External Memory Interface Handbook Volume 3

Section III. QDR II and QDR II+ SRAM Controller with UniPHY User Guide

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

EMI_QDRII_UG-2.1

Document last updated for Altera Complete Design Suite version: 10.1
Document publication date: December 2010

Subscribe

Contents

1. About This IP

The Altera QDR II and QDR II+ SRAM controllers with UniPHY provide simplified interfaces to industry-standard QDR II and QDR II+ SRAM.

The QDR II and QDR II+ SRAM controllers with UniPHY offer full-rate or half-rate QDR II and QDR II+ SRAM interfaces. The UniPHY IP is an interface between a memory controller and memory devices and performs read and write operations to the memory. The UniPHY IP creates the datapath between the memory device and the memory controller and user logic in various Altera devices.

The Quartus II software generates an example top-level project, consisting of an example driver, and your QDR II or QDR II+ SRAM controller custom variation. The controller instantiates an instance of the UniPHY.

The example top-level project 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 or fail and test-complete signals.

For device families not supported by the UniPHY IP, use the Altera legacy integrated static datapath and controller MegaCore functions.

You can, alternatively, create your own memory interface datapath using the ALTDLL and ALTDQ_DQS megafunctions, available in the Quartus II software, but you then must consider all of the aspects of the design including timing analysis and design constraints.

The UniPHY IP offers the Altera PHY interface (AFI). The AFI results in a simple connection between the PHY and controller.

Release Information

Table 1–1 provides information about this release of the QDR II and QDR II+ SRAM controllers with UniPHY.

Table 1–1. Release Information

Item	Description
Version	10.1
Release Date	December 2010
Ordering Codes	IP-QDRII/UNI
Vendor ID	6AF7

Device Family Support

IP cores provide the following levels of support for target Altera device families:

- For FPGA device support:
  - Preliminary—verified with preliminary timing models for this device
  - Final—verified with final timing models for this device
- For ASIC devices (HardCopy families)
  - HardCopy companion—verified with preliminary timing models for HardCopy companion device
  - HardCopy compilation—verifed with final timing models for HardCopy device

Table 1–2 shows the level of support offered by the QDR II and QDR II+ SRAM controllers to each of the Altera device families.

For information about supported clock rates for external memory interfaces, refer to the *External Memory Interface System Specifications* section in volume 1 of the *External Memory Interface Handbook*.

Table 1–2. Device Family Support

Device Family	Support
Arria® II GX	Final
Arria II GZ	Preliminary
HardCopy® III	HardCopy companion
HardCopy IV	HardCopy companion
Stratix® III	Final
Stratix IV	Final
Stratix V	Preliminary
Other device families	No support

Features

Table 1–3 summarizes key feature support for the QDR II and QDR II+ SRAM Controllers with UniPHY.

Table 1–3. Key Feature Support for QDR II and QDR II+ SRAM Controllers with UniPHY (Part 1 of 2)

Key Feature	QDR II SDRAM UniPHY	QDR II+ SDRAM UniPHY
High-performance controller (HPC)	—	—
High-performance controller II (HPC II)	✓	✓
Half-rate core logic and user interface	✓	✓
Full-rate core logic and user interface	✓	✓
Burst length (half-rate)	4	4
Burst length (full-rate)	2 or 4	2 or 4

Table 1–3. Key Feature Support for QDR II and QDR II+ SRAM Controllers with UniPHY (Part 2 of 2)

Key Feature	QDR II SDRAM UniPHY	QDR II+ SDRAM UniPHY
Reduced controller latency (1) (2)	✓	✓
Read latency	1.5	2 or 2.5
Maximum data width	72 bits	72 bits
ODT (in memory device)	—	✓
x36 emulation mode(3)	✓	✓

Notes for Table 1–3:

(1) The maximum achievable clock rate when reduced controller latency is selected must be attained through Quatrus II software timing analysis of your complete design.

(2) Not available in Arria II GX devices.

(3) Emulation mode allows emulation of a larger memory-width interface using multiple smaller memory-width interfaces. For example, an x36 QDR II or QDR II+ interface can be emulated using two x18 interfaces.

Unsupported Features

Table 1–4 summarizes unsupported features for the QDR II and QDR II+ SRAM Controllers with UniPHY.

Table 1–4. Unsupported Features for the QDR II and QDR II+ SRAM Controllers with UniPHY

Memory Protocol	Unsupported Feature
QDR II SRAM	Cyclone III devices
	Cyclone IV devices
	Deterministic latency
	ECC
	Memory device ODT
	Multiple chip select
	VHDL support for Arria II GX, Arria II GZ, Stratix III, Stratix IV, and Stratix V devices
	x36 emulation mode for Stratix V devices
QDR II+ SRAM	Cyclone III devices
	Cyclone IV devices
	Deterministic latency
	ECC
	Multiple chip select
	VHDL support for Arria II GX, Arria II GZ, Stratix III, Stratix IV, and Stratix V devices
	x36 emulation mode for Stratix V devices

MegaCore Verification

Altera has carried out extensive random, directed tests with functional test coverage using industry-standard models to ensure the functionality of the QDR II and QDR II+ SRAM controllers with UniPHY.

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.

Resource Utilization

This section lists resource utilization for the QDR II and QDR II+ SRAM controllers with UniPHY for supported device families. Resource utilizations are derived with all parameters at their default values.

Table 1–5 shows the typical resource usage of the QDR II and QDR II+ SRAM controllers with UniPHY in the current version of Quartus II software for Arria II GX devices.

Table 1–5. Resource Utilization in Arria II GX Devices

PHY Rate	Memory Width (Bits)	Combinational ALUTS	Logic Registers	Memory (Bits)	M9K Blocks
Half	9	620	701	0	0
	18	921	1122	0	0
	36	1534	1964	0	0
Full	9	584	708	0	0
	18	850	1126	0	0
	36	1387	1962	0	0

Table 1–6 shows the typical resource usage of the QDR II and QDR II+ SRAM controllers with UniPHY in the current version of Quartus II software for Arria II GZ, Stratix III, Stratix IV, and Stratix V devices.

Table 1–6. Resource Utilization in Arria II GZ, Stratix III, Stratix IV, and Stratix V Devices

PHY Rate	Memory Width (Bits)	Combinational ALUTS	Logic Registers	Memory (Bits)	M9K Blocks
Half	9	602	641	0	0
	18	883	1002	0	0
	36	1457	1724	0	0
Full	9	586	708	0	0
	18	851	1126	0	0
	36	1392	1962	0	0

System Requirements

The QDR II and QDR II+ SRAM controllers with UniPHY are part of the MegaCore IP Library, which is distributed with the Quartus II software.

For system requirements and installation instructions, refer to *Altera Software Installation & Licensing*.

2. Getting Started

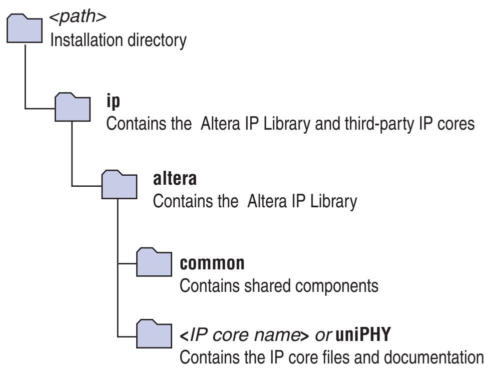
This chapter provides a general overview of the Altera IP core design flow to help you quickly get started with any Altera IP core. The Altera IP Library is installed as part of the Quartus II installation process. You can select and parameterize any Altera IP core from the library. Altera provides an integrated parameter editor that allows you to customize IP cores to support a wide variety of applications. The parameter editor guides you through the setting of parameter values and selection of optional ports. The following sections describe the general design flow and use of Altera IP cores.

Installation and Licensing

The Altera IP Library is distributed with the Quartus II software and downloadable from the Altera website (www.altera.com).

Figure 2–1 shows the directory structure after you install an Altera IP core, where *<path>* is the installation directory. The default installation directory on Windows is **C:\altera\\<***version number***>**; on Linux it is **/opt/altera<***version number***>.**

Figure 2–1. IP core Directory Structure



<path>
Installation directory
ip
Contains the Altera IP Library and third-party IP cores
altera
Contains the Altera IP Library
common
Contains shared components
<IP core name> or **uniPHY**
Contains the IP core files and documentation

You can evaluate an IP core in simulation and in hardware until you are satisfied with its functionality and performance. Some IP cores require that you purchase a license for the IP core when you want to take your design to production. After you purchase a license for an Altera IP core, you can request a license file from the Altera Licensing page of the Altera website and install the license on your computer. For additional information, refer to *Altera Software Installation and Licensing*.

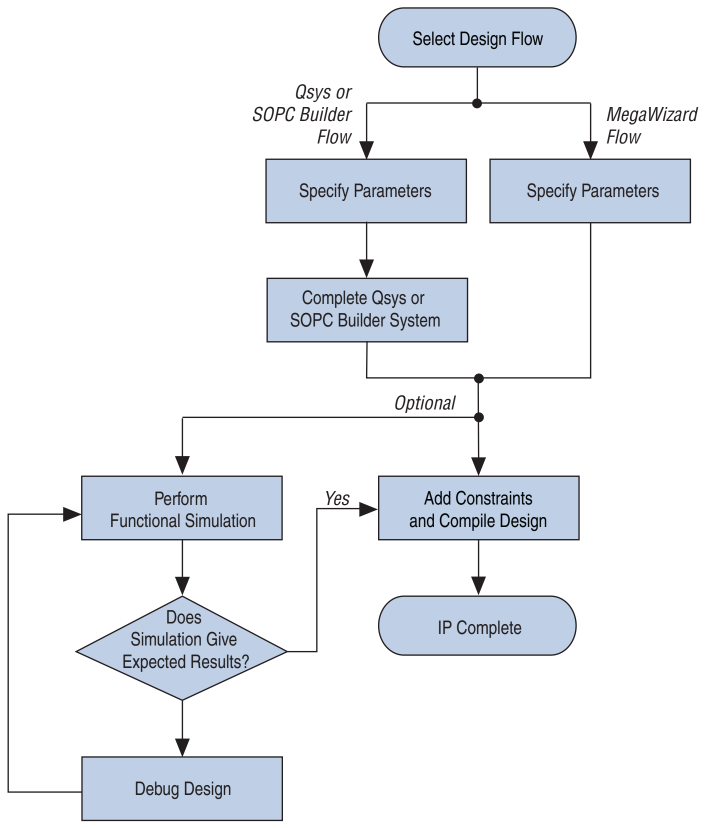
Design Flows

You can use the following flow(s) to parameterize Altera IP cores:

- MegaWizard Plug-In Manager Flow
- SOPC Builder Flow
- Qsys Flow

CAUTION Altera's Qsys system integration tool is now available as beta for evaluation in the Quartus II software subscription edition version 10.1. Altera does not recommend using the beta release of Qsys in the Quartus II software version 10.1 for designs that are close to completion and are meeting design requirements. Before using Qsys, review the *Quartus II Software Version 10.1 Release Notes* and *AN 632: SOPC Builder to Qsys Migration Guidelines* for known issues and limitations. To submit general feedback or technical support on the beta release of Qsys, submit a service request through **mysupport.altera.com**. Alternatively, to submit general feedback, click **Feedback** on the Quartus II software Help menu.

Figure 2–2. Design Flows


Select Design Flow
Qsys or SOPC Builder Flow
MegaWizard Flow
Specify Parameters
Specify Parameters
Complete Qsys or SOPC Builder System
Optional
Perform Functional Simulation
Yes
Add Constraints and Compile Design
Does Simulation Give Expected Results?
IP Complete
Debug Design

The MegaWizard Plug-In Manager flow offers the following advantages:

- Allows you to parameterize an IP core variant and instantiate into an existing design
- For some IP cores, this flow generates a complete example design and testbench.

The SOPC Builder flow offer the following advantages:

- Generates simulation environment
- Allows you to integrate Altera-provided custom components
- Uses Avalon®memory-mapped (Avalon-MM) interfaces

The Qsys flow offers the following additional advantages over SOPC Builder:

- Provides visualization of hierarchical designs
- Allows greater performance through interconnect elements and pipelining
- Provides closer integration with the Quartus II software

MegaWizard Plug-In Manager Flow

The MegaWizard Plug-In Manager flow allows you to customize your IP core and manually integrate the function into your design.

Specifying Parameters

To specify IP core parameters with the MegaWizard Plug-In Manager, follow these steps:

1. Create a Quartus II project using the **New Project Wizard** available from the File menu.
2. In the Quartus II software, launch the **MegaWizard Plug-in Manager** from the Tools menu, and follow the prompts in the MegaWizard Plug-In Manager interface to create or edit a custom IP core variation.
3. To select a specific Altera IP core, click the IP core in the **Installed Plug-Ins** list in the MegaWizard Plug-In Manager.
4. Specify the parameters on the **Parameter Settings** pages. For detailed explanations of these parameters, refer to the "*Parameter Settings*" chapter in this document.

   ☞ Some IP cores provide preset parameters for specific applications. If you wish to use preset parameters, click the arrow to expand the **Presets** list, select the desired preset, and then click **Apply**. To modify preset settings, in a text editor edit the **<*installation directory*>\ip\altera\uniphy\lib\<*IP core*>.qprs** file.

5. If the IP core provides a simulation model, specify appropriate options in the wizard to generate a simulation model.

   ☞ Altera IP supports a variety of simulation models, including simulation-specific IP functional simulation models and encrypted RTL models, and plain text RTL models. These are all cycle-accurate models. The models allow for fast functional simulation of your IP core instance using industry-standard VHDL or Verilog HDL simulators. For some cores, only the plain text RTL model is generated, and you can simulate that model.

   For more information about functional simulation models for Altera IP cores, refer to *Simulating Altera Designs* in volume 3 of the *Quartus II Handbook*.

   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.

6. If the parameter editor includes **EDA** and **Summary** tabs, follow these steps:
   a. Some third-party synthesis tools can use a netlist that contains the structure of an IP core but no detailed logic to optimize timing and performance of the design containing it. To use this feature if your synthesis tool and IP core support it, turn on **Generate netlist**.
   b. On the **Summary** tab, if available, select the files you want to generate. A gray checkmark indicates a file that is automatically generated. All other files are optional.

   If file selection is supported for your IP core, after you generate the core, a generation report (<*variation name*>**.html)** appears in your project directory. This file contains information about the generated files.

7. Click the **Finish** button, the parameter editor generates the top-level HDL code for your IP core, and a simulation directory which includes files for simulation.

   The **Finish** button may be unavailable until all parameterization errors listed in the messages window are corrected.

8. Click **Yes** if you are prompted to add the Quartus II IP File (**.qip**) to the current Quartus II project. You can also turn on **Automatically add Quartus II IP Files to all projects**.

You can now integrate your custom IP core instance in your design, simulate, and compile. While integrating your IP core instance into your design, you must make appropriate pin assignments. You can create virtual pin to avoid making specific pin assignments for top-level signals while you are simulating and not ready to map the design to hardware.

For some IP cores, the generation process also creates a complete example design in the <*variation_name*>**_example_design_fileset/example_project/** directory. This example demonstrates how to instantiate and connect the IP core.

For information about the Quartus II software, including virtual pins and the MegaWizard Plug-In Manager, refer to Quartus II Help.

Simulate the IP Core

You can simulate your IP core variation with the functional simulation model and the testbench or example design generated with your IP core. The functional simulation model and testbench files are generated in a project subdirectory. This directory may also include scripts to compile and run the testbench.

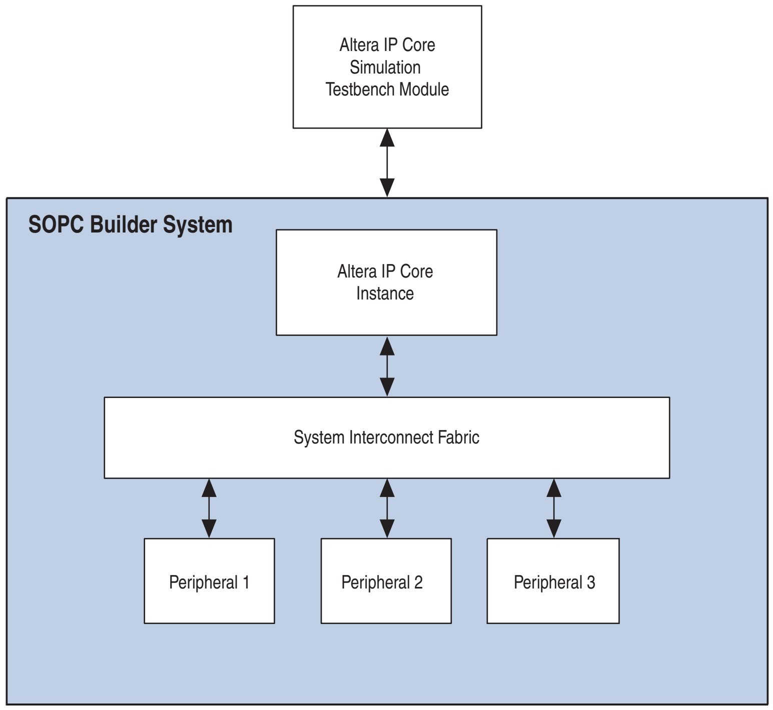
For a complete list of models or libraries required to simulate your IP core, refer to the scripts provided with the testbench.

