QDRII SRAM Controller

MegaCore Function User Guide

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

MegaCore Version:	8.0
Document Date:	May 2008

I.S. EN ISO 9001

UG-IPQDRII-6.0

Contents

1. About This MegaCore Function

Release Information

Table 1–1 provides information about this release of the Altera® QDRII SRAM Controller MegaCore® function.

Table 1–1. Release Information

Item	Description
Version	8.0
Release Date	May 2008
Ordering Code	IP-SRAM/QDRII
Product ID	00A4
Vendor ID	6AF7

For more information about this release, refer to the *MegaCore IP Library Release Notes and Errata*.

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

Device Family Support

MegaCore functions provide either full or preliminary support for target Altera device families:

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

Table 1–2 shows the level of support offered by the QDRII SRAM Controller MegaCore function to each Altera device family.

Table 1–2. Device Family Support

Device Family	Support
HardCopy® II	Preliminary
Stratix IV	Preliminary
Stratix®	Full
Stratix II	Full
Stratix II GX	Full
Stratix GX	Full
Other device families *(1)*	No support

Note to Table 1–2:

(1) For more information on support for Stratix III devices, contact Altera.

Features

- Support for burst of two and four memory type
- Support for 8-, 18-, and 36-bit QDRII interfaces
- Support for two-times and four-times data width on the local side (four-times for burst of four only)
- Operates at 300 MHz for QDRII and QDRII+ SRAM
- Automatic concatenation of consecutive reads and writes (narrow local bus width mode only)
- Easy-to-use IP Toolbench interface
- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators
- Support for OpenCore Plus evaluation

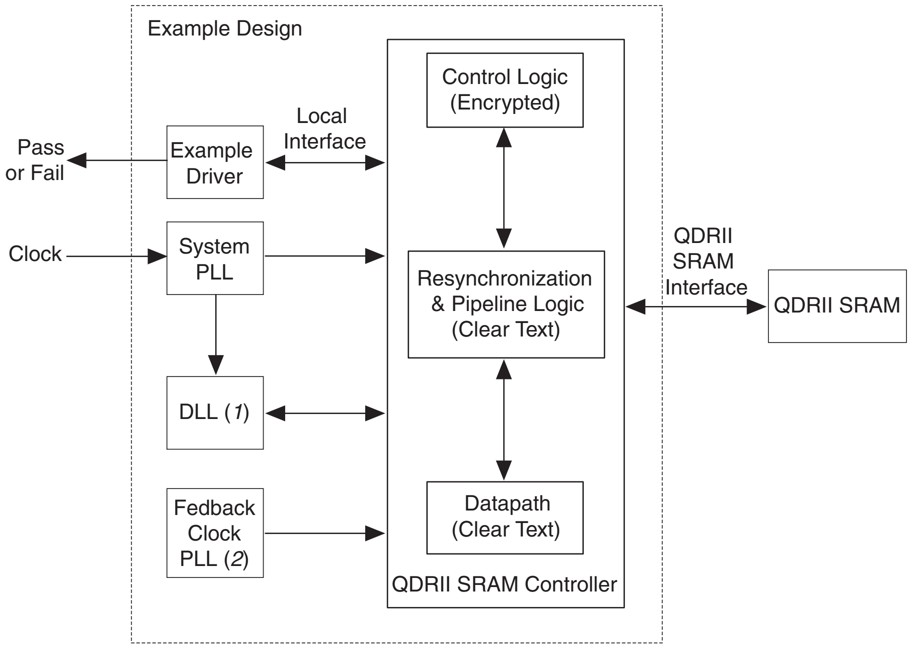
General Description

The QDRII SRAM Controller MegaCore function provides an easy-to-use interface to QDRII SRAM modules. The QDRII SRAM Controller ensures that the placement and timing are in line with QDRII specifications.

The QDRII SRAM Controller is optimized for Altera Stratix series. The advanced features available in these devices allow you to interface directly to QDRII SRAM devices.

Figure 1–1 shows a system-level diagram including the example design that the QDRII SRAM Controller MegaCore function creates for you.

Figure 1–1. QDRII SRAM Controller System-Level Diagram



Notes to Figure 1–1:
(1) Optional, for Stratix II devices only.
(2) Non-DQS mode only.

The IP Toolbench-generated example design instantiates a phase-locked loop (PLL), an optional DLL (for Stratix II devices only), an example driver, and your QDRII SRAM Controller custom variation. The example design is a fully-functional example design that can be simulated, synthesized, and used in hardware. The example driver is a self-test module that issues read and write commands to the controller and checks the read data to produce the pass/fail and test complete signals.

You can replace the QDRII SRAM controller encrypted control logic in the example design with your own custom logic, which allows you to use the Altera clear-text resynchronization and pipeline logic and datapath with your own control logic.

OpenCore Plus Evaluation

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

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPP^{SM} megafunction) within your system
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily
- Generate time-limited device programming files for designs that include megafunctions
- Program a device and verify your design in hardware

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

For more information on OpenCore Plus hardware evaluation using the QDRII SRAM Controller, see "OpenCore Plus Time-Out Behavior" on page 3–10 and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

Performance and Resource Utilization

Table 1–3 shows typical expected performance for the QDRII SRAM Controller MegaCore function, with the Quartus® II software version 8.0.

The example driver, which only demonstrates basic read and write operation, can limit the performance, particularly in wide interfaces. To improve performance, replace the example driver or remove it and use the virtual pins on the controller.

Table 1–3. Performance

Device	f_{MAX} (MHz)
Stratix II (EP2S60F1020C3)	300
Stratix II GX (EP2SGX30CF780C3)	300
Stratix (EP1S25F780C5)	200

Stratix II and Stratix II GX devices support QDRII SRAM at up to 300 MHz/1,200 Megabits per second (Mbps). Stratix and Stratix GX devices support QDRII SRAM at up to 200 MHz/800 Mbps. Tables 1–4 through 1–6 show the clock frequency support for each device family, with the Quartus II software version 8.0.

☞ These numbers apply to both commercial and industrial devices.

Table 1–4. QDRII SDRAM Maximum Clock Frequency Support in Stratix II & Stratix GX Devices (1)

Speed Grade	Frequency (MHz)	
	DLL-Based Implementation	PLL-Based Implementation
–3	300	200
–4	200	167
–5	200	167

Notes to Table 1–4:
(1) This analysis is based on the EP2S90F1020 device. Ensure you perform a timing analysis for your chosen FPGA.

Table 1–5. QDRII SRAM Maximum Clock Frequency Supported in Stratix & Stratix GX Devices (EP1S10 to EP1S40 & EP1SGX10 to EP1SGX40 Devices) (1)

Speed Grade	Frequency (MHz)
–5	200
–6	167
–7	133

Notes to Table 1–5:
(1) This analysis is based on the EP1S25F1020 device. Ensure you perform a timing analysis for your chosen FPGA.

Table 1–6. QDRII SRAM Maximum Clock Frequency Supported in Stratix Devices (EP1S60 to EP1S80 Devices) (1)

Speed Grade	Frequency (MHz)
–5	167
–6	167
–7	133

Notes to Table 1–6:
(1) This analysis is based on the EP1S60F1020 device. Ensure you perform a timing analysis for your chosen FPGA.

Table 1–7 shows typical sizes in combinational adaptive look-up tables (ALUTs) and logic registers for a QDRII SRAM controller with a burst length of 4 in narrow mode.

Table 1–7. Typical Size (1)

Device	Memory Width (Bits)	Combinational ALUTs	Logic Registers	Memory Blocks	
				M4K	M512
Stratix II	9	360	598	–	1
	18	369	633	1	–
	36	390	708	2	–
	72 (2 × 36)	459	880	4	–

Notes to Table 1–7:
(1) These sizes are a guide only and vary with different choices of parameters.

2. Getting Started

Design Flow

To evaluate the QDRII SRAM Controller using the OpenCore Plus feature, include these steps in your design flow:

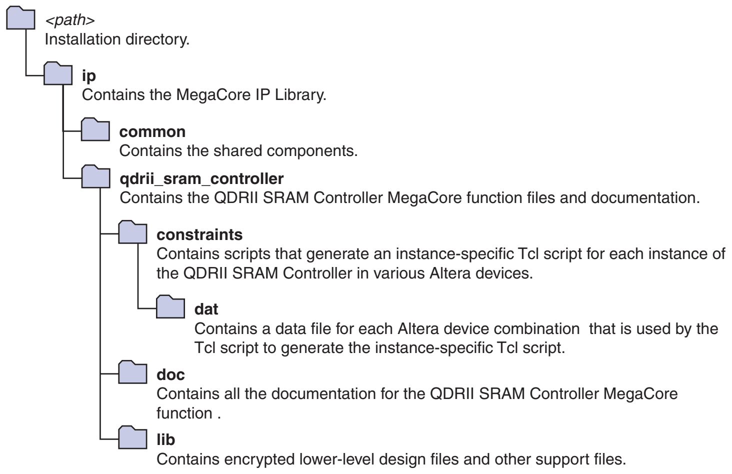
1. Obtain and install the QDRII SRAM Controller.

The QDRII SRAM Controller is part of the MegaCore® IP Library, which is distributed with the Quartus® II software and downloadable from the Altera® website, **www.altera.com**.

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

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

Figure 2–1. Directory Structure



<path>
Installation directory.
ip
Contains the MegaCore IP Library.
common
Contains the shared components.
qdrii_sram_controller
Contains the QDRII SRAM Controller MegaCore function files and documentation.
constraints
Contains scripts that generate an instance-specific Tcl script for each instance of the QDRII SRAM Controller in various Altera devices.
dat
Contains a data file for each Altera device combination that is used by the Tcl script to generate the instance-specific Tcl script.
doc
Contains all the documentation for the QDRII SRAM Controller MegaCore function .
lib
Contains encrypted lower-level design files and other support files.

2. Create a custom variation of the QDRII SRAM Controller MegaCore function using IP Toolbench.

IP Toolbench is a toolbar from which you quickly and easily view documentation, specify parameters, and generate all of the files necessary for integrating the parameterized MegaCore function into your design.

3. Implement the rest of your design using the design entry method of your choice.

4. Use the IP Toolbench-generated IP functional simulation model to verify the operation of your design.

For more information on IP functional simulation models, see the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

5. Edit the PLL(s).

6. Use the Quartus II software to add constraints to the example design and compile the example design.

7. Perform gate-level timing simulation, or if you have a suitable development board, you can generate an OpenCore Plus time-limited programming file, which you can use to verify the operation of the example design in hardware.

8. Either obtain a license for the QDRII SRAM controller MegaCore function or replace the encrypted QDRII SRAM controller control logic with your own logic and use the clear-text data path.

   If you obtain a license for the QDRII SRAM controller, you must set up licensing.

9. Generate a programming file for the Altera device(s) on your board.

10. Program the Altera device(s) with the completed design.

QDRII SRAM Controller Walkthrough

This walkthrough explains how to create a QDRII SRAM controller using the Altera QDRII SRAM controller IP Toolbench and the Quartus II software. When you are finished generating a custom variation of the QDRII SRAM Controller MegaCore function, you can incorporate it into your overall project.

IP Toolbench only allows you to select legal combinations of parameters, and warns you of any invalid configurations.

This walkthrough requires the following steps:

Create a New Quartus II Project

Before you begin, you must create a new Quartus II project. With the New Project wizard, you specify the working directory for the project, assign the project name, and designate the name of the top-level design entity. You will also specify the QDRII SRAM Controller user library. To create a new project, follow these steps:

You need to create a new Quartus II project with the **New Project Wizard**, which specifies the working directory for the project, assigns the project name, and designates the name of the top-level design entity. To create a new project follow these steps:

1. Choose **Programs > Altera > Quartus II** <*version*> (Windows Start menu) to run the Quartus II software. Alternatively, you can use the Quartus II Web Edition software.

2. Choose **New Project Wizard** (File menu).

3. Click **Next** in the **New Project Wizard Introduction** page (the introduction page does not display if you turned it off previously).

4. In the **New Project Wizard: Directory, Name, Top-Level Entity** page, enter the following information:

   a. Specify the working directory for your project. For example, this walkthrough uses the **c:\altera\temp\qdr_project** directory.

   b. Specify the name of the project. This walkthrough uses **project** for the project name.

   ☞ The Quartus II software automatically specifies a top-level design entity that has the same name as the project. Do not change it.

5. Click **Next** to close this page and display the **New Project Wizard: Add Files** page.

When you specify a directory that does not already exist, a message asks if the specified directory should be created. Click **Yes** to create the directory.

6. If you installed the MegaCore IP Library in a different directory from where you installed the Quartus II software, you must add the user libraries:

   a. Click **User Libraries**.

   b. Type <*path*>`\ip` into the **Library name** box, where <*path*> is the directory in which you installed the QDRII SRAM Controller.

   c. Click **Add to** add the path to the Quartus II project.

   d. Click **OK** to save the library path in the project.

7. Click **Next** to close this page and display the **New Project Wizard: Family & Device Settings** page.

8. On the **New Project Wizard: Family & Device Settings** page, choose the target device family in the **Family** list.

9. The remaining pages in the **New Project Wizard** are optional. Click **Finish** to complete the Quartus II project.

You have finished creating your new Quartus II project.

