



Quartus II Version 8.0 Handbook

Volume 4: SOPC Builder



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

Copyright © 2008 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.





Chapter Revision Dates	xi
-------------------------------------	-----------

About this Handbook.....	xiii
---------------------------------	-------------

How to Contact Altera	xiii
-----------------------------	------

Typographic Conventions	xiii
-------------------------------	------

Section I. SOPC Builder Features

Chapter 1. Introduction to SOPC Builder

Quick Start Guide	1-1
-------------------------	-----

Overview	1-1
----------------	-----

Architecture of SOPC Builder Systems	1-2
--	-----

SOPC Builder Modules	1-2
----------------------------	-----

Example System	1-3
----------------------	-----

Custom Components	1-4
-------------------------	-----

Functions of SOPC Builder	1-5
---------------------------------	-----

Defining and Generating the System Hardware	1-5
---	-----

Creating a Memory Map for Software Development	1-6
--	-----

Creating a Simulation Model and Test Bench	1-6
--	-----

Operating System Support	1-6
--------------------------------	-----

Talkback Support	1-6
------------------------	-----

Referenced Documents	1-7
----------------------------	-----

Document Revision History	1-8
---------------------------------	-----

Chapter 2. System Interconnect Fabric for Memory-Mapped Interfaces

Introduction	2-1
--------------------	-----

High-Level Description	2-1
------------------------------	-----

Fundamentals of Implementation	2-4
--------------------------------------	-----

Functions of System Interconnect Fabric	2-4
---	-----

Address Decoding	2-5
------------------------	-----

Datapath Multiplexing	2-6
-----------------------------	-----

Wait-State Insertion	2-7
----------------------------	-----

Pipeline Read Transfers	2-7
-------------------------------	-----

Native Address Alignment and Dynamic Bus Sizing	2-8
---	-----

Dynamic Bus Sizing	2-9
--------------------------	-----

Wider Master	2-10
--------------------	------

Narrower Master	2-10
-----------------------	------

Arbitration for Multimaster Systems	2-11
---	------

Traditional Shared Bus Architectures	2-11
Slave-Side Arbitration	2-12
Arbiter Details	2-13
Arbitration Rules	2-14
Setting Arbitration Parameters in SOPC Builder	2-15
Fairness-Based Shares	2-15
Round-Robin Scheduling	2-16
Burst Transfers	2-16
Minimum Share Value	2-16
Burst Adapters	2-17
Interrupts	2-18
Individual Requests IRQ Scheme	2-18
Priority Encoded Interrupt Scheme	2-19
Assigning IRQs in SOPC Builder	2-19
Reset Distribution	2-20
Referenced Documents	2-20
Document Revision History	2-21

Chapter 3. System Interconnect Fabric for Streaming Interfaces

Introduction	3-1
High-Level Description	3-1
Avalon Streaming and Avalon Memory-Mapped Interfaces	3-2
Adapters	3-3
Data Format Adapter	3-4
Timing Adapter	3-5
Channel Adapter	3-5
Multiplexer Examples	3-5
Example to Double Clock Frequency	3-5
Example to Double Data Width and Maintain Frequency	3-6
Example to Boost the Frequency	3-6
Referenced Documents	3-7
Document Revision History	3-8

Chapter 4. SOPC Builder Components

Introduction	4-1
Component Providers	4-1
Component Hardware Structure	4-2
Components Inside the SOPC Builder System	4-3
Components That Interface to Logic Outside the SOPC Builder System	4-4
Exported Connection Points	4-4
Selecting Components in SOPC Builder	4-5
Component Structure	4-5
Component Description File (_hw.tcl)	4-6
Component File Organization	4-6
Using Classic Components in SOPC Builder	4-7
Referenced Document	4-7
Document Revision History	4-8

Chapter 5. Using SOPC Builder with the Quartus II Software

Introduction	5-1
Quartus IP File	5-1
Quartus II Incremental Compilation	5-2
TimeQuest Timing Analyzer	5-2
Analyzing PLLs	5-3
Analyzing Slow Asynchronous I/O Paths	5-5
Analyzing Single Data Rate SDRAM and SSRAM	5-5
Analyzing Tri-state Bridge and Asynchronous Devices	5-8
Analyzing DDR and DDR2 Memories	5-9
Referenced Documents	5-10
Document Revision History	5-10

Chapter 6. Component Editor

Introduction	6-1
Component Hardware Structure	6-2
Starting the Component Editor	6-2
HDL Files Tab	6-3
Signals Tab	6-4
Naming Signals for Automatic Type and Interface Recognition	6-4
Templates for Interfaces to External Logic	6-6
Interfaces Tab	6-6
Component Wizard Tab	6-6
Identifying Information	6-6
Parameters	6-7
Saving a Component	6-7
Editing a Component	6-8
Software Assignments	6-8
Component GUI	6-8
Referenced Documents	6-8
Document Revision History	6-9

Chapter 7. Component Interface Tcl Reference

Overview	7-1
Information in a Hardware Tcl File	7-1
Component Phases	7-2
Writing a Hardware Tcl File	7-2
Providing Basic Information	7-2
Declaring Parameters	7-3
Declaring Interfaces	7-3
Adding Files and Guiding Generation	7-4
Default Behavior	7-5
Edit Time Behavior	7-5
Validation Time Behavior	7-6
Elaboration Behavior	7-6
Generation Behavior	7-7
Overriding Default Behaviors	7-7

Edit Callback	7-7
Validation Callback	7-8
Elaboration Callback	7-10
Generation Callback	7-11
Hardware Tcl Command Reference	7-12
Module Definition	7-13
get_module_properties	7-13
get_module_property	7-15
set_module_property	7-15
add_file	7-16
get_files	7-16
get_file_property	7-17
set_file_property	7-17
send_message	7-17
Parameters	7-19
add_parameter	7-19
get_parameters	7-19
get_parameter_properties	7-19
get_parameter_property	7-21
set_parameter_property	7-21
get_parameter_value	7-21
set_parameter_value	7-22
Interfaces and Ports	7-23
get_interfaces	7-23
get_interface_properties	7-24
get_interface_property	7-24
set_interface_property	7-24
add_interface_port	7-25
get_interface_ports	7-25
get_port_property	7-26
set_port_property	7-27
Generation	7-27
get_generation_property	7-28
get_project_property	7-28
Referenced Documents	7-29
Document Revision History	7-30

Chapter 8. Archiving SOPC Builder Projects

Introduction	8-1
Limitations	8-1
Required Files	8-2
SOPC Builder Design Files	8-2
Referenced Documents	8-3
Document Revision History	8-4

Section II. Building Systems with SOPC Builder

Chapter 9. SOPC Builder Memory Subsystem Development Walkthrough

Introduction	9-1
Example Design	9-2
Example Design Structure	9-2
Example Design Starting Point	9-4
Hardware and Software Requirements	9-4
Design Flow	9-5
Component-Level Design in SOPC Builder	9-5
SOPC Builder System-Level Design	9-6
Simulation	9-6
Quartus II Project-Level Design	9-6
Board-Level Design	9-6
Simulation Considerations	9-6
Generic Memory Models	9-7
Vendor-Specific Memory Models	9-7
On-Chip RAM and ROM	9-7
Component-Level Design for On-Chip Memory	9-8
Memory Type	9-8
Size	9-8
Read Latency	9-8
Non-Default Memory Initialization	9-9
Enable In-System Memory Content Editor Feature	9-9
SOPC Builder System-Level Design for On-Chip Memory	9-9
Simulation for On-Chip Memory	9-9
Quartus II Project-Level Design for On-Chip Memory	9-9
Board-Level Design for On-Chip Memory	9-10
Example Design with On-Chip Memory	9-10
EPCS Serial Configuration Device	9-11
Component-Level Design for an EPCS Device	9-11
SOPC Builder System-Level Design for an EPCS Device	9-11
Simulation for an EPCS Device	9-12
Quartus II Project-Level Design for an EPCS Device	9-12
Board-Level Design for an EPCS Device	9-12
Example Design with an EPCS Device	9-12
(SDR) SDRAM	9-13
Component-Level Design for SDRAM	9-14
SOPC Builder System-Level Design for SDRAM	9-14
Simulation for SDRAM	9-14
Quartus II Project-Level Design for SDRAM	9-14
Connecting and Assigning the SDRAM-Related Pins	9-15
Accommodating Clock Skew	9-15
Board-Level Design for SDRAM	9-15
Example Design with (SDR) SDRAM	9-15
(DDR) SDRAM	9-17
(DDR2) SDRAM	9-17

Parallel Flash Loader	9–18
Off-Chip SRAM and Flash Memory	9–18
Component-Level Design for SRAM and Flash Memory	9–19
Avalon-MM Tristate Bridge	9–20
Flash Memory	9–20
SRAM	9–21
SOPC Builder System-Level Design for SRAM and Flash Memory	9–21
Simulation for SRAM and Flash Memory	9–22
Quartus II Project-Level Design for SRAM and Flash Memory	9–22
Board-Level Design for SRAM and Flash Memory	9–23
Aligning the Least-Significant Address Bits	9–23
Aligning the Most-Significant Address Bits	9–24
Example Design with SRAM and Flash Memory	9–25
Adding the Avalon-MM Tristate Bridge	9–25
Adding the Flash Memory Interface	9–25
Adding the SRAM Interface	9–25
SOPC Builder System Contents Tab	9–26
Connecting and Assigning Pins in the Quartus II Project	9–27
Connecting FPGA Pins to Devices on the Board	9–27
Referenced Documents	9–29
Document Revision History	9–29

Chapter 10. SOPC Builder Component Development Walkthrough

Introduction	10–1
SOPC Builder Components and the Component Editor	10–1
Prerequisites	10–2
Hardware and Software Requirements	10–2
Component Development Flow	10–2
Typical Design Steps	10–2
Hardware Design	10–4
Design Example: Checksum Hardware Accelerator	10–4
Software Design	10–7
Verifying the Component	10–8
System Console	10–8
System-Level Verification	10–8
Sharing Components	10–9
.sopcinfo Files	10–10
Referenced Documents	10–10
Document Revision History	10–11

Section III. Interconnect Components

Chapter 11. Avalon Memory-Mapped Bridges

Introduction to Bridges	11–1
Structure of a Bridge	11–1

Reasons for Using a Bridge	11-2
Address Mapping for Systems with Avalon-MM Bridges	11-7
Tools for Visualizing the Address Map	11-8
Differences between Avalon-MM Bridges and Avalon-MM Tristate Bridges	11-8
Avalon-MM Pipeline Bridge	11-8
Component Overview	11-9
Functional Description	11-9
Interfaces	11-10
Pipeline Stages and Effects on Latency	11-10
Burst Support	11-11
Example System with Avalon-MM Pipeline Bridges	11-11
Clock Crossing Bridge	11-13
Component Overview	11-13
Choosing Clock Crossing Methodology	11-13
Functional Description	11-14
Interfaces	11-14
Clock Domain Adapter and FIFOs	11-14
Burst Support	11-15
Example System with Avalon-MM Clock-Crossing Bridges	11-16
Instantiating the Avalon-MM Clock-Crossing Bridge in SOPC Builder	11-18
Clock Domain Crossing	11-19
Description of Clock Domain Adapter	11-19
Location of Clock Domain Adapter	11-21
Duration of Transfers Crossing Clock Domains	11-21
Implementing Multiple Clock Domains in SOPC Builder	11-22
Device Support	11-23
Installation and Licensing	11-23
Hardware Simulation Considerations	11-24
Software Programming Model	11-24
Referenced Documents	11-24
Document Revision History	11-24

Chapter 12. Avalon Streaming Interconnect Components

Introduction to Interconnect Components	12-1
Interconnect Component Usage	12-1
Address Mapping	12-2
Timing Adapter	12-4
Resource Usage and Performance	12-4
Instantiating the Timing Adapter in SOPC Builder	12-6
Input Interface Parameters	12-6
Output Interface Parameters	12-6
Common to Input and Output Interfaces	12-6
Channel Signal Width (Bits)	12-6
Max Channel	12-6
Bits Per Symbol	12-6
Symbols Per Beat	12-6
Include Packet Support	12-7

Error Signal Width (Bits)	12-7
Data Format Adapter	12-7
Resource Usage and Performance	12-7
Instantiating the Data Format Adapter in SOPC Builder	12-9
Input Interface Parameters	12-9
Data Symbols Per Beat	12-9
Output Interface Parameters	12-9
Data Symbols Per Beat	12-9
Common to Input and Output	12-9
Support Backpressure with the Ready Signal	12-9
Data Bits Per Symbol	12-9
Channel Signal Width (Bits)	12-9
Max Channel	12-9
Include Packet Support	12-10
Error Signal Width (Bits)	12-10
Channel Adapter	12-10
Resource Usage and Performance	12-10
Instantiating the Channel Adapter in SOPC Builder	12-11
Input Interface Parameters	12-11
Channel Signal Width (Bits)	12-11
Max Channel	12-11
Output Interface Parameters	12-11
Channel Signal Width (Bits)	12-11
Max Channel	12-11
Common to Input and Output Interfaces	12-11
Data Bits Per Symbol	12-11
Symbols Per Beat	12-12
Include Packet Support	12-12
Error Signal Width (Bits)	12-12
Installation and Licensing	12-12
Hardware Simulation Considerations	12-12
Software Programming Model	12-12
Referenced Documents	12-12
Document Revision History	12-13



Chapter Revision Dates

The chapters in this book, *Quartus II Handbook, Volume 4*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1. Introduction to SOPC Builder
Revised: May 2008
Part number: QII54001-8.0.0

Chapter 2. System Interconnect Fabric for Memory-Mapped Interfaces
Revised: May 2008
Part number: QII54003-8.0.0

Chapter 3. System Interconnect Fabric for Streaming Interfaces
Revised: May 2008
Part number: QII54019-8.0.0

Chapter 4. SOPC Builder Components
Revised: May 2008
Part number: QII54004-8.0.0

Chapter 5. Using SOPC Builder with the Quartus II Software
Revised: May 2008
Part number: QII54023-8.0.0

Chapter 6. Component Editor
Revised: May 2008
Part number: QII54005-8.0.0

Chapter 7. Component Interface Tcl Reference
Revised: May 2008
Part number: QII54022-8.0.0

Chapter 8. Archiving SOPC Builder Projects
Revised: May 2008
Part number: QII54017-8.0.0

Chapter 9. SOPC Builder Memory Subsystem Development Walkthrough
Revised: May 2008
Part number: QII54006-8.0.0

Chapter 10. SOPC Builder Component Development Walkthrough

Revised: *May 2008*Part number: *QII54007-8.0.0*

Chapter 11. Avalon Memory-Mapped Bridges

Revised: *May 2008*Part number: *QII54020-8.0.0*

Chapter 12. Avalon Streaming Interconnect Components

Revised: *May 2008*Part number: *QII54021-8.0.0*



About this Handbook

This handbook provides comprehensive information about the Altera® SOPC Builder tool.