For more information about simulating Altera IP cores, refer to *Simulating Altera Designs* in volume 3 of the *Quartus II Handbook*.

SOPC Builder Design Flow

You can use SOPC Builder to build a system that includes your customized IP core. You easily can add other components and quickly create an SOPC Builder system. SOPC Builder automatically generates HDL files that include all of the specified components and interconnections. SOPC Builder defines default connections, which you can modify. The HDL files are ready to be compiled by the Quartus II software to produce output files for programming an Altera device. SOPC Builder generates a simulation testbench module for supported cores that includes basic transactions to validate the HDL files. Figure 2–3 shows a block diagram of an example SOPC Builder system.

Figure 2–3. SOPC Builder System



For more information about system interconnect fabric, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* and *System Interconnect Fabric for Streaming Interfaces* chapters in the *SOPC Builder User Guide* and to the *Avalon Interface Specifications*.

For more information about SOPC Builder and the Quartus II software, refer to the *SOPC Builder Features* and *Building Systems with SOPC Builder* sections in the *SOPC Builder User Guide* and to Quartus II Help.

Specify Parameters

To specify IP core parameters in the SOPC Builder flow, follow these steps:

1. Create a new Quartus II project using the **New Project Wizard** available from the File menu.
2. On the Tools menu, click **SOPC Builder**.
3. For a new system, specify the system name and language.
4. On the **System Contents** tab, double-click the name of your IP core to add it to your system. The relevant parameter editor appears.
5. Specify the required parameters in the parameter editor. For detailed explanations of these parameters, refer to the *"Parameter Settings"* chapter in this document.

   Some IP cores provide preset parameters for specific applications. If you wish to use preset parameters, click the arrow to expand the **Presets** list, select the desired preset, and then click **Apply**. To modify preset settings, in a text editor edit the <*installation directory*>**\ip\altera\uniphy\lib\**<*IP core*>**.qprs** file.

   If your design includes external memory interface IP cores, you must turn on **Generate power of two bus widths** on the **PHY Settings** tab when parameterizing those cores.

6. Click **Finish** to complete the IP core instance and add it to the system.

   The **Finish** button may be unavailable until all parameterization errors listed in the messages window are corrected.

Complete the SOPC Builder System

To complete the SOPC Builder system, follow these steps:

1. Add and parameterize any additional components. Some IP cores include a complete SOPC Builder system design example.
2. Use the Connection panel on the **System Contents** tab to connect the components.
3. By default, clock names are not displayed. To display clock names in the **Module Name** column and the clocks in the **Clock** column in the **System Contents** tab, click **Filters** to display the **Filters** dialog box. In the **Filter** list, click **All.**
4. If you intend to simulate your SOPC builder system, on the **System Generation** tab, turn on **Simulation** to generate simulation files for your system.
5. Click **Generate** to generate the system. SOPC Builder generates the system and produces the <*system name*>**.qip** file that contains the assignments and information required to process the IP core or system in the Quartus II Compiler.
6. In the Quartus II software, click **Add/Remove Files in Project** and add the **.qip** file to the project.
7. Compile your design in the Quartus II software.

Simulate the System

During system generation, you can specify whether SOPC Builder 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 and plain-text RTL design files that describe your system in the ModelSim simulation software.

For information about the latest Altera-supported simulation tools, refer to the *Quartus II Software Release Notes*.

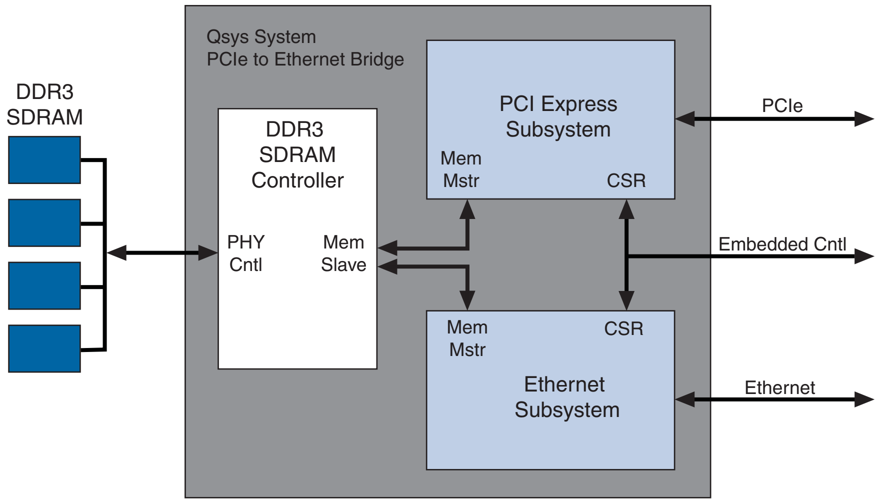
For information about simulating SOPC Builder systems, refer to the *SOPC Builder User Guide* and *AN 351: Simulating Nios II Embedded Processor Designs*.

For general information about simulating Altera IP cores, refer to *Simulating Altera Designs* in volume 3 of the *Quartus II Handbook*.

Qsys System Integration Tool Design Flow

You can use the Qsys system integration tool to build a system that includes your customized IP core. You easily can add other components and quickly create a Qsys system. Qsys automatically generates HDL files that include all of the specified components and interconnections. In Qsys, you specify the connections you want. The HDL files are ready to be compiled by the Quartus II software to produce output files for programming an Altera device. Qsys generates Verilog HDL simulation models for the IP cores that comprise your system. Figure 2–4 shows a high level block diagram of an example Qsys system.

Figure 2–4. Example Qsys System



For more information about the Qsys system interconnect, refer to the *Qsys Interconnect* chapter in volume 1 of the *Quartus II Handbook* and to the *Avalon Interface Specifications*.

For more information about the Qsys tool and the Quartus II software, refer to the *System Design with Qsys* section in volume 1 of the *Quartus II Handbook* and to Quartus II Help.

Specify Parameters

To specify parameters for your IP core using the Qsys flow, follow these steps:

1. Create a new Quartus II project using the **New Project Wizard** available from the File menu.
2. On the Tools menu, click **Qsys (Beta)**.
3. On the **System Contents** tab, double-click the name of your IP core to add it to your system. The relevant parameter editor appears.
4. Specify the required parameters in all tabs in the Qsys tool. For detailed explanations of these parameters, refer to the *"Parameter Settings"* chapter in this document.

   If your design includes external memory interface IP cores, you must turn on **Generate power of two bus widths** on the **PHY Settings** tab when parameterizing those cores.

   Some IP cores provide preset parameters for specific applications. If you wish to use preset parameters, click the arrow to expand the **Presets** list, select the desired preset, and then click **Apply**. To modify preset settings, in a text editor edit the **<*installation directory*>\ip\altera\uniphy\lib\<*IP core*>.qprs** file.

5. Click **Finish** to complete the IP core instance and add it to the system.

   The **Finish** button may be unavailable until all parameterization errors listed in the messages window are corrected.

Complete the Qsys System

To complete the Qsys system, follow these steps:

1. Add and parameterize any additional components.
2. Connect the components using the Connection panel on the **System Contents** tab.
3. In the **Export As** column, enter the name of any connections that should be a top-level Qsys system port. If the **Export As** column is not present, click the **Project Settings** tab and turn off **Use SOPC Builder port naming**.
4. If you intend to simulate your Qsys system, on the **Generation** tab, turn on one or more options under **Simulation** to generate desired simulation files.
5. If your system is not part of a Quartus II project and you want to generate synthesis RTL files, turn on **Create synthesis RTL files**.

6. Click **Generate** to generate the system. Qsys generates the system and produces the <*system name*>**.qip** file that contains the assignments and information required to process the IP core or system in the Quartus II Compiler.
7. In the Quartus II software, click **Add/Remove Files in Project** and add the **.qip** file to the project.
8. Compile your project in the Quartus II software.

Simulate the System

During system generation, Qsys generates a functional simulation model—or example design that includes a testbench—which you can use to simulate your system in any Altera-supported simulation tool.

For information about the latest Altera-supported simulation tools, refer to the *Quartus II Software Release Notes*.

For general information about simulating Altera IP cores, refer to *Simulating Altera Designs* in volume 3 of the *Quartus II Handbook*.

For information about simulating Qsys systems, refer to the *System Design with Qsys* section in volume 1 of the *Quartus II Handbook*.

HardCopy Migration Design Guidelines

If you intend to target your design to a HardCopy® device, ensure you use the following design guidelines:

- On the **General Settings** page of the **DDR2 SDRAM Controller with UniPHY** or **DDR3 SDRAM Controller with UniPHY** MegaWizard, turn on **HardCopy Compatibility Mode**, and then specify whether the **Reconfigurable PLL Location** is **Top_Bottom** or **Left_Right**.

  Altera recommends that you set the **Reconfigurable PLL Location** to the same side as your memory interface.

  When turned on, the **HardCopy Compatibility Mode** option enables run-time reconfiguration for all phase-locked loops (PLLs) and delay-locked loops (DLLs) instantiated in memory interfaces that are configured in PLL and DLL masters, and brings the necessary reconfiguration signals to the top level of the design.

  "Top-Level HardCopy Migration Signals" on page 6–12 lists the top-level signals generated for HardCopy migration.

- Enable run-time reconfiguration mode for all PLLs and DLLs instantiated in interfaces that are configured in PLL and DLL slaves.

For information about PLL megafunctions, refer to the *Phase-Locked Loop (ALTPLL) Megafunction User Guide* and the *Phase-Locked Loops Reconfiguration (ALTPLL_RECONFIG) Megafunctions User Guide*. For information about DLL megafunctions, refer to the *ALTDLL and ALTDQ_DQS Megafunctions User Guide*.

- Ensure that you place all memory interface pins close together. If, for example, address pins are located far away from data pins, closing timing might be difficult.

You can use the example top-level project that is generated when you turn on **HardCopy Migration** as a guide to help you connect the necessary signals in your design.

Differences in UniPHY IP Generated with HardCopy Migration Support

When you generate a UniPHY memory interface for HardCopy device support, certain features in the IP are enabled that do not exist when you generate the IP core for only the FPGA. This section discusses those additional enabled features.

ROM Loader for Designs Using Nios II Sequencer

An additional ROM loader is intantiated in the design for UniPHY designs that use the Nios II sequencer. The Nios II sequencer instruction code resides in RAM on either the HardCopy or FPGA device.

When you target only an FPGA device, the RAM is initialized when the device is programmed; however, HardCopy devices are not programmed and therefore the RAM cannot be initialized in this fashion. Instead, the Nios II sequencer instruction code must be stored in an external, non-volatile, ROM that loads the Nios II sequencer RAM through a ROM loader. You must attach the ROM loader to the appropriate pins connected to the external non-volatile ROM.

Table 2–1 summarizes the ports exposed at the top level of the PHY+Controller wrapper to expose the ROM loader utilized by the Nios II-based sequencer within the DDR2 or DDR3 PHY.

Table 2–1. Top-level Ports that Connect to External ROM for Loading Nios II Code Memory (Part 1 of 2)

Port Name	Direction	Description
`hc_rom_config_clock`	Input	Write clock for the ROM loader. This clock is the write clock for the Nios II code memory.
`hc_rom_config_datain`	Input	Data input from external ROM.
`hc_rom_config_rom_data_ready`	Input	Asserts to the code memory loader that the word of memory is ready to be loaded.
`hc_rom_config_init`	Input	Signals that the Nios II code memory is being loaded from the external ROM.
`hc_rom_config_init_busy`	Output	Remains asserted throughout initialization and becomes inactive when initialization is complete. `soft_reset_n` can be issued after `hc_rom_config_init_busy` is deasserted.

Table 2–1. Top-level Ports that Connect to External ROM for Loading Nios II Code Memory (Part 2 of 2)

Port Name	Direction	Description
`hc_rom_config_rom_rden`	Output	Read-enable signal that connects to the external ROM.
`hc_rom_config_rom_address`	Output	ROM address that connects to the external ROM.

PLL/DLL Run-time Reconfiguration

The PLLs and DLLs in the HardCopy design have run-time reconfiguration enabled—provided that they are not in PLL/DLL slave mode.

When the PLLs and DLLs are generated with reconfiguration enabled, there are extra signals that must be connected and driven by user logic. In the example design generated during IP core generation, the PLL/DLL reconfiguration signals are brought to the top level and connected to constants, as shown in Figure 2–5.

For information about PLL megafunctions and reconfiguration, refer to the *Phase-Locked Loop (ALTPLL) Megafunction User Guide* and the *Phase-Locked Loops Reconfiguration (ALTPLL_RECONFIG) Megafunctions User Guide*.

Figure 2–5. HardCopy UniPHY Example Design

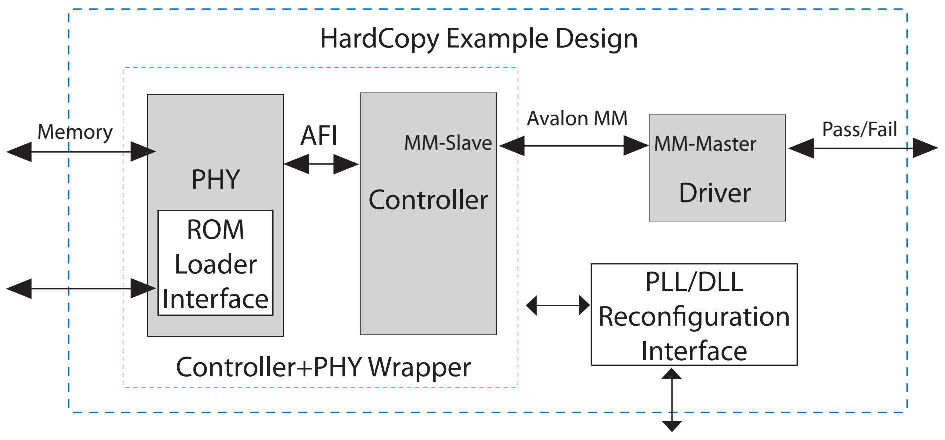


Table 2–2 summarizes the DLL reconfiguration ports exposed at the top level of the Controller+PHY.

Table 2–2. DLL Reconfiguration Ports Exposed at Top-Level of Controller+PHY Wrapper (Part 1 of

Port Name	Direction	Description
`hc_dll_config_dll_offset_ctrl_addnsub`	Input	Addition/subtraction control port for the DLL. This port controls if the delay-offset setting on hc_dll_config_dll_offset_ctrl_offset is added or subtracted.

Table 2–2. DLL Reconfiguration Ports Exposed at Top-Level of Controller+PHY Wrapper (Part 2 of

Port Name	Direction	Description
`hc_dll_config_dll_offset_ctrl_offset`	Input	Offset input setting for the PLL. This is a Gray-coded offset that is added or subtracted from the current value of the DLL's delay chain.
`hc_dll_config_dll_offset_ctrl_offsetctrlout`	Output	The registered and Gray-coded value of the current delay-offset setting.

Table 2–3 summarizes the ports exposed at the top level of the Controller and PHY wrapper to allow PLL reconfiguration.

Table 2–3. PLL Reconfiguration Ports Exposed at the Top-Level of Controller+PHY Wrapper

Port Name	Direction	Description
`hc_pll_config_configupdate`	Input	Control signal to enable PLL reconfiguration. (Applies to RLDRAMII and QDRII only, the phase reconfiguration feature for DDR2/3 is included in the CSR port.)
`hc_pll_config_phasecounterselect`	Input	Specifies the counter select for dynamic phase adjustment. (Applies to RLDRAMII and QDR II only.)
`hc_pll_config_phasestep`	Input	Specifies the phase step for dynamic phase shifting. (Applies to RLDRAMII and QDR II only.)
`hc_pll_config_phaseupdown`	Input	Specifies if the phase adjustment should be up or down. (Applies to RLDRAMII and QDR II only.)
`hc_pll_config_scanclk`	Input	PLL reconfiguration scan chain clock.
`hc_pll_config_scanclkena`	Input	Clock enable port of the `hc_pll_config_scanclk` clock.
`hc_pll_config_scandata`	Input	Serial input data for the PLL reconfiguration scan chain.
`hc_pll_config_phasedone`	Output	When asserted, this signal indicates to core logic that phase adjustment is completed and that the PLL is ready to act on a possible second adjustment pulse.
`hc_pll_config_scandataout`	Output	The data output of the serial scan chain.
`hc_pll_config_scandone`	Output	Asserted when the scan chain write operation is in progress and is deasserted when the write operation is complete.

To facilitate placement and timing closure and help compensate for PLLs adjacent to I/Os and vertical I/O overhang issues that can occur when targeting HardCopy III and HardCopy IV devices, an additional pipeline stage is added to the write path in the RTL when you turn on **HardCopy Compatibility**. The additional pipeline stage is added in all cases, except when CAS write latency equals 2 (for DDR3) or CAS latency equals 3 (for DDR2), where the additional pipeline stage is not required to meet timing requirements. The additional pipeline stage does not affect the overall latency of the controller.

For information about HardCopy issues such as vertical I/O overhang, PLLs adjacent to I/Os, and timing closure, refer to HardCopy III Device I/O Features in the *HardCopy III Device Handbook, Volume 1*, and HardCopy IV Device I/O Features in the *HardCopy IV Device Handbook, Volume 1*.