Launch IP Toolbench

To launch IP Toolbench in the Quartus II software, follow these steps:

1. Start the MegaWizard® Plug-In Manager by choosing **MegaWizard Plug-In Manager** (Tools menu). The **MegaWizard Plug-In Manager** dialog box displays.

   Refer to Quartus II Help for more information on how to use the MegaWizard Plug-In Manager.

2. Specify that you want to create a new custom megafunction variation and click **Next**.

3. Expand the **Interfaces > Memory Controllers** directory then click **QDRII SRAM Controller-v8.0**.

4. Select the output file type for your design; the wizard supports VHDL and Verilog HDL.

5. The MegaWizard Plug-In Manager shows the project path that you specified in the **New Project Wizard**. Append a variation name for the MegaCore function output files <*project path*>\<*variation name*>.

   The <*variation name*> must be a different name from the project name and the top-level design entity name.

6. Click **Next** to launch IP Toolbench.

Step 1: Parameterize

To parameterize your MegaCore function, follow these steps:

1. Click **Step 1: Parameterize** in IP Toolbench.

For more information on the parameters, refer to "Parameters" on page 3–29).

2. Set the memory type:

   a. Choose the **Memory device**.

   b. Select either **QDRII** or **QDRII+**.

   c. Set the **Clock speed**.

   d. Choose the **Voltage**.

   e. Choose the **Burst length**.

   f. Choose the **Data bus width**.

   g. Choose the **Address bus width**.

   h. Choose the **Memory Latency**.

   i. Select the **Narrow mode** or **Wide mode** to set the local bus width.

3. Set the memory interface.

   a. Set **Device width**.

   b. Set **Device depth**.

   c. Turn off **Use ALTDDIO pin**, if you are targeting HardCopy II devices.

4. Click **Board & Controller** tab or **Next**.

For more information on board and controller parameters, see "Board & Controller" on page 3–31.

5. Choose the number of pipeline registers.

6. To set the read latency, turn on **Manual read latency setting** and specify the latency at **Set latency to clock cycle**.

7. Turn on the appropriate capture mode—DQS or non-DQS capture mode. If you turn off **Enable DQS mode** (non-DQS capture mode), you can turn on **Use migratable bytegroups**.

8. Enter the pin loading for the FPGA pins.

9. Click **Project Settings** tab or **Next**.

For more information on the project settings, see "Project Settings" on page 3–33.

10. Altera recommends that you turn on **Automatically apply QDRII SRAM controller-specific constraints to the Quartus II project** so that the Quartus II software automatically applies the constraints script when you compile the example design.

11. Ensure **Update the example design that instantiates the QDRII SRAM controller variation** is turned on, for IP Toolbench to automatically update the example design file.

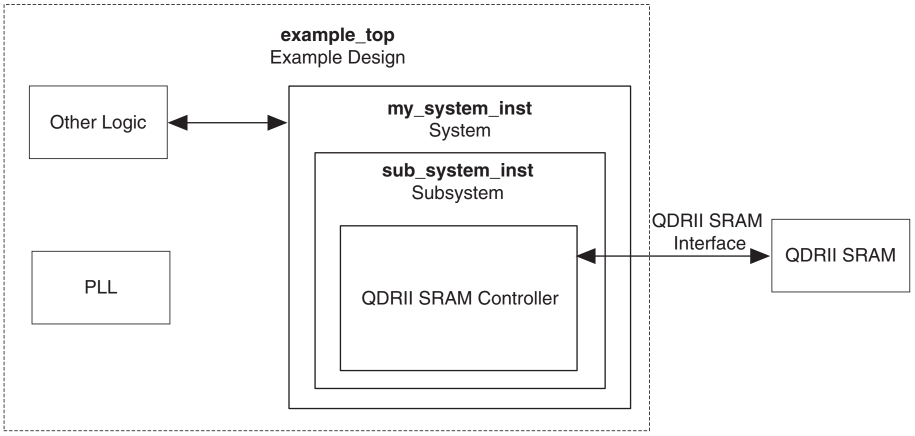
12. Turn off **Update example design system PLL**, if you have edited the PLL and you do not want the wizard to regenerate the PLL when you regenerate the variation.

    The first time you create a custom variation, you must turn on **Update example design system PLL**.

13. The constraints script automatically detects the hierarchy of your design. The constraints script analyzes and elaborates your design to automatically extract the hierarchy to your variation. To prevent the constraints script analyzing and elaborating your design, turn on **Enable hierarchy control**, and enter the correct hierarchy path to your variation. The hierarchy path is the path to your QDRII SRAM controller, without the top-level name. Figure 2–2 shows the following example hierarchy:

```
my_system:my_system_inst|sub_system:sub_system_inst|
```

Figure 2–2. System Naming



14. IP Toolbench uses a prefix (e.g., **qdrii_**) for the names of all memory interface pins. Enter a prefix for all memory interface pins associated with this custom variation.

15. Click **Finish**.

Step 2: Constraints

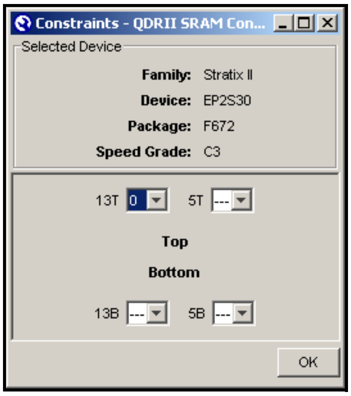
To choose the constraints for your device, follow these steps:

1. Click **Step 2: Constraints** in IP Toolbench.

2. Choose the positions on the device for each of the QDRII SRAM byte groups (see Figure 2–3 on page 2–8). To place a byte group, select the byte group in the drop-down box at your chosen position.

   ☞ The floorplan matches the orientation of the Quartus II floorplanner. The layout represents the die as viewed from above. A byte group consists of a `cq` pin and a number of `q` pins (the same number as the data width).

Figure 2–3. Choose the Constraints


Constraints - QDRII SRAM Con...

Selected Device

Family: Stratix II

Device: EP2S30

Package: F672

Speed Grade: C3

Step 3: Set Up Simulation

An IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus II software. The model allows for fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.

CAUTION You may only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis will create a nonfunctional design.

To generate an IP functional simulation model for your MegaCore function, follow these steps:

1. Click **Step 3: Set Up Simulation** in IP Toolbench.

2. Turn on **Generate Simulation Model**.

3. Choose the language in the **Language** list.

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

5. Click **OK**.

Step 4: Generate

1. To generate your MegaCore function, click **Step 4: Generate** in IP Toolbench.

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

Table 2–1 describes the generated files and other files that may be in your project directory. The names and types of files specified in the IP Toolbench report vary based on whether you created your design with VHDL or Verilog HDL

Table 2–1. Generated Files (Part 1 of 3) (1), (2) & (3)

Filename	Description
<variation name>.**bsf**	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<variation name>.**html**	MegaCore function report file.
<variation name>.**vhd**, or **.v**	A MegaCore function variation file, which defines a VHDL or Verilog HDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
*<variation name>*_**bb.v**	Verilog HDL black-box file for the MegaCore function variation. Use this file when using a third-party EDA tool to synthesize your design.
*<variation name>*_**auk_qdrii_sram.vhd** or **.v**	File that instantiates the control logic and the datapath.
*<variation name>*_**auk_qdrii_sram_addr_cmd_reg.vhd** or **.v**	The address and command output registers.
*<variation name>*_**auk_qdrii_sram_avalon_controller_ipfs_wrap.vhd** or **.v**	File that instantiates the controller.
*<variation name>*_**auk_qdrii_sram_avalon_controller_ipfs_wrap.vho** or **.vo**	VHDL or Verilog HDL IP functional simulation model.

Table 2–1. Generated Files (Part 2 of 3) *(1)*, *(2)* & *(3)*

Filename	Description
*<variation name>*_**auk_qdrii_sram_capture_group_wrapper .vhd** or **.v**	File that contains all the capture group modules (CQ and CQN group modules and read capture registers).
*<variation name>*_**auk_qdrii_sram_clk_gen.vhd** or **.v**	The clock output generators.
*<variation name>*_**auk_qdrii_sram_cq_cqn_group.vhd** or **.v**	The CQ and CQN module.
*<variation name>*_**auk_qdrii_sram_datapath.vhd** or **.v**	Datapath.
*<variation name>*_**auk_qdrii_sram_dll.vhd** or **.v**	DLL.
*<variation name>*_**auk_qdrii_sram_example_driver .vhd** or **.v**	The example driver.
*<variation name>*_**auk_qdrii_sram_read_group.vhd** or **.v**	The read capture registers.
*<variation name>*_**auk_qdrii_sram_pipe_resynch_wrapper. vhd** or **.v**	File that includes the write data pipeline and includes the address and command, read command, write data, and write command pipeline.
*<variation name>*_**auk_qdrii_sram_pipeline_addr_cmd.vhd** or **.v**	Address and command pipeline.
*<variation name>*_**auk_qdrii_sram_pipeline_rdata.vhd** or **.v**	Read data pipeline.
*<variation name>*_**auk_qdrii_sram_pipeline_wdata.vhd** or **.v**	Write data pipeline.
*<variation name>*_**auk_qdrii_sram_read_group.vhd** or **.v**	The read registers.
*<variation name>*_**auk_qdrii_sram_resynch_reg.vhd** or **.v**	The resynchronization FIFO buffers.
*<variation name>*_**auk_qdrii_sram_train_wrapper.vhd** or **.v**	File that contains all the training group modules.
*<variation name>*_**auk_qdrii_sram_test_group.vhd** or **.v**	Training module, which realigns latency.
*<variation name>*_**auk_qdrii_sram_write_group.vhd** or **.v**	The write registers.
<variation name>.**qip**	Contains Quartus II project information for your MegaCore function variations.
<top-level name>.**vhd** or **.v** *(1)*	Example design file.
add_constraints_for_*<variation name>***.tcl**	The add constraints script.

Table 2–1. Generated Files (Part 3 of 3) (1), (2) & (3)

Filename	Description
qdrii_pll_stratixii.vhd or **.v**	Stratix II PLL.

Notes to Table 2–1:
(1) *<top-level name>* is the name of the Quartus II project top-level entity.
(2) *<variation name>* is the name you give to the controller you create with the Megawizard.
(3) IP Tooblench replaces the string **qdrii_sram** with **qdriiplus_sram** for QDRII+ SRAM controllers.

2. After you review the generation report, click **Exit** to close IP Toolbench.

You have finished the walkthrough. Now, simulate the example design (see "Simulate the Example Design" on page 2–11), edit the PLL(s) (see "Edit the PLL" on page 2–18), and compile (see "Compile the Example Design" on page 2–19).

Simulate the Example Design

This section describes the following simulation techniques:

- Simulate with IP Functional Simulation Models
- Simulating With the ModelSim Simulator
- Simulating With Other Simulators
- Simulating in Third-Party Simulation Tools Using NativeLink

Simulate with IP Functional Simulation Models

You can simulate the example design using the IP Toolbench-generated IP functional simulation models. IP Toolbench generates a VHDL or Verilog HDL testbench for your example design, which is in the **testbench** directory in your project directory.

For more information on the testbench, see "Example Design" on page 3–27.

You can use the IP functional simulation model with any Altera-supported VHDL or Verilog HDL simulator. The instructions for the ModelSim simulator are different to other simulators.

Simulating With the ModelSim Simulator

Altera supplies a generic memory model, **lib\qdrii_model.v**, which allows you to simulate the example design with the ModelSim simulator. To simulate the example design with the ModelSim® simulator, follow these steps:

1. Copy the generic memory model to the <*directory name*>**testbench** directory.

2. Open the memory model and the testbench (<*top-level name*>**_vsim.v** or **.vhd**) in a text editor and ensure the signal names have the same capitalization in both files.

3. Start the ModelSim-Altera simulator.

4. Change your working directory to your IP Toolbench-generated file directory <*directory name*>**testbench\modelsim.**

5. To simulate with an IP functional simulation model simulation, type the following command:

   ```
   source <variation name>_vsim.tcl↵
   ```

6. For a gate-level timing simulation (VHDL or Verilog HDL ModelSim output from the Quartus II software), type the following commands:

   ```
   set use_gate_model 1↵
   source <variation name>_vsim.tcl↵
   ```

Simulating With Other Simulators

The IP Toollbench-generated Tcl script is for the ModelSim simulator only. If you prefer to use a different simulation tool, follow these instructions. You can also use the generated script as a guide. You also need to download and compile an appropriate memory model.