How to Contact Altera





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

Information Type	USA and Canada
Technical support	www.altera.com/mysupport/
Technical training	www.altera.com/training/custrain@altera.com
Product literature	www.altera.com/literature
Altera literature services	literature@altera.com
FTP site	ftp.altera.com

Typographic Conventions

This document uses the typographic conventions shown below.

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, filenames, filename extensions, and software utility names are shown in bold type. Examples: f_{MAX} , lqdesigns directory, d: drive, chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75: High-Speed Board Design</i> .
<i>Italic type</i>	Internal timing parameters and variables are shown in italic type. Examples: <i>t_{PIA}</i> , <i>n + 1</i> . 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."

Visual Cue	Meaning
Courier type	Signal and port names are shown in lowercase Courier type. Examples: <code>data1</code> , <code>tdi</code> , <code>input</code> . Active-low signals are denoted by suffix <code>n</code> , e.g., <code>resetsn</code> . Anything that must be typed exactly as it appears is shown in Courier type. For example: <code>c:\qdesigns\tutorial\chiptrip.gdf</code> . Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword <code>SUBDESIGN</code>), as well as logic function names (e.g., <code>TRI</code>) 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.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
	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.

Section I of this volume introduces the SOPC Builder system integration tool. Chapters in this section answer the following questions:

- What is SOPC Builder?
- What features does SOPC Builder provide?

This section includes the following chapters:

- Chapter 1, Introduction to SOPC Builder
- Chapter 2, System Interconnect Fabric for Memory-Mapped Interfaces
- Chapter 3, System Interconnect Fabric for Streaming Interfaces
- Chapter 4, SOPC Builder Components
- Chapter 5, Using SOPC Builder with the Quartus II Software
- Chapter 6, Component Editor
- Chapter 7, Component Interface Tcl Reference
- Chapter 8, Archiving SOPC Builder Projects



For information about the revision history for chapters in this section, refer to each individual chapter's revision history.

Quick Start Guide

For a quick introduction on how to use SOPC Builder, follow these general steps:

- Install the Quartus® II software, which includes SOPC Builder. This is available at www.altera.com.
- Download and run the checksum sample design described in the *SOPC Builder Component Development Walkthrough* chapter in volume 4 of the *Quartus II Handbook*.
- Study the introduction to SOPC Builder in the *Nios II Hardware Development Tutorial*.

Overview

SOPC Builder is a powerful system development tool. SOPC Builder enables you to define and generate a complete system-on-a-programmable-chip (SOPC) in much less time than using traditional, manual integration methods. SOPC Builder is included as part of the Quartus II software.

You may have used SOPC Builder to create systems based on the Nios® II processor. However, SOPC Builder is more than a Nios II system builder; it is a general-purpose tool for creating systems that may or may not contain a processor.

SOPC Builder automates the task of integrating hardware components. Using traditional design methods, you must manually write HDL modules to wire together the pieces of the system. Using SOPC Builder, you specify the system components in a GUI, and SOPC Builder generates the interconnect logic automatically. SOPC Builder outputs HDL files that define all components of the system, and a top-level HDL file that connects all the components together. SOPC Builder generates either Verilog HDL or VHDL equally.

In addition to its role as a system generation tool, SOPC Builder provides features to ease writing software and to accelerate system simulation. This chapter includes the following sections:

- “Architecture of SOPC Builder Systems” on page 1–2
- “Functions of SOPC Builder” on page 1–5
- “Operating System Support” on page 1–6
- “Talkback Support” on page 1–6

Architecture of SOPC Builder Systems

An SOPC Builder component is a design module that SOPC Builder recognizes and can automatically integrate into a system. You can also define and add custom components. SOPC Builder connects multiple modules together to create a top-level HDL file called the SOPC Builder system. SOPC Builder generates system interconnect fabric that contains logic to manage the connectivity of all modules in the system.

SOPC Builder Modules



This document refers to "components" as the class definition for a module, while "module" is the instance of the component class.

SOPC Builder modules are the building blocks for creating an SOPC Builder system. SOPC Builder modules use Avalon® interfaces, such as memory-mapped, streaming, and IRQ, for the physical connection of components. You can use SOPC Builder to connect any logical device (either on-chip or off-chip) that has an Avalon interface. There are different types of Avalon interfaces, as described in the *Avalon Interface Specifications*.

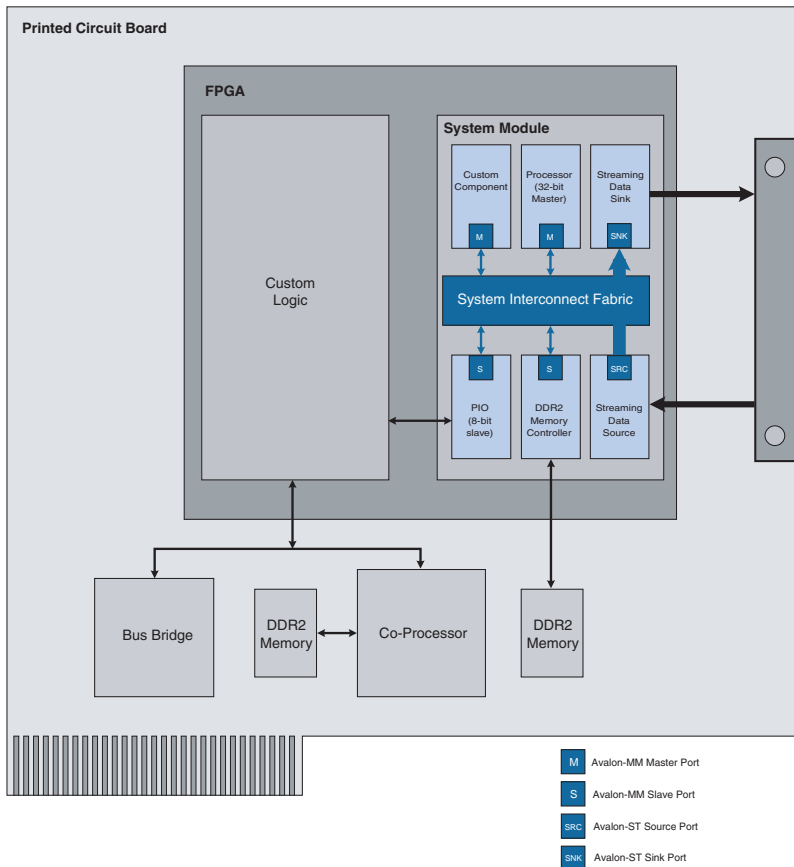


For details on the Avalon-MM interface and the Avalon-ST interface, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface protocol, refer to *Avalon Interface Specifications*.

Example System

Figure 1–1 shows an FPGA design that includes an SOPC Builder system and custom logic modules. You can integrate custom logic inside or outside the SOPC Builder system. In this example, the custom component inside the SOPC Builder system is an SOPC Builder component that communicates with other modules through an Avalon-MM master interface. The custom logic outside of the SOPC Builder system is connected to the SOPC Builder system through a PIO interface. The SOPC Builder system includes two SOPC Builder components with Avalon-ST source and sink interfaces. The system interconnect fabric connects all of the SOPC Builder components using the Avalon-MM or Avalon-ST system interconnect as appropriate.

Figure 1–1. Example of an FPGA with a SOPC Builder System Generated by SOPC Builder



A component can be a logical device that is entirely contained within the SOPC Builder system, such as the processor component shown in [Figure 1-1](#). Alternately, a component can act as an interface to an off-chip device, such as the DDR2 interface component in [Figure 1-1](#). In addition to the Avalon interface, a component can have other signals that connect to logic outside the SOPC Builder system. Non-Avalon signals can provide a special-purpose interface to the SOPC Builder system, such as the PIO in [Figure 1-1](#). You can instantiate more than one component in a system. These non-Avalon signals are described in [Conduit Interface chapter in the Avalon Interface Specifications](#).

Altera and third-party developers provide ready-to-use SOPC Builder components, including:

- Microprocessors, such as the Nios II processor
- Microcontroller peripherals, such as a scatter-gather DMA controller and timer
- Serial communication interfaces, such as a UART and a serial peripheral interface (SPI)
- General purpose I/O
- Communications peripherals, such as a 10/100/1000 Ethernet MAC
- Interfaces to off-chip devices

Custom Components

You can import HDL modules and entities that you write using Verilog HDL or VHDL into SOPC builder as custom components. You use the following design flow to integrate custom logic into an SOPC Builder system:

1. Determine the interfaces used to interact with your custom component.
2. Create the component logic using either Verilog HDL or VHDL.
3. Use the SOPC Builder component editor to create an SOPC Builder component with your HDL files.
4. Instantiate your component in the system.

Once you have created an SOPC Builder component, you can use the component in other SOPC Builder systems, and share the component with other design teams.



For instructions on developing a custom SOPC Builder component, the details about the file structure of a component, or the component editor, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.



For further details, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* and *System Interconnect Fabric for Streaming Interfaces* chapters in volume 4 of the *Quartus II Handbook*.

Functions of SOPC Builder

This section describes the functions of SOPC Builder.

Defining and Generating the System Hardware

SOPC Builder allows you to design the structure of a hardware system. The GUI allows you to add components to a system, configure the components, and specify how they connect.

After you add and parameterize components, SOPC Builder generates the system interconnect fabric, and outputs HDL files. During system generation, SOPC Builder creates the following items:

- An HDL file for the top-level SOPC Builder system and for each component in the system
- Synopsis Design Constraints (.sdc) and .sopcinfo files
- A Block Symbol File (.bsf) representation of the top-level SOPC Builder system for use in Quartus II Block Diagram Files (.bdf)
- Memory-map header file and component drivers for the Nios II processor tool chain



The header file is generated if you have the Nios II IDE in your system. Otherwise, the Nios II IDE generates a **system.h** file that calls the SOPC Builder.

- Functional test bench for the SOPC Builder system and ModelSim® simulation project files

After you generate the SOPC Builder system, you can compile it with the Quartus II software, or you can instantiate it in a larger FPGA design. An .sopcinfo file describes all of the components and connections in your system. This file is a complete system description, and is used by downstream tools such as the Nios II tool chain.

Creating a Memory Map for Software Development

When your SOPC Builder system includes a Nios II processor, SOPC Builder generates a header file that provides the address of each Avalon-MM slave component. In addition, each slave component can provide software drivers and other software functions and libraries for the processor.

For more details about how to provide Nios II software drivers for components, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. The Nios II EDS is separate from SOPC Builder, but it uses the output of SOPC Builder as the foundation for software development.

Creating a Simulation Model and Test Bench

You can simulate your systems after generating the system with SOPC Builder. During system generation, SOPC Builder outputs a simulation test bench and a ModelSim setup script that eases the system simulation effort. The test bench does the following:

- Instantiates the SOPC Builder system
- Drives all clocks and resets
- Instantiates simulation models for off-chip devices when available

Operating System Support

SOPC Builder supports the following operating systems:

- Windows 2000
- Windows XP (32- and 64-bit)
- Windows Vista (Business)
- SUSE 9 (32- and 64-bit)
- RedHAT Linux v3.0 (32- and 64-bit)
- RedHAT Linux v4.0 (32- and 64-bit)
- RedHAT Linux v5.0 (32- and 64-bit) (Beta)

Talkback Support

Talkback is a Quartus II software feature that provides feedback to Altera on tool and IP feature usage. Altera uses the data to help guide future product planning efforts.