Generated Files

When you complete the IP generation flow, there are generated files created in your project directory. The directory structure created varies somewhat, depending on the tool used to parameterize and generate the IP.

The PLL parameters are statically defined in the <*variation_name*>**_parameters.tcl** at generation time. To ensure timing constraints and timing reports are correct, when you use the GUI to make changes to the PLL component, apply those changes to the PLL parameters in this file.

MegaWizard Plug-in Manager Flow

The tables in this section list the generated directory structure and key files of interest to users, resulting from the MegaWizard Plug-in Manager flow.

Synthesis

Table 2–4 lists the generated directory structure and key files created by the synthesis flow with the MegaWizard Plug-in Manager.

Table 2–4. Generated Directory Structure and Key Files—MegaWizard Plug-In Manager Synthesis Flow

Directory	File Name	Description
<*working_dir*>/	<*variation_name*>.qip	QIP file which refers to all generated files in the synthesis fileset.
<*working_dir*>/	<*variation_name*>.v (for Verilog), or <*variation_name*>.vhd (for VHDL)	Top-level wrapper for synthesis files.
<*working_dir*>/<*variation_name*>/	<*variation_name*>_<*stamp*>.v (1)	UniPHY top-level wrapper.
<*working_dir*>/<*variation_name*>/	<*variation_name*>_<*stamp*>_*.v (1)	UniPHY Verilog RTL files.
<*working_dir*>/<*variation_name*>/	<*variation_name*>_<*stamp*>_*.sv (1)	UniPHY SystemVerilog RTL files.
<*working_dir*>/<*variation_name*>/	<*variation_name*>_<*stamp*>.sdc (1)	Synopsys constraints file.
<*working_dir*>/<*variation_name*>/	<*variation_name*>_<*stamp*>.ppf (1)	Pin Planner file.
<*working_dir*>/<*variation_name*>/	<*variation_name*>_<*stamp*>_pin_assignments.tcl (1)	Pin constraints script to be run after synthesis.
<*working_dir*>/<*variation_name*>/	<*variation_name*>_<*stamp*>_*.tcl (1)	Other Tcl scripts.
<*working_dir*>/<*variation_name*>/	<*variation_name*>_<*stamp*>_readme.txt (1)	Readme text file.

Note to Table 2–4:

(1) <*stamp*> is a unique identifier determined by the MegaWizard Plug-in Manager at generation time.

Simulation

Table 2–5 lists the generated directory structure and key files created by the Verilog simulation flow with the MegaWizard Plug-in Manager.

Table 2–5. Generated Directory Structure and Key Files—MegaWizard Plug-In Manager Simulation Flow (Verilog)

Directory	File Name	Description
<*working_dir*>/<*variation_name*> _sim/	<*variation_name*>.v (for Verilog), or <*variation_name*>.vho (for VHDL)	UniPHY top-level wrapper.
<*working_dir*>/<*variation_name*> _sim/	<*variation_name*>_*.v	UniPHY Verilog RTL files.
<*working_dir*>/<*variation_name*> _sim/	<*variation_name*>_*.sv	UniPHY SystemVerilog RTL files.
<*working_dir*>/<*variation_name*> _sim/	<*variation_name*>_readme.txt	Readme text file.

Table 2–6 lists the generated directory structure and key files created by the VHDL simulation flow with the MegaWizard Plug-in Manager.

Table 2–6. Generated Directory Structure and Key Files—MegaWizard Plug-In Manager Simulation Flow (VHDL)

Directory	File Name	Description
<*working_dir*>/<*variation_name*> _sim/	<*variation_name*>.vho	UniPHY VHDL top-level module.
<*working_dir*>/<*variation_name*> _sim/	<*variation_name*>_*.vhd <*variation_name*>_*.vho	UniPHY simulation VHDL files.
<*working_dir*>/<*variation_name*> _sim/	vhdl_files.txt	File list text file.

Example Design

Table 2–7 lists the generated directory structure and key files created for the example design with the MegaWizard Plug-in Manager

Table 2–7. Generated Directory Structure and Key Files—MegaWizard Plug-In Manager Example Design (Part 1 of 2)

Directory	File Name‘	Description
<*working_dir*>/<*variation_name*> _example_design_fileset/	<*variation_name*>.qip	QIP which refers to UniPHY RTL in this fileset. This is distinct from ../<*variation_name*>.qip. This file is included automatically in the example project.
<*working_dir*>/<*variation_name*> _example_design_fileset/	<*variation_name*>.v	UniPHY top-level wrapper.
<*working_dir*>/<*variation_name*> _example_design_fileset	<*variation_name*>_*.v	UniPHY Verilog RTL files.
<*working_dir*>/<*variation_name*> _example_design_fileset	<*variation_name*>_*.sv	UniPHY SystemVerilog RTL fies.
<*working_dir*>/<*variation_name*> _example_design_fileset/	<*variation_name*>.sdc	Synopsys constraints file.

Table 2–7. Generated Directory Structure and Key Files—MegaWizard Plug-In Manager Example Design (Part 2 of 2)

Directory	File Name‘	Description
<*working_dir*>/<*variation_name*>_example_design_fileset/	<*variation_name*>.ppf	Pin Planner file.
<*working_dir*>/<*variation_name*>_example_design_fileset/	<*variation_name*>_pin_assignments.tcl	Pin constraints script to be run after synthesis.
<*working_dir*>/<*variation_name*>_example_design_fileset/	<*variation_name*>_*.tcl	Other Tcl scripts.
<*working_dir*>/<*variation_name*>_example_design_fileset/	<*variation_name*>_readme.txt	Readme text file.
<*working_dir*>/<*variation_name*>_example_design_fileset/example_project/	<*variation_name*>_example_top.qpf	Example design project file.
<*working_dir*>/<*variation_name*>_example_design_fileset/example_project/	<*variation_name*>_example_top.qsf	Example design project settings file.
<*working_dir*>/<*variation_name*>_example_design_fileset/example_project/	<*variation_name*>_example_top.v	Top-level wrapper including UniPHY, traffic generator, and memory model.
<*working_dir*>/<*variation_name*>_example_design_fileset/example_project/	<*variation_name*>_*.v	Other example design Verilog RTL files.
<*working_dir*>/<*variation_name*>_example_design_fileset/example_project/	<*variation_name*>_*.sv	Other example design SystemVerilog RTL files.
<*working_dir*>/<*variation_name*>_example_design_fileset/example_project/	<*prefix*>_mem_model.sv (1)	Generic memory model.
<*working_dir*>/<*variation_name*>_example_design_fileset/rtl_sim/	<*variation_name*>_example_top_tb.v	Top-level test bench.

Note to Table 2–7:

(1) <*prefix*> varies depending on protocol and type of memory model.

SOPC Builder Flow

Table 2–8 lists the generated directory structure and key files created by the SOPC Builder flow.

Table 2–8. Generated Directory Structure and Key Files—SOPC Builder Flow (Part 1 of 2)

Directory	File Name‘	Description
<*working_dir*>/	<*system_name*>.qip	QIP which refers to all generated files in the SOPC Builder project.
<*working_dir*>/	<*system_name*>.v	SOPC Builder system top-level wrapper.
<*working_dir*>/	<*core_name*>_<*stamp*>.v (1)	UniPHY top-level wrapper.
<*working_dir*>/	<*core_name*>_<*stamp*>_*.v (1)	UniPHY Verilog RTL files.
<*working_dir*>/	<*core_name*>_<*stamp*>_*.sv (1)	UniPHY SystemVerilog RTL files.
<*working_dir*>/	<*core_name*>_<*stamp*>.sdc (1)	Synopsys constraints file.

Table 2–8. Generated Directory Structure and Key Files—SOPC Builder Flow (Part 2 of 2)

Directory	File Name‘	Description
<working_dir>/	*<core_name>*_*<stamp>*.ppf (1)	Pin Planner file.
<working_dir>/	*<core_name>*_*<stamp>*_pin_assignments.tcl (1)	Pin constraints script to be run after synthesis.
<working_dir>/	*<core_name>*_*<stamp>*_*.tcl (1)	Other Tcl scripts.
<working_dir>/	*<core_name>*_*<stamp>*_readme.txt (1)	Readme text file.
<working_dir>/	*Other IP core files.*	Other IP cores.

Note to Table 2–8:

(1) *<stamp>* is a unique identifier determined by SOPC Builder at generation time.

Qsys Flow

The tables in this section list the generated directory structure and key files of interest to users, resulting from the Qsys flow

Synthesis

Table 2–9 lists the generated directory structure and key files created by the synthesis flow with Qsys.

Table 2–9. Generated Directory Structure and Key Files—Qsys Synthesis Flow (Part 1 of 2)

Directory	File Name	Description
<working_dir>/*<system_name>*/ synthesis/	*<system_name>*.qip	QIP which refers to all generated files in the Qsys system synthesis fileset.
<working_dir>/*<system_name>*/ synthesis/	*<system_name>*.v	Qsys system top-level wrapper.
<working_dir>/*<system_name>*/ synthesis/submodules/	*<core_name>*_*<stamp>*.v (1)	UniPHY top-level wrapper.
<working_dir>/*<system_name>*/ synthesis/submodules/	*<core_name>*_*<stamp>*_*.v (1)	UniPHY Verilog RTL files.
<working_dir>/*<system_name>*/ synthesis/submodules/	*<core_name>*_*<stamp>*_*.sv (1)	UniPHY SystemVerilog RTL files.
<working_dir>/*<system_name>*/ synthesis/submodules/	*<core_name>*_*<stamp>*.sdc (1)	Synopsys constraints file.
<working_dir>/*<system_name>*/ synthesis/submodules/	*<core_name>*_*<stamp>*.ppf (1)	Pin Planner file.
<working_dir>/*<system_name>*/ synthesis/submodules/	*<core_name>*_*<stamp>*_pin_assignments.tcl (1)	Pin constraints script to be run after synthesis.
<working_dir>/*<system_name>*/ synthesis/submodules/	*<core_name>*_*<stamp>*_*.tcl (1)	Other Tcl scripts.
<working_dir>/*<system_name>*/ synthesis/submodules/	*<core_name>*_*<stamp>*_readme.txt (1)	Readme text file.

Table 2–9. Generated Directory Structure and Key Files—Qsys Synthesis Flow (Part 2 of 2)

Directory	File Name	Description
<*working_dir*>/<*system_name*>/ synthesis/submodules/	*Other IP core files*	Other IP core files.

Note to Table 2–9

(1) <*stamp*> is a unique identifier created by Qsys during generation.

Verilog Simulation

Table 2–10 lists the generated directory structure and key files created by the Verilog HDL simulation flow with Qsys.

Table 2–10. Generated Directory Structure and Key Files—Qsys Verilog Simulation

Directory	File Name	Description
<*working_dir*>/<*system_name*>/ sim_verilog/	<*system_name*>.v	Qsys system top-level wrapper.
<*working_dir*>/<*system_name*>/ sim_verilog/submodules/	<*core_name*>_<*stamp*>.v	UniPHY top-level wrapper.
<*working_dir*>/<*system_name*>/ sim_verilog/submodules/	<*core_name*>_<*stamp*>_*.v	UniPHY Verilog RTL files.
<*working_dir*>/<*system_name*>/ sim_verilog/submodules/	<*core_name*>_<*stamp*>_*.sv	UniPHY SystemVerilog RTL files.
<*working_dir*>/<*system_name*>/ sim_verilog/submodules/	<*core_name*>_<*stamp*>_readme.txt (1)	Readme text file.
<*working_dir*>/<*system_name*>/ sim_verilog/submodules/	*Other IP core files*	Other IP core files.

Note for Table 2–10:

(1) <*stamp*> is a unique identifier created by Qsys during generation.

VHDL Simulation

Table 2–11 lists the generated directory structure and key files created by the VHDL simulation flow with Qsys.

Table 2–11. Generated Directory Structure and Key Files—Qsys VHDL Simulation

Directory	File Name	Description
<*working_dir*>/<*system_name*>/ sim_vhdl/	<*system_name*>.vhd	Qsys system top-level wrapper.
<*working_dir*>/<*system_name*>/ sim_vhdl/submodules/	<*core_name*>_<*stamp*>.vho (1)	UniPHY VHDL top-level module.
<*working_dir*>/<*system_name*>/ sim_vhdl/submodules/	<*core_name*>_<*stamp*>_*.vhd <*core_name*>_<*stamp*>_*.vho (1)	UniPHY VHDL simulation files.
<*working_dir*>/<*system_name*>/ sim_vhdl/submodules/	vhdl_files.txt	File list text file.

Note to Table 2–11:

(1) <*stamp*> is a unique identifier created by Qsys during generation.

3. Parameter Settings

This chapter describes the QDR II and QDR II+ SRAM Controller with UniPHY IP core parameters that you can set in the GUI.

General Settings

The **General Settings** tab allows you to configure the following parameter settings.

Clocks

Table 3–1 describes the clock settings.

Table 3–1. Clock Settings

Parameter	Description
Memory clock frequency	The frequency of the clock that drives the memory device.
PLL reference clock frequency	The frequency of the clock that feeds the PLL.
Full or half rate on Avalon-MM interface	Defines the width of the data bus on the Avalon-MM interface. A setting of **Full** results in a width twice the memory data width. A setting of **Half** results in a width of four times the memory data width.
Additional address/command clock phase	Increases or decreases the phase shift of the address/command clock. The base phase shift center-aligns the address/command clock at the memory device. In some circumstances, you can improve timing by increasing or decreasing the phase shift.

Advanced PHY Settings

Table 3–2 describes the advanced PHY settings.

Table 3–2. Advanced PHY Settings

Parameter	Description
Generate power-of-2 bus widths	Rounds down the Avalon-MM side data bus to the nearest power of 2.
Maximum Avalon-MM burst length	Specifies the maximum burst length on the Avalon-MM bus.
I/O standard	Specifies the I/O standard voltage.
Master for PLL/DLL sharing	Causes UniPHY to instantiate its own PLL and DLL. All of the PLL clocks and DLL delay values are exported for use by other identical UniPHY cores that have this option turned on.

Table 3–2. Advanced PHY Settings

Parameter	Description
Master for OCT control block	Causes UniPHY to instantiate the required OCT control block. When this parameter is turned off, you must instantiate this block and connect the termination control bus signals to the PHY, or share an OCT control block from another UniPHY instantiation that is in master mode.
Hardcopy compatibility mode	Causes the generated UniPHY memory interface to have all required HardCopy compatibility options enabled. For example, PLLs and DLLs will have their reconfiguration ports exposed.
Example Testbench Simulation options	
Skip memory initialization	Causes the example testbench to skip the memory initialization sequence. This setting does not change the generated RTL, but can speed up simulation.

Topology

Table 3–3 describes the topology settings.

Table 3–3. Topology Settings

Parameter	Description
Device width	Specifies the number of devices used for width expansion.
Device depth	Specifies the number of devices (ranks) used for depth expansion.

Controller Settings

Table 3–4 describes the controller settings.

Table 3–4. Controller Settings

Parameter	Description
Controller latency	Specifies the number of clock cycles required for a request to pass through an idling controller.

Memory Parameters

The **Memory Parameters** tab allows you to configure memory device parameters. You can enter parameters manually from the manufacturer’s device data sheet, or you can populate the fields automatically by selecting the required device from the list of presets.

Table 3–5 describes the memory parameters.

Table 3–5. Memory Parameters (Part 1 of 2)

Parameter	Description
Address width	The width of the address bus on the memory device.
Data width	The width of the data bus on the memory device.

Table 3–5. Memory Parameters (Part 2 of 2)

Parameter	Description
Data-mask width	The width of the data-mask on the memory device,
CQ width	The width of the CQ (read strobe) bus on the memory device.
K width	The width of the K (write strobe) bus on the memory device.
Burst length	The burst length supported by the memory device.

Memory Timing

The **Memory Timing** tab allows you to configure memory device timing parameters. You can enter timing parameters manually from the manufacturer's device data sheet, or you can populate the fields automatically by selecting the required device from the list of presets.

Table 3–6 describes the memory timing parameters.

Table 3–6. Memory Timing Parameters

Parameters	Description
tWL (cycles)	The write latency.
tRL (cycles)	The read latency.
tSA	The address and control setup to K clock rise.
tHA	The address and control hold after K clock rise.
tSD	The data setup to clock (K/K#) rise.
tHD	The data hold after clock (K/K#) rise.
tCQD	Echo clock high to data valid.
tCQDOH	Echo clock high to data invalid.
Internal jitter	The QDRII/II+ internal jitter.
TCQHCQnH	The CQ clock rise to CQn clock rise (rising edge to rising edge).
TKHKnH	The K clock rise to Kn clock rise (rising edge to rising edge).

Board Settings

The **Board Settings** tab allows you to enter values derived from board simulation.

Intersymbol Interference

Intersymbol interference (ISI), occurs when a signal is distorted due to the interference of one symbol with subsequent symbols. Typically, ISI is greater at device depths greater than 1 because there are multiple stubs causing reflections. The advanced I/O timing analysis capabilities of the Quartus II software already includes ISI effects for device depth of 1.

Table 3–7 describes the intersymbol interference settings.