☞ The following variables apply in this section:

- <*QUARTUS ROOTDIR*> is the Quartus II installation directory
- <*simulator name*> is the name of your simulation tool
- <*device name*> is the Altera device family name
- <*project name*> is the name of your Quartus II top-level entity or module.
- <*MegaCore install directory*> is the QDRII SRAM Controller installation directory

VHDL IP Functional Simulations

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

1. Create a directory in the <*project directory*>**testbench** directory.

2. Launch your simulation tool inside this directory and create the following libraries:

   - **altera_mf**
   - **lpm**
   - **sgate**
   - *<device name>*
   - **auk_qdrii_lib**

3. Compile the files in Table 2–2 into the appropriate library. The files are in VHDL93 format.

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

Library	Filename
altera_mf	*<QUARTUS ROOTDIR>***/eda/sim_lib/altera_mf_components.vhd**
	*<QUARTUS ROOTDIR>***/eda/sim_lib/altera_mf.vhd**
lpm	*<QUARTUS ROOTDIR>***/eda/sim_lib/220pack.vhd**
	*<QUARTUS ROOTDIR>***/eda/sim_lib/220model.vhd**
sgate	*<QUARTUS ROOTDIR>***/eda/sim_lib/sgate_pack.vhd**
	*<QUARTUS ROOTDIR>***/eda/sim_lib/sgate.vhd**
<device name>	*<QUARTUS ROOTDIR>***/eda/sim_lib/<***device name***>_atoms.vhd**
	*<QUARTUS ROOTDIR>***/eda/sim_lib/***<device name>***_components.vhd**

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

Library	Filename
auk_qdrii_lib	*<project directory>*/*<variation name>***_auk_qdrii_sram_clk_gen.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_addr_cmd_reg.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_cq_cqn_group.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_read_group.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_capture_group_wrapper.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_resynch_reg.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_write_group.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_datapath.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_test_group.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_train_wrapper.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_pipeline_wdata.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_pipeline_rdata.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_pipeline_addr_cmd.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_pipe_resynch_wrapper.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_avalon_controller_ipfs_wrap.vho**
	<project directory>/*<variation name>***_auk_qdrii_sram.vhd**
	<project directory>/*<variation name>***.vhd**
	*<project directory>***/qdrii_pll_stratixii.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_dll.vhd**
	<project directory>/*<variation name>***_auk_qdrii_sram_example_driver.vhd**
	<project directory>/*<project name>***.vhd**
	*<project directory>***/testbench/***<project name>***_tb.vhd**

4. Set the Tcl variable `gRTL_DELAYS` to 1, which tells the testbench to model the extra delays in the system necessary for RTL simulation

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

VHDL Gate-Level Simulations

For VHDL simulations with gate-level models, follow these steps:

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

2. Launch your simulation tool inside this directory and create the following libraries.

   - *<device name>*
   - **auk_qdrii_lib**

3. Compile the files in Table 2–3 into the appropriate library. The files are in VHDL93 format.

Table 2–3. Files to Compile—VHDL Gate-Level Simulations

Library	Filename
<device name>	*<QUARTUS ROOTDIR>***/eda/sim_lib/<***device name***>_atoms.vhd**
	*<QUARTUS ROOTDIR>***/eda/sim_lib/<***device name***>_components.vhd**
auk_qdrii_lib	*<project directory>***/simulation/***<simulator name>***/***<project name>***.vho**
	*<project directory>***/testbench/***<project name>***_tb.vhd**

4. Set the Tcl variable `gRTL_DELAYS` to 0, which tells the testbench not to use the insert extra delays in the system, because these are applied inside the gate-level model.

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

Verilog HDL IP Functional Simulations

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

1. Create a directory in the *<project directory>***testbench** directory.

2. Launch your simulation tool inside this directory and create the following libraries.:

   - **altera_mf_ver**
   - **lpm_ver**
   - **sgate_ver**
   - *<device name>***_ver**
   - **auk_qdrii_lib**

3. Compile the files in Table 2–4 into the appropriate library.

Table 2–4. Files to Compile—Verilog HDL IP Functional Simulation Models

Library	Filename
altera_mf_ver	*<QUARTUS ROOTDIR>***/eda/sim_lib/altera_mf.v**
lpm_ver	*<QUARTUS ROOTDIR>***/eda/sim_lib/220model.v**
sgate_ver	*<QUARTUS ROOTDIR>***/eda/sim_lib/sgate.v**
*<device name>***_ver**	*<QUARTUS ROOTDIR>***/eda/sim_lib/<***device name***>_atoms.v**
auk_qdrii_lib	*<project directory>***/***<variation name>***_auk_qdrii_sram_clk_gen.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_addr_cmd_reg.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_cq_cqn_group.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_read_group.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_capture_group_wrapper.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_resynch_reg.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_write_group.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_datapath.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_test_group.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_train_wrapper.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_pipeline_wdata.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_pipeline_rdata.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_pipeline_addr_cmd.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_pipe_resynch_wrapper.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_avalon_controller_ipfs_wrap.vo**
	*<project directory>***/***<variation name>***_auk_qdrii_sram.v**
	*<project directory>***/***<variation name>***.v**
	*<project directory>***/qdrii_pll_stratixii.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_dll.v**
	*<project directory>***/***<variation name>***_auk_qdrii_sram_example_driver.v**
	*<project directory>***/***<project name>***.v**
	*<project directory>***/testbench/***<project name>***_tb.vhd**

4. Set the Tcl variable `gRTL_DELAYS` to 1, which tells the testbench to model the extra delays in the system necessary for RTL simulation.

5. Configure your simulator to use transport delays, a timestep of picoseconds and to include the **auk_qdrii_lib, sgate_ver**, **lpm_ver**, **altera_mf_ver**, and <*device name*>**_ver** libraries.

Verilog HDL Gate-Level Simulations

For Verilog HDL simulations with gate-level models, follow these steps:

1. Create a directory in the <*project directory*>**testbench** directory.

2. Launch your simulation tool inside this directory and create the following libraries:

   - <*device name*>**_ver**
   - **auk_qdrii_lib**

3. Copy the **<***project directory***>/simulation/**<*simulator name*>**_v.sdo** file into the compilation directory.

4. Compile the files in Table 2–5 into the appropriate library.

Table 2–5. Files to Compile—Verilog HDL Gate-Level Simulations

Library	Filename
<*device name*>**_ver**	<*QUARTUS ROOTDIR*>**/eda/sim_lib/**<*device name*>**_atoms.v**
auk_qdrii_lib	<*project directory*>**/simulation/**<*simulator name*>**/<toplevel_name>.vo**
	<*project directory*>**/testbench/**<*project name*>**_tb.v**

5. Set the Tcl variable `gRTL_DELAYS` to 0, which tells the testbench not to use the insert extra delays in the system, because these are applied inside the gate level model. Configure your simulator to use transport delays, a timestep of picoseconds, and to include the **auk_qdrii_lib** and <*device name*>**_ver** library.

Simulating in Third-Party Simulation Tools Using NativeLink

You can perform a simulation in a third-party simulation tool from within the Quartus II software, using NativeLink.

For more information on NativeLink, refer to the *Simulating Altera IP Using NativeLink* chapter in volume 3 of the *Quartus II Handbook*.

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

1. Create a custom variation with an IP functional simulation model.

2. Obtain and copy a memory model to a suitable location, for example, the testbench directory.

   Before running the simulation you may also need to edit the testbench to match the chosen memory model.

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

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

5. On the Assignments menu click **Settings**, expand **EDA Tool Settings** and select **Simulation**. Select a simulator under **Tool Name** and in **NativeLink Settings**, select **Compile Test Bench** and click **Test Benches**.

6. Click **New**.

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

8. Enter the name of the automatically generated testbench, <*project name*>`_tb`, in **Test bench entity**.

9. Enter the name of the top-level instance in **Instance**.

10. Change **Run for** to **500 µs**.

11. Add the testbench files. In the **File name** field browse to the location of the memory model and the testbench, <*project name*>**_tb**, click **OK** and click **Add**.

12. Click **OK**.

13. Click **OK**.

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

Edit the PLL

The IP Toolbench-generated example design includes up to two PLLs (system PLL and fedback clock PLL), which have an input to output clock ratio of 1:1 and a clock frequency that you entered in IP Toolbench. In addition, IP Toolbench correctly sets all the phase offsets of all the relevant clock outputs for your design. You can edit either PLLs' input clock to make it conform to your system requirements. If you re-run IP Toolbench, it does not overwrite the system PLL, if you turn off **Reset the PLL to the default setting**, so your edits are not lost.

For more information on the PLL, see "PLL Configuration" on page 3–26.

To edit the example PLL, follow these steps:

1. Choose **MegaWizard Plug-In Manager** (Tools menu).

2. Select **Edit an existing custom megafunction variation** and click **Next**.

3. In your Quartus II project directory, for VHDL choose **qdrii_pll_**<*device name*>**.vhd**; for Verilog HDL choose **qdrii_pll_**<*device name*>**.v**.

4. Click **Next**.

5. Edit the PLL parameters in the `altpll` MegaWizard Plug-In.

For more information on the `altpll` megafunction, refer to the Quartus II Help or click **Documentation** in the `altpll` MegaWizard Plug-In.

Compile the Example Design

Before the Quartus II software compiles the example design it runs the IP Toolbench-generated Tcl constraints script, **auto_add_constraints.tcl**.

The **auto_add_qdrii_constraints.tcl** script calls the **add_constraints_for_**<*variation name*>**.tcl** script for each variation in your design. The **add_constraints_for_**<*variation name*>**.tcl** script checks for any previously added constraints, removes them, and then adds constraints for that variation.

The constraints script analyzes and elaborates your design, to automatically extract the hierarchy to your variation. To prevent the constraints script analyzing and elaborating your design, turn on **Enable hierarchy control** in the wizard, and enter the correct hierarchy path to your data path (see step 13 on page 2–6).

When the constraints script runs, it creates another script, **remove_constraints_for_**<*variation name*>**.tcl**, which you can use to remove the constraints from your design.

To compile the example instance, follow these steps:

1. *Optional*. Enable TimeQuest Timing Analyzer.

   a. On the Assignments menu click **Settings**, expand **Timing Analysis Settings**, and select **Use TimeQuest Timing Analyzer**.

   b. Use the DDR timing wizard (DTW) to generate the required QDRII SRAM Synopsys design constraint (SDC) TimeQuest constraints for the design.

For more information on the DTW, refer to the *DTW User Guide*.

2. Choose **Start Compilation** (Processing menu), which runs the add constraints scripts, compiles the example design, and performs timing analysis.

3. View the Classic or TimeQuest Timing Analyzer to verify your design meets timing.

If your design does not meet timing requirements, add the following lines to you **.qsf** file:

```
set_instance_assignment -name GLOBAL_SIGNAL OFF -to soft_reset_n
set_global_assignment -name OPTIMIZE_FAST_CORNER_TIMING ON
```

If the compilation does not reach the frequency requirements, follow these steps:

1. Choose **Settings** (Assignments menu).

2. Choose **Analysis and Synthesis Settings** in the category list.

3. Select **Speed** in Optimization Technique.

4. Click **OK**.

5. Re-compile the example design by choosing **Start Compilation** (Processing menu).

To view the constraints in the Quartus II Assignment Editor, choose **Assignment Editor** (Assignments menu).

If you have "?" characters in the Quartus II Assignment Editor, the Quartus II software cannot find the entity to which it is applying the constraints, probably because of a hierarchy mismatch. Either edit the constraints script, or enter the correct hierarchy path in the Hierarchy tab (see step 13 on page 2–6).

For more information on constraints, see "Constraints" on page 3–29.

Program a Device

After you have compiled the example design, you can perform gate-level simulation (see "Simulate the Example Design" on page 2–11) or program your targeted Altera device to verify the example design in hardware.

With Altera's free OpenCore Plus evaluation feature, you can evaluate the QDRII SRAM Controller MegaCore function before you obtain a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model, and produce a time-limited programming file.

For more information on OpenCore Plus hardware evaluation using the QDRII SRAM Controller MegaCore function, see "OpenCore Plus Evaluation" on page 1–3, "OpenCore Plus Time-Out Behavior" on page 3–10, and *Application Note 320: OpenCore Plus Evaluation of Megafunctions*.

Implement Your Design

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

Set Up Licensing

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

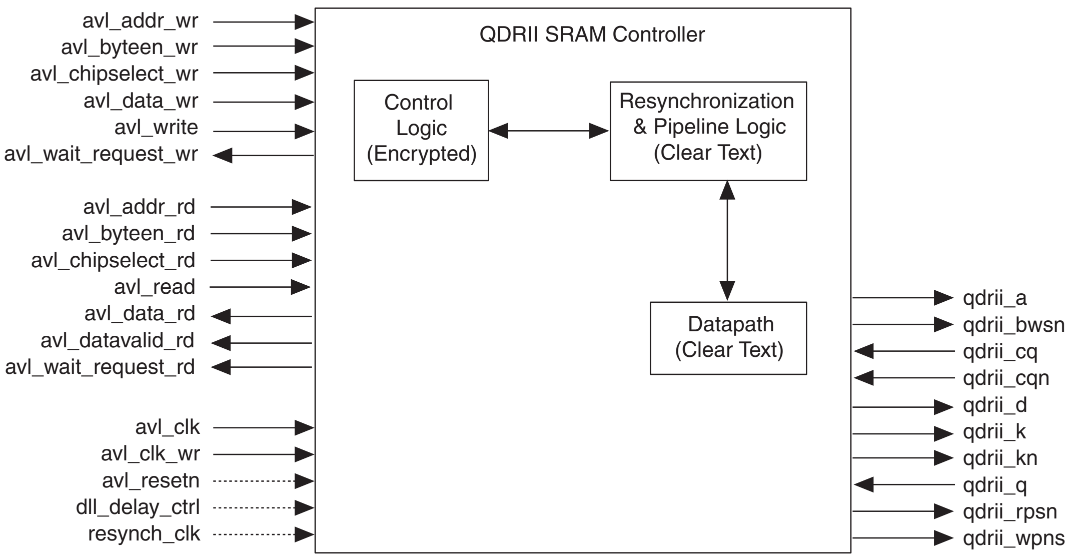
After you obtain a license for QDRII SRAM Controller, you can request a license file from the Altera web site at **www.altera.com/licensing** and install it on your computer. When you request a license file, Altera emails you a **license.dat** file. If you do not have Internet access, contact your local Altera representative.