The default when installing the Quartus II software is for Talkback to be enabled. You can disable Talkback if you do not wish to share your tool usage data with Altera.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- *Conduit Interface* chapter in the *Avalon Interface Specification*
- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Hardware Development Tutorial*
- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook*
- *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 1–1 shows the revision history for this chapter.

Table 1–1. Document Revision History		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	<ul style="list-style-type: none"> Updated references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications. Add Quick Start Guide. Add list of OS support. 	The two specifications have been combined into one for all Avalon interfaces.
October 2007, v7.2.0	<ul style="list-style-type: none"> Updated with new 7.2 functionality and terminology. Deleted unneeded description of SOPC Builder Ready Components. 	—
May 2007, v7.1.0	<ul style="list-style-type: none"> Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon switch fabric” term to “system interconnect fabric.” Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.” Added new information on Avalon Streaming (Avalon-ST) interface. Revised SOPC Builder system block diagram Added Referenced Documents section. 	This chapter was revised to introduce the Avalon streaming interface in addition to the Avalon Memory-Mapped interface. The block diagram was made more comprehensive.
March 2007, v7.0.0	No change from previous release	—
November 2007, v6.1.0	No change from previous release.	—
May 2006, v6.0.0	No change from previous release.	—
October 2005, v5.1.0	No change from previous release.	—
May 2005, v5.0.0	No change from previous release.	—
February 2005, v1.0	Initial release.	—

Introduction

The system interconnect fabric for memory-mapped interfaces is a high-bandwidth interconnect structure for connecting components that use the Avalon® Memory-Mapped (Avalon-MM) interface. The system interconnect fabric consumes minimal logic resources and provides greater flexibility than a typical shared system bus. This is a cross-connect fabric and not a tristated or time domain multiplexed bus. This chapter describes the functions of system interconnect fabric for memory-mapped interfaces and the implementation of those functions.

High-Level Description

The system interconnect fabric is the collection of interconnect and logic resources that connects Avalon-MM master and slaves on components in a system. SOPC Builder generates the system interconnect fabric to match the needs of the components in a system. The system interconnect fabric implements the connection details of a system. It guarantees that signals are routed correctly between master and slaves, as long as the ports adhere to the rules of the *Avalon Interface Specifications*. This chapter provides information on the following topics:

- “Address Decoding” on page 2-5
- “Datapath Multiplexing” on page 2-6
- “Wait-State Insertion” on page 2-7
- “Pipeline Read Transfers” on page 2-7
- “Native Address Alignment and Dynamic Bus Sizing” on page 2-8
- “Arbitration for Multimaster Systems” on page 2-11
- “Burst Adapters” on page 2-17
- “Interrupts” on page 2-18
- “Reset Distribution” on page 2-20



For details about the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

System interconnect fabric for memory-mapped interfaces supports the following items:

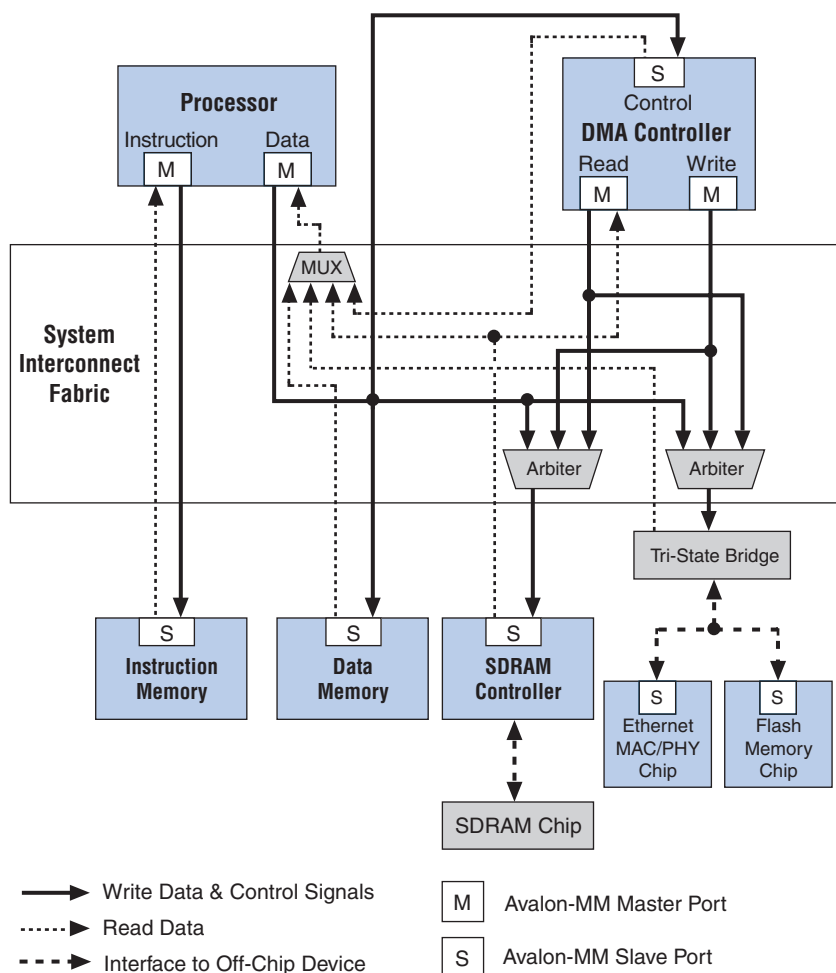
- Any number of master and slave components. The master-to-slave relationship can be one-to-one, one-to-many, many-to-one, or many-to-many.
- On-chip components.
- Interfaces to off-chip devices.

- Master and slaves of different data widths.
- Components operating in different clock domains.
- Components using multiple Avalon-MM ports.

Figure 2–1 shows a simplified diagram of the system interconnect fabric in an example memory-mapped system with multiple masters.



All figures in this chapter are simplified to show only the particular function being discussed. In a complete system, the system interconnect fabric might alter the address, data, and control paths beyond what is shown in any one particular figure.

Figure 2–1. System Interconnect Fabric—Example System

SOPC Builder supports components with multiple Avalon-MM interfaces, such as the processor component shown in [Figure 2–1](#). Because SOPC Builder can create system interconnect fabric to connect components with multiple interfaces, you can create complex interfaces that provide more functionality than a single Avalon-MM interface. For example, you can create a component with two different Avalon-MM slaves, each with an associated interrupt interface.

System interconnect fabric can connect any combination of components, as long as each interface conforms to the Avalon Interface Specifications. It can, for example, connect a system comprised of only two components with unidirectional dataflow between them. Avalon-MM interfaces are suitable for random address transactions, such as to memories or embedded peripherals.

Generating system interconnect fabric is SOPC Builder's primary purpose. In most cases, you are not required to modify the generated HDL; however, a basic understanding of how HDL works can help you optimize your systems. For example, knowledge of the arbitration algorithm can help designers of multimaster systems minimize the impact of arbitration on the system throughput.

Fundamentals of Implementation

System interconnect fabric for memory-mapped interfaces implements a switched interconnect structure that provides concurrent paths between master and slaves. System interconnect fabric consists of synchronous logic and routing resources inside the FPGA.

For each component interface, system interconnect fabric manages Avalon-MM transfers, interacting with signals on the connected component. The signals on masters and slaves can be different. In the path between master and slaves, the system interconnect fabric might introduce registers for timing synchronization, finite state machines for event sequencing, or nothing at all, depending on the services required by the specific interfaces.



For more information, refer to the *Avalon Memory-Mapped Design Optimizations* chapter in the *Embedded Design Handbook*.

Functions of System Interconnect Fabric

System interconnect fabric logic provides the following functions:

- "Address Decoding" on page 2-5
- "Datapath Multiplexing" on page 2-6
- "Wait-State Insertion" on page 2-7
- "Pipeline Read Transfers" on page 2-7
- "Arbitration for Multimaster Systems" on page 2-11
- "Burst Adapters" on page 2-17
- "Interrupts" on page 2-18
- "Reset Distribution" on page 2-20

The behavior of these functions in a specific SOPC Builder system depends on the design of the components in the system and the settings made in SOPC Builder. The remaining sections of this chapter describe how SOPC Builder implements each function.

Address Decoding

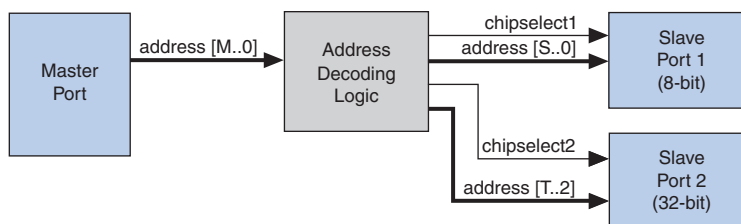
Address decoding logic in the system interconnect fabric distributes an appropriate address and produces a `chipselect` signal for each slave. Address decoding logic simplifies component design in the following ways:

- The system interconnect fabric selects a slave whenever it is being addressed by a master. Slave components do not need to decode the address to determine when they are selected.
- Slave addresses are properly aligned to the slave interface.
- Changing the system memory map does not involve manually editing HDL.

Figure 2–2 shows a block diagram of the address-decoding logic for one master and two slaves. Separate address-decoding logic is generated for every master in a system.

As shown in Figure 2–2, the address decoding logic handles the difference between the master address width (M) and the individual slave address widths (S and T). It also maps only the necessary master address bits to access words in each slave's address space.

Figure 2–2. Block Diagram of Address Decoding Logic



In SOPC Builder, the user-configurable aspects of address decoding logic are controlled by the **Base** setting in the list of active components on the **System Contents** tab, as shown in Figure 2–3.

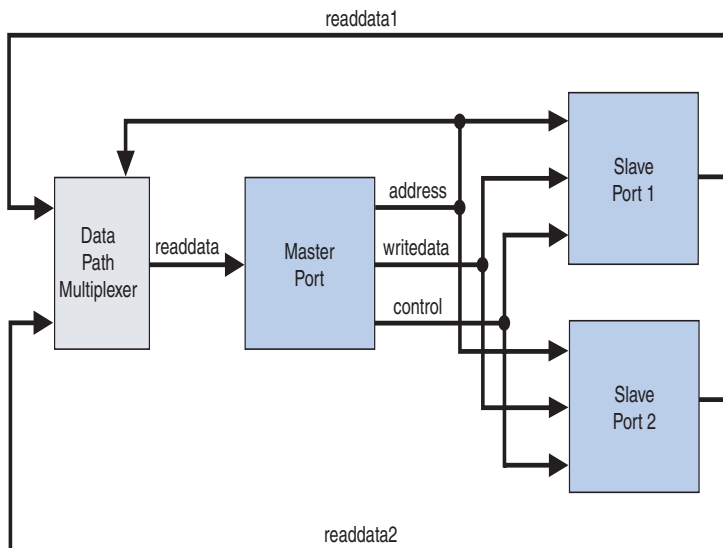
Figure 2–3. Base Settings in SOPC Builder Control Address Decoding

Module Name	Description	Base	End	IRQ
<input type="checkbox"/> cpu	Nios II Proces...			
instruction_master	Master port			
data_master	Master port			
jtag_debug_mod...	Slave port	0x02120000	0x021207FF	← IRQ 31
<input checked="" type="checkbox"/> ext_flash	Flash Memory...	0x00000000	0x007FFFFFFF	
<input checked="" type="checkbox"/> ext_ram	IDT71V416 S...	0x02000000	0x020FFFFFFF	
<input checked="" type="checkbox"/> ext_ram_bus	Avalon Tri-St...			
<input checked="" type="checkbox"/> button_pio	PIO (Parallel I/O)	0x02120860	0x0212086F	2
<input checked="" type="checkbox"/> high_res_timer	Interval timer	0x02120820	0x0212083F	3

Datapath Multiplexing

Datapath multiplexing logic in the system interconnect fabric drives the `writedata` signal from the granted master to the selected slave, and the `readdata` signal from the selected slave back to the requesting master.

Figure 2–4 shows a block diagram of the datapath multiplexing logic for one master and two slaves. SOPC Builder generates separate datapath multiplexing logic for every master in the system.

Figure 2–4. Block Diagram of Datapath Multiplexing Logic

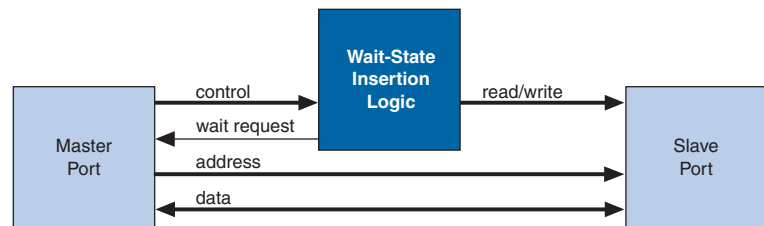
In SOPC Builder, the generation of datapath multiplexing logic is specified using the connections panel on the **System Contents** tab.