Table 3–7. Intersymbol Interference Settings

Parameter	Description
Address/command eye reduction (setup)	The reduction in the eye diagram on the setup side (or left side of the eye) due to ISI on the address/command signals compared to a case where there is no ISI.
Address/command eye reduction (hold)	The reduction in the eye diagram on the hold side (or right side of the eye) due to ISI on the address/command signals compared to a case where there is no ISI.
D eye reduction	The total reduction in the eye diagram due to ISI on DQ signals compared to a case where there is no ISI. (It is assumed that the ISI reduces the eye width symmetrically on the left and right sides of the eye.)
Delta K arrival time	The increase in variation on the range of arrival times of DQS compared to a case when there is no ISI. (It is assumed that the ISI causes DQS to further vary symmetrically to the left and right.)

Board Skews

Skews between PCB traces can reduce timing margins.

Table 3–8 describes the board skew settings.

Table 3–8. Board Skews Settings (Part 1 of 2) (Part 1 of 2)

Parameter	Description
Maximum delay difference between devices	The maximun delay difference of data signals between devices. For example, in a two-device configuration there is greater propagation delay for data signals going to and returning from the furthest device relative to the nearest device.
Maximum skew within write data group (ie, K group)	The maximum skew between D and BWS signals referenced by a common K signal.
Maximun slkew within read data group (ie, CQ group)	The maximum skew between Q signals referenced by a common CQ signal.
Maximum skew between CQ groups	The maximum skew between CQ signals of different read data groups.

Table 3–8. Board Skews Settings (Part 2 of 2) (Part 2 of 2)

Parameter	Description
Maximun skew within address/command bus	The maximum skew between the address/command signals.
Average delay difference between address/command and K	A value equal to the average of the longest and smallest address/command signal delay values, minus the delay of the K signal. The value can be positive or negative.
Average delay difference between write data signals and K	A value equal to the average of the longest and smallest write data signal delay values, minus the delay of the K signal. Write data signals include the D and BWS signals. The value can be positive or negative.
Average delay difference between read data signals and CQ	A value equal to the average of the longest and smallest read data signal delay values, minus the delay of the CQ signal. The value can be positive or negative.

4. Constraining and Compiling

The parameter editor generates a Synopsis Design Constraint (**.sdc**) script. <*variation_name*>**.sdc**, and a pin assignment script, <*variation_name*>**_pin_assignments.tcl**. Both the **.sdc** and the <*variation name*>**_pin_assignments.tcl** support multiple instances. These scripts iterate through all instances of the core and apply the same constraints to all of them.

Add Pin and DQ Group Assignments

The pin assignment script, <*variation_name*>**_pin_assignments.tcl**, sets up the I/O standards and the input/output termination for the QDR II or QDR II+ SRAM controller with UniPHY. This script also helps to relate the `DQ` and `QK` pin groups together for the Fitter to place them correctly in the device.

The pin assignment script does not create a clock for the design. You must create a clock for the design and provide pin assignments for the signals of both the example driver and testbench that the MegaCore variation generates.

Run the <*variation_name*>**_pin_assignments.tcl** to add the input and output termination, I/O standards, and DQ group assignments to the example design. To run the pin assignment script, follow these steps:

1. On Processing menu, point to **Start**, and click **Start Analysis and Synthesis**.
2. On the Tools menu click **Tcl Scripts**.
3. Specify the **pin_assignments.tcl** file and click **Run**.

If the PLL input reference clock pin does not have the same I/O standard as the memory interface I/Os, the design might not fit into the device because incompatible I/O standards cannot be placed in the same I/O bank.

Board Settings

The **Board Settings** tab allows you to enter board-related data. In the **Intersymbol Interference** and **Board Skews** sections, you enter information derived during your PCB development process of prelayout (line) and postlayout (board) simulation.

Timing analysis does not consider bus turnaround; consequently, the controller dead times are based on assumptions about the user board trace lengths. For timing analysis to be accurate, board trace delays must be limited to 0.6 ns from FPGA to memory and from memory to FPGA.

For more information about how to include your board simulation results in the Quartus II software and how to assign pins using pin planners, refer to *Volume 6: Design Flow Tutorials* of the *External Memory Interface Handbook*.

Compile the Design

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

After you have compiled the top-level file, you can perform RTL simulation or program your targeted Altera device to verify the top-level file in hardware.

For more information about simulating, refer to the *Simulation* section in volume 4 of the *External Memory Interface Handbook*.

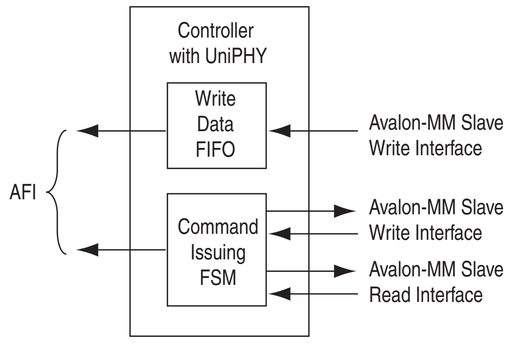
5. Functional Description—Controller

The controller translates memory requests from the Avalon Memory-Mapped (Avalon-MM) interface to AFI, while satisfying timing requirements imposed by the memory configurations. QDR II and QDR II+ SRAM has unidirectional data buses, therefore read and write operations are highly independent of each other and each has its own interface and state machine.

Block Description

This topic describes the blocks in the IP. Figure 5–1 shows a block diagram of the QDR II and QDR II+ SRAM controller architecture.

Figure 5–1. QDR II and QDR II+ SRAM Controller Architecture Block Diagram


Controller with UniPHY
Write Data FIFO
Command Issuing FSM
AFI
Avalon-MM Slave Write Interface
Avalon-MM Slave Write Interface
Avalon-MM Slave Read Interface

Avalon-MM Slave Read and Write Interfaces

The read and write blocks accept from the Avalon-MM interface read and write requests respectively. Each block has a simple state machine that represents the state of the command and address registers, which stores the command and address when a request arrives.

The read data passes through without the controller registering it, as the PHY takes care of read latency. The write data goes through a pipeline stage to delay for a fixed number of cycles as specified by the write latency. In the full-rate burst length of four controller, the write data is also multiplexed into a burst of 2, which is then multiplexed again in the PHY to become a burst of 4 in DDR.

The user interface to the controller has separate read and write Avalon-MM interfaces because reads and writes are independent of each other in the memory device. The separate channels give efficient use of available bandwidth.

Command Issuing FSM

The command-issuing full-state machine (FSM) has two states: `INIT` and `INIT_COMPLETE`. In the `INIT_COMPLETE` state, commands are issued immediately as requests arrive using combinational logic and do not require state transitions.

AFI

In the full-rate burst length of two configuration, the controller can issue both read and write commands in the same clock cycle. In the memory device, both commands are clocked on the positive edge, but the read address is clocked on the positive edge, while the write address is clocked on the negative edge. Care must be taken on how these signals are ordered in the AFI.

For the half-rate burst length of four configuration the controller also issues both read and write commands, but the AFI width is doubled to fill two memory clocks per controller clock. As the controller only issues one write command and one read command per controller clock, the AFI read and write signals corresponding to the other memory cycle are tied to no operation (NOP).

For information on the AFI, refer to "Functional Description—UniPHY" on page 6–1.

Avalon-MM and Memory Data Width

Table 5–1 shows the data width ratio between the memory interface and the Avalon-MM interface. The half-rate controller does not support burst-of-2 devices because it under-uses the available memory bandwidth. Regardless of full or half-rate decision and the device burst length, the Avalon-MM interface must supply all the data for the entire memory burst in a single clock cycle. Therefore the Avalon-MM data width of the full-rate controller with burst-of-4 devices is four times the memory data width. For width-expanded configurations, the data width is further multiplied by the expansion factor (not shown in table 5-1 and 5-2).

Table 5–1. Data Width Ratio

Memory Burst Length	Half-Rate Designs	Full-Rate Designs
QDR II 2-word burst	No Support	2:1
QDR II and II+ 4-word burst	4:1	

Signal Description

This topic discusses the signals for each interface.

For information on the AFI signals, refer to "UniPHY Signals" on page 6–10.

Avalon-MM Slave Read Interface

Table 5–2 shows the list of signals of the controller's Avalon-MM slave read interface.

Table 5–2. Avalon-MM Slave Read Signals (Part 1 of 2)

Signal	Width	Direction	Avalon-MM Signal Type	Description
`avl_r_ready`	1	Out	`waitrequest_n`	—
`avl_r_read_req`	1	In	`read`	—
`avl_r_addr`	≤20	In	`address`	—
`avl_r_rdata_valid`	1	Out	`readdatavalid`	—

Table 5–2. Avalon-MM Slave Read Signals (Part 2 of 2)

Signal	Width	Direction	Avalon-MM Signal Type	Description
avl_r_rdata	16, 18, 36, 72, 144	Out	readdata	—
avl_r_size	log_2(MAX_BURST_SIZE) + 1		—	—

The data width of the Avalon-MM interface is restricted to powers of two when using SOPC Builder or Qsys. Non-power-of-two data widths are supported when using the MegaWizard Plug-In Manager.

Avalon-MM Slave Write Interface

Table 5–3 shows the list of signals of the controller's Avalon-MM slave write interface.

Table 5–3. Avalon-MM Slave Write Signals

Signal	Width	Direction	Avalon-MM Signal Type	Description
avl_w_ready	1	Out	waitrequest_n	—
avl_w_write_req	1	In	write	—
avl_w_addr	≤20	In	address	—
avl_w_wdata	18, 36, 72, 144	In	writedata	—
avl_w_be	2,4,8,16	In	byteenable	—
avl_w_size	log_2(MAX_BURST_SIZE) + 1		—	—

6. Functional Description—UniPHY

This chapter describes the PHY part of the QDR II and QDR II+ SRAM controllers with UniPHY.

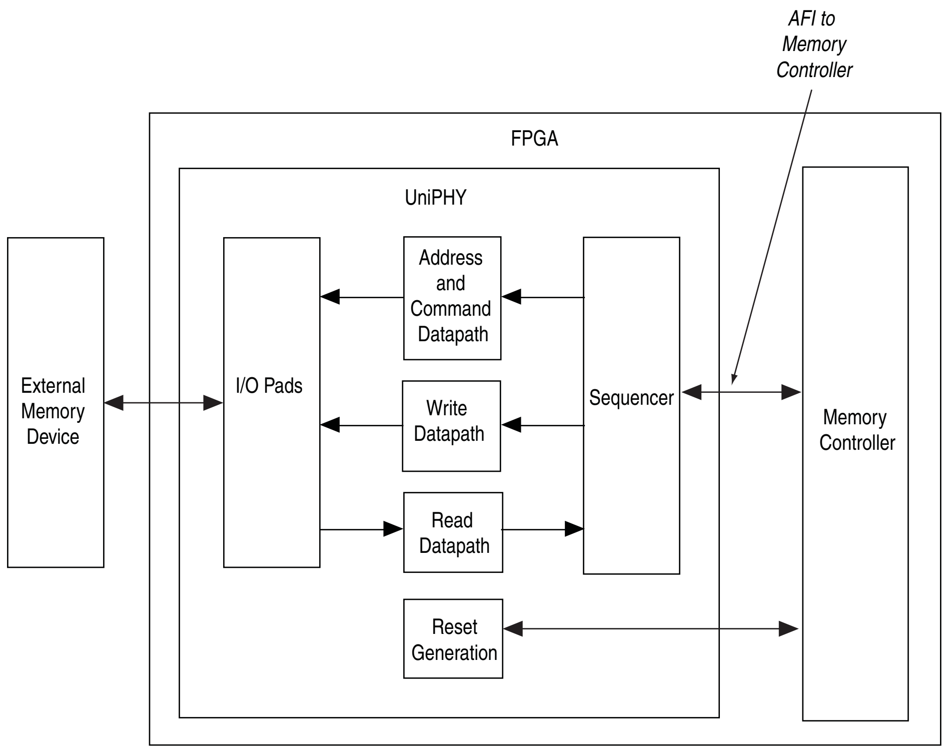
Block Description

The PHY comprises the following major functional units:

- Reset and Clock Generation
- Address and Command Datapath
- Write Datapath
- Read Datapath
- Sequencer

Figure 6–1 shows the PHY block diagram.

Figure 6–1. PHY Block Diagram



I/O Pads

The I/O pads contain all the I/O instantiations. The bulk of the UniPHY I/O circuitry is encapsulated in the ALTDQ_DQS megafunction (ALTDQ_DQS2 for Stratix V series devices).

Reset and Clock Generation

The clocking operation in the PHY can be classified into two domains: the PHY-memory domain and the PHY-AFI domain. The PHY-memory domain interfaces with the external memory device and is always at full-rate. The PHY-AFI domain interfaces with the memory controller and can be either a full-rate or half-rate clock based on the choice of the controller. Table 6–1 lists the clocks required for half-rate designs.

Table 6–1. Clocks—Half-Rate Designs

Clock	Source	Clock Rate	Phase	Clock Network Type	Description
`pll_afi_clk`	PLL: C0	Half	0°	Unconstrained	Clock for AFI logic.
`pll_mem_clk`	PLL: C1	Full	0°(1) -45°(2)	Dual-regional(4)	Output clock to memory.
`pll_write_clk`	PLL: C2	Full	-90°(1) -135°(2) 45° (3)	Dual-regional(4)	Clock for write data out to memory (data is center aligned with the delayed pll_write_clk).
`pll_addr_cmd_clk`	PLL: C3	Half	Set in wizard (default 270°)	Dual-regional	Clock for the address and command out to memory (address and command is center aligned with memory clock).
`read_capture_clk`	Memory	Full	90°	Local	A continuous running clock from the memory device for capturing read data.

Notes for Table 6–1:

(1) For memory frequencies >240 MHz.
(2) For memory frequencies <=240 MHz.
(3) For memory frequencies >=240 MHz, for Stratix V devices only.
(4) For parameterizations with interface width >36, pll_mem_clk and pll_write_clk are assigned to use the global network.

Table 6–2 lists the clocks required for full-rate designs.

Table 6–2. Clocks—Full-Rate Designs (Part 1 of 2)

Clock	Source	Clock Rate	Phase	Clock Network Type	Description
`pll_afi_clk`	PLL: C0	Full	0°	Unconstrained	Clock for AFI logic.
`pll_mem_clk`	PLL: C1	Full	90°(1) 0°(2)	Dual-regional(3)	Output clock to memory.
`pll_write_clk`	PLL: C2	Full	180°(1) -90°(2)	Dual-regional(3)	Clock for write data out to memory (data is center aligned with the delayed pll_write_clk).
`pll_addr_cmd_clk`	PLL: C3	Full	Set in wizard (default 225°)	Dual-regional	Clocks address/command out to memory. 180° gives adress and command center aligned with memory clock; 225° produces best overall timing results.

Table 6–2. Clocks—Full-Rate Designs (Part 2 of 2)

Clock	Source	Clock Rate	Phase	Clock Network Type	Description
`read_capture_clk`	Memory	Full	90°	Local	A continuous running clock from the memory device for capturing read data.

Notes for Table 6–2:

(1) For memory frequencies >240 MHz.

(2) For memory frequencies <=240 MHz.

(3) For parameterizations with interface width >36, pll_mem_clk and pll_write_clk are assigned to use the global network.

The UniPHY uses an active-low, asynchronous assert and synchronous deassert reset scheme. The global reset signal resets the PLL in the PHY and the rest of the system waits in reset until after the PLL is locked. The number of synchronization pipeline stages is 4.

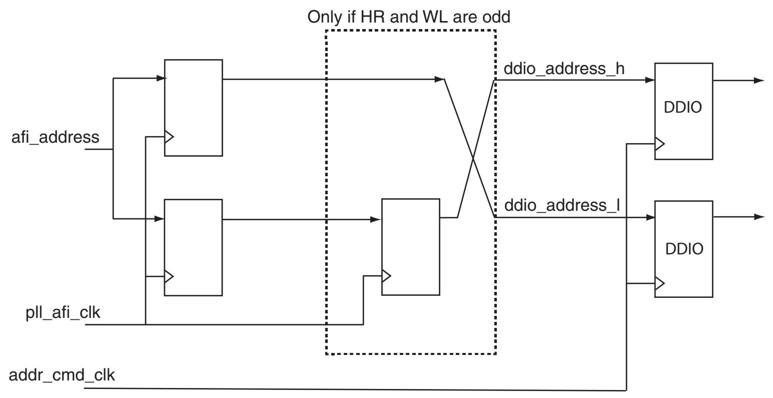
Address and Command Datapath

The memory controller controls the read and write addresses and commands to meet the memory specifications. The PHY simply passes the address and command received from the memory controller to the memory device. The PHY is also indifferent to address or command; the circuitry is the same for both.

The address and command datapath outputs are connected to the inputs of the address and command I/Os . An ALTDDIO_OUT megafunction converts the addresses from SDR to DDR. An ALTDDIO_OUT megafunction with an ALTIOBUF megafunction delivers a pair of address and command clock to the memory.

Figure 6–2 illustrates the address and command datapath. The controller only requires the registry-and- address-swapping circuitry inside the dotted box when it is operating in half-rate (HR) mode with odd write latency (WL). In full-rate mode, `ddio_address_h` and `ddio_addesss_l` are the same.