3. Functional Description

Block Description

Figure 3–1 shows a block diagram of the QDR SRAM controller MegaCore® function.

Figure 3–1. QDRII SRAM Controller Block Diagram (1)



Notes to Figure 3–1:
(1) You can edit the `qdrii_` prefix.

The QDRII SRAM Controller comprises the following three parts:

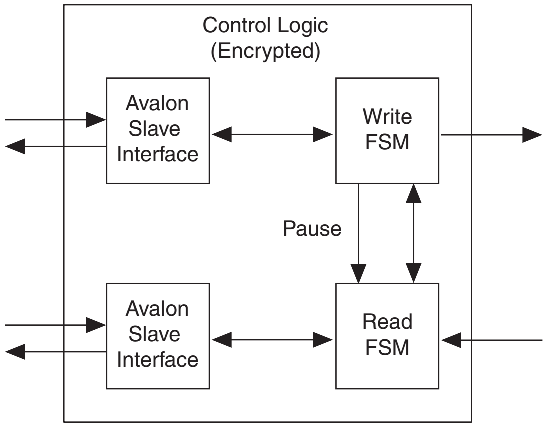
- The control logic gets read and write requests from the Avalon® interface and turn them into QDRII SRAM read and write requests, with the correct timing and concatenating consecutive addresses where applicable.
- The resynchronization and pipeline logic provides the resynchronization system, the training block, and the optional pipeline logic.
- The datapath contains all the I/O and the clock generation.

You can use the datapath on its own if you want to create you own resynchronization scheme or want to have an interface similar to the QDRII SRAM v1.0.0 interface.

Control Logic

Figure 3–2 shows the control logic block diagram.

Figure 3–2. Control Logic Block Diagram



The basic architecture comprises two separate almost independent channels. The write channel sends data to the memory. The read channel receives the data. The address port on the QDRII SRAM interface is shared— a write takes precedence when simultaneous reads and writes occur. On the Avalon interface, all the signals are independent.

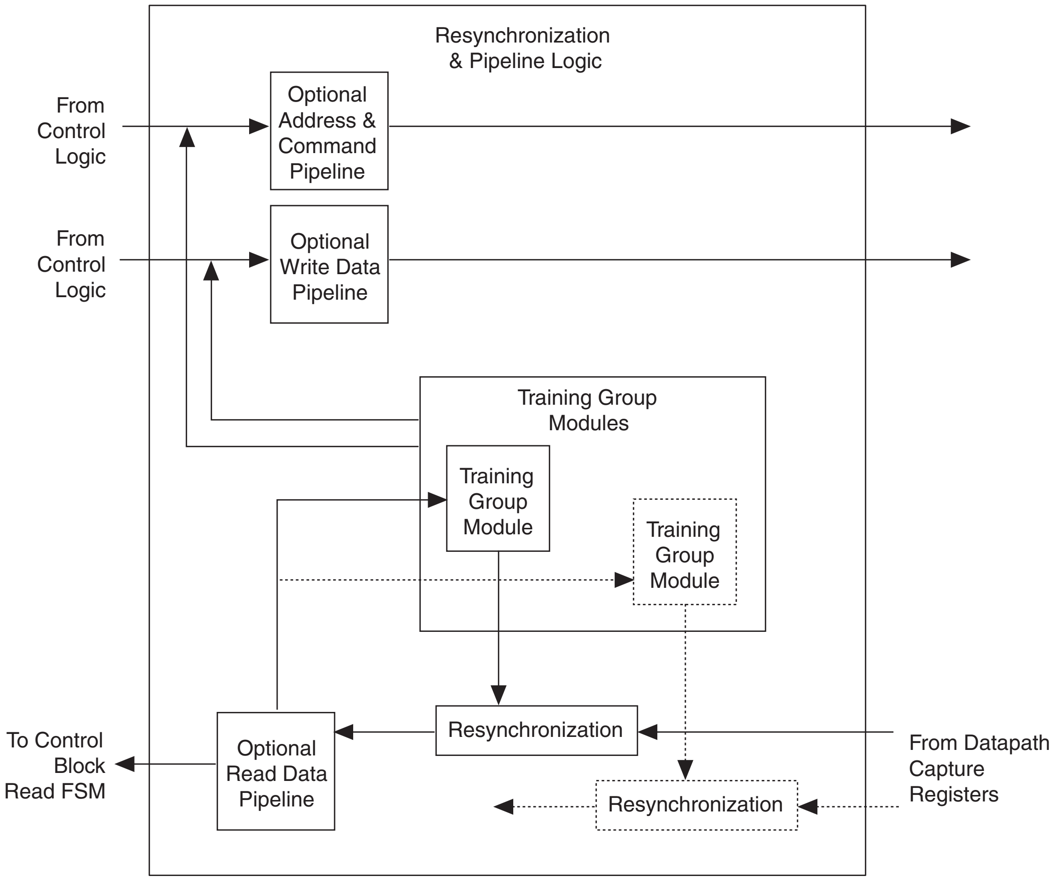
The write channel comprises an Avalon interface and a small pipeline to perform two-cycle bursts. A finite state machine (FSM) controls the signaling to the Avalon interface and deals with the data from Avalon interface. The data and address are then passed to the I/O and sent to the memory.

Similarly for the read channel, a FSM controls the signaling to the Avalon interface and deals with the data going to Avalon interface. The read command is passed to the QDRII SRAM interface and the data is captured when arriving back. Simultaneous read and write operations may lead to pauses on the Avalon read interface.

Resynchronization & Pipeline Logic

Figure 3–3 shows the resynchronization and pipeline logic block diagram.

Figure 3–3. Resynchronization & Pipeline Logic Block Diagram


Resynchronization & Pipeline Logic
From Control Logic
Optional Address & Command Pipeline
From Control Logic
Optional Write Data Pipeline
Training Group Modules
Training Group Module
Training Group Module
To Control Block Read FSM
Optional Read Data Pipeline
Resynchronization
Resynchronization
From Datapath Capture Registers

Address & Command Pipeline

The optional address and command pipeline pipelines all commands and addresses by a predefined number of cycles.

Write Data Pipeline

The write data pipeline pipelines the write data by a specified number of clock cycles.The number of pipelines is equal to the address and command pipelines, because the controller already aligns the data, address and command correctly, therefore the amount of delay going to the I/O is identical.

Training Group Module

The training group module sends all the control, data, and address during training; it reverts to the controller-issued signals after training. It also pauses the controllers for the duration of the training and sends some feedback to the resynchronization logic to realign the pointers to get to the desired latency. To ensure stability the read pointer is aligned only after the DLL is stable. The write pointer is synchronously reset after the read pointer. You can view the training signals from outside the example design.

Read Data Pipeline

The optional read data pipeline pipelines the data after it is resynchronized by a predefined number of cycles.

Resynchronization Logic

The resynchronization logic transfers the data from the QDRII SRAM clock domain onto the system clock domain.

A small dual-port RAM block resynchronizes the data onto the system clock. It writes and reads data every cycle. The frequency is the same on either side.

The amount of buffering in the dual-port RAM automatically compensates for any phase effects. However, there is no way of knowing in which cycle the data is valid. Also the latency may vary from board to board, even device to device depending on the timing relationship of the clocks. Thus the training group module guarantees that each QDRII device has the same read latency and that the latency is fixed and known at startup.

Data is sent to a specific address. The same address is read at the same time. It takes a certain amount of time to propagate the first data to the memory and read it back. This first set of clock cycles is deemed invalid and is not taken into account.

When this initialization time has elapsed, the training group module monitors the data coming back and checks for its validity.

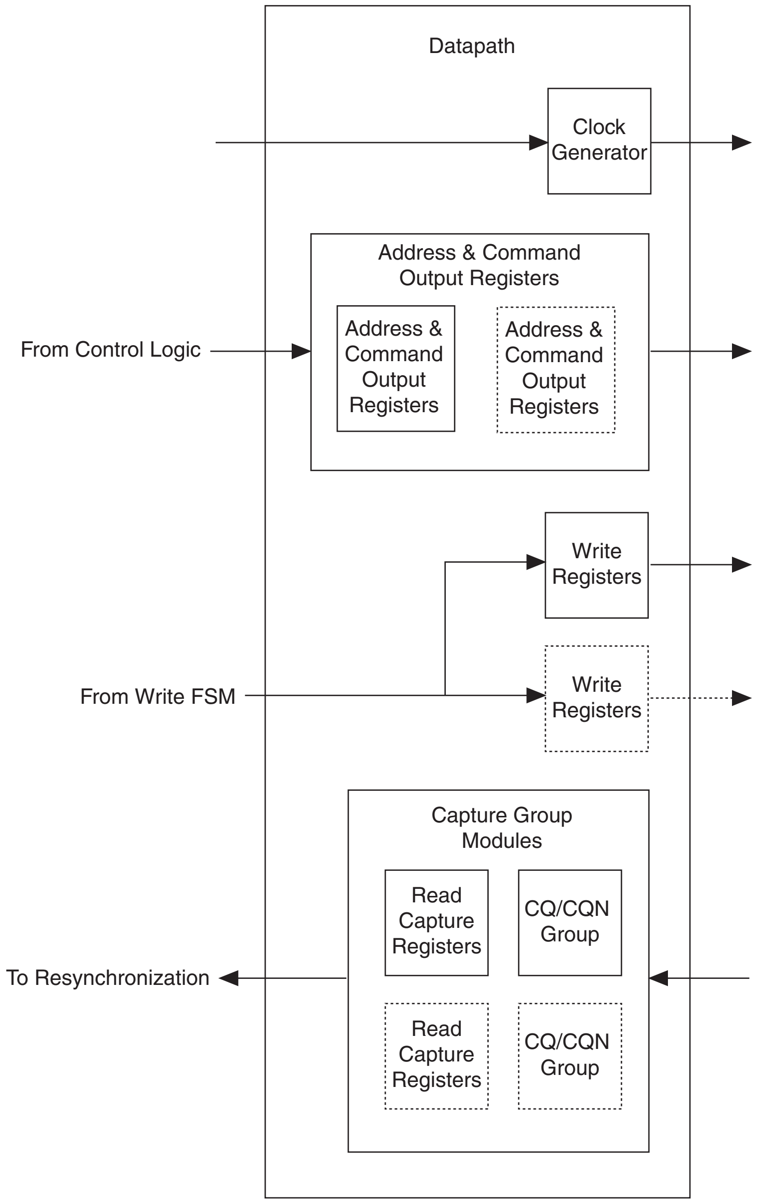
When the training group module detects a pattern, it checks to see if it is too early, too late, or on time. If the pattern is too early, the pointer moves by one; too late, the pointer moves by one in the other direction. The training group module retrains until the pointer is correct.

The RAM size ensures there is minimal latency, but there is enough slack to compensate for the training pattern realignment.

Datapath

Figure 3–4 on page 3–6 shows the datapath block diagram.

Figure 3–4. Datapath Block Diagram



Clock Generator

The clock generator generates the memory signals `k` and `kn`. The clocks are derived from the PLL-generated clock and are shifted by 90° to the system clock.

Address & Command Output Registers

The address and command output registers generate the following outputs:

- Address
- Read
- Write
- Write byte enable

There is one set of signals per device on a board.

With more than one device on a board, a suffix indicates the width position and depth position. The width can be anything up to what the device supports (for example, you can make a 72-bit interface out of four 18-bit interfaces). The depth is limited to 2.

For a device depth of two, you must connect the reads and writes to each device. The top address bit going into the address command top-level file is a device select, which selects device 0 or 1 by setting the read and write of the unused device to 1.

Write Registers

The write registers comprise write I/O blocks going to the memory. For each memory in width, the controller creates a data bus. For a device depth of two, the controller shares the data bus between the two devices.

The Capture Group Module

The capture group module comprises the following elements:

- CQ/CQN group module
- Read capture registers

The controller uses the 90°–shifted `cq` and `cqn` clocks for the capture registers of the `q` bus.

When captured, the controller synchronizes the two words on a double width bus.

With more than one device, one `cq`/`cqn` pair and `q` bus are connected per device in the width direction. For a device depth of two, it shares the `q` and `cq`/`cqn` signals.

All the signals go out of the block with their associated internal `cq` clock, so you can use Altera's resynchronization scheme or implement your own.

Altera recommends the following read capture implementation for data captures from QDRII SRAM devices when using complementary echo clocks (`cq` and `cqn` signals).