Wait-State Insertion

Wait states extend the duration of a transfer by one or more cycles. Wait-state insertion logic accommodates the timing needs of each slave, and causes the master to wait until the slave can proceed. System interconnect fabric inserts wait states into a transfer when the target slave cannot respond in a single clock cycle. System interconnect fabric also inserts wait states in cases when slave `read-enable` and `write-enable` signals have setup or hold time requirements.

Wait-state insertion logic is a small finite-state machine that translates control signal sequencing between the slave side and the master side. [Figure 2–5](#) shows a block diagram of the wait-state insertion logic between one master and one slave.

Figure 2–5. Block Diagram of Wait-State Insertion Logic



System interconnect fabric can force a master to wait for several reasons in addition to the wait state needs of a slave. For example, arbitration logic in a multimaster system can force a master to wait until it is granted access to a slave.

SOPC Builder generates wait-state insertion logic based on the properties of all slaves in the system.

Pipeline Read Transfers

The Avalon-MM interface supports pipelined read transfers, allowing a pipelined master to start multiple read transfers in succession without waiting for the prior transfers to complete. Pipelined transfers allow master-slave pairs to achieve higher throughput, even though the slave might require one or more cycles of latency to return data for each transfer.

SOPC Builder generates system interconnect fabric with pipeline management logic to take advantage of pipelined components wherever possible, based on the pipeline properties of each master-slave pair in the

system. Regardless of the pipeline latency of a target slave, SOPC Builder guarantees that read data arrives at each master in the order requested. Because master and slaves often have mismatched pipeline latency, system interconnect fabric often contains logic to reconcile the differences. Many cases of pipeline latency are possible, as shown in [Table 2–1](#).

<i>Table 2–1. Various Cases of Pipeline Latency in a Master-Slave Pair</i>		
Master	Slave	Pipeline Management Logic Structure
No Pipeline	No Pipeline	The system interconnect fabric does not instantiate logic to handle pipeline latency.
No Pipeline	Pipelined with Fixed or Variable Latency	The system interconnect fabric forces the master to wait through any slave-side latency cycles. This master-slave pair gains no benefits of pipelining, because the master waits for each transfer to complete before beginning a new transfer. However, while the master is waiting, the slave can accept transfers from a different master.
Pipelined	No Pipeline	The system interconnect fabric carries out the transfer as if neither master nor slave were pipelined, causing the master to wait until the slave returns data.
Pipelined	Pipelined with Fixed Latency	The system interconnect fabric allows the master to capture data at the exact clock cycle when data from the slave is valid. This enables the master-slave pair to achieve maximum throughput performance.
Pipelined	Pipelined with Variable Latency	This is the most simple pipelined case, in which the slave asserts a signal when its <code>readdata</code> is valid, and the master captures the data. This case enables this master-slave pair to achieve maximum throughput.

SOPC Builder generates logic to handle pipeline latency based on the properties of the master and slaves in the system. When configuring a system in SOPC Builder, there are no settings that directly control the pipeline management logic in the system interconnect fabric.

Native Address Alignment and Dynamic Bus Sizing

SOPC Builder generates system interconnect fabric to accommodate master and slaves with unmatched data widths. Address alignment affects how slave data is aligned in a master's address space, in the case that the master and slave data widths are different. Address alignment is a property of each slave, and can be different for each slave in a system. A slave can declare itself to use one of the following:

- Native address alignment
- Dynamic bus sizing

Table 2–2 demonstrates native address alignment and dynamic bus sizing for a 32-bit master connected to a 16-bit slave (a 2:1 ratio). In this example, the slave is mapped to base address *BASE* in the master's address space. In Table 2–2, *OFFSET* refers to the offset into the 16-bit slave address space.

Table 2–2. 32-Bit Master View of 16-Bit Slave Data		
32-bit Master Address	Data with Native Alignment	Data with Dynamic Bus Sizing
<i>BASE</i> + 0x0 (word 0)	0x0000 : <i>OFFSET</i> [0]	<i>OFFSET</i> [1] : <i>OFFSET</i> [0]
<i>BASE</i> + 0x4 (word 1)	0x0000 : <i>OFFSET</i> [1]	<i>OFFSET</i> [3] : <i>OFFSET</i> [2]
<i>BASE</i> + 0x8 (word 2)	0x0000 : <i>OFFSET</i> [2]	<i>OFFSET</i> [5] : <i>OFFSET</i> [4]
<i>BASE</i> + 0xC (word 3)	0x0000 : <i>OFFSET</i> [3]	<i>OFFSET</i> [7] : <i>OFFSET</i> [6]
...
<i>BASE</i> + 4 <i>N</i> (word <i>N</i>)	0x0000 : <i>OFFSET</i> [<i>N</i>]	<i>OFFSET</i> [2 <i>N</i> +1] : <i>OFFSET</i> [2 <i>N</i>]

SOPC Builder generates appropriate address-alignment logic based on the properties of the master and slaves in the system. When configuring a system in SOPC Builder, there are no settings that directly control the address alignment in the system interconnect fabric.

Dynamic Bus Sizing

Dynamic bus sizing hides the details of interfacing a narrow component device to a wider master, and vice versa. When an *N*-bit master accesses a slave with dynamic bus sizing, the master operates exclusively on full *N*-bit words of data, without awareness of the slave data width.



When using dynamic bus sizing, the slave data width in units of bytes must be a power of two.

Dynamic bus sizing provides the following benefits:

- Eliminates the need to create address-alignment hardware manually.
- Reduces design complexity of the master component.
- Enables any master to access any memory device, regardless of the data width.

In the case of dynamic bus sizing, the system interconnect fabric includes a small finite state machine that reconciles the difference between master and slave data widths. The behavior is different depending on whether the master data width is wider or narrower than the slave.

Wider Master

In the case of a wider master, the dynamic bus-sizing logic accepts a single, wide transfer on the master side, and then performs multiple narrow transfers on the slave side. For a data-width ratio of $N:1$, the dynamic bus-sizing logic generates up to N slave transfers for each master transfer. The master waits while multiple slave-side transfers complete; the master transfer ends when all slave-side transfers end.

Dynamic bus-sizing logic uses the master-side byte-enable signals to generate appropriate slave transfers. The dynamic bus-sizing logic performs as many slave-side transfers as necessary to write or read the specified byte lanes.

Narrower Master

In the case of a narrower master, one transfer on the master side generates one transfer on the slave side. In this case, multiple master word addresses map to a single offset in the slave memory space. The dynamic bus-sizing logic maps each master address to a subset of byte lanes in the appropriate slave offset. All bytes of the slave memory are accessible in the master address space.

Table 2–3 demonstrates the case of a 32-bit master accessing a 64-bit slave with dynamic bus sizing. In the table, offset refers to the offset into the slave memory space.

Table 2–3. 32-Bit Master View of 64-Bit Slave with Dynamic Bus Sizing	
32-bit Address	Data
0x00000000 (word 0)	OFFSET [0] _{31..0}
0x00000004 (word 1)	OFFSET [0] _{63..32}
0x00000008 (word 2)	OFFSET [1] _{31..0}
0x0000000C (word 3)	OFFSET [1] _{63..32}

In the case of a read transfer, the dynamic bus-sizing logic multiplexes the appropriate byte lanes of the slave data to the narrow master. In the case of a write transfer, the dynamic bus-sizing logic uses slave-side byte-enable signals to write only to the appropriate byte lanes.

Arbitration for Multimaster Systems

System interconnect fabric supports systems with multiple master components. In a system with multiple masters, such as the system pictured in [Figure 2–1 on page 2–3](#), the system interconnect fabric provides shared access to slaves using a technique called slave-side arbitration. Slave-side arbitration moves the arbitration logic close to the slave, such that the algorithm that determines which master gains access to a specific slave in the event that multiple masters attempt to access the same slave at the same time.

The multimaster architecture used by system interconnect fabric offers the following benefits:

- Eliminates the need to create arbitration hardware manually.
- Allows multiple masters to transfer data simultaneously. Unlike traditional host-side arbitration architectures where each master must wait until it is granted access to the shared bus, multiple Avalon-MM masters can simultaneously perform transfers with independent slaves. Arbitration logic stalls a master only when multiple masters attempt to access the same slave during the same cycle.
- Eliminates unnecessary master-slave connections. The connection between a master and a slave exists only if it is specified in SOPC Builder. If a master never initiates transfers to a specific slave, no connection is necessary, and therefore SOPC Builder does not waste logic resources to connect the two ports.
- Provides configurable arbitration settings, and arbitration for each slave is specified independently. For example, you can grant one master more arbitration shares than others, allowing it to gain more access cycles to the slave. The arbitration share settings are defined for each slave independently.
- Simplifies master component design. The details of arbitration are encapsulated inside the system interconnect fabric. Each Avalon-MM master connects to the system interconnect fabric as if it is the only master in the system. As a result, you can reuse a component in single-master and multimaster systems without requiring design changes to the component.

This section discusses the architecture of the system interconnect fabric generated by SOPC Builder for multimaster systems.

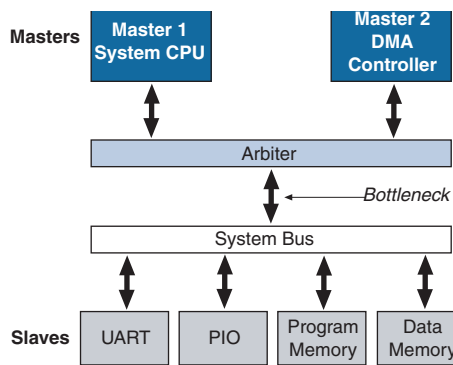
Traditional Shared Bus Architectures

As a frame of reference for the discussion of multiple masters and arbitration, this section describes traditional bus architectures.

In traditional bus architectures, one or more bus masters and bus slaves connect to a shared bus, consisting of wires on a printed circuit board or on-chip routing. A single arbiter controls the bus (that is, the path between bus masters and bus slaves), so that multiple bus masters do not simultaneously drive the bus. Each bus master requests control of the bus from the arbiter, and the arbiter grants access to a single master at a time. Once a master has control of the bus, the master performs transfers with any bus slave. When multiple masters attempt to access the bus at the same time, the arbiter allocates the bus resources to a single master, forcing all other masters to wait.

Figure 2–6 illustrates the bus architecture for a traditional processor system. Access to the shared system bus becomes the bottleneck for throughput: only one master has access to the bus at a time, which means that other masters are forced to wait and only one slave can transfer data at a time.

Figure 2–6. Bus Architecture in a Traditional Microprocessor System

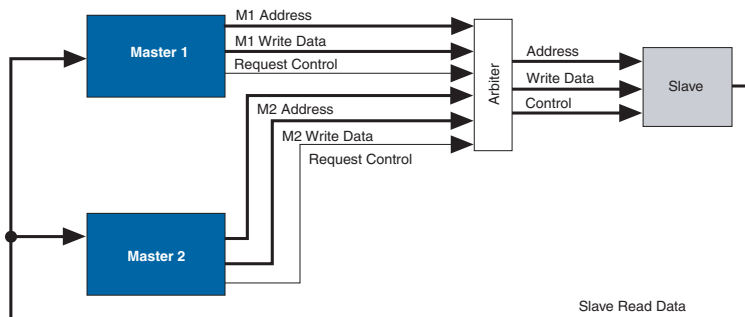


Slave-Side Arbitration

The system interconnect fabric uses multimaster architecture to eliminate the bottleneck for access to a shared bus. Multiple masters can be active at the same time, simultaneously transferring data with independent slaves. For example, Figure 2–1 on page 2–3 demonstrates a system with two masters (a CPU and a DMA controller) sharing a slave (an SDRAM controller). Arbitration is performed at the SDRAM slave; the arbiter dictates which master gains access to the slave if both masters initiate a transfer with the slave in the same cycle.

Figure 2–7 focuses on the two masters and the shared slave and shows additional detail of the data, address, and control paths. The arbiter logic multiplexes all address, data, and control signals from a master to a shared slave.