Figure 6–2. Address and Command Datapath

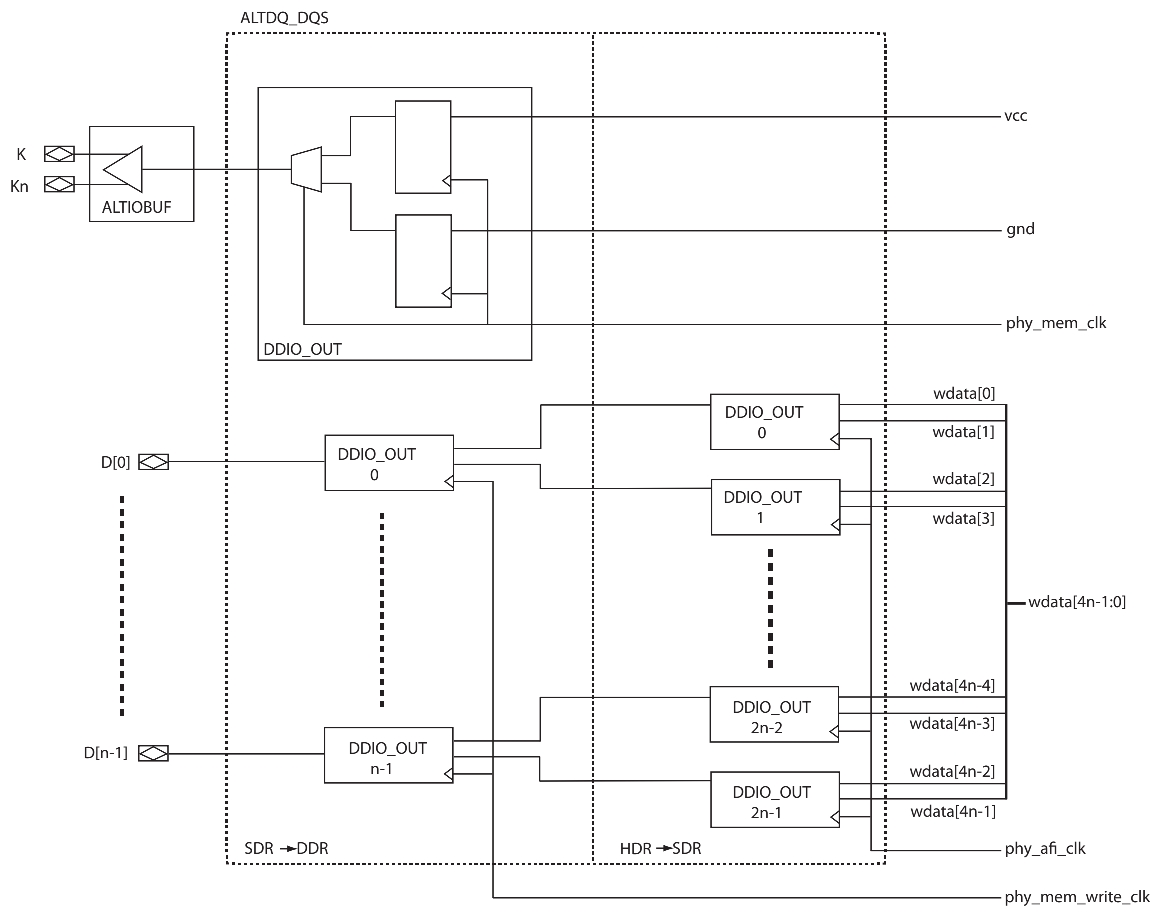


Write Datapath

The write datapath passes write data from the memory controller to the I/O. The QDR II interface has separate unidirectional read and write pins. The write pins are not controlled by the output-enable signal, and are always driven. Because the pins are unidirectional, there is no need for dynamic termination control.

Figure 6–3 illustrates the write datapath. The full-rate PLL output clock phy_mem_clk is sent to a DDIO_OUT cell. The output of ALTDQ_DQS feeds an ALTIOBUF buffer which creates a pair of pseudodifferential clocks that connects to the memory. In full-rate mode, only the SDR-DDR portion of the ALTDQ_DQS logic is used; in half-rate mode, the HDR-SDR circuitry is also required. The *<variation_name>*_**pin_assignments.tcl** script automatically specifies the logic option that associates all DQ pins to the DQS pin. The Fitter treats the pins as a DQS/DQ pin group.

Figure 6–3. Write Datapath



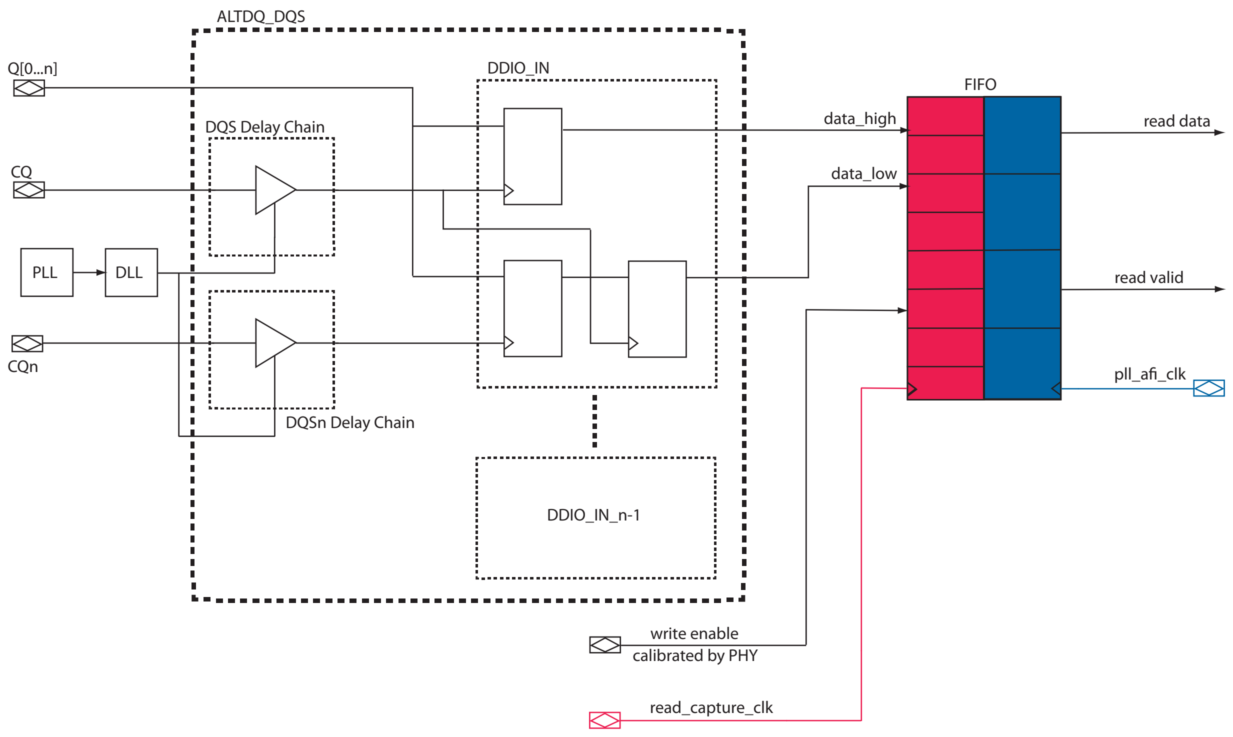
Read Datapath

The read data is captured in the input mode ALTDQ_DQS in the I/O. The captured data is then forwarded to the read datapath. The read datapath synchronizes read data from the read capture clock domain to the AFI clock domain and converts data from SDR to HDR (half-rate designs only).

In half-rate designs, the write side of the FIFO buffer should be double the size of the read side of the FIFO buffer. The read side only reads one entry after the write side has written into two entries, which effectively converts data from SDR to HDR. In full-rate designs, the size of the FIFO buffer is the same for both write and read as both sides operate at the same rate. For half-rate designs, the FIFO operates at half-rate on both read and write sides, and contains 4 half-rate entries; for full-rate designs, the FIFO operates at full-rate on both read and write sides, and contains 8 full-rate entries.

Figure 6–4 illustrates the read datapath. The DQS and DQSn clocks and the read data (DQ) returned from memory are edge-aligned; the DQS and DQSn delay chains shift the clocks to achieve center alignment.

Figure 6–4. Read Datapath



Sequencer

The sequencer is a state machine that processes the calibration algorithm. The sequencer assumes control of the interface at reset (whether at initial startup or when the IP is reset) and maintains control throughout the calibration process, relinquishing control to the memory controller only after successful calibration. Table 6–3 shows the major states in the sequencer.

Table 6–3. Sequencer States

State	Description
RESET	Remain in this state until reset is released.
LOAD_INIT	Load any initialization values for simulation purposes.
STABLE	Wait until the memory device is stable. The QDR II and QDR II+ specification requires 2,048 cycles of power up wait time.

Table 6–3. Sequencer States

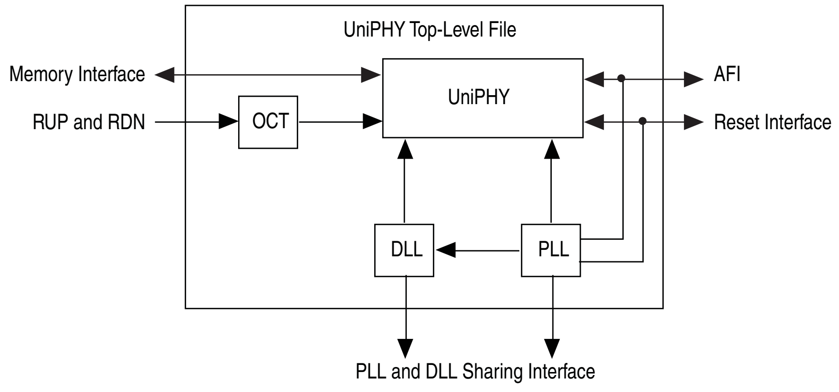
State	Description
WRITE_ZERO	Issue write command to address 0.
WAIT_WRITE_ZERO	Write all 0s to address 0.
WRITE_ONE	Issue write command to address 1.
WAIT_WRITE_ONE	Write all 1s to address 1.
Valid Calibration States	
V_READ_ZERO	Issue read command to address 0 (expected data is all 0s).
V_READ_NOP	This state represents the minimum number of cycles required between 2 back-to-back read commands. The number of NOP states depends on the burst length.
V_READ_ONE	Issue read command to address 1 (expected data is all 1s).
V_WAIT_READ	Wait for read valid signal.
V_COMPARE_READ_ZERO_READ_ONE	Parameterizable number of cycles to wait before making the read data comparisons.
V_CHECK_READ_FAIL	When a read fails, the write pointer (in the AFI clock domain) of the valid FIFO buffer is incremented. The read pointer of the valid FIFO buffer is in the DQS clock domain. The gap between the read and write pointers is effectively the latency between the time when the PHY receives the read command and the time valid data is returned to the PHY.
V_ADD_FULL_RATE	Advance the read valid FIFO buffer write pointer by an extra full rate cycle.
V_ADD_HALF_RATE	Advance the read valid FIFO buffer write pointer by an extra half rate cycle. In full-rate designs, equivalent to V_ADD_FULL_RATE.
V_READ_FIFO_RESET	Reset the read and write pointers of the read data synchronization FIFO buffer.
V_CALIB_DONE	Valid calibration is successful.
Latency Calibration States	
L_READ_ONE	Issue read command to address 1 (expected data is all 1s).
L_WAIT_READ	Wait for read valid signal from read datapath. Initial read latency is set to a predefined maximum value.
L_COMPARE_READ_ONE	Check returned read data against expected data. If data is correct, go to L_REDUCE_LATENCY; otherwise go to L_ADD_MARGIN.
L_REDUCE_LATENCY	Reduce the latency counter by 1.
L_READ_FLUSH	Read from address 0 (expected data is all 0s), to flush the contents of the read data resynchronization FIFO buffer.
L_WAIT_READ_FLUSH	Wait until the whole FIFO buffer is flushed, then go back to L_READ and try again.
L_ADD_MARGIN	Increment latency counter by 3 (1 cycle to get the correct data, 2 more cycles of margin for run time variations). If latency counter value is smaller than predefined ideal condition minimum, then go to CALIB_FAIL.
CALIB_DONE	Calibration is successful.
CALIB_FAIL	Calibration is not successful.

Interfaces

Figure 6–5 shows the major blocks of the UniPHY and how it interfaces with the external memory device and the controller.

Instantiating the DLL and the PLL on the same level as the UniPHY eases DLL and PLL sharing.

Figure 6–5. UniPHY Interfaces with the Controller and the External Memory



The following interfaces are on the UniPHY top-level file:

- AFI
- Memory interface
- DLL and PLL sharing interface
- OCT interface

The Memory Interface

For more information on the memory interface, refer to "UniPHY Signals" on page 6–10.

The DLL and PLL Sharing Interface

You can generate the UniPHY memory interface as a master or as a slave, depending on the setting of the **Master for PLL/DLL sharing** option on the **General Settings** tab of the parameter editor. The top level of a generated UniPHY variant contains both a PLL and a DLL; the PLL produces a variety of required clock signals derived from the reference clock, and the DLL produces a delay codeword. The top-level file defines the PLL and DLL output signals as outputs for the master, and as inputs for the slave. When you instantiate master and slave variants into your HDL code, you must connect the PLL outputs from the master to the clock inputs of the slaves.

The master **.qip** file must appear before the slave **.qip** file in the Quartus II Settings File (**.qsf**).

The UniPHY memory interface requires one PLL and one DLL to produce the clocks and delay codeword. The PLL and DLL can be shared using a master and slave scenario. The top-level file defines the PLL and DLL output signals as inputs and outputs and an additional parameter `PLL_DLL_MASTER` is also defined. If `PLL_DLL_MASTER` is 1, the RTL instantiates the PLL and DLL, which drives the clock and DLL codeword inputs and outputs. If the parameter is 0, the wires previously connected to the output of the PLL and DLL are assigned to the clock and DLL codeword input and outputs. Inputs and outputs are specified based on the setting of the **PLL/DLL sharing** option.

☞ If you generate a slave IP core, you must modify the timing scripts to allow the timing analysis to correctly resolve clock names and analyze the IP core. Otherwise the software issues critical warnings and an incorrect timing report.

To modify the timing script, follow these steps:

1. In a text editor, open the <*IP core name*>**/constraints directory/**<*IP core name*>**_timing.tcl** file.
2. Search for the following 2 lines:
   - `set master_corename "_MASTER_CORE_"`
   - `set master_instname "_MASTER_INST_"`
3. Replace `_MASTER_CORE_` with the core name and `_MASTER_INST_` with instance name of the UniPHY master to which the slave is connected.

   ☞ The instance name is the full path to the instance and is in the <*IP core name*>**_all_pins.txt** file that is automatically generated after the <*IP core name*>**_pin_assignments.tcl** script runs.

4. If the slave component is connected to a user-defined PLL rather than a UniPHY master, you must manually enter all clock names.
   - Remove the `master_corename` and `master_instname` variables with the checks performed in the eight lines following them.
   - You can use all clock name assignments as templates. For example set `local_pll_afi_clk "mycomponent|mypll|my_afi_clk"`.

   CAUTION You must be extremely careful when connecting clock signals to the slave. Connecting to clocks with frequency or phase different than what the core expects may result in hardware failures.

☞ The DLL and PLL Sharing interface is not available with SOPC Builder.

The OCT Sharing Interface

By default, the UniPHY IP generates the required OCT control block in the top-level RTL file for the PHY. If you want, you can instantiate this block elsewhere in your code and feed the required termination control signals into the IP core by turning off **Master for OCT Control Block** on the **PHY Settings** tab. If you turn off **Master for OCT Control Block**, you must instantiate the OCT control block, or use another UniPHY instance as a master, and ensure that the parallel and series termination control bus signals are connected to the PHY.

Termination Control Block assignments must be created for all calibrated input-only pins, to designate which OCT control block to use for those pins. If the UniPHY IP is in OCT Control Block master mode, these assignments are included in the *<variation_name>***_pin_assignments.tcl** file which must be run after analysis and synthesis. If the UniPHY IP is not using OCT Control Block master mode you must manually create the required assignments to connect the input-only pins to the relevant OCT control block. For QDRII this is the input clocks and input data pins, all output and bidirectional pins are hard coded between the pin's I/O buffer and the series and parallel termination control signals.

Figure 6–6 and Figure 6–7, respectively, show the PHY architecture with and without Master for OCT Control Block.