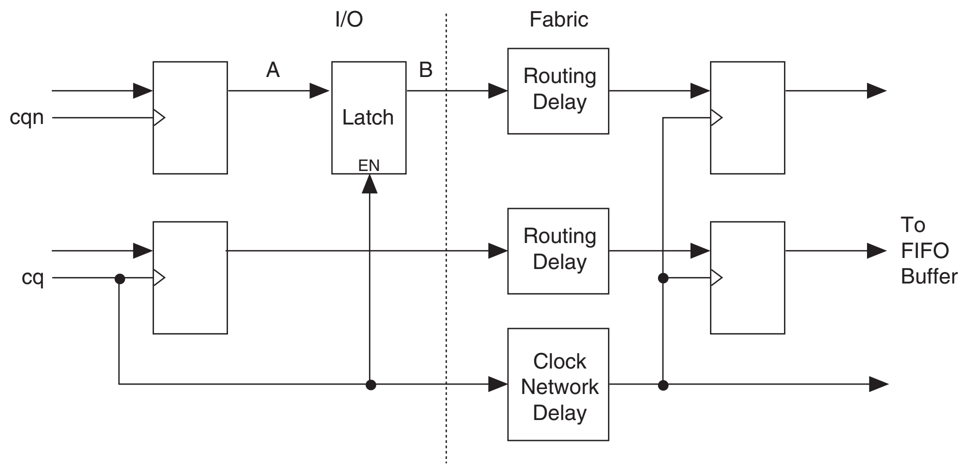
The Stratix II IOE contains two input registers and a latch. The `cq` and `cqn` echo clock signals clock the positive and negative half-cycle registers during reads. The latch holds the negative half-cycle data until the next rising edge on `cq`. However, the latch in the IOE is not recommended when the complementary clocks do not have 50% duty cycle or skew, because the latch, controlled by the `cq` clock, is still transparent until just after the register clocked on the `cqn` signal captures the data.

Instead, the captured read data is recaptured with the `cq` echo clock in the FPGA fabric using a zero-cycle path. The `cq` echo clock is routed into the FPGA fabric using dedicated clock routing (Altera recommends global routing) to provide minimum clock skew across all recapture registers. If you do not have enough global clock network resources, you have the option of using the regional clock network. Routing the `cq` over a clock network adds delay. The Quartus II software fitter places and routes the recapture registers so that the data delay is sufficient to meet the setup and hold requirements at the device registers.

☞ You should only use regional routing if you run out of the global clock networks.

Figure 4 shows a block diagram of the new read capture implementation.

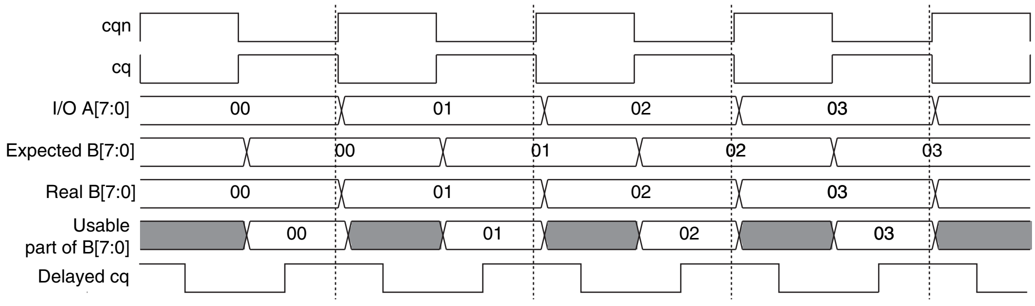
Figure 4. Block Diagram of the New Read Capture Implementation



The data from the latch becomes valid following the rising edge of the `cq` signal (when the latch becomes transparent) and, in a worst-case condition, becomes invalid following the rising edge of `cqn` signal (when roughly half a cycle = t_{KHKH}), which is done by creating a zero-cycle path between the latch and a device register. The data is re-captured in the device using the same edge of the `cq` signal that makes the latch transparent. Both the `cq` signal and the data cross the IOE-to-device boundary where they are delayed. The `cq` signal is delayed by slightly more than by the data needed to meet the setup time for this register. However, the delay is not enough to violate its hold time,which is related to the rising edge of `cqn` signal. Because the data is recaptured in the FPGA while the latch is valid, the IOE capture register timing margins are not impacted.

Figure 5 is a timing diagram of the IOE that assumes the latch is still transparent when `cqn` rising edge occurs. The real `B`, expected `B`, and delayed `cq` signals represent the data and clock to the re-capture registers. The output of latch B is either real `B` or expected `B`, depending on the relationship between `cq` and `cqn`. To cover both cases, the usable part of `B` signal should be captured before going to the resynchronization FIFO buffers. Routing delay aligns the data with the clock.

Figure 5. Timing Diagram of the IOE



OpenCore Plus Time-Out Behavior

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

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

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

For MegaCore functions, the untethered timeout is 1 hour; the tethered timeout value is indefinite.

Your design stops working after the hardware evaluation time expires, the reads and writes go low, and the `wait` output goes high.

For more information on OpenCore Plus hardware evaluation, see "OpenCore Plus Evaluation" on page 1–3 and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

Interfaces & Signals

This section describes the following topics:

- "Interface Description" on page 3–10
- "Signals" on page 3–22

Interface Description

This section describes the following Avalon interface requests:

- Writes
- Reads
- Simultaneous Read & Write Timing

For more information on the Avalon interface, see the *Avalon Bus Specification Reference Manual*.

Writes

This section discusses the following topics:

- "Isolated Write" on page 3–11
- "Bursts" on page 3–13
- "Bursts with Pauses" on page 3–14

If the address is the consecutive, you can have consecutive write cycles (see "Bursts" on page 3–13). Non-consecutive addresses are split into two transfers and you must pause a transfer (see "Bursts with Pauses" on page 3–14).

Isolated Write

Figure 3–1 shows an isolated write transaction on a burst of four (narrow mode). The Avalon interface receives a write request, which the controller immediately accepts. It then transfers the write data (the exact timing may vary) to the QDRII SRAM interface. As it receives only half the required data for a burst of four, it masks the second part of the burst on the QDRII SRAM interface as invalid.

Figure 3–1. Isolated Write—Burst of Four (Narrow Mode)

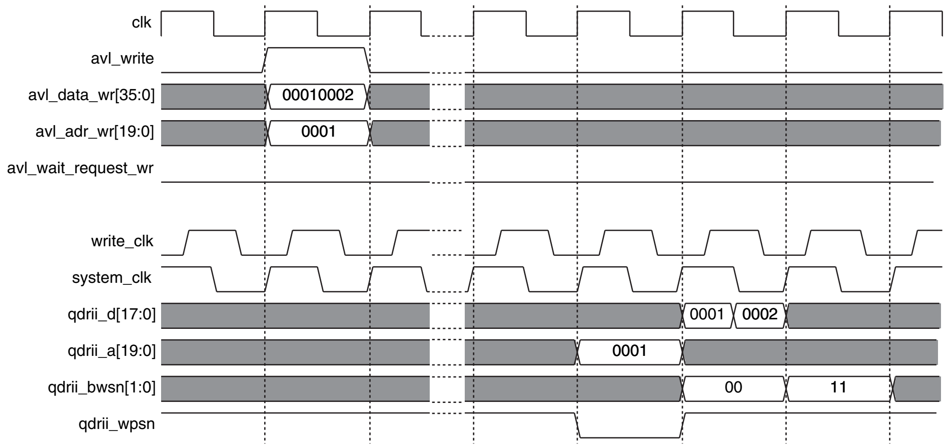


Figure 3–2 on page 3–12 shows a burst of two, the controller takes the data straight away and puts it on the QDRII SRAM interface a few cycle later (the exact timing may change). Because it takes as many Avalon clock cycles as QDRII SRAM clock cycles to write the data, you can put write accesses back-to-back. The write cycles have no influence on the read cycles as the address is put on half a clock cycle.

Figure 3–2. Write—Burst of Two

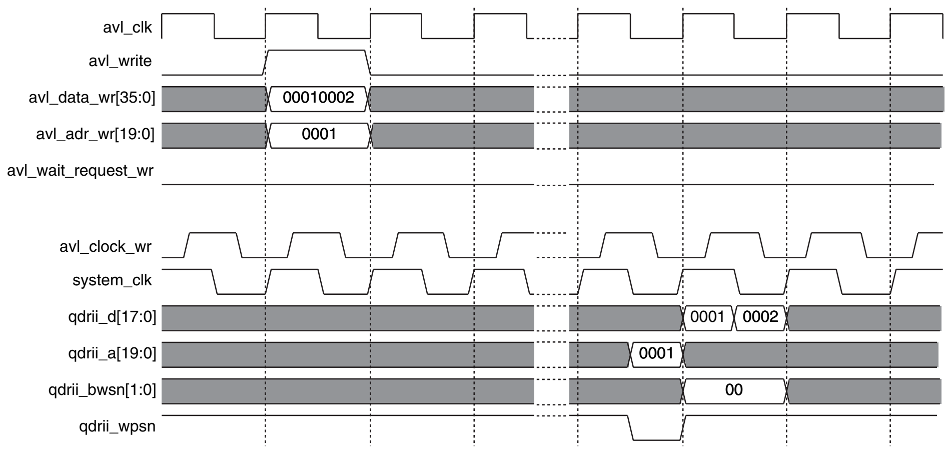
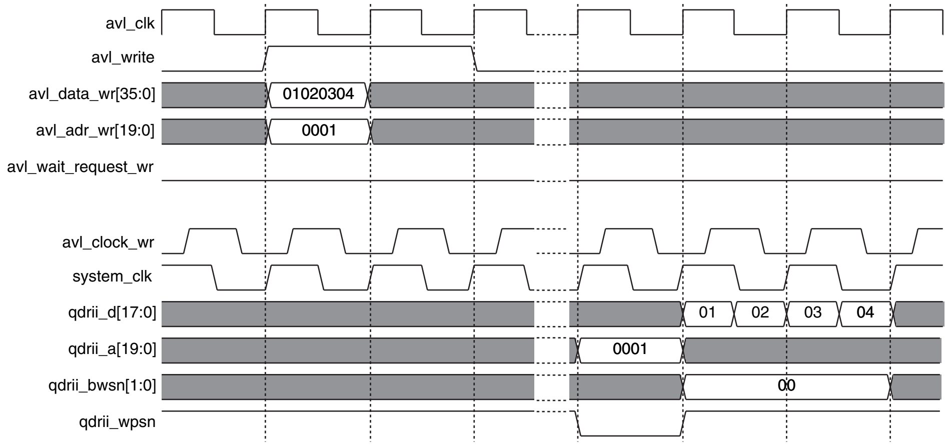


Figure 3–3 shows a burst of four (wide mode), all the data is present in one clock cycle. After one Avalon write, you can transfer data for two clock cycles on the QDRII SRAM interface. In this example, all the data bits are valid and the byte mask is set to enable the whole transfer.

Figure 3–3. Write—Burst of Four (Wide Mode)

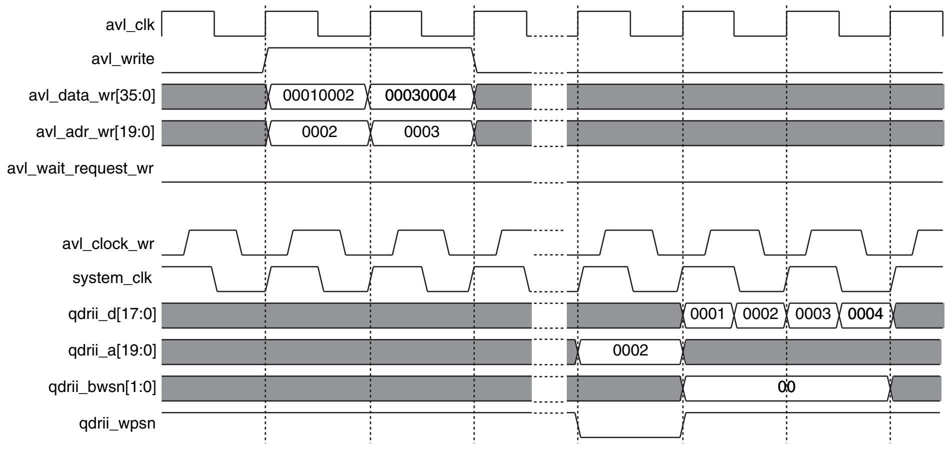


Bursts

Bursts are only possible on the Avalon side in the burst of two mode, where you can transfer data every clock cycle and in bursts of four (narrow mode). It is not possible in the burst of four (wide mode), because it takes two QDRII SRAM clock cycles to transfer one Avalon clock cycle of data.

Figure 3–4 shows the burst of four (narrow mode). When two write requests are sent on the Avalon interface at consecutive addresses, the controller automatically concatenates them and transfers them to the QDRII SRAM, if the first one is an even address. If more data is coming in the following cycle, it is also sent straight away, without any pause.

Figure 3–4. Write—Burst of Four (Narrow Mode)



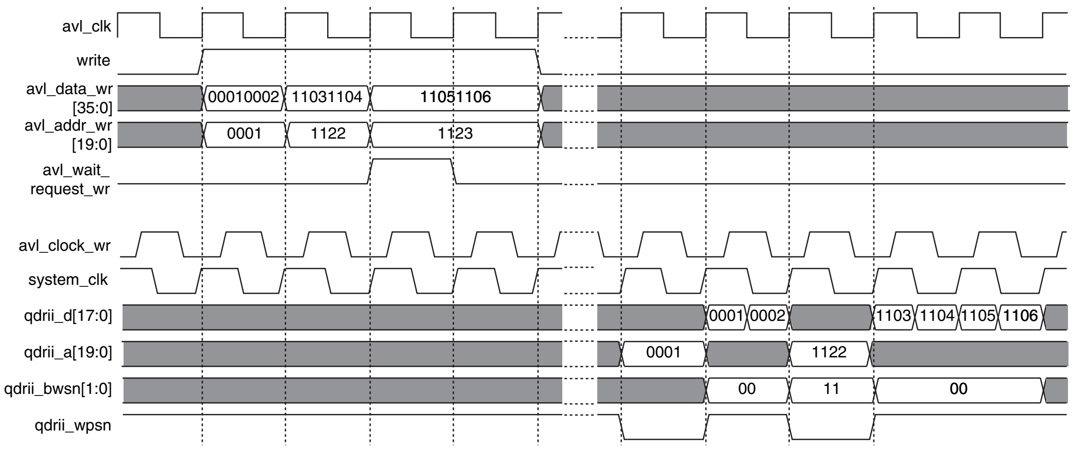
This section does not illustrate the burst of two example, because you can transfer any data at any address in every Avalon clock cycle. The timing of the `qdrii_a` signal is different, see Figure 3–1.