Figure 2–7. Detailed View of Multimaster Connections



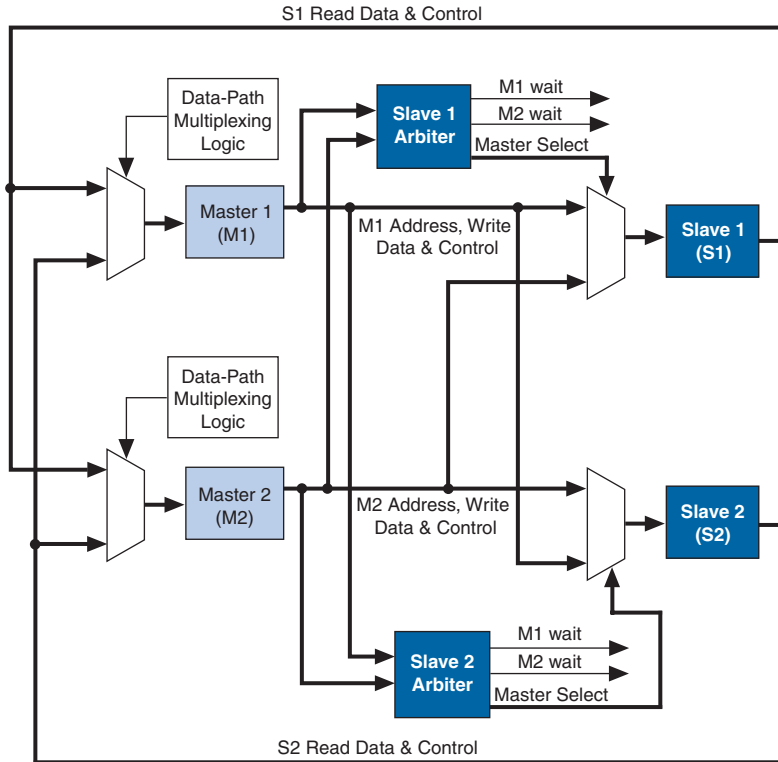
Arbiter Details

SOPC Builder generates an arbiter for every slave, based on arbitration parameters specified in SOPC Builder. The arbiter logic performs the following functions for its slave:

- Evaluates the address and control signals from each master and determines which master, if any, gains access to the slave next.
- Grants access to the chosen master and forces all other requesting masters to wait.
- Uses multiplexers to connect address, control, and datapaths between the multiple masters and the slave.

Figure 2–8 shows the arbiter logic in an example multimaster system with two masters, each connected to two slaves.

Figure 2–8. Block Diagram of Arbiter Logic



Arbitration Rules

This section describes the rules by which the arbiter grants access to masters when they contend.

Setting Arbitration Parameters in SOPC Builder

You specify the arbitration shares for each master using the connection panel on the **System Contents** tab of SOPC Builder, as shown in [Figure 2–9](#).

Figure 2–9. Arbitration Settings on the System Contents Tab

Module Name	Description	Clock
<input type="checkbox"/> cpu	Nios II Processor - Alte...	clk
<input type="checkbox"/> instruction_master	Master port	
<input type="checkbox"/> data_master	Master port	
1 <input type="checkbox"/> jtag_debug_module	Slave port	
1 <input checked="" type="checkbox"/> sys_clk_timer	Interval timer	clk
1 1 <input checked="" type="checkbox"/> ext_ram_bus	Avalon Tri-State Bridge	clk
<input checked="" type="checkbox"/> ext_flash	Flash Memory (Commo...	
<input checked="" type="checkbox"/> ext_ram	IDT71V416 SRAM	
1 1 <input checked="" type="checkbox"/> epcs_controller	EPCS Serial Flash Cont...	clk
<input checked="" type="checkbox"/> lan91c111	LAN91c111 Interface (...)	
1 1 <input checked="" type="checkbox"/> jtag_uart	JTAG UART	clk

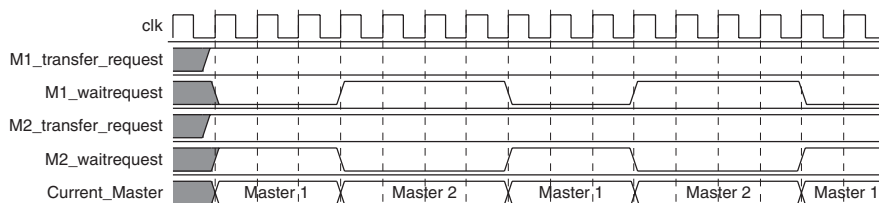
The arbitration settings are hidden by default. To see them, on the View menu, click **Show Arbitration**.

Fairness-Based Shares

Arbiter logic uses a fairness-based arbitration scheme. In a fairness-based arbitration scheme, each master pair has an integer value of *transfer shares* with respect to a slave. One share represents permission to perform one transfer.

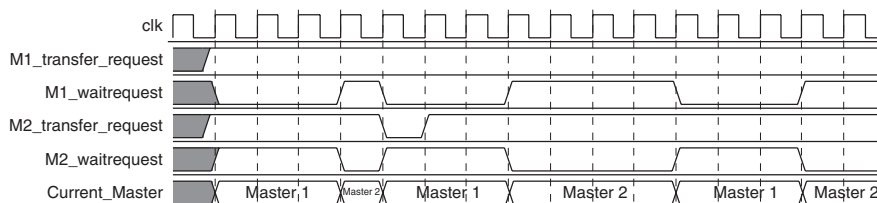
For example, assume that two masters continuously attempt to perform back-to-back transfers to a slave. Master 1 is assigned three shares and Master 2 is assigned four shares. In this case, the arbiter grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely. [Figure 2–10](#) demonstrates this case, showing each master's transfer request output, wait request input (which is driven by the arbiter logic), and the current master with control of the slave.

Figure 2–10. Arbitration of Continuous Transfer Requests from Two Masters



If a master stops requesting transfers before it exhausts its shares, it forfeits all its remaining shares, and the arbiter grants access to another requesting master. See [Figure 2-11](#). After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbiter grants access back to Master 1, which gets a replenished supply of shares.

Figure 2-11. Arbitration of Two Masters with a Gap in Transfer Requests



Round-Robin Scheduling

When multiple masters contend for access to a slave, the arbiter grants shares in round-robin order. Round-robin scheduling drives a request interface according to space available and data available credit interfaces. At every slave transfer, only requesting masters are included in the arbitration.

Burst Transfers

Avalon-MM burst transfers grant a master uninterrupted access to a slave for a specified number of transfers. The master specifies the number of transfers when it initiates the burst. Once a burst begins between a master-slave pair, arbiter logic does not allow any other master to access the slave until the burst completes. For burst masters, the size of the burst controls the determined the number of cycles that the master has access to the slave, and the selected transfer shares have no effect. For further information, refer to [“Burst Adapters” on page 2-17](#).

Minimum Share Value

A component design can declare the minimum number of shares in each round-robin cycle, which affects how the arbiter grants access. For example, if a slave has a minimum share value of ten, then the arbiter grants at least ten shares to any master when it begins a sequence of transfer requests. The arbiter might grant more shares, if the master is assigned more shares in SOPC Builder.

By declaring a minimum share value of N , a slave declares that it is more efficient at handling continuous sequential transfers of length N . Accessing the slave in sequences less than N incurs performance penalties that might prevent the slave from achieving higher performance. By nature, continuous back-to-back master transfers tend to access sequential addresses. However, there is no requirement that the master perform transfers to sequential addresses.



Burst transfers provide even higher performance for continuous transfers when they are guaranteed to access sequential addresses. The minimum share value does not apply to slaves that support bursts; the burst length takes precedence over minimum share value. Refer to “[Burst Adapters](#)” for information.

Burst Adapters

System interconnect fabric provides burst adaptation logic to accommodate the burst capabilities of each port in the system, including ports that do not support burst transfers. Burst adaptation logic consists of a finite state machine that translates the sequencing of address and control signals between the slave side and the master side.

The maximum burst length for each port is determined by the component design and is independent of other ports in the system. Therefore, a particular master might be capable of initiating a burst longer than a slave's maximum supported burst length. In this case, the burst management logic translates the master burst into smaller slave bursts, or into individual slave transfers if the slave does not support bursts. Until the master completes the burst, the arbiter logic prevents other masters from accessing the target slave.

For example, if a master initiates a burst of 16 transfers to a slave with maximum burst length of 8, the burst adapter logic initiates two bursts of length 8 to the slave. If the master initiates a burst of 14, the burst adapter logic will segment the burst transfer into a burst of 8 words followed by a burst of 6 words, because the slave can only handle a maximum burst length of 8. If a master initiates a burst of 16 transfers to a slave that does not support bursts, the burst management logic initiates 16 separate transfers to the slave.

Interrupts

In systems where components have interrupt request (IRQ) sender interfaces, the system interconnect fabric includes interrupt controller logic. A separate interrupt controller is generated for each interrupt receiver. The interrupt controller aggregates IRQ signals from all interrupt senders, and maps them to user-specified values on the receiver inputs.



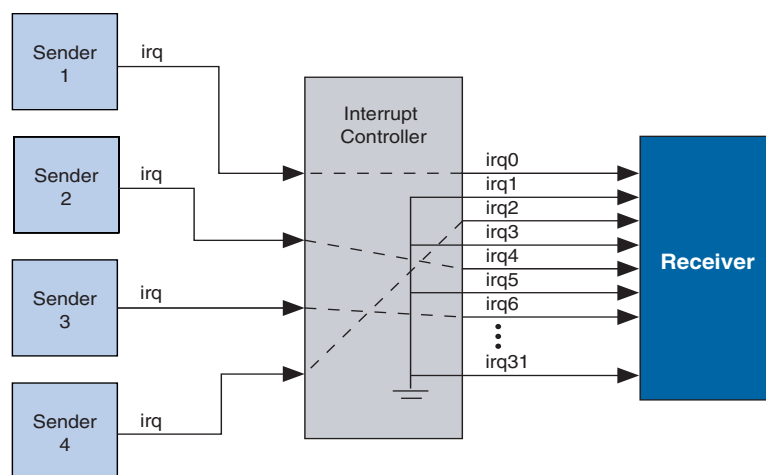
For further information to the *Interrupt Interfaces* chapter in the *Avalon Interface Specifications*.

Individual Requests IRQ Scheme

In the individual requests IRQ scheme, the system interconnect fabric passes IRQs directly from the sender to the receiver, without making any assumptions about IRQ priority. In the event that multiple senders assert their IRQs simultaneously, the receiver logic (presumably under software control) determines which IRQ has highest priority, then responds appropriately.

Using individual requests, the interrupt controller can handle up to 32 IRQ inputs. The interrupt controller generates a 32-bit signal `irq[31..0]` to the receiver, and simply maps slave IRQ signals to the bits of `irq[31..0]`. Any unassigned bits of `irq[31..0]` are disabled. [Figure 2-12](#) shows an example of the interrupt controller mapping the IRQs on four senders to `irq[31..0]` on a receiver.

Figure 2-12. IRQ Mapping Using Software Priority

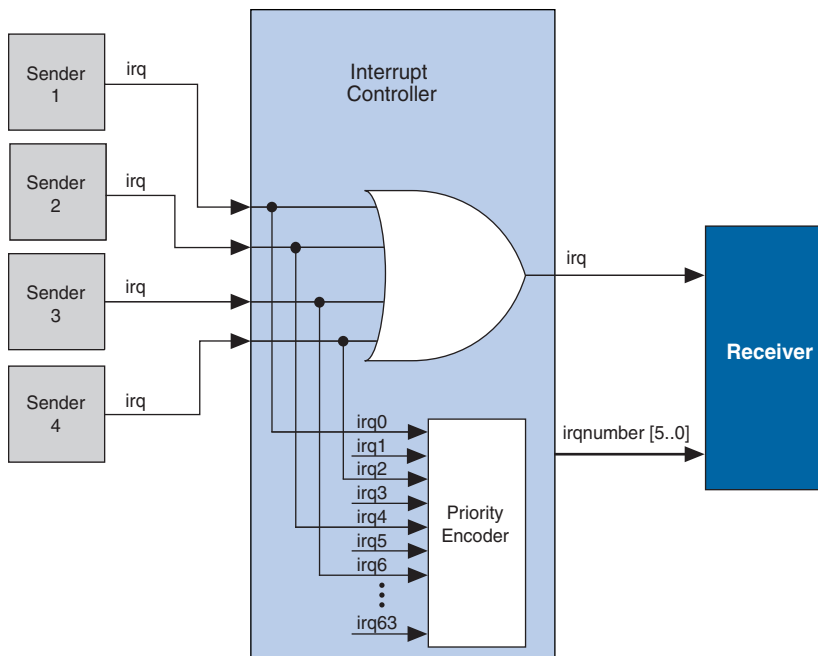


Priority Encoded Interrupt Scheme

In the priority encoded interrupt scheme, in the event that multiple slaves assert their IRQs simultaneously, the system interconnect fabric provides the interrupt receiver with a 1-bit interrupt signal, and the number of the highest priority active interrupt. An IRQ of lesser priority is undetectable until all IRQs of higher priority have been serviced.

Using priority encoded interrupts, the interrupt controller can handle up to 64 slave IRQ signals. The interrupt controller generates a 1-bit `irq` signal to the receiver, signifying that one or more senders have generated an IRQ. The controller also generates a 6-bit `irqnumber` signal, which outputs the encoded value of the highest pending IRQ. See [Figure 2–13](#).

Figure 2–13. IRQ Mapping Using Hardware Priority



Assigning IRQs in SOPC Builder

You specify IRQ settings on the **System Contents** tab of SOPC Builder. After adding all components to the system, you make IRQ settings for all interrupt senders, with respect to each interrupt receiver. For each slave, you can either specify an IRQ number, or specify not to connect the IRQ.

Reset Distribution

SOPC Builder generates the logic used in the system interconnect fabric, which drives the reset pulse to all the logic. The system interconnect fabric distributes the reset signal conditioned for each clock domain. The duration of the reset signal is at least one clock period.