Figure 6–6. PHY Architecture with Master for OCT Control Block

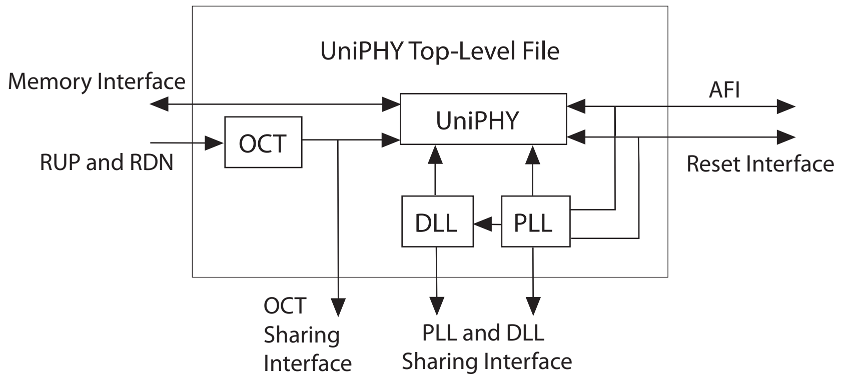
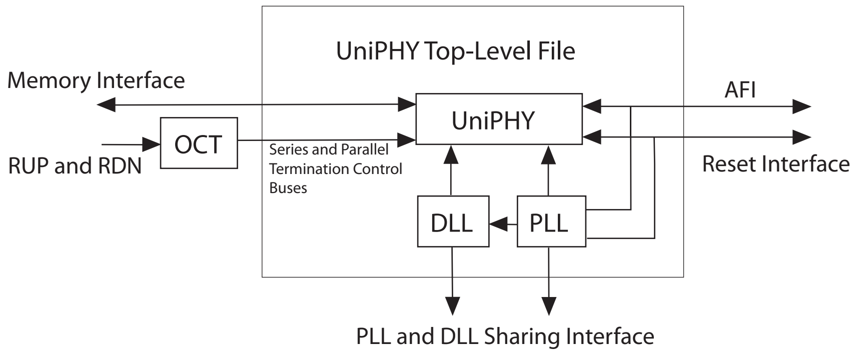


Figure 6–7. PHY Architecture without Master for OCT Control Block



The OCT Sharing Interface and OCT slave mode are not available with SOPC Builder.

UniPHY Signals

This section describes the UniPHY signals. Table 6–4 shows the clock and reset signals.

Table 6–4. Clock and Reset Signals

Name	Direction	Description
`pll_ref_clk`	Input	PLL reference clock input.
`global_reset_n`	Input	Active low global reset for PLL and all logic in the PHY, which causes a complete reset of the whole system.
`soft_reset_n`	Input	Holding `soft_reset_n` low holds the PHY in a reset state. However it does not reset the PLL, which keeps running. It also holds the `afi_reset_n` output low. Mainly for use by SOPC Builder.
`reset_request_n`	Output	When the PLL is locked, `reset_request_n` is high. When the PLL is out of lock, `reset_request_n` is low.
`seriesterminationcontrol`	Input (for OCT slave)	Required signal for PHY to provide series termination calibration value. Must be connected to a user-instantiated OCT control block (alt_oct) or another UniPHY instance that is set to OCT master mode.
	Output(for OCT master)	Unconnected PHY signal, available for sharing with another PHY.
`parallelterminationcontrol`	Input (for OCT slave)	Required signal for PHY to provide series termination calibration value. Must be connected to a user-instantiated OCT control block (alt_oct) or another UniPHY instance that is set to OCT master mode.
	Output(for OCT master)	Unconnected PHY signal, available for sharing with another PHY.
`oct_rdn`	Input (for OCT master)	Must connect to calibration resistor tied to GND on the appropriate RDN pin on the device. (See appropriate device handbook.)
`oct_rup`	Input (for OCT master)	Must connect to calibration resistor tied to V_{ccio} on the appropriate RUP pin on the device. (See appropriate device handbook.)

Table 6–5 shows the AFI signals.

Table 6–5. AFI Signals (Part 1 of 2)

Name	Direction	Width	Description
Clocks and Reset			
`afi_clk`	Output	1	Half-rate or full-rate clock supplied to controller and system logic.
`afi_reset_n`	Output	1	Reset output on `afi_clk` clock domain. For use as asynchronous reset. This signal is asynchronously asserted and synchronously deasserted.

Table 6–5. AFI Signals (Part 2 of 2)

Name	Direction	Width	Description
Address and Command			
afi_addr	Input	MEM_ADDRESS_WIDTH	For full-rate, burst-of-4 configuration.
	Input	2 x MEM_ADDRESS_WIDTH	For full-rate, burst-of-2 configuration. The upper memory address bits contain the write address, while the lower memory address bits contain the read address.
	Input	2 x MEM_ADDRESS_WIDTH	For half-rate, burst-of-4 configuration. The upper memory address bits contain the read address, while the lower memory address bits contain the write address.
afi_wps_n	Output	MEM_CONTROL_WIDTH	Write enable.
afi_rps_n	Output	MEM_CONTROL_WIDTH	Read enable.
Write Data			
afi_dm	Input	MEM_DM_WIDTH × AFI_RATIO	Data mask input that generates mem_dm.
afi_wdata	Input	MEM_DQ_WIDTH × 2 × AFI_RATIO	Write data input that generates mem_dq.
afi_wdata_valid	Input	MEM_WRITE_DQS_WIDTH × AFI_RATIO	Write data valid that generates mem_dq and mem_dm output enables.
Read Data			
afi_rdata	Output	MEM_DQ_WIDTH × 2 × AFI_RATIO	Read data
afi_rdata_en	Input	MEM_READ_DQS_WIDTH × AFI_RATIO	Doing read input. Indicates that the memory controller is currently performing a read operation.
afi_rdata_valid	Output	AFI_RATIO	Read data valid indicating valid read data on afi_rdata, in the byte lanes and alignments that were indicated on afi_rdata_en.
Calibration Control and Status			
afi_cal_success	Output	1	'1' signals that calibration has completed
afi_cal_fail	Output	1	'1' signals that calibration has failed

Table 6–6 shows the QDR II and QDR II+ SRAM interface signals.

Table 6–6. QDR II and QDR II+ SRAM Interface Signals

Name	Direction	Width	Description
mem_address	Output	MEM_ADDRESS_WIDTH	Address.
mem_bws_n	Output	MEM_DM_WIDTH	Data mask.
mem_wps_n	Output	MEM_CONTROL_WIDTH	Write enable.
mem_rps_n	Output	MEM_CONTROL_WIDTH	Read enable.
mem_doff_n	Output	MEM_CONTROL_WIDTH	Connecting this pin to ground turns off the DLL inside the device.

Table 6–6. QDR II and QDR II+ SRAM Interface Signals

Name	Direction	Width	Description
`mem_k, mem_kn`	Output	`MEM_WRITE_DQS_ WIDTH`	Write clock(s) to memory, 1 clock per DQS group.
`mem_cq, mem_cq_n`	Input	`MEM_READ_DQS_ WIDTH`	Read clock(s) from memory, 1 clock per DQS group
`mem_d`	Input	`MEM_DQ_WIDTH`	Input data bus.
`mem_q`	Output	`MEM_DQ_WIDTH`	Output data bus.

Table 6–7 shows the top-level signals generated for HardCopy migration.

Table 6–7. Top-Level HardCopy Migration Signals (Part 1 of 2)

Name	Direction	Description
altsyncram Signals		
`hc_rom_config_clock`	Input	Write clock for the ROM loader. This clock is used as the write clock of the NIOS code memory.
`hc_rom_config_datain`	Input	Data input from external ROM.
`hc_rom_config_rom_data_ready`	Input	Asserts to the code memory loader that the word memory is ready to be loaded.
`hc_rom_config_init`	Input	Triggers the ROM loading process. Should be asserted for one hc_rom_config_clock cycle after PLL is locked
`hc_rom_config_init_busy`	Output	When asserted, indicates ROM loading is in progress. The soft_reset_n signal should be de-asserted if the ROM data is not loaded, and also when the ROM is being loaded. The falling edge of hc_rom_config_init_busy indicates the completion of the ROM loading process, at which time, soft_reset_n can be asserted.
`hc_rom_config_rom_rden`	Output	Read-enable signal that connects to the external ROM.
`hc_rom_config_rom_address`	Output	ROM address that connects to the external ROM.
DLL Reconfiguration Signals		
`hc_dll_config_dll_offset_ctrl_addnsub`	Input	Addition and subtraction control port for the DLL. This port controls if the delay-offset setting on `hc_dll_config_ dll_offset_ctrl_offset` is added or subtracted.
`hc_dll_config_offset_ctrl_offset`	Input	Offset input setting for the DLL. This setting is a Gray-coded offset that is added or subtracted from the current value of the DLL's delay chain.
`hc_dll_config_dll_offset_ctrl_offsetctrlout`	Output	The registered and Gray-coded value of the current delay-offset setting.

Table 6–7. Top-Level HardCopy Migration Signals (Part 2 of 2)

Name	Direction	Description
PLL Reconfiguration Signals		
`hc_pll_config_configupdate`	Input	Control signal to enable PLL reconfiguration.
`hc_pll_config_phasecounterselect`	Input	Specifies the counter select for dynamic phase adjustment.
`hc_pll_config_phasestep`	Input	Specifies the phase step for dynamic phase shifting.
`hc_pll_config_phaseupdown`	Input	Specifies whether the phase shift is up or down.
`hc_pll_config_scanclk`	Input	PLL reconfiguration scan chain clock.
`hc_pll_config_scanclkena`	Input	Clock enable port of the `hc_pll_config_scanclk` clock.
`hc_pll_config_scandata`	Input	Serial input data for the PLL reconfiguration scan chain.
`hc_pll_config_scandataout`	Output	Data output of the serial scan chain.
`hc_pll_config_scandone`	Output	This signal is asserted when the scan chain write operation is in progress. This signal is deasserted when the write operation is complete.

Table 6–8 shows the parameters that Table 6–5 through Table 6–7 mention.

Table 6–8. Parameters (Part 1 of 2)

Parameter Name	Description
`AFI_RATIO`	`AFI_RATIO` is 1 in full-rate designs. `AFI_RATIO` is 2 for half-rate designs.
`MEM_IF_DQS_WIDTH`	The number of DQS pins in the interface.
`MEM_ADDRESS_WIDTH`	The address width of the specified memory device.
`MEM_BANK_WIDTH`	The bank width of the specified memory device.
`MEM_CHIP_SELECT_WIDTH`	The chip-select width of the specified memory device.
`MEM_CONTROL_WIDTH`	The control width of the specified memory device.
`MEM_DM_WIDTH`	The DM width of the specified memory device.
`MEM_DQ_WIDTH`	The DQ width of the specified memory device.
`MEM_READ_DQS_WIDTH`	The READ DQS width of the specified memory device.
`MEM_WRITE_DQS_WIDTH`	The WRITE DQS width of the specified memory device.
`OCT_SERIES_TERM_ CONTROL_WIDTH`	—
`OCT_PARALLEL_TERM_ CONTROL_WIDTH`	—
`AFI_ADDRESS_WIDTH`	The AFI address width, derived from the corresponding memory interface width.
`AFI_BANK_WIDTH`	The AFI bank width, derived from the corresponding memory interface width.
`AFI_CHIP_SELECT_WIDTH`	The AFI chip select width, derived from the corresponding memory interface width.
`AFI_DATA_MASK_WIDTH`	The AFI data mask width.

Table 6–8. Parameters (Part 2 of 2)

Parameter Name	Description
AFI_CONTROL_WIDTH	The AFI control width, derived from the corresponding memory interface width.
AFI_DATA_WIDTH	The AFI data width.
AFI_DQS_WIDTH	The AFI DQS width.
DLL_DELAY_CTRL_WIDTH	The DLL delay output control width.
NUM_SUBGROUP_PER_READ_DQS	A read datapath parameter for timing purposes.
QVLD_EXTRA_FLOP_STAGES	A read datapath parameter for timing purposes.
READ_VALID_TIMEOUT_WIDTH	A read datapath parameter; calibration fails when the timeout counter expires.
READ_VALID_FIFO_WRITE_ADDR_WIDTH	A read datapath parameter; the write address width for half-rate clocks.
READ_VALID_FIFO_READ_ADDR_WIDTH	A read datapath parameter; the read address width for full-rate clocks.
MAX_LATENCY_COUNT_WIDTH	A latency calibration parameter; the maximum latency count width.
MAX_READ_LATENCY	A latency calibration parameter; the maximum read latency.
READ_FIFO_READ_ADDR_WIDTH	—
READ_FIFO_WRITE_ADDR_WIDTH	—
MAX_WRITE_LATENCY_COUNT_WIDTH	A write datapath parameter; the maximum write latency count width.
INIT_COUNT_WIDTH	An initailization sequence.
SEQ_BURST_COUNT_WIDTH	The burst count width for the sequencer.
VCALIB_COUNT_WIDTH	The width of a counter used by the sequencer.
DOUBLE_MEM_DQ_WIDTH	—
HALF_AFI_DATA_WIDTH	—
CALIB_REG_WIDTH	The width of the calibration status register.
NUM_AFI_RESET	The number of AFI resets to generate.

AFI Signal Names

The QDR II and QDR II+ SRAM controllers with UniPHY use AFI.

The AFI timing is identical to the DDR3 SDRAM AFI in the Quartus II software version 9.0. However, some signals have been renamed, some added, and others removed from the AFI definition. The AFI includes only signals that are part of the controller-to-PHY interface, clocks, and reset. All signals on the controller-to-PHY interface have the **afi_** prefix to the signal name. Table 6–9 shows the renamed AFI signals and original (Quartus II software version 9.0) names.

Table 6–9. AFI New Signal Names

AFI Name	Old Name
afi_clk	ctl_clk
afi_reset_n	ctl_reset_n

Table 6–9. AFI New Signal Names

AFI Name	Old Name
afi_addr	ctl_addr
afi_ba	ctl_ba
afi_cke	ctl_cke
afi_cs_n	ctl_cs_n
afi_ras_n	ctl_ras_n
afi_we_n	ctl_we_n
afi_cas_n	ctl_cas_n
afi_dqs_burst	ctl_dqs_burst
afi_wdata_valid	ctl_wdata_valid
afi_wdata	ctl_wdata
afi_dm	ctl_dm
afi_wlat	ctl_wlat
afi_rdata_en	ctl_doing_read
afi_rdata	ctl_rdata
afi_mem_clk_disable	ctl_mem_clk_disable
afi_cal_success	ctl_cal_success
afi_cal_fail	ctl_cal_fail
afi_cal_req	ctl_cal_req

PHY-to-Controller Interfaces

This section describes the typical modules that are connected to the UniPHY PHY and the port name prefixes each module uses. This section describes using a custom controller and describes the AFI.

The AFI standardizes and simplifies the interface between controller and PHY for all Altera memory designs, thus allowing you to easily interchange your own controller code with Altera's high-performance controllers. The AFI PHY includes an administration block that configures the memory for calibration and performs necessary accesses to mode registers that configure the memory as required (these calibration processes are different).

For half-rate designs, the address and command signals in the UniPHY are asserted for one `mem_clk` cycle (1T addressing), such that there are two input bits per address and command pin in half-rate designs. If you require a more conservative 2T addressing, drive both input bits (of the address and command signal) identically in half-rate designs.

Figure 6–8 shows the half-rate write operation.

Figure 6–8. Half-Rate Write with Word-Aligned Data

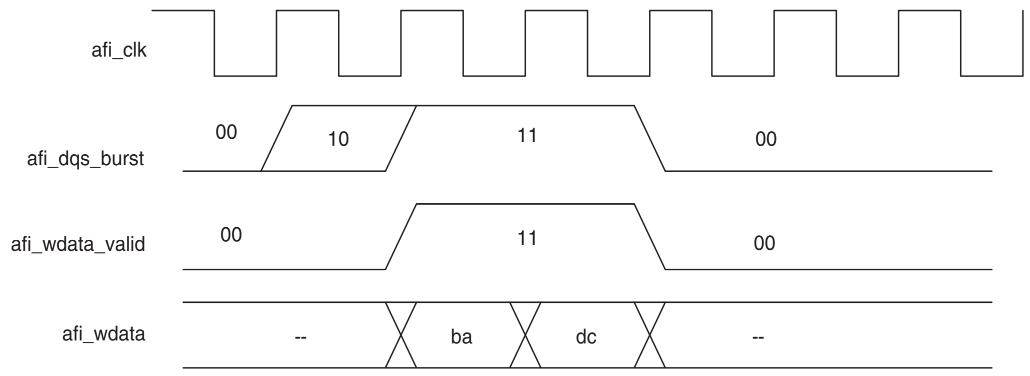


Figure 6–9 shows a full-rate write.

Figure 6–9. Full-Rate Write

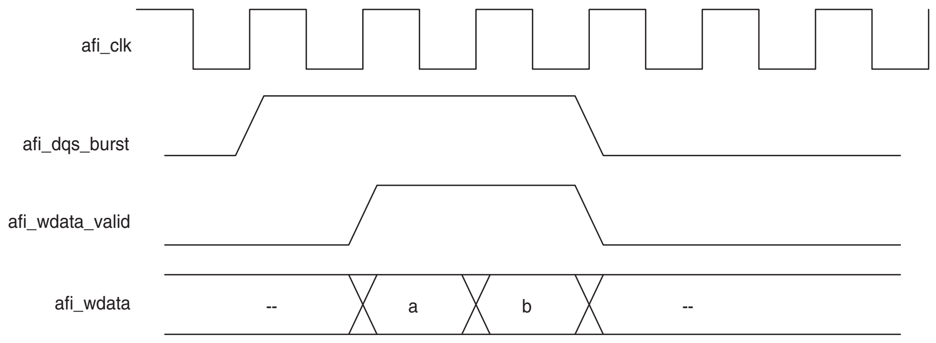


Figure 6–10 shows full-rate reads; Figure 6–11 shows half-rate reads.