Bursts with Pauses

There are no pauses when using a burst of two memories. For the burst of four, there are some pauses (depending on the mode). In narrow mode, if the transfers are to consecutive addresses all the time, no pause occurs. If the transfers are to non-consecutive addresses, a pause may occur, see Figure 3–5. a pause occurs only in the following conditions:

- A one-cycle write to address <a> followed straight away by a two-cycle transfer to addresses <b> and <b + 1>
- The second half of the transfer to <b> is paused for a clock cycle

Figure 3–5. Write Burst with Pause—Burst of Four (Narrow Mode)



For a burst of four (wide mode), you cannot transfer more than one write request every other cycle, because it takes two cycles on the QDRII SRAM side to send the data. Therefore, if two consecutive writes arrive, the controller pauses the second one for one clock cycle.

Reads

This section discusses the following topics:

- "Isolated Read" on page 3–15
- "Burst" on page 3–17
- "Bursts with Pauses" on page 3–18

Isolated Read

Figure 3–6 shows a read request from the Avalon read interface for a burst of four. The Avalon read FSM issues a latent read and transfers the data back at a later stage, which frees the Avalon interface. The controller transfers the read to the QDRII SRAM. A few cycles later (timing is not accurate), the data arrives, in synchronization with the `cq` and `cqN` clocks. Even though only one set of data was requested, the memory send two sets of data. The controller captures and resynchronizes the data onto the system clock and it appears on the Avalon interface a few cycles later. The controller asserts `avl_data_read_valid` with the data to validate the data cycle.

Figure 3–6. Isolated Read—Burst of Four (Narrow Mode)

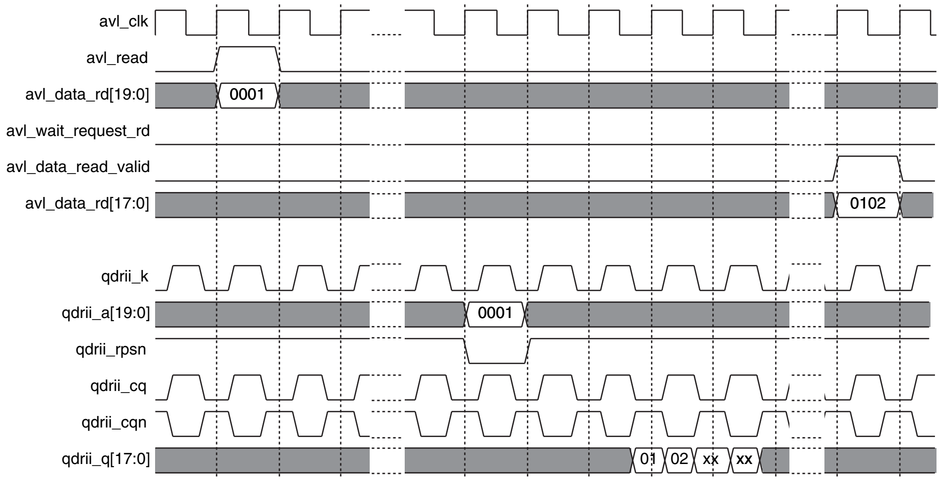


Figure 3–7 shows a single read request from the Avalon interface for a burst of two. The principle is identical to the burst of four, but all the data bits coming back are transferred onto the Avalon interface. The timing on the QDRII SRAM interface is slightly different as the address is only present for half a clock cycle.

Figure 3–7. Isolated Read—Burst of Two (Wide Mode)

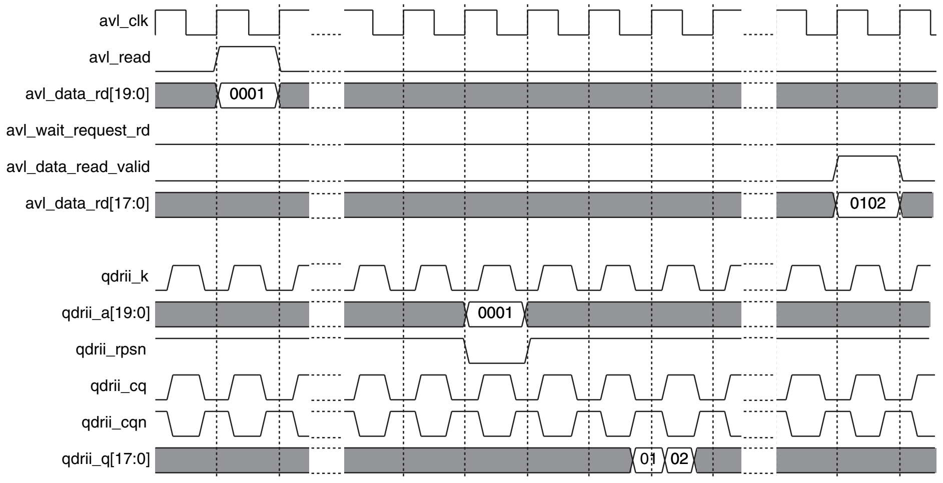
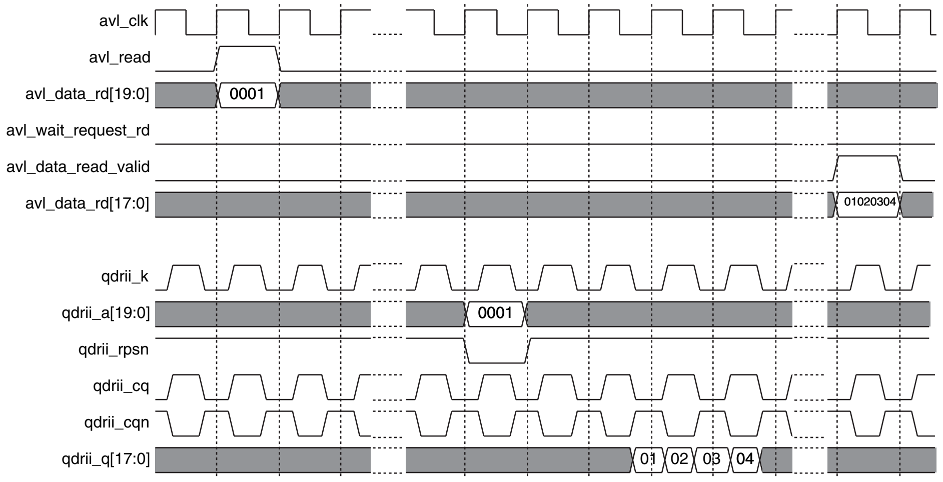


Figure 3–8 shows the behavior of a single read request for a burst of four (wide mode). The read occurs on the Avalon interface, the slave issues a latent read answer. The read command is sent to the memory with the address.

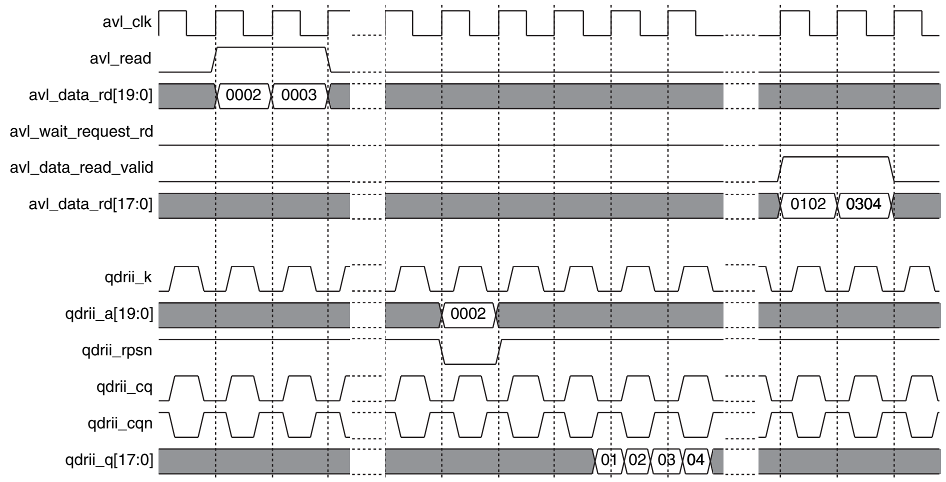
Figure 3–8. Isolated Read—Burst of Four (Wide Mode)



Burst

Bursts only apply to burst of four, (narrow mode), see Figure 3–9. For the other two modes, there is no such concept as all the data required on the QDRII SRAM interface is available for a single Avalon read. The burst consists of two consecutive read requests. The controller sends one read request to the memory, which returns the four half cycles of value. After resynchronization, the data is sent back to the Avalon interface.

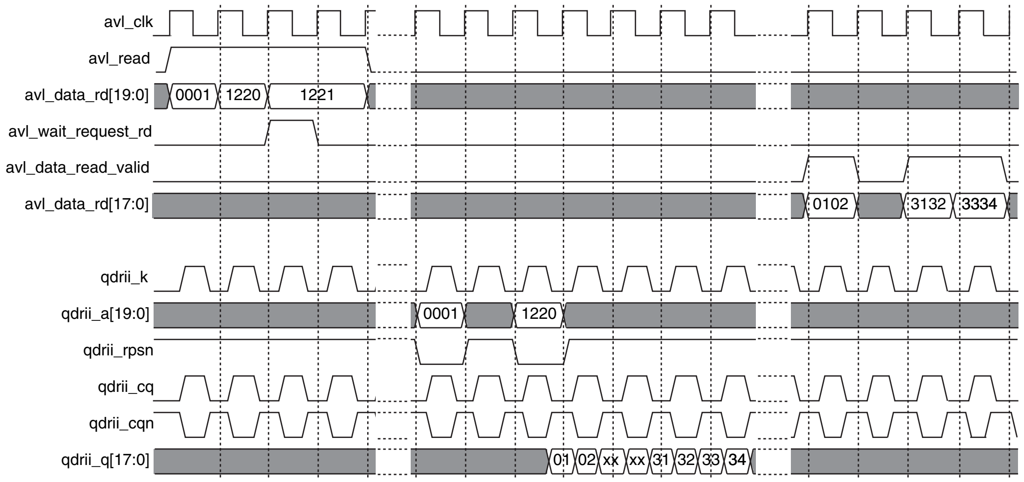
Figure 3–9. Burst—Burst of Four (Narrow Mode)



Bursts with Pauses

Bursts with pauses only applies to bursts of four, (narrow mode). When several read requests to non-consecutive addresses occur, it takes more time to get the data from the memory (it take two cycles per read access) than time needed to request them. Figure 3–10 on page 3–19 shows a read followed by two reads to consecutive addresses. As the first two requests are not to consecutive addresses, the controller has to pause the read requests to insert a clock cycle. The following two reads still get concatenated to make a burst of four, avoiding loss of bandwidth.

Figure 3–10. Burst with Non-Consecutive Address—Burst of Four (Narrow Mode)



Simultaneous Read & Write Timing

This section discusses the following topics:

- "Burst of Four (Narrow Mode)" on page 3–19
- "Burst of Two" on page 3–20
- "Burst of Four (Wide Mode)" on page 3–21

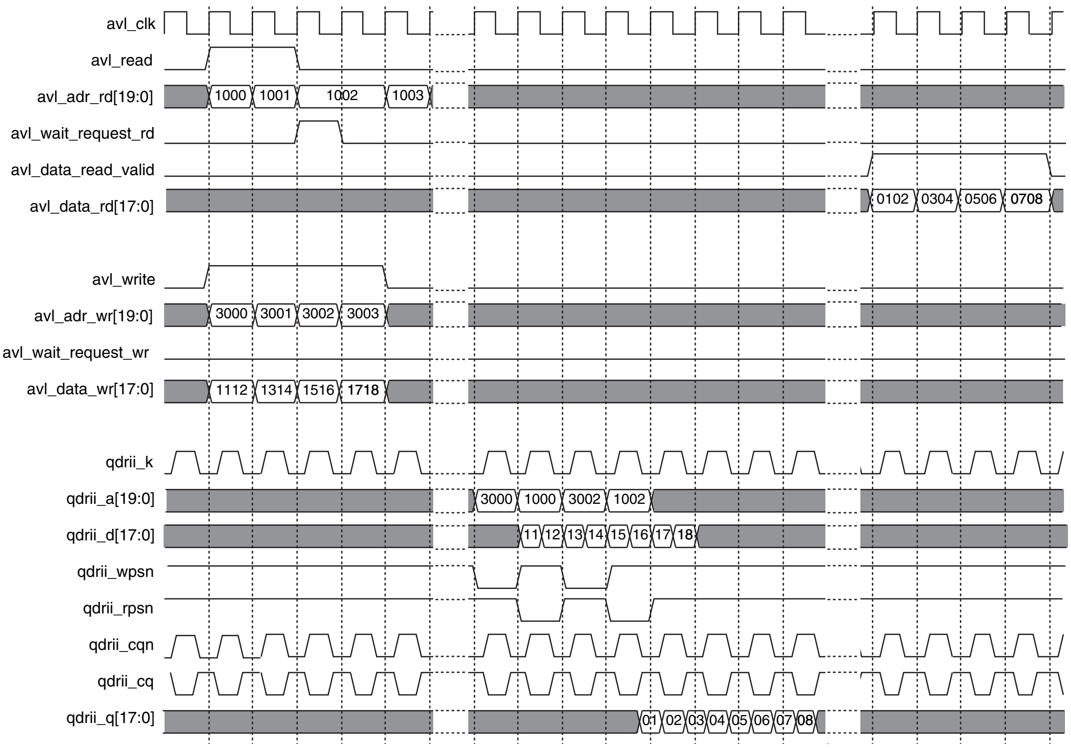
The QDRII SRAM protocol allows simultaneous reads and writes to the memory. As the address bus is shared between the read and write, if a concurrent read and write occurs, some arbitration may be necessary.