The system interconnect fabric asserts the system-wide reset in the following conditions:

- The global reset input to the SOPC Builder system is asserted.
- Any component asserts its `resetrequest` signal.

The global reset and reset requests are "ORed" together. This signal is then synchronized to each clock domain associated to an Avalon-MM port, which causes the asynchronous resets to be de-asserted synchronously.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Avalon Memory-Mapped Bridges* chapter in the *Avalon Interface Specifications*
- *Avalon Memory-Mapped Design Optimizations* chapter in the *Embedded Design Handbook*
- *Interrupt Interfaces* chapter in the *Avalon Interface Specifications*

Document Revision History

Table 2–4 shows the revision history for this chapter.

Table 2–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	<ul style="list-style-type: none"> Updated references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications. Moved clock-crossing bridge section from this chapter to chapter 11. 	The two specifications have been combined into one for all Avalon interfaces.
October 2007 v7.2.0	<ul style="list-style-type: none"> Updated to match 7.2 features. Deleted paragraphs discussing “Pipelining for High Performance”, “Endian Conversion”, and added new screenshots. Moved clock-crossing bridge discussion to this chapter from chapter 10. 	—
May 2007, v7.1.0	<ul style="list-style-type: none"> Chapter 3 was previously titled Avalon Switch Fabric. Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon switch fabric” term to “system interconnect fabric.” Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.” Rearranged content in section “Introduction” on page 2–1 to enhance clarity and to acknowledge the existence of the new Avalon Streaming interface. In section “Pipelining for High Performance” on page 2–7, noted that automatic pipelining for high performance is a deprecated feature. Added the recommendation to use the Avalon-MM Pipeline Bridge component instead. Updated Table 2–2 on page 2–9 for improved clarity. Updated section “Dynamic Bus Sizing” on page 2–9 to reflect new behavior of system interconnect fabric with respect to byte enables during read transfers. For a master-to-slave data-width ratio of $N:1$, the system interconnect fabric might not need to perform N slave-side read transfers, depending on how the master asserts its byte-enable signals. Added three paragraphs explaining when clock signals are automatically connected to SOPC Builder components. Added paragraph referencing the higher performance Avalon-MM Clock-Crossing Bridge which can be used instead of the CDC logic for systems requiring higher throughput. 	For the 7.1 release, Altera released the Avalon Streaming Interface, which necessitated some re-phrasing of existing Avalon terminology. The newly-released Avalon-MM Pipeline Bridge component provides a more effective means to improve f_{MAX} performance than the traditional pipeline option in SOPC Builder. The behavior of <code>byteenable</code> signals in the Avalon Interface Specifications was updated, necessitating changes to this document.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	No change from previous release.	—

Table 2–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
May 2006, v6.0.0	No change from previous release.	—
October 2005, v5.1.0	No change from previous release.	—
August 2005, v5.0.1	Updated for the Quartus II software version 5.1.	—
May 2005, v5.0.0	<ul style="list-style-type: none"> • Added burst transfer management details. • Updated pipeline management details. 	—
February 2005, v1.0	Initial release.	—

Introduction

Avalon® Streaming interconnect fabric connects high-bandwidth, low latency components that use the Avalon Streaming (Avalon-ST) interface. This interconnect fabric creates datapaths for unidirectional traffic including multichannel streams, packets, and DSP data. This chapter describes the Avalon-ST interconnect fabric and its use in connecting components with Avalon-ST interfaces. Descriptions of specific adapters and their use in streaming systems can be found in the following sections:

- “Adapters” on page 3-3
- “Multiplexer Examples” on page 3-5

High-Level Description

Avalon-ST interconnect fabric is logic generated by SOPC Builder. Using SOPC Builder, you specify how Avalon-ST source and sink ports connect. SOPC Builder then creates a high performance point-to-point interconnect between the two components. The Avalon-ST interconnect is flexible and can be used to implement on-chip interfaces for industry standard telecommunications and data communications cores, such as Ethernet IEEE 802.3 MAC and SPI 4.2. In all cases, bus widths, packets, and error conditions are custom-defined.

Figure 3-1 illustrates the simplest system example that generates an interconnect between the source and sink. This source-sink pair includes only the data and valid signals.

Figure 3-1. Interconnect for a Simple Avalon Streaming Source-Sink Pair

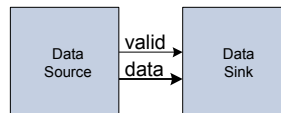
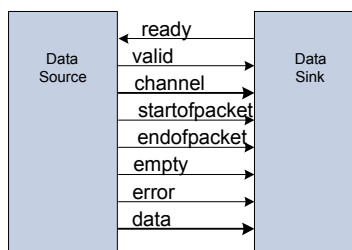


Figure 3-2 illustrates a more extensive interface that includes signals indicating the start and end of packets, channel numbers, error conditions, and back pressure.

Figure 3–2. Avalon Streaming Interface for Packet Data

All data transfers using Avalon-ST interconnect occur synchronously to the rising edge of the associated clock interface. All outputs from the source interface, including the data, channel, and error signals, must be registered on the rising edge of the clock. Registers are not required for inputs at the sink interface. Registering signals at the source provides for high frequency operation while eliminating back-to-back registration with no intervening logic. There is no inherent maximum performance of the interconnect. Throughput for a system depends on the components and how they are connected.



Although you do not have to register signals in the sink-to-source direction, you must register such signals if more than a trivial amount of logic is needed to generate them. Registering signals at both ends of the source-to-sink connection can increase the f_{MAX} at which the system can run.



For details about the Avalon-ST interface protocol, refer to the *Avalon Interface Specifications*, which are available on www.altera.com.

Avalon Streaming and Avalon Memory-Mapped Interfaces

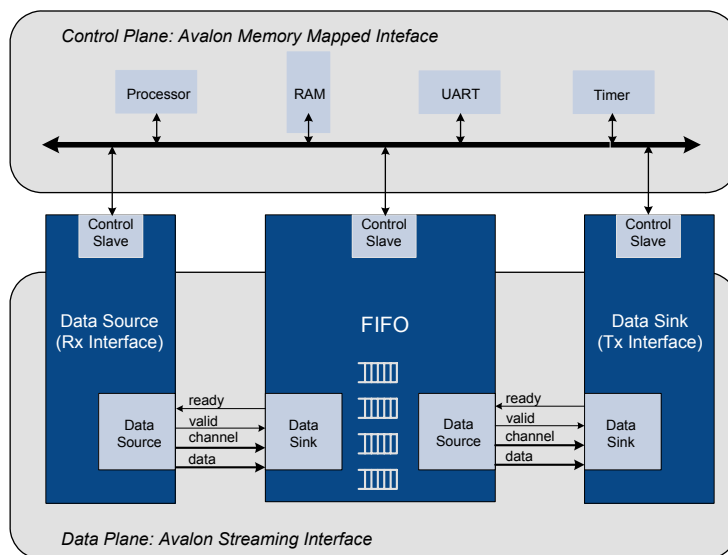
The Avalon-ST and Avalon Memory-Mapped (Avalon-MM) interfaces are complementary. High bandwidth components with streaming data typically use Avalon-ST interfaces for the high throughput datapath. These components can also use Avalon-MM connection interfaces to provide an access point for control. In contrast to the Avalon-MM interconnect, which can be used to create a wide variety of topologies, the Avalon-ST interconnect fabric always creates a point-to-point between a single data source and data sink, as Figure 3–3 illustrates. There are two connection pairs in this figure:

- The Data Source in the Rx Interface transfers data to the Data Sink in the FIFO.

- The Data Source in the FIFO transfers data to the Tx Interface Data Sink.

In Figure 3–3, the Avalon-MM interface allows a processor to access the data source, FIFO, or data sink to provide system control.

Figure 3–3. Use of the Avalon Memory-Mapped and Streaming Interfaces



Adapters

Adapters are configurable SOPC Builder components that are part of the streaming interconnect fabric. They are used to connect source and sink interfaces that are not exactly the same without affecting the semantics of the data. SOPC Builder includes the following three adapters:

- Data Format Adapter
- Timing Adapter
- Channel Adapter

You can add Avalon-ST adapters between two components of mismatched size. This allows you to connect a data source to a data sink of differing byte sizes. If you connect mismatched Avalon-ST sources and sinks in SOPC Builder without inserting adapters, SOPC Builder generates error messages. Inserting adapters into the system does not change the types of components that SOPC Builder allows you to connect.

The **Insert Avalon-ST Adapters** command on the System menu attempts to correct these errors automatically, if possible, by inserting the appropriate adapter types.



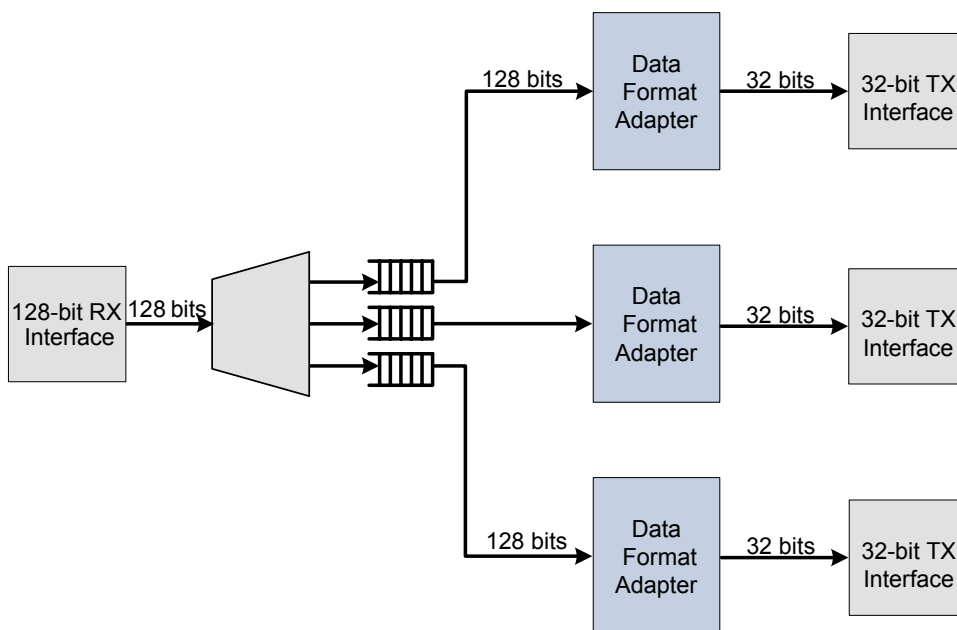
For complete information about these adapters, refer to the *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*.

The following sections provide an overview of these adapters.

Data Format Adapter

The data format adapter allows you to connect interfaces that have different values for the parameters defining the data signal. One of the most common uses of this adapter is to convert data streams of different widths. [Figure 3–4](#) shows an adapter that allows a connection between a 128-bit input data stream and three 32-bit output data streams.

Figure 3–4. Avalon Streaming Interconnect Fabric with Data Format Adapter



Timing Adapter

The timing adapter allows you to connect component interfaces that require a different number of cycles before driving or receiving data. This adapter inserts a FIFO between the source and sink to buffer data or pipeline stages to delay the back pressure signals. The timing adapter can also be used to connect interfaces that support the `ready` signal and those that do not.

Channel Adapter

The channel adapter provides adaptations between interfaces that have different support for the channel signal or channel-related parameters. For example, if the source channel is narrower than the sink channel, you can use this adapter to connect them. The high-order bits of the sink channel are connected to zero. You can also use this adapter to connect a source with a wider channel to a sink with a narrower channel.

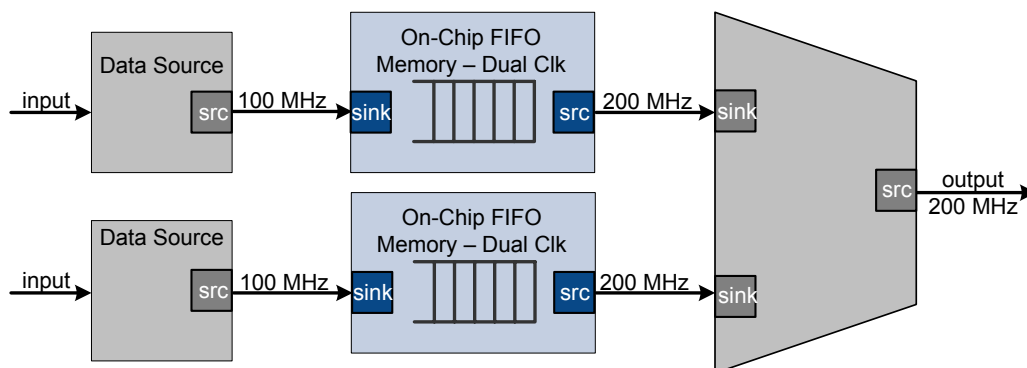
Multiplexer Examples

You can combine these adapters described above with streaming components to create datapaths whose input and output streams have different properties. The following sections provide examples of datapaths constructed using SOPC Builder in which the output stream is higher performance than the input stream:

- The first example shows an output with double the throughput of each interface with a corresponding doubling of the clock frequency.
- The second example doubles the data width.
- The third example boosts the frequency of a stream by 10% by multiplexing input data from two sources.

