data Latency		Total Read Latency *(4)*	
				FPGA	IO		FPGA	I/O	Cycles	Time (ns)
DDR3 SDRAM	400	Half rate	6	4	1	3	7	2	23	115

Notes to Table B–2:
(1) These are typical latency values using the assumptions listed in the beginning of the section. Your actual latency may be different than shown. You need to perform your own simulation for your actual latency.
(2) Numbers shown may have been rounded up to the nearest higher integer.
(3) The controller latency value is from the Altera high-performance controller.
(4) Total read latency is the sum of controller, address and command, CAS, and read data latencies.

Table B–3. Typical Write Latency in Stratix III & IV High-Performance Controller *Note (1), (2)*								
Memory Standard	**Frequency (MHz)**	**Interface Mode**	**Controller Latency** *(3)*	**Address and Command Latency**		**Memory Write Latency** *(4)*	**Total Write Latency** *(5)*	
				FPGA	**I/O**		**Cycles**	**Time (ns)**
DDR3 SDRAM	400	Half rate	6	4	1	2.5	13.5	67.5

Notes to Table B–3:
(1) These are typical latency values using the assumptions listed in the beginning of the section. Your actual latency may be different than shown. You need to perform your own simulation for your actual latency.
(2) Numbers shown may have been rounded up to the nearest higher integer.
(3) The controller latency value is from the Altera high-performance controller.
(4) Memory write latency is per memory device specification. This is the latency from when you provide the command to write to when you need to provide data at the memory device.
(5) Total write latency is the sum of controller, address and command, and memory write latencies.

X-ON Electronics

Largest Supplier of Electrical and Electronic Components

Click to view similar products for Development Software *category:*

Click to view products by Intel *manufacturer:*

Other Similar products are found below :

RAPPID-567XFSW SRP004001-01 SW163052 SYSWINEV21 Core429-SA WS01NCTF1E W128E13 SW89CN0-ZCC IPS-EMBEDDED IP-UART-16550 MPROG-PRO535E AFLCF-08-LX-CE060-R21 WS02-CFSC1-EV3-UP SYSMAC-STUDIO-EIPCPLR LIB-PL-PC-N-1YR-DISKID LIB-PL-A-F SW006026-COV 1120270005 1120270006 MIKROBASIC PRO FOR FT90X (USB DONGLE) MIKROC PRO FOR FT90X (USB DONGLE) MIKROC PRO FOR PIC (USB DONGLE LICENSE) MIKROBASIC PRO FOR AVR (USB DONGLE LICEN MIKROBASIC PRO FOR FT90X MIKROC PRO FOR DSPIC30/33 (USB DONGLE LI MIKROPASCAL PRO FOR ARM (USB DONGLE LICE MIKROPASCAL PRO FOR FT90X MIKROPASCAL PRO FOR FT90X (USB DONGLE) MIKROPASCAL PRO FOR PIC32 (USB DONGLE LI SW006021-2H ATATMELSTUDIO 2400573 2702579 2988609 2702546 SW006022-DGL 2400303 2701356 VDSP-21XX-PCFLOAT VDSP-BLKFN-PC-FULL 88970111 DG-ACC-NET-CD 55195101-102 SW1A-W1C MDK-ARM PCI-EXP1-E3-US PCI-T32-E3-US SW006021-2NH SW006021-1H SW006021-2