Burst of Four (Narrow Mode)

For a burst of four, you cannot send a read and a write request during the same clock cycle. Because it takes two clock cycle per transfer, you can alternate reads and writes every other cycle. Thus you lose no bandwidth apart from an initial one clock cycle on either the read or the write.

When a read and a write arrive at the same time, the write takes priority over the read. For a continuous read and write, there is a one off pause on the read side, see Figure 3–11 on page 3–20.

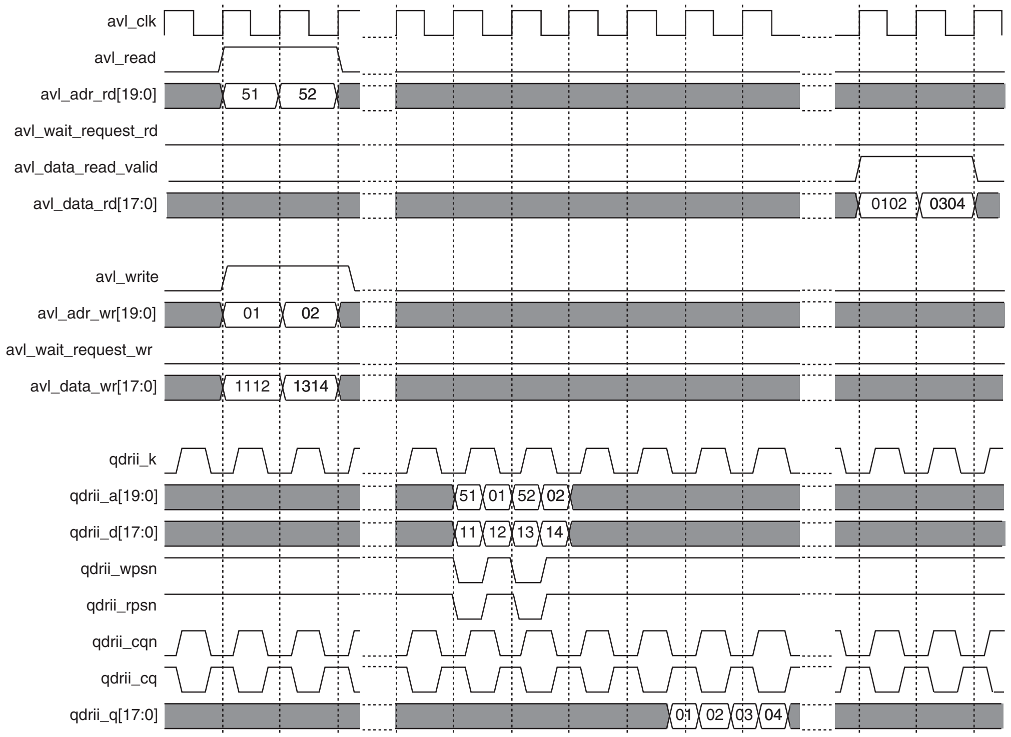
Figure 3–11. Simultaneous Read & Write—Burst of Four (Narrow Mode)



Burst of Two

For the burst of two, the protocol already allows simultaneous reads and writes by asserting `readn` and `writen` and their respective addresses for only half a clock cycle. No arbitration on the Avalon interface is required and you can use the full bandwidth, without even losing any initial cycles. Figure 3–12 on page 3–21 shows concurrent reads and writes in a burst of two configuration.

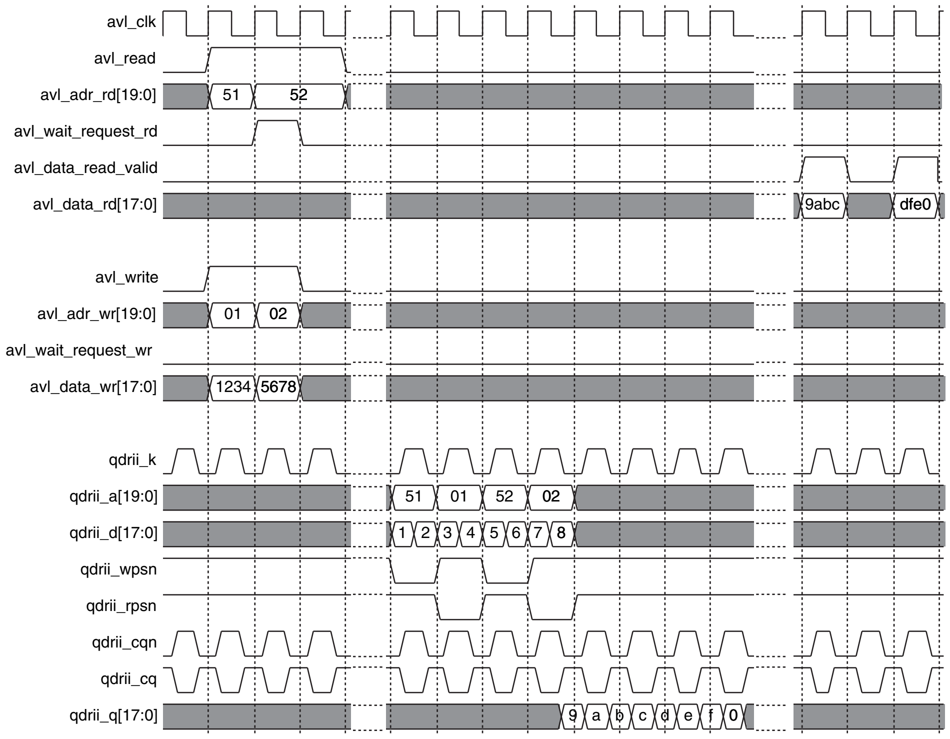
Figure 3–12. Simultaneous Read & Write—Burst of Two



Burst of Four (Wide Mode)

For the burst of four (wide mode) all the data is present in one clock cycle. Similarly to the two cycles, you must alternate the read and write commands on the QDRII SRAM interface. As a result, there is a pause when both the read and write commands arrive simultaneously on the Avalon interfaces. The first read is buffered and then the consecutive read is delayed by one clock cycle, see Figure 3–13 on page 3–22.

Figure 3–13. Simultaneous Read & Write—Burst of Four (Wide Mode)



Signals

Table 3–1 shows the system signals.

Table 3–1. System Signals (Part 1 of 2)

Signal	Direction	Description
`avl_clk`	Input	System clock derived from the PLL.
`avl_clk_wr`	Input	Write clock derived from the PLL.
`avl_resetn`	Input	Reset signal, which you can assert asynchronously, but you must deassert synchronously to `avl_clk`.
`dll_delay_ctrl[6]`	Input	Delay bus for DLL to shift DQS inputs. DQS mode only.

Table 3–1. System Signals (Part 2 of 2)

Signal	Direction	Description
`non_dqs_capture_clock`	Input	Non-DQS capture mode clock.
`training_done`	Output	Asserted when the training of the core is complete.
`training_incorrect`	Output	The core is nonfunctional. Asserted when the training reaches the maximum number of iterations but fails to adjust the pointers.
`training_pattern_not_found`	Output	The core is nonfunctional. The training must find a positive edge on the bit 0 of data. The core did not find this edge.

Table 3–2 shows the Avalon write signals.

Table 3–2. Avalon Write Signals

Signal	Width (Bits)	Direction	Description
`avl_addr_wr`	≤ 21	Input	Avalon write address.
`avl_byteen_wr`	2, 4, 8, or 16	Input	Byte enable (active low).
`avl_chipselect_wr`	1	Input	Device select for the write port.
`avl_data_wr`	18, 36, 72, 144, or 288	Input	Avalon data write from master.
`avl_write`	1	Input	Avalon write request.
`avl_wait_request_wr`	1	Output	Avalon write wait—the transaction does not occur on this cycle.

Table 3–3 shows the Avalon read signals.

Table 3–3. Avalon Read Signals (Part 1 of 2)

Signal	Width (Bits)	Direction	Description
`avl_addr_rd`	≤ 21	Input	Avalon read address.
`avl_byteen_rd`	2 to 16	Input	Byte enable (active low).
`avl_chipselect_rd`	1	Input	Device select for the read port.
`avl_read`	1	Input	Avalon read request.
`avl_data_rd`	18, 36, 72, 144, or 288	Output	Avalon read data to master.

Table 3–3. Avalon Read Signals (Part 2 of 2)

Signal	Width (Bits)	Direction	Description
`avl_datavalid_rd`	1	Output	Avalon read data valid—the data is sent concurrent to the signal.
`avl_wait_request_rd`	1	Output	Avalon read wait—the transaction does not occur on this cycle.

Table 3–4 shows the QDRII memory signals.

Table 3–4. QDRII Memory Signals

Signal	Width (Bits)	Direction	Description
`qdrii_a`	≤ 21	Output	Address bus.
`qdrii_bwsn`	≤ 8	Output	Byte enable to memory.
`qdrii_cq`	≤ 9	Input	Free running clock from memory.
`qdrii_cqn`	≤ 9	Input	Free running clock from memory.
`qdrii_d`	≤ 72	Output	Data out.
`qdrii_k`	≤ 9	Output	Free running clock to memory.
`qdrii_kn`	≤ 9	Output	Free running clock to memory.
`qdrii_q`	≤ 72	Input	Data in from memory.
`qdrii_rpsn`	≤ 8	Output	Read signal to memory. Active low and reset in the inactive state.
`qdrii_wpsn`	≤ 8	Output	Write signal to memory. Active low and reset in the inactive state.

Table 3–5 shows the datapath interface signals.

Table 3–5. Datapath Interface Signals (Part 1 of 2)

Name	Width (Bits)	Direction	Description
`clk`	–	Input	Clock.
`control_a_rd`	17:0	Input	Read address from the pipeline and resynchronization logic.
`control_a_wr`	17:0	Input	Write address from the pipeline and resynchronization logic.
`control_bwsn`	3:0	Input	Byte enable from the pipeline and resynchronization logic.
`control_rpsn`	–	Input	Read from the pipeline and resynchronization logic.
`control_wdata`	35:0	Input	Write data from the pipeline and resynchronization logic.

Table 3–5. Datapath Interface Signals (Part 2 of 2)

Name	Width (Bits)	Direction	Description
control_wpsn	–	Input	Write signal from the pipeline and resynchronization logic.
dll_delay_ctrl	5:0	Input	DLL delay control from the top-level design to shift the CQ by a nominal 90 degrees.
capture_clock	–	Output	Capture clocks (CQ into soft logic) to the pipeline and resynchronization logic.
captured_data	35:0	Output	Captured data—data after the IO to pipeline and resynchronization logic.

Table 3–6 shows the datapath.

Table 3–6. Pipeline & Resynchronization Logic Signals

Name	Width (Bits)	Direction	Description
avl_control_a_rd	17:0	Input	Read address from the control logic.
avl_control_a_wr	17:0	Input	Write address from the control logic.
avl_control_bwsn	3:0	Input	Byte enable from the control logic.
avl_control_rpsn	–	Input	Read from the control logic.
avl_control_wdata	35:0	Input	Write data from the control logic.
avl_control_wpsn	–	Input	Write from the control logic.
capture_clock	–	Input	Clocks from the datapath (CQ into soft logic).
captured_data	35:0	Input	Data captured by IO from datapath.
clk	–	Input	Clock.
reset	–	Input	Reset.
control_a_rd	17:0	Output	Read address to datapath.
control_a_wr	17:0	Output	Write address to datapath.
control_bwsn	3:0	Output	Byte enable to datapath.
control_rdata	35:0	Output	Read data after resynchronization to control logic.
control_rpsn	–	Output	Read to datapath.
control_wdata	35:0	Output	Write data to datapath.
control_wpsn	–	Output	Write to datapath.
training_done	–	Output	Initial training done to control logic.

Device-Level Configuration

This section describes the following topics:

- "PLL Configuration" on page 3–26
- "Example Design" on page 3–27
- "Constraints" on page 3–29

PLL Configuration

IP Toolbench creates up to two example PLLs in your project directory, which you can parameterize to meet your exact requirements. IP Toolbench generates the example PLLs with an input to output clock ratio of 1:1 and a clock frequency you entered in IP Toolbench. In addition IP Toolbench sets the correct phase outputs on the PLLs' clocks. You can edit the PLLs to meet your requirements with the `altpll` MegaWizard Plug-In. IP Toolbench overwrites your PLLs in your project directory unless you turn off the **Reset PLL to default setting** option.