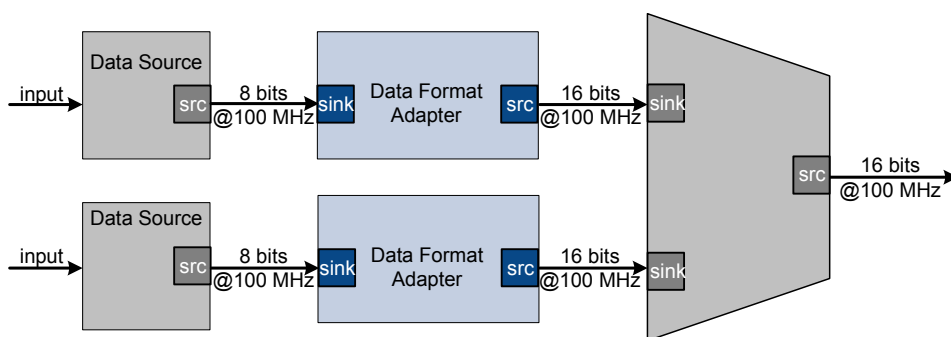
Example to Double Clock Frequency

Figure 3–5 illustrates a datapath that uses the dual clock version of the on-chip FIFO memory and Avalon-ST channel multiplexer to merge the 100 MHz input from two streaming data sources into a single 200 MHz streaming output. As Figure 3–5 illustrates, this example increases throughput by increasing the frequency and combining inputs.

Figure 3–5. Datapath that Doubles the Clock Frequency

Example to Double Data Width and Maintain Frequency

Figure 3–6 illustrates a datapath that uses the data format adapter and Avalon-ST channel multiplexer to convert two, 8-bit inputs running at 100 MHz to a single 16-bit output at 100 MHz.

Figure 3–6. Datapath to Double Data Width and Maintain Original Frequency

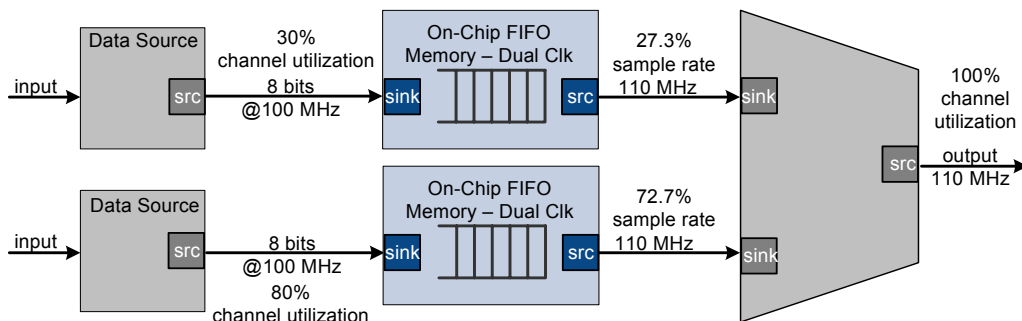
Example to Boost the Frequency

Figure 3–7 illustrates a datapath that uses the dual clock version of the on-chip FIFO memory to boost the frequency of input data from 100 MHz to 110 MHz by sampling two input streams at differential rates. In this example, the on-chip FIFO memory has an input clock frequency of

100 MHz and an output clock frequency of 110 MHz. The channel multiplexer runs at 110 MHz and samples one input stream 27.3 percent of the time and the second 72.7 percent of the time.

You do not need to know what the typical and maximum input channel utilizations are before attempting this. For example, if the first channel hits 50% utilization, then the output stream exceeds 100% utilization.

Figure 3–7. Datapath to Boost the Clock Frequency



Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*, available at www.altera.com
- *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 3–1 shows the revision history for this chapter.

<i>Table 3–1. Document Revision History</i>		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	Updated references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications.	—
October 2007, v7.2.0	No changes for this release.	—
May 2007, v7.1.0	Initial release.	The Avalon-ST Data Format Adapter, Timing Adapter and Channel Adapter are new components provided in the Quartus II software v7.1 release.

Introduction

An SOPC Builder *component* is a hardware design block available within SOPC Builder that can be instantiated in an SOPC Builder system. This chapter defines SOPC Builder components, with emphasis on the structure of custom components.

A component includes the following:

- The Hardware Description Language (HDL) description of the component's hardware.
- A description of the interface to the component hardware, such as the names and types of I/O signals.
- A description of any parameters that specify the structure of the component logic and component.
- A GUI for configuring an instance of the component in SOPC Builder.
- Scripts and other information SOPC Builder needs to generate the HDL files for the component and integrate the component instance into the SOPC Builder system.
- Other component-related information, such as reference to software drivers, necessary for development steps downstream of SOPC Builder.

This chapter discusses the design flow for new and classic custom-defined SOPC Builder components, in the following sections:

- “Component Providers” on page 4-1
- “Component Hardware Structure” on page 4-2
- “Exported Connection Points” on page 4-4
- “Selecting Components in SOPC Builder” on page 4-5
- “Component Structure” on page 4-5
- “Using Classic Components in SOPC Builder” on page 4-7

Component Providers

SOPC Builder components can be obtained from many providers, including:

- The components accepted include those installed with the Quartus® II software.

- Third-party IP developers can provide IP blocks as SOPC Builder ready components, including software drivers and documentation. A list of select of third-party components can be found in SOPC Builder by clicking **IP MegaStore** on the Tools menu.
- Altera development kits, such as the Nios® II Development Kit, can provide SOPC Builder components as features.
- You can use the SOPC Builder component editor to convert your own HDL files into custom components.



The GUI interfaces for classic components run slower in newer versions of SOPC Builder when you add or modify your component settings. You have better performance when you upgrade to the new **_hw.tcl** component format in newer versions of SOPC Builder.

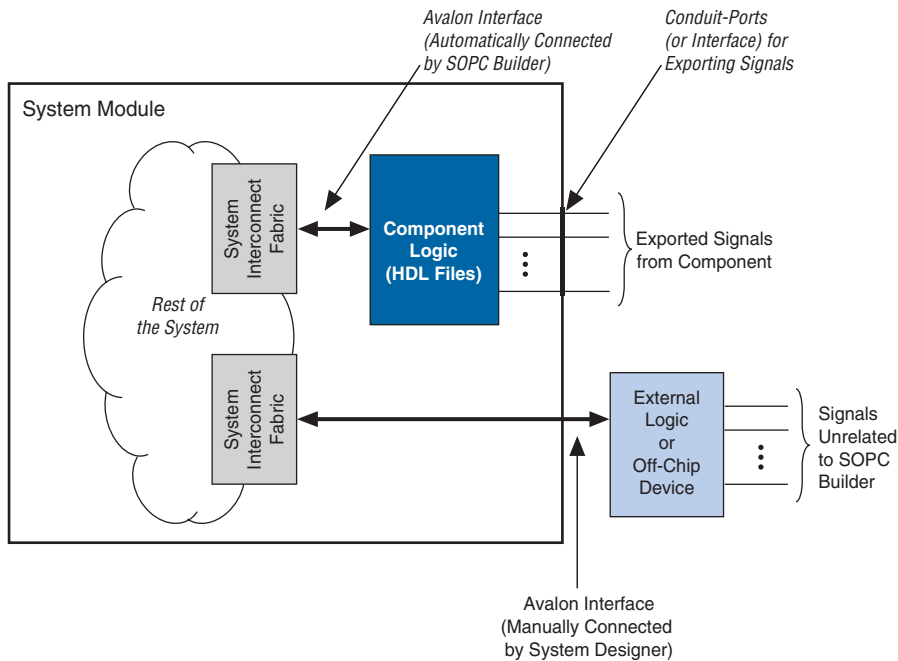
Component Hardware Structure

There are the following types of components in an SOPC Builder system, based on where the associated component logic resides:

- Components that include their associated logic inside the SOPC Builder system
- Components that interface to logic outside the SOPC Builder system

Figure 4–1 shows an example of both types of components.

Figure 4–1. Component Logic Inside and Outside the SOPC Builder System



Components Inside the SOPC Builder System

For components that are instantiated inside the SOPC Builder system, the component defines its logic in an associated HDL file. During system generation, SOPC Builder instantiates the component and connects it to the rest of the system. The component can include exported signals in conduit interfaces. Conduit interfaces become ports on the system, so they can be connected to logic outside the SOPC Builder system in the board-level schematic.



For more information about conduit interfaces, refer to the [Conduit Interfaces](#) chapter in the *Avalon Interface Specifications*.

In general, components connect to the system interconnect fabric using the Avalon® Memory-Mapped (Avalon-MM) interface or the Avalon Streaming (Avalon-ST) interface. A single component can provide more

than one Avalon port. For example, a component might provide an Avalon-ST source port for high-throughput data, in addition to an Avalon-MM slave for control.

Components That Interface to Logic Outside the SOPC Builder System

For components that interface to logic outside the SOPC Builder system, the component files describe only the interface to the external logic. During system generation, SOPC Builder exports an interface for the component in the top-level SOPC Builder system. You must manually connect that pop out of SOPC Builder that need to be routed to pins or other logic, or logic defined outside the system that already has Avalon-compatible signals.



For example, you might have a four-bit output PIO to drive LEDs in your design, so you would see four PIO wires at the top level of SOPC Builder that you would then have to assign to I/O pins of your FPGA.

Exported Connection Points

Conduit interfaces are brought to the top level of the system as additional ports. Exported signals are usually either application-specific signals or the Avalon interface signals.

Application-specific signals are exported to the top level of the system by the conduit interface(s) defined in the component description file. These are I/O signals in a component's HDL logic that are not used by any Avalon interfaces and connect to an external device (i.e. DDR SDRAM memory), or logic defined outside of the system that is not in HDL format. You use these conduit interfaces to create the interface logic to the Avalon interface between external devices or logic defined outside of the system.

You can also export the Avalon interfaces to manually connect them to external devices, or logic defined outside a system with Avalon-compatible signals. This method allows a direct connection to the Avalon interface from any device that has Avalon-compatible signals. You can also export the Avalon interface in either an HDL file using conduit interfaces, or in the **_hw.tcl** file without an HDL file.

You export the Avalon interface signals as an HDL file with simple wire connections in the HDL description. The Avalon interface port signals are directly connected to external I/O signals in the HDL description. The conduit interface in the **_hw.tcl** file exports the external I/O signals to the top level of the system.

In the `_hw.tcl` file, no HDL files are specified and only the Avalon signals and interface ports are declared in the file.

Selecting Components in SOPC Builder



Each time SOPC Builder starts, it searches for component files. The components that SOPC Builder finds are displayed in the list of available components on the SOPC Builder **System Contents** tab.

For more information refer to the *Embedded Peripherals* section in volume 5 of the *Quartus II Handbook*, which describes the components that appear in the latest version of SOPC Builder.

SOPC Builder uses the following mechanisms to generate the list of available components:

- SOPC Builder automatically searches the `/ip` subdirectory of your Quartus II project directory. Adding a component to a project is as easy as copying it to an `/ip` subdirectory. This method is recommended for all project-specific components.
- SOPC Builder searches all of the paths entered in SOPC Builder/Tools/Options/IP Search Path to support a global library of components. This method is recommended for all global components. This search path is shared by all of your SOPC Builder projects.
- SOPC Builder identifies component files stored in the current Quartus II project directory.
- SOPC Builder searches the paths where previous versions of SOPC Builder expected to find component files, including classic component search paths.

Component Structure



Most components are defined with a `_hw.tcl` file, which is a text file written in the Tcl scripting language that describes the components in your SOPC Builder system. You can add a component to SOPC Builder by either writing a Tcl description or you can use the component editor to generate an automatic Tcl description of it. This section describes the structure of Tcl components and how they are stored.

For details about the SOPC Builder component editor, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*. For details about the SOPC Builder Tcl commands, refer to the *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook*.

Component Description File (`_hw.tcl`)

A Tcl component consists of:

- A component description file, which is a Tcl file with file name of the form `<entity name>_hw.tcl`.
- Verilog HDL, HDL, or VHDL files that define the top-level module of the custom component (optional).

The `_hw.tcl` file defines everything that SOPC Builder requires about the name and location of component design files.

The SOPC Builder component saves components in the `_hw.tcl` format. These Tcl files can be used as a template for editing components by hand.

Component File Organization

A typical component uses the following directory structure. The names of the directories are not significant.

- **component_directory/**
 - **hdl/**—a directory that contains the component HDL design files and the `_hw.tcl` file
 - `<component name>_hw.tcl`—the component description file
 - `<component name>.v` or `.vhd`—the HDL file that contains the top-level module
 - `<component_name>_sw.tcl`—the software driver configuration file for the Nios II command line flow.
 - You are not required to create a special sub-directory for component HDL files. However, you are required to follow the naming conventions given here.
 - `component_dir/`
 - `<name>_hw.tcl`
 - `<name>.v` or `.vhd`
 - `<name>_sw.tcl`
- **software/**—a directory that contains software drivers or libraries related to the component, if any. The component directory often includes a `_sw.tcl` file and the software definitions and drivers to which it refers.