Figure 6–10. Full-Rate Reads

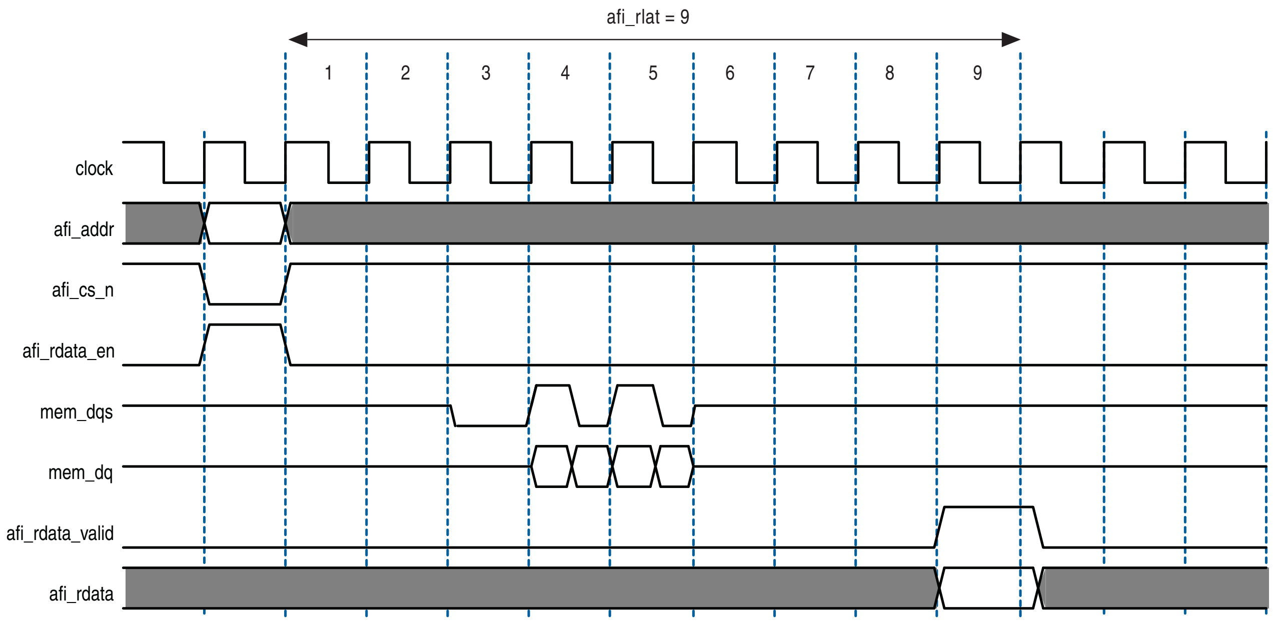


Figure 6–11. Half-Rate Reads

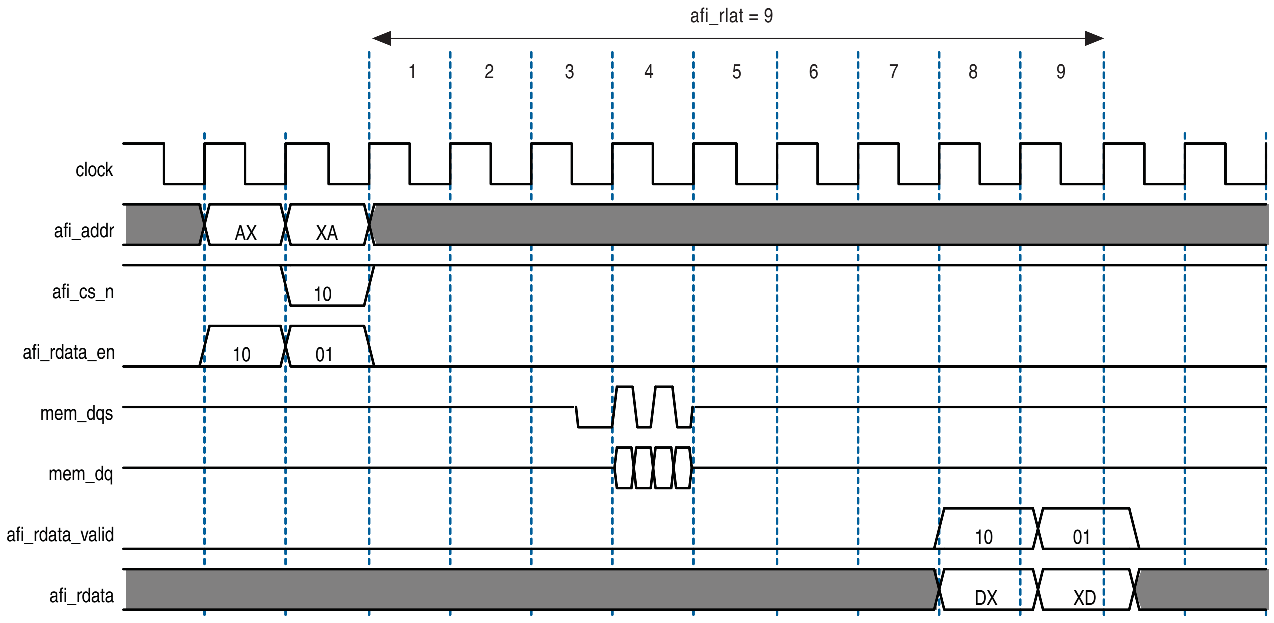


Figure 6–12 and Figure 6–13 show writes and reads, where the data is written to and read from the same address. In each example, `afi_rdata` and `afi_wdata` are aligned with controller clock (`afi_clk`) cycles. All the data in the bit vector is valid at once.

The AFI has the following conventions:

- With the AFI, high and low signals are combined in one signal, so for a single chip-select (`afi_cs_n`) interface, `afi_cs_n[1:0]`, where location 0 appears on the memory bus on one `mem_clk` cycle and location 1 on the next `mem_clk` cycle.

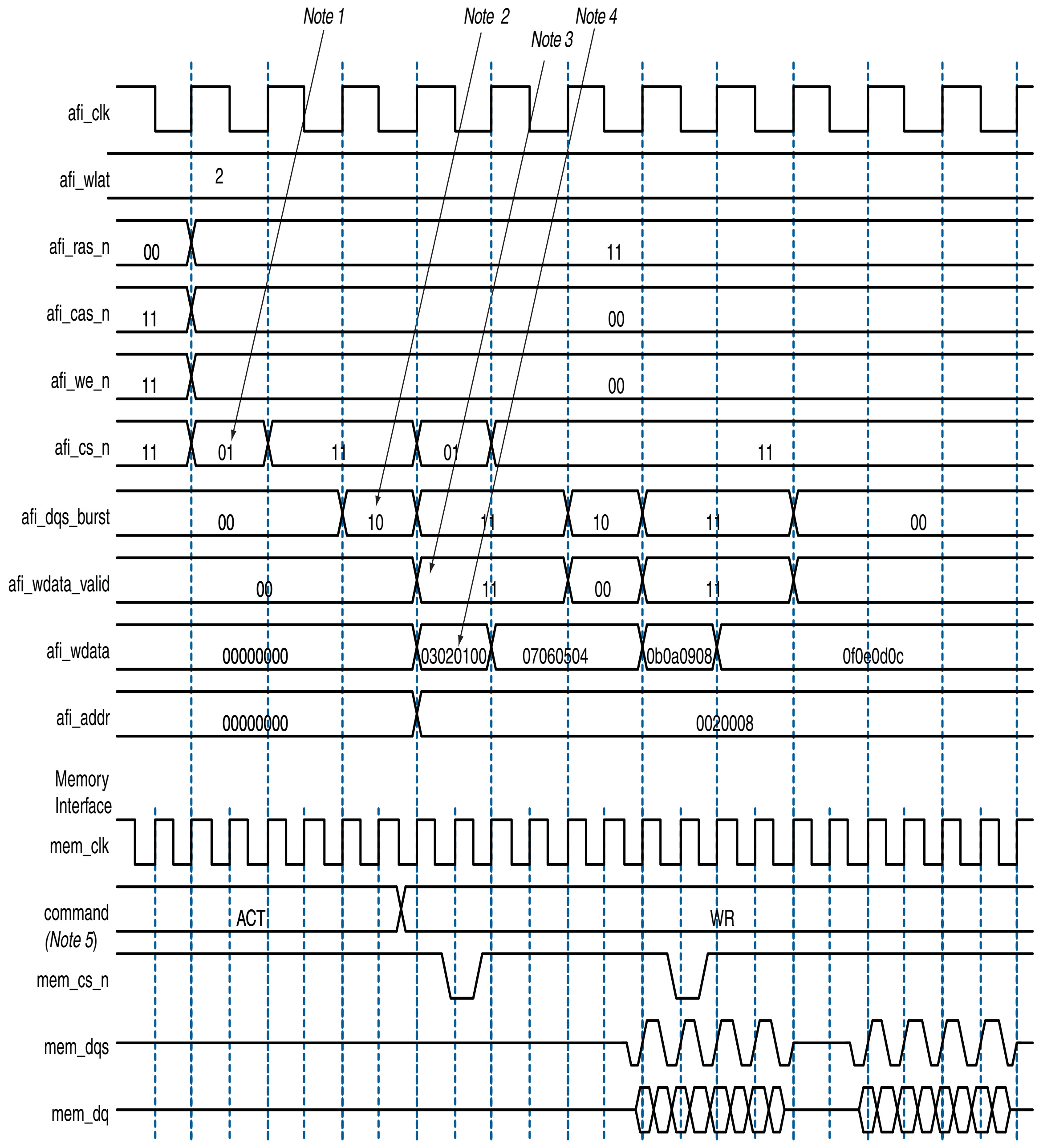
  ☞ This convention is maintained for all signals, so for an 8 bit memory interface, the write data (`afi_wdata`) signal is `afi_wdata[31:0]`, where the first data on the DQ pins is `afi_wdata[7:0]`, then `afi_wdata[15:8]`, then `afi_wdata[23:16]`, then `afi_wdata[31:24]`.

- Spaced reads and writes have the following definitions:
  - Spaced writes—write commands separated by a gap of one controller clock (`afi_clk`) cycle.
  - Spaced reads—read commands separated by a gap of one controller clock (`afi_clk`) cycle.

Figure 6–12 through Figure 6–13 assume the following general points:

- The burst length is four.
- An 8-bit interface with one chip-select.
- The data for one controller clock (afi_clk) cycle represents data for two memory clock (mem_clk) cycles (half-rate interface).

Figure 6–12. Word-Aligned Writes



Notes to Figure 6–12:

(1) To show the even alignment of `afi_cs_n`, expand the signal (this convention applies for all other signals).

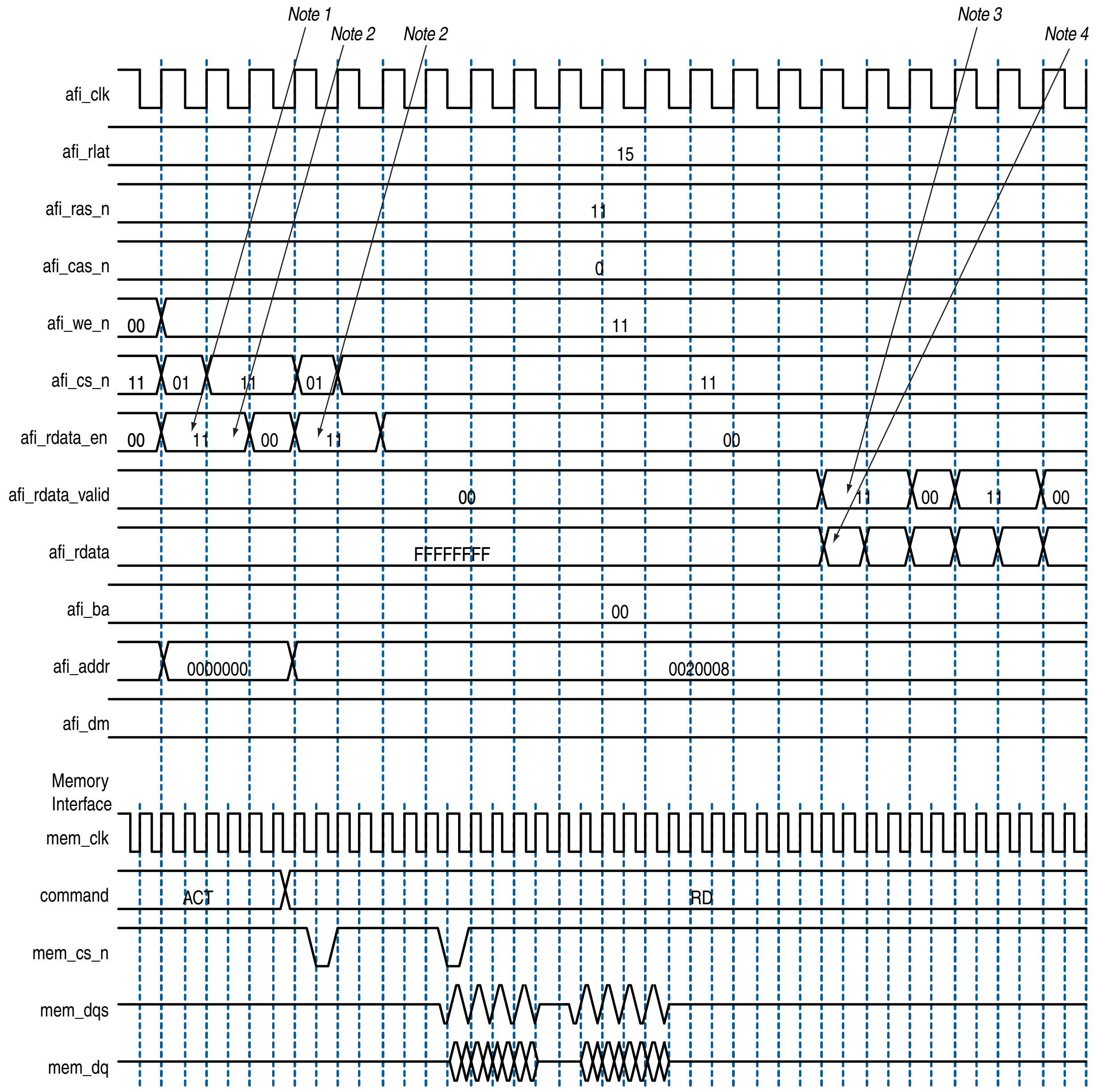
(2) The `afi_dqs_burst` must go high one memory clock cycle before `afi_wdata_valid`. Compare with the word-unaligned case.

(3) The `afi_wdata_valid` is asserted two `afi_wlat` controller clock (`afi_clk`) cycles after chip select (`afi_cs_n`) is asserted. The `afi_wlat` indicates the required write latency in the system. The value is determined during calibration and is dependant upon the relative delays in the address and command path and the write datapath in both the PHY and the external DDR SDRAM subsystem. The controller must drive `afi_cs_n` and then wait `afi_wlat` (two in this example) `afi_clks` before driving `afi_wdata_valid`.

(4) Observe the ordering of write data (`afi_wdata`). Compare this to data on the `mem_dq` signal.

(5) In all waveforms a command record is added that combines the memory pins `ras_n`, `cas_n` and `we_n` into the current command that is issued. This command is registered by the memory when chip select (`mem_cs_n`) is low. The important commands in the presented waveforms are WR = write, ACT = activate.

Figure 6–13. Word-Aligned Reads



Notes to Figure 6–13:

(1) For AFI, `afi_rdata_en` is required to be asserted one memory clock cycle before chip select (`afi_cs_n`) is asserted. In the half-rate `afi_clk` domain, this requirement manifests as the controller driving `11` (as opposed to the `01`) on `afi_rdata_en`.

(2) AFI requires that `afi_rdata_en` is driven for the duration of the read. In this example, it is driven to `11` for two half-rate `afi_clks`, which equates to driving to `1`, for the four memory clock cycles of this four-beat burst.

(3) The `afi_rdata_valid` returns 15 (`afi_rlat`) controller clock (`afi_clk`) cycles after `afi_rdata_en` is asserted. Returned is when the `afi_rdata_valid` signal is observed at the output of a register within the controller. A controller can use the `afi_rlat` value to determine when to register to returned data, but this is unnecessary as the `afi_rdata_valid` is provided for the controller to use as an enable when registering read data.

(4) Observe the alignment of returned read data relative to data on the bus.

Using a Custom Controller

The UniPHY-based memory interface IP cores integrate the PHY and the memory controller. To replace the Altera high-performance memory controller with a custom memory controller, perform the following steps:

1. Parameterize and generate your variation of the UniPHY-based memory controller IP as described in "Getting Started" on page 2–1.

   This step creates a top-level HDL file called <*variation_name*>**.v** (or <*variation_name*>**.vhd**), and a sub-directory called <*variation_name*>.

   The top-level module instantiates the <*variation_name*>_<*stamp*>**_controller_phy** module in the <*variation_name*> subdirectory. The <*variation_name*>_<*stamp*>**_controller_phy** module instantiates the PHY and the controller.

   ☞ <*stamp*> is a unique identifier determined by the MegaWizard Plug-in Manager, SOPC Builder, or Qsys, during generation.