The external clocks are generated using standard I/O pins in double data rate I/O (DDIO) mode (using the `altddio_out` megafunction). This generation matches the way in which the write data is generated and allows better control of the skew between the clock and the data to meet the timing requirements of the QDRII SRAM.

The PLL has the following outputs:

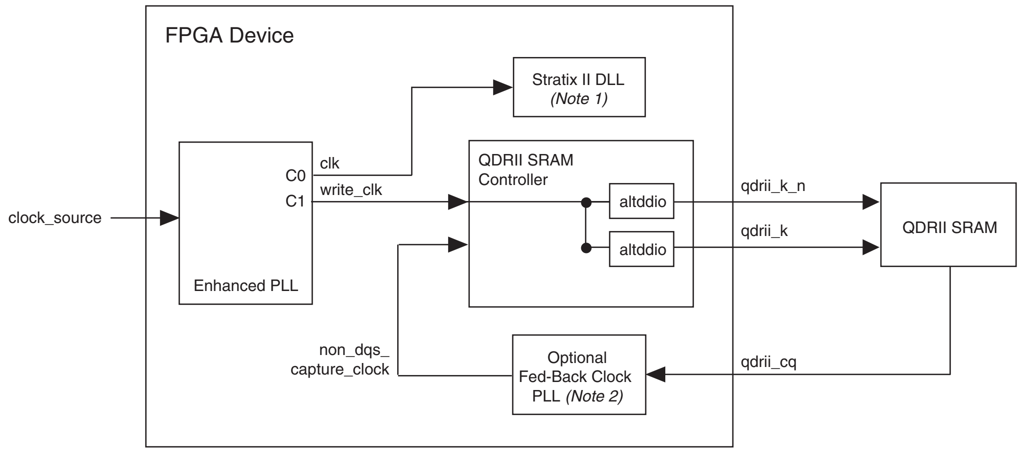
- Output c0 drives the system clock that clocks most of the controller including the state machine and the local interface.
- Output c1 drives the write clock that lags the system clock by 90°.

The recommended configuration for implementing the QDRII SRAM controller in a Stratix® series is to use a single enhanced PLL to produce all the required clock signals. No external clock buffer is required as the Altera® device can generate clock signals for the QDRII SRAM devices.

For Stratix II devices, if you turn off **DQS mode**, you enable fed-back resynchronization, which uses a fed-back clock to resynchronize the data.

Figure 3–14 on page 3–27 shows the recommended PLL configuration.

Figure 3–14. PLL Configuration



Notes to Figure 3–14:
(1) Stratix II devices only.
(2) Non-DQS mode only.

Example Design

IP Toolbench creates an example design that shows you how to instantiate and connect up the QDRII SRAM controller. The example design is a working system that can be compiled and used for both static timing checks and board tests. It also instantiates an example PLL and shows you how to generate the external clocks for the QDRII SRAM device.

The example design consists of the QDRII SRAM controller, some driver logic to issue read and write requests to the controller, and a PLL to create the necessary clocks. The asynchronous reset, `avl_resetn`, drives the reset logic, which resets the PLL and all the logic. When the PLL is locked and `avl_resetn` is deasserted, the reset to the core, `soft_reset_n`, is also deasserted. If the PLL lock is lost, the reset logic issues a reset.

Figure 3–15 on page 3–28 shows the testbench and the example design.

Figure 3–15. Testbench & Example Design

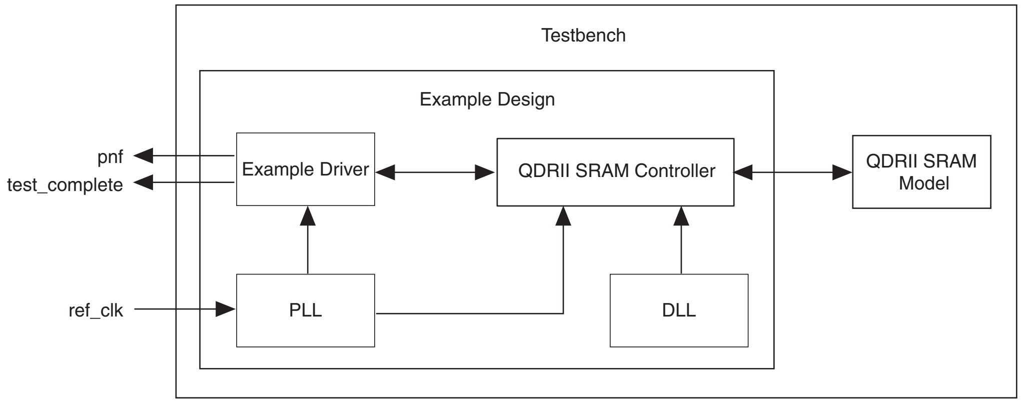


Table 3–7 describes the files that are associated with the example design and the testbench.

Table 3–7. Example Design & Testbench Files

Filename	Description
<*top-level name*>**_tb.v** or **.vhd** *(1)*	Testbench for the example design.
<*top-level name*>**.vhd** or **.v** *(1)*	Example design.
qdrii_pll_stratixii.vhd	Example PLL, which you should configure to match your frequency.
<*variation name*>**_example_driver.v** or **.vhd** *(2)*	Example driver.
<*variation name*> **.v** or **.vhd** *(2)*	QDRII SRAM controller.

Notes to Table 3–7:

(1) <*top-level name*> is the name of the Quartus® II project top-level entity.

(2) <*variation name*> is the is the name you give to the controller you create with the Megawizard.

The example driver is a self-checking test generator for the QDRII SRAM controller. It uses a state machine to write data patterns to all memory banks. It then reads back the data and checks that the data matches. If any read data fails the comparison, the fail output transitions high for one cycle and the fail permanent output transitions high and stays high.

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

The testbench instantiates a QDRII SRAM model, a reference clock for the PLL, and model for the system board memory trace delays.

Altera provides a Verilog HDL simulation model. The model is a behavioral model to verify the design but does not simulate any delays. Altera recommends that you replace the model with the specific model from your memory vendor.

For more details on how to run the simulation script, see "Simulate the Example Design" on page 2–11.

Constraints

IP Toolbench generates a constraints script, **add_constraints_for_**<*variation name*>**.tcl**, which is a set of Quartus II assignments that are required to successfully compile the example design.

When the constraints script runs, it creates another script, **remove_constraints_for_**<*variation name*>**.tcl**, which you can use to remove the constraints from your design.

The constraints script implements the following types of assignments:

- `cqn`, `cq`, and `q` capture pins placement
- Capacitance loading
- `cq` pin set to non-global signal
- I/O type for all interface pins
- Cut timing assignments for false timing paths

Parameters

The parameters can only be set in IP Toolbench (see "Step 1: Parameterize" on page 2–5).

Memory

Table 3–8 shows the memory type parameters.

Table 3–8. Memory Type Parameters

Parameter	Value	Description
Memory device	Part number	A part number for a particular memory device. Choosing an entry other than **Custom** sets many of the parameters in the wizard to the correct value for the specified part. If any such parameter is changed to a value that is not supported by the specified device, the preset automatically changes to custom. You can add your own devices to this list by editing the **memory_types.dat** file in the **\constraints** directory.
	QDRII or QDRII+	Selects QDRII or QDRII+ SRAM devices.
Clock speed	Up to 300 MHz *(1)*	The memory controller clock frequency. The constraints script and the datapath use this clock speed. It must be set to the value that you intend to use. The first time you use IP Toolbench or if you turn on **Automatically generate the PLL**, it uses this value for the IP Toolbench-generated PLL's input and output clocks .
Voltage	1.5 or 1.8 V	Memory device voltage.
Burst length	2/4	Burst length.
Data bus width	8, 9, 16, 18, 32, 36	QDRII SRAM device width.
Address bus width	15 to 23	Memory space.
Memory latency	For QDRII, 1.5; for QDRII+, 2.0 or 2.5	The memory latency. QDRII+ is bursts of four only.

Note to Table 3–8:

(1) IP Toolbench allows you to enter up to 600 MHz, but Altera only supports the QDRII SRAM controller up to 300 MHz.

Table 3–9 shows the local bus width parameter (only available with burst length of four).

Table 3–9. Local Bus Width Parameters

Parameter	Value	Description
Local bus width	Narrow mode or wide mode	Narrow mode is twice the width of the memory; wide mode is four times the width of the memory.

Table 3–10 shows the memory interface parameters.

Table 3–10. Memory Interface Parameters

Parameter	Value	Description
Device width	1 to 4	Specifies the number of devices to increase the width of the data bus.
Device depth	1 to 2	Choose 2 to double the memory space.
Use `altddio` pin	On or off	When turned on `altddio` outputs generate the clock outputs. Turn off to use dedicated PLL outputs to generate the clocks, which is recommended for HardCopy II devices.

Board & Controller

Table 3–11 shows the pipelining parameters.

Table 3–11. Pipelining Parameters

Parameter	Value	Description
Number of pipeline registers on address, command, and data outputs	0 to 4	You can choose 1, 2, or 3 pipeline registers between the memory controller and the address, command, and data outputs. These registers help to achieve the required performance at higher frequencies.
Number of pipeline registers on read data	0 to 4	You can choose 1, 2, or 3 pipeline registers between the memory controller and the read data input. These registers help to achieve the required performance at higher frequencies.

Table 3–12 shows the read latency options.

Table 3–12. Read Latency Options

Parameter	Value	Description
Manual read latency setting	On or off	Turn on if you want to choose the latency clock cycle.
Set latency to clock cycle	–2 < current clock cycle < +4	Choose the latency clock cycle. For example, if the default is 13, you can choose any value from 11 to 17. However, Altera recommends that you do not alter this parameter.

Table 3–13 shows the capture modes.

Table 3–13. Capture Modes

Parameter	Value	Description
DQS mode	On or off	Turn on for DQS capture mode (Stratix II devices only). The controller is in non-DQS mode only for Stratix devices.
Use migratable byte groups	On or off	When turned on, you can migrate the design to a migration device (Stratix II devices only). When turned off the wizard allows much greater flexibility in the placement of byte groups.

Table 3–14 shows the pin loading parameters.

Table 3–14. Pin Loading Parameters

Parameter	Range (pF)	Description
Pin loading on data pins	Any	Enter the pin loading to match your board and memory devices.
Pin loading on FPGA address and command pins	Any	Enter the pin loading to match your board and memory devices.
Pin loading on FPGA clock pins	Any	Enter the pin loading to match your board and memory devices.

Project Settings

Table 3–15 shows the example settings.

Table 3–15. Example Settings

Parameter	Description
Automatically apply QDR SRAM controller-specific constraints to the Quartus II project	When this option is turned on, the next time you compile, the Quartus II software automatically runs the add constraints script. Turn off this option if you do not want the script to run automatically.

Update the example design file that instantiates the QDRII SRAM controller variation	When this option is turned on, IP Toolbench parses and updates the example design file. It only updates sections that are between the following markers:
	`<<START MEGAWIZARD INSERT <tagname>`
	`<<END MEGAWIZARD INSERT <tagname>`
	
	If you edit the example design file, ensure that your changes are outside of the markers or remove the markers. Once you remove the markers, you must keep the file updated, because IP Toolbench can no longer update the file.

Update example design system PLL	When this option is turned on, IP Toolbench automatically overwrites the PLL.Turn off this option, if you do not want the wizard to overwrite the PLL. The first time you create a custom variation, you must turn on **Update example design system PLL**.

Table 3–16 shows the variation path parameters.

Table 3–16. Variation Path Parameters

Parameter	Description
Enable hierarchy control	The constraints script analyzes your design, to automatically extract the hierarchy to your variation. To prevent the constraints script analyzing your design, turn on **Enable hierarchy control**, and enter the correct hierarchy path to your controller.
Hierarchy path to variation	The hierarchy path is the path to your QDRII SRAM controller, minus the top-level name. The hierarchy entered in the wizard must match your design, because the constraints scripts rely on this path for correct operation.

Table 3–17 shows the pin prefixes parameter.

Table 3–17. Pin Prefixes

Parameter	Description
Prefix all QDRII SRAM pins with	This string prefixes the pin names for the FPGA pins that are connected to the QDRII SRAM controller.

MegaCore Verification

MegaCore verification involves simulation testing and hardware testing.

Simulation Environment

Altera has carried out extensive tests using industry-standard models to ensure the functionality of the QDRII SRAM controller. In addition, Altera has carried out a wide variety of gate-level tests of the QDRII SRAM controller to verify the post-compilation functionality of the controller.

Hardware Testing

Table 3–18 shows the Altera development board on which Altera hardware tested the QDRII SRAM controller.

Table 3–18. Altera Development Boards

Development Board	Altera Device	Memory Device
Stratix II Memory Demonstration Board 2	EP2S60F1020C3	Samsung 18-bit QDRII SDRAM

Additional Information

Revision History

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

Date	Version	Changes Made
May 2008	8.0	Updated the device support.
October 2007	7.2	Added compilation timing tips.
May 2007	7.1	● Updated the device support. ● Corrected burst of two timing diagram. ● Added information for new reset block in example design. ● Added new training signals and updated training group module description. ● Added extra resynchronization and pipeline logic information. ● Updated description of `wpsn` and `rpsn` signals.
March 2007	7.0	No changes.
December 2006	6.1	Updated format.

How to Contact Altera

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

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

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

Typographic Conventions

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

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