2. Open the <*variation_name*>**/**<*variation_name*>_<*stamp*>**_controller_phy.sv** file.
3. Replace the <*variation_name*>_<*stamp*>**_alt_qdr_controller** module with your custom controller module.
4. Delete the ports of the Altera high-performance memory controller, and add the top-level ports of your custom controller.
5. Similarily, update the port names in the top-level module in the <*variation_name*>**.v** or <*variation_name*>**.vhd** file.
6. Compile and simulate the design to confirm correct operation.

☞ Regenerating the UniPHY memory interface IP erases all modifications made to the HDL files. The parameters you select in the parameter editor are stored in the top-level <*variation_name*> module; hence, you must repeat the above steps every time you regenerate the IP variation.

☞ For half-rate controllers, AFI signals are double the bus width of the memory interface. Half rate controllers have double the width of the signal and run at half the speed. Hence, the overall bandwidth is maintained. Such double-width signals are divided into two signals for transmission to the memory interface, with a higher order bits representing the most-significant bit (MSB) and a lower order bits representing the least-significant bit (LSB). The LSB is transmitted first, and is followed by the MSB.

Using a Vendor-Specific Memory Model

You can replace the Altera-supplied memory model with a vendor-specific memory model. In general, you may find vendor-specific models to be standardized, thorough, and well supported, but sometimes more complex to setup and use.

If you do want to replace the Altera-supplied memory model with a vendor-supplied memory model, observe the following guidelines:

- Ensure that the vendor-supplied memory model that you have is correct for your memory device.
- Disconnect all signals from the default memory model and reconnect them to the vendor-supplied memory model.
- If you intend to run simulation from the Quartus II software, ensure that the **.qip** file points to the vendor-supplied memory model.

7. Functional Description—Example Top-Level Project

IP generation creates an example top-level project that shows you how to instantiate and connect the controller.

The example top-level project contains a testbench, which is for use with Verilog HDL only language simulators such as ModelSim-AE Verilog, and shows simple operation of the memory interface.

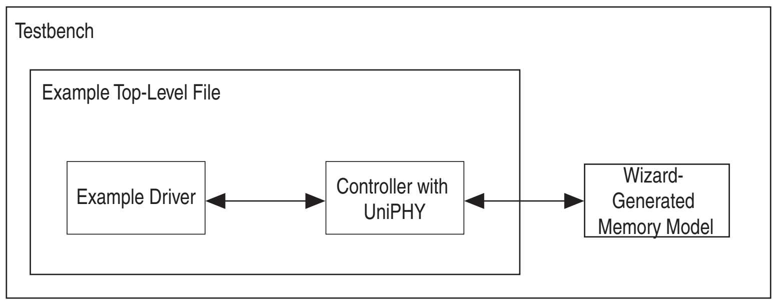
☞ For a VHDL-only simulation, use the VHDL IP functional simulation model.

The testbench contains the following blocks:

- A synthesizable Avalon-MM example driver, which acts as a traffic generator block and implements a pseudo-random pattern of reads and writes to a parameterized number of addresses. The driver also monitors the data read from the memory to ensure it matches the written data and asserts a failure otherwise.
- An instance of the controller, which interfaces between the Avalon-MM interface and the AFI.
- The UniPHY IP, which serves as an interface between the memory controller and external memory device(s) to perform read and write operations to the memory.
- A memory model, which acts as a generic model that adheres to the memory protocol specifications. Memory vendors also provide simulation models for specific memory components that can be downloaded from their websites. This block is available in Verilog HDL only.

Figure 7–1 shows the testbench and the example top-level file.

Figure 7–1. Testbench and Example Top-Level File



Example Driver

The example driver for Avalon-MM memory interfaces generates Avalon-MM traffic on an Avalon-MM master interface. As the read and write traffic is generated, the expected read response is stored internally and compared to the read responses as they arrive. If all reads report their expected response, the pass signal is asserted; however, if any read responds with unexpected data a fail signal is asserted.

Each operation generated by the driver is a single write or block of writes followed by a single read or block of reads to the same addresses, which allows the driver to precisely determine the data that should be expected when the read data is returned by the memory interface. The driver comprises a traffic generation block, the Avalon-MM interface and a read comparison block. The traffic generation block generates addresses and write data, which are then sent out over the Avalon-MM interface. The read comparison block compares the read data received from the Avalon-MM interface to the write data from the traffic generator. If at any time the data received is not the expected data, the read comparison block records the failure, finishes reading all the data, and then signals that there is a failure and the driver enters a fail state. If all patterns have been generated and compared successfully, the driver enters a pass state.

Within the driver, there are the following main states:

- Generation of individual read and writes
- Generation of block read and writes
- The pass state
- The fail state

Within each of the generation states there are the following substates:

- Sequential address generation
- Random address generation
- Mixed sequential and random address generation

For each of the states and substates, the order and number of operations generated for each substate is parameterizable—you can decide how many of each address pattern to generate, or can disable certain patterns entirely if you want. The sequential and random interleave substate takes in additions to the number of operations to generate. An additional parameter specifies the ratio of sequential to random addresses to generate randomly.

Read and Write Generation

The traffic generator block can perform individual or block read and write generation.

Individual Read and Write Generation

During the individual read and write generation stage of the driver, the traffic generation block generates individual write followed by individual read Avalon-MM transactions, where the address for the transactions are chosen according to the specific substate. The width of the Avalon-MM interface is a global parameter for the driver, but each substate can have a parameterizable range of burst lengths for each operation.

Block Read and Write Generation

During the block read and write generation state of the driver, the traffic generator block generates a parameterizable number of write operations followed by the same number of read operations. The specific addresses generated for the blocks are chosen by the specific substates. The burst length of each block operation can be parameterized by a range of acceptable burst lengths.

Address and Burst Length Generation

The traffic generator block can perform sequential or random addressing.

Sequential Addressing

The sequential addressing substate defines a traffic pattern where addresses are chosen in sequential order starting from a user definable address. The number of operations in this substate is parameterizable.

Random Addressing

The random addressing substate defines a traffic pattern where addresses are chosen randomly over a parameterizable range. The number of operations in this substate is parameterizable.

Sequential and Random Interleaved Addressing

The sequential and random interleaved addressing substate defines a traffic pattern where addresses are chosen to be either sequential or random based on a parameterizable ratio. The acceptable address range is parameterizable as is the number of operations to perform in this substate.

Example Driver Signals

Table 7–1 lists the signals used by the example driver.

Table 7–1. Driver Signals (Part 1 of 2)

Signal	Width	Signal Type
`clk`		
`reset_n`		
`avl_ready`		avl_ready
`avl_write_req`		avl_write_req
`avl_read_req`		avl_read_req
`avl_addr`	24	avl_addr
`avl_size`	3	avl_size
`avl_wdata`	72	avl_wdata
`avl_rdata`	72	avl_rdata
`avl_rdata_valid`		avl_rdata_valid
`pnf_per_bit`		pnf_per_bit
`pnf_per_bit_ persist`		pnf_per_bit_persist

Table 7–1. Driver Signals (Part 2 of 2)

Signal	Width	Signal Type
`pass`		pass
`fail`		fail
`test_complete`		test_complete

Example Driver Add-Ons

Some optional components that can be useful for verifying aspects of the controller and PHY operation are generated in conjunction with certain user-specified options. These add-on components are self-contained, and are not part of the controller or PHY, nor the example driver.

User Refresh Generator

The user refresh generator sends refresh requests to the memory controller when user refresh is enabled. The memory controller returns an acknowledgement signal and then issues the refresh command to the memory device.

The user refresh generator is created when you turn on **Enable User Refresh** under **Controller Settings** on the **General Settings** tab of the parameter editor. The user refresh generator is instantiated by **example_top_v**. and resides in the example_project subdirectory.

Refresh Monitor

As its name implies, the refresh monitor monitors refresh commands from the controller and verifies that those commands conform to the necessary refresh timing parameters.

The refresh monitor is created when you turn on **Enable User Refresh** under **Controller Settings** on the **General Settings** tab of the parameter editor. The refresh monitor is instantiated by **example_top_tb.v** and resides in **refresh_monitor.sv** in the **rtl_sim** subdirectory.

Data Corrupter

The data corrupter intercepts read data in the memory interface bus and introduces errors to that data to test the error detection function in the memory controller. Both the rate of error injection and the number of error bits are configurable (although the per-byte parity protection feature supports only 1 bit error detection).

The data corrupter employs four types of error injection:

- per-bit data corruption in a single memory burst
- per-byte corruption in a single memory burst
- per-bit all-burst corruption
- per-byte all burst corruption

Throughout the four types of error injection tests, the data corrupter exercises a walking-one pattern to confirm correctness.

The data corrupter is created when you turn on **Enable Error Detection Parity** under **Controller Settings** on the **General Settings** tab of the parameter editor. The data corrupter resides in **data_corrupter.sv** in the **rtl_sim** subdirectory.

Altera defines read and write latencies in terms of memory clock cycles. These latencies apply to supported device families (Table 1–2 on page 1–2). There are two types of latencies that exists while designing with memory controllers—read and write latencies, which have the following definitions:

- Read latency—the amount of time it takes for the read data to appear at the local interface after initiating the read request.
- Write latency—the amount of time it takes for the write data to appear at the memory interface after initiating the write request.

For a half-rate controller, the local side frequency is half of the memory interface frequency. For a full-rate controller, the local side frequency is equal to the memory interface frequency.

Table 8–1 shows the latency in full rate memory clock cycles.

Table 8–1. Latency (In Full-Rate Memory Clock Cycles) *(Note 2)*

Rate	Controller Address and Command *(3)*	PHY Address and Command	Memory Maximum Read	PHY Read Return	Round Trip	Round Trip without Memory
Full	1	1	1.5, 2.0, 2.5	RL 1.5: 4.5 RL 2.0: 4.0 RL 2.5: 4.5 *(1)*	RL 1.5: 8 RL 2.0: 8 RL 2.5: 9 *(1)*	RL 1.5: 6.5 RL 2.0: 6.0 RL 2.5: 6.5 *(1)*
Half	2	2	1.5, 2.0, 2.5	RL 1.5: 6.5 RL 2.0: 6.0 RL 2.5: 7.5 *(1)*	RL 1.5: 12 RL 2.0: 12 RL 2.5: 14 *(1)*	RL 1.5: 10.5 RL 2.0: 10.0 RL 2.5: 11.5 *(1)*

Notes to Table 8–1:

(1) RL = read latency.

(2) Latency is the number of cycles between the first register of the current stage capturing cmd/data, and the first register in the next stage capturing cmd/data.

(3) Latency shown is best case, for maximum performance specifications. Latency may be higher due to protocol requirements; controller latency may be lower for slower frequencies.

Variable Controller Latency

The variable controller latency feature allows you to take advantage of lower latency for variations designed to run at lower frequency. When deciding whether to vary the controller latency from the default value of 1, be aware of the following considerations:

- Reduced latency can help acheive a reduction in resource usage and clock cycles in the controller, but might result in lower f_{MAX}.
- Increased latency can help acheive greater f_{MAX}, but might consume more clock cycles in the controller and result in increased resource usage.

If you select a latency value that is inappropriate for the target frequency, the system displays a warning message in the text area at the bottom of the parameter editor.

You can change the controller latency by altering the value of the **Controller Latency** setting in the **Controller Settings** section of the **General Settings** tab of the QDR II and QDR II+ SRAM controller with UniPHY parameter editor.

9. Timing Diagrams

The timing diagrams in this chapter are for QDR II SRAM with the following parameters:

- ×18
- Half rate
- Burst length 4
- Latency 2.5

Figure 9–1 shows back-to-back write to addresses 0 and 1.

You can set the `avl_w_size` to `0x2` and hold `avl_w_addr` constant at `0x0` to perform the same back-to-back write.

Figure 9–1. Back-to-Back Writes

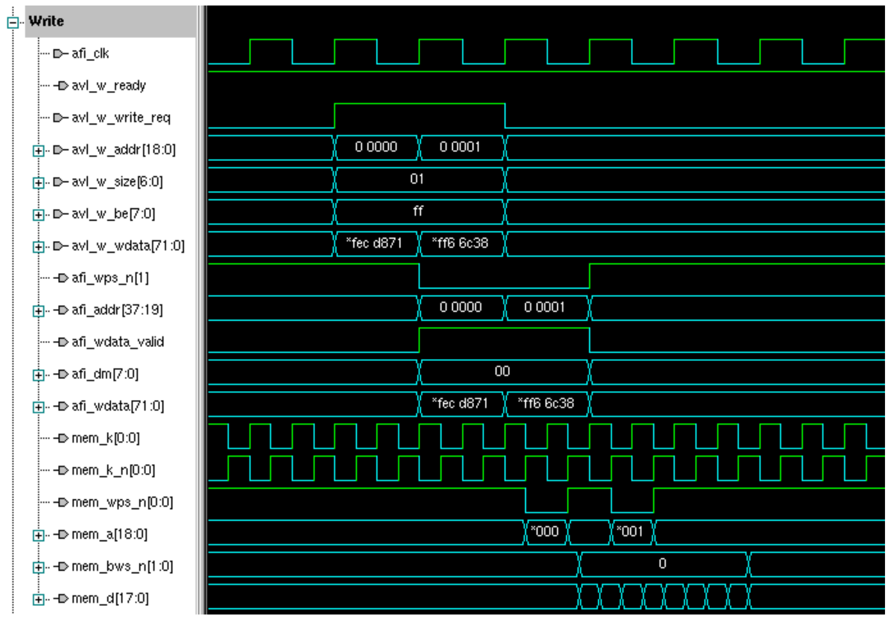
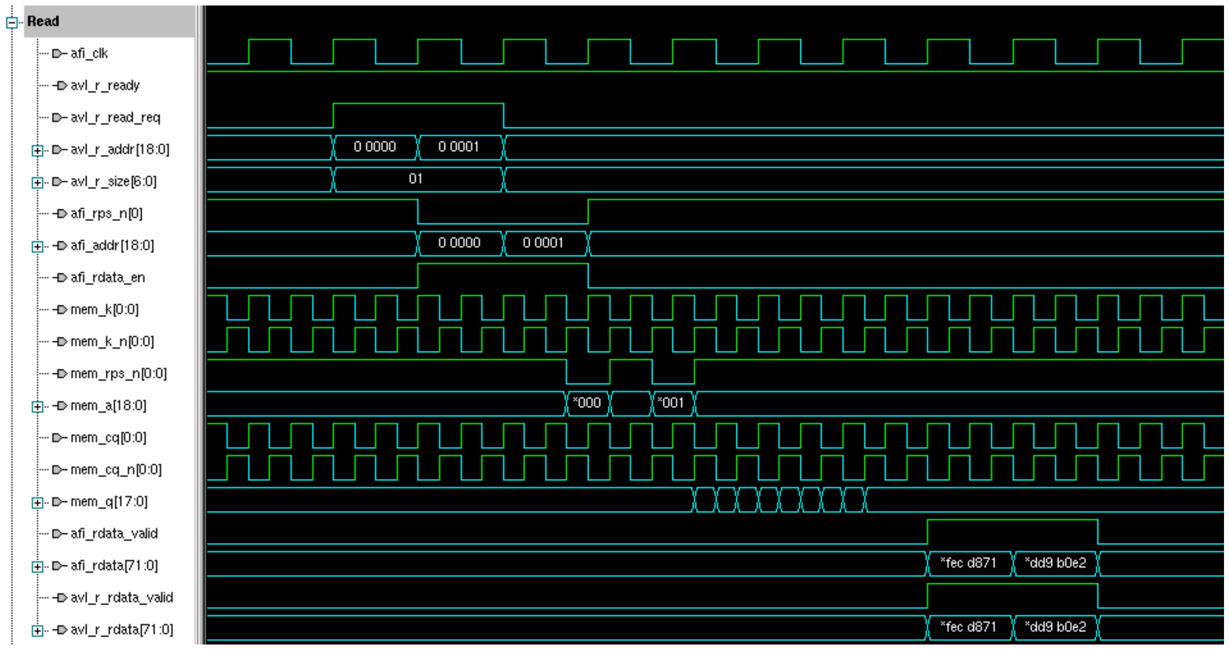


Figure 9–2 shows back-to-back read from addresses 0 and 1.

You can set the `avl_w_size` to `0x2` and hold `avl_w_addr` constant at `0x0` to perform the same back-to-back read.

Figure 9–2. Back-to-Back Reads



Additional Information

This chapter provides additional information about the document and Altera.

Document Revision History

The following table shows the revision history for this document.

Date	Version	Changes
December 2010	2.1	■ Updated Device Family Support, Features list, and Resource Utilization tables ■ Updated Design Flows, added new Generated Files information ■ Added information to HardCopy Migration Design Guidelines ■ Added new Parameter Settings chapter ■ Updated Reset and Clock Generation, Write Datapath, and Read Datapath information ■ Added The DLL and PLL Sharing Interface and Using a Custom Controller information ■ Updated Latency data
July 2010	2.0	Updated for the Altera Complete Design Suite version 10.0 release.
February 2010	1.2	Corrected typos.
January 2010	1.1	Updated features.
November 2009	1.0	First published.

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to 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 to Table:

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

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, <*file name*> and <*project name*>**.pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the 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, `data1`, `tdi`, and `input`. The suffix `n` denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
↵	An angled arrow instructs you to press the Enter key.
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 question mark directs you to a software help system with related information.
👣	The feet direct you to another document or website with related information.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
⚠ WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
✉	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.