For information on writing a device driver or software package suitable for use with the Nios II IDE design flow, please refer to the *Hardware Abstraction Layer* section of the *Nios II Software Developer's Handbook*. The *Nios II Software Build Tool Reference* chapter of the *Nios II Software Developer's Handbook* describes the commands you can use in the Tcl script.

Using Classic Components in SOPC Builder

If you use classic components created with an earlier version of SOPC Builder, read through this section to familiarize yourself with the differences. This document uses the term “classic components” to refer to components created with a previous version of the Quartus II software. If you do not have any classic components, skip this section.

Classic components are compatible with newer versions of SOPC Builder, with the following caveats:

- Classic components that use a **More Options** tab in SOPC Builder, such as complex IP components provided by third-party IP developers, are not supported in the Quartus II software in version 7.1 and beyond. If your component has a “bind” program, you cannot use the component without recreating it with the component editor or with Tcl scripting.
- To make changes to a classic component with the component editor, you must first upgrade the component by editing the classic component and saving it in the `_hw.tcl` component format in the component editor.

Referenced Document

This chapter references the following documents:

- *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook*
- *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- *Conduit Interfaces* chapter in the *Avalon Interface Specifications*
- Volume 5 of the *Quartus II Handbook*
- *Hardware Abstraction Layer* section of the *Nios II Software Developer's Handbook*
- *Nios II Software Build Tool Reference* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

Table 4–1 shows the revision history for this chapter.

Table 4–1. Document Revision History		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	<ul style="list-style-type: none"> Added paragraph about IP Search Path. 	—
October 2007, v7.2.0	<ul style="list-style-type: none"> Description added of Tcl components and removal of custom-defined components. Added warning that SOPC Builder does not support parameter values > 31 bits 	—
May 2007, v7.1.0	<ul style="list-style-type: none"> Described the new structure of components which is new in 7.1. Added and updated the sources of components list. Reorganized content of the chapter. Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon switch fabric” term to “system interconnect fabric.” Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.” Removed description of SOPC Builder MegaWizard® Plug-In Manager component discovery mechanism that was inaccurate. 	Version 7.1 of the Quartus II software provides a new mechanism for storing and finding SOPC Builder component files located on your computer, which necessitates significant changes to this chapter.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	No change from previous release.	—
May 2006, v6.0.0	No change from previous release.	—
October 2005, v5.1.0	No change from previous release.	—
August 2005, v5.0.1	Corrected reference to figure.	—
May 2005, v5.0.0	No change from previous release.	—
February 2005, v1.0	Initial release.	—

Introduction

This chapter describes the Quartus® II software tools that interface with SOPC Builder, including the following:

- “Quartus IP File”
- “Quartus II Incremental Compilation” on page 5–2
- “TimeQuest Timing Analyzer” on page 5–2

Quartus IP File

The Quartus IP File (**.qip**) generated by SOPC Builder provides the Quartus II software with all required information about your SOPC Builder system. SOPC Builder creates the **.qip** during system generation and adds a reference to it in the Quartus II Settings File (**.qsf**).

The **.qip** file includes references to the following information:

- Hardware description language (HDL) files used in the SOPC Builder system
- TimeQuest timing analyzer Synopsys Design Constraint (SDC) files
- Component definition files for archiving purposes

The **.qip** file is based on Tcl scripting syntax and is similar to the **.qsf** file. The information required to process most components is included in the system's single **.qip** file. Some complex components provide their own **.qip** file, in which case the system's **.qip** file references the component **.qip** file.



The **.qip** file is normally added to your project automatically by SOPC Builder. If it does not get added automatically you can add the file in the same way that you add other source files to your project. You can also have a **.qip** file for each component in your design. When you generate a design, each **.qip** is pulled into the main **.qip** file for your system by reference.

Quartus II Incremental Compilation

SOPC Builder supports the Quartus II incremental compilation feature, which allows you to separately compile isolated portions, or partitions, of a design.

From within the Quartus II software, you can designate an entire SOPC Builder system as a design partition, or you can designate individual SOPC Builder components as design partitions.



Changing the parameters of a component and regenerating your system only prompts other partitions within the same system to recompile if the HDL in that partition depends on the changed parameters. This means that the HDL you generate for the Nios II processor is optimized as related to components to which the Nios II processor is connected.



For more information about Incremental Compilation, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

TimeQuest Timing Analyzer

Altera recommends the TimeQuest timing analyzer in the Quartus II software for analysis of all new designs. SOPC Builder automatically generates TimeQuest Synopsys Design Constraints (.sdc) for SOPC Builder systems and components. In most cases, you use the TimeQuest constraints to declare false paths for signals that cross clock domains within a component, so that the TimeQuest timing analyzer does not perform normal setup and hold analysis for them. You add the TimeQuest SDC to the Quartus II project by including them in the .qip file.

The Classic timing analysis was provided in earlier versions of the Quartus II software. In the past you could use the Classic timing analyzer to check the timing results after compilation in the Quartus II software, especially for SOPC Builder based designs, where much of the external interface timing is easy to meet. However, Altera now recommends that you constrain designs before compilation, because in the Quartus II software any unconstrained paths are reported and analyzed by default during the compilation process.



Refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* for further description of the TimeQuest timing analyzer.



Refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* for a description of the benefits of using the TimeQuest timing analyzer rather than the Classic timing analyzer.



Refer to *TimeQuest Example: Basic SDC Example* on www.altera.com for a working example of using the TimeQuest timing analyzer.



Refer to *TimeQuest Design Examples* on www.altera.com for further details about how to constrain different types of circuits for the TimeQuest timing analyzer.

Analyzing PLLs

You must constrain PLL clocks for proper analysis by the TimeQuest timing analyzer. You can define clocks generated by PLLs using one of the following methods:

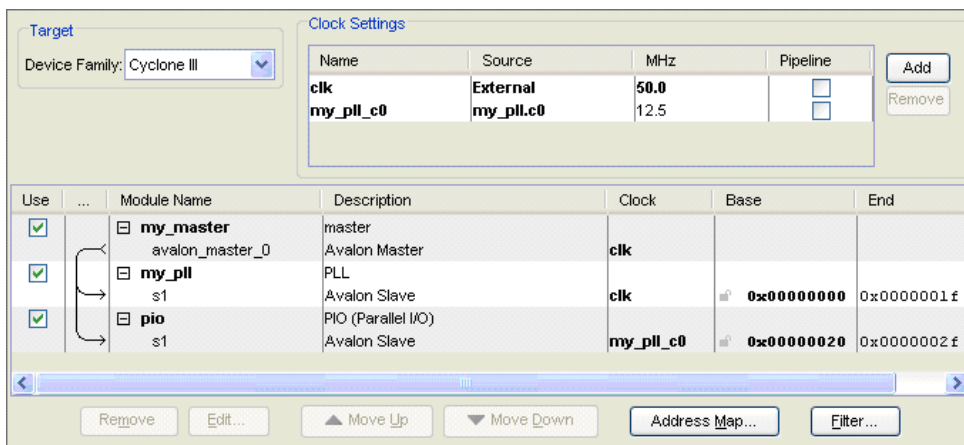
- Use the **derive_pll_clocks** command to derive clocks for all PLL outputs in the design. This is the best method.
- Use the **create_generated_clock** command to designate each clock output.
- Use the **-create_base_clocks** option of the **derive_pll_clock** assignments to designate the base clock feeding the PLL.

The following example focuses on the use of the `derive_pll_clocks` assignment, because this method automatically defines clock frequencies and phase shifts.



Note that `derive_pll_clocks` generates clocks for all PLLs in the Quartus II software project, not just for the PLLs in the SOPC Builder system.

The SOPC system shown in [Figure 5–1](#) illustrates the use of the `derive_pll_clocks` assignment in the case of a single clock input and one PLL using a single output.

Figure 5–1. Example SOPC System

After running the following commands in the TimeQuest timing analyzer, two clocks are generated:

```
create_clock -name master_clk -period 20 [get_ports {clk}]
```

```
derive_pll_clocks
```

The TimeQuest timing analyzer analyzes and reports performance of the constrained clocks in the Clocks Summary report. This displays a report as shown in [Figure 5–2](#).

Figure 5–2. Clocks Summary Report

Clocks Summary			
	Clock Name	Type	Period
1	master_clk	Base	20.000
2	the_my_pll the_pll altpll_component auto_generated pll1 clk[0]	Generated	80.000

master_clk is defined by the **create_clock** command, and the_my_pll clock is derived from the **derive_pll_clocks** command.

Analyzing Slow Asynchronous I/O Paths

If you use slow asynchronous I/O in an SOPC Builder system, such as PIO and UART peripherals, you do not need to analyze these paths because they are asynchronous to the clock that is used to capture or output data. In this case you must designate false paths to produce an accurate analysis.

For outputs, set a false path between the launch clock and the output. For inputs, a false path should be set between the input and the latching clock. For bi-directional signals, set a false path from the launching clock to the bidirectional pin and also from the bidirectional pin to the latching clock. Launch and latch clocks are typically the clocks associated with the SOPC Builder module that includes the I/O.

For the system described in the PLL section above, the following command sets false paths for the PLL outputs:

```
set_false_path -to [get_ports {*_pio[*]}]
```

Because design contains a 4-bit PIO, the filter `*_pio[*]` includes the following I/O pins.

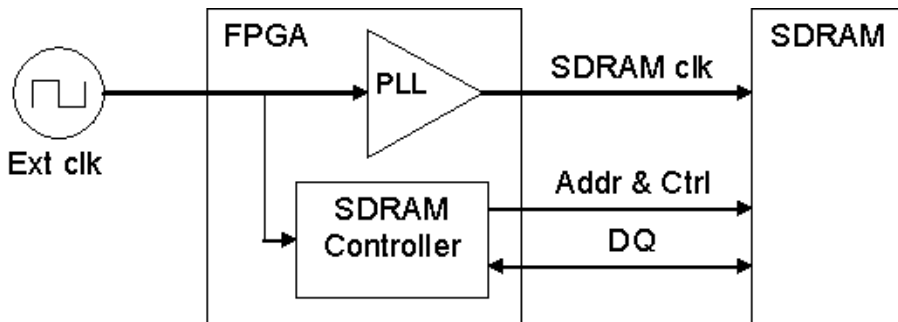
- `out_port_from_the_pio[0]`
- `out_port_from_the_pio[1]`
- `out_port_from_the_pio[2]`
- `out_port_from_the_pio[3]`



Ensure that filters using wildcard filters do not include unintended nodes.

Analyzing Single Data Rate SDRAM and SSRAM

Single data SDRAM interfaces in SOPC Builder typically use the type of circuit shown in [Figure 5–3](#). You can use a PLL to fine tune the phase shift to the external memory to meet I/O timing requirements.

Figure 5–3. Typical Single Data Rate SDRAM Circuit

To constrain this interface, you must create a clock that is recognized by the external SDRAM; then you must set the I/O timing relative to that clock.

The following example shows how to constrain a PLL output clock and the setting of a Tcl variable for that clock as follows:

```
create_clock -period 20.000 -name ext_clk [get_ports {clk}]
derive_pll_clocks
set sdram_clk\
my_pll_inst|altpll_component|auto_generated|pll1|clk[0]
```

You can then use the **create_generated_clock** command to define a clock as recognized by the external memory. This generated clock automatically adds delays associated with routing to the clock output pin and the delay of the pin itself. You must also account for some board delay due to the PCB trace between the FPGA and SDRAM by using the **offset** option.

The following command shows the creation of the `sdram_clk_pin` generated clock derived from the output pin `sdram_clk` clock. A 0.5 ns offset accounts for PCB routing delay.

```
create_generated_clock -name sdram_clk_pin -source
$sdram_clk \
-offset 0.5 [get_ports {sdram_clk}]
```

There may be some uncertainty associated with the PCB delay not accounted for in this command. The uncertainty can be included in the I/O constraints that are specific to input or output and minimum or maximum delays.

The I/O constraints must be defined in relation to the data sheet for the external memory. Figure 5–4 shows an extract from a data sheet for an SDRAM device with the worst case input and output timing highlighted for a CAS latency of 3.

Figure 5–4. Extract from SDRAM Device Datasheet

AC Characteristics Parameter		Symbol	-6		-7		Units	Notes
			Min	Max	Min	Max		
Access time from CLK (pos. edge)	CL = 3	$t_{AC}^{(3)}$		5.5		5.5	ns	
	CL = 2	$t_{AC}^{(2)}$		7.5		8	ns	
	CL = 1	$t_{AC}^{(1)}$		17		17	ns	
Address hold time		t_{AH}	1		1		ns	
Address setup time		t_{AS}	1.5		2		ns	
CLK high-level width		t_{CH}	2.5		2.75		ns	
CLK low-level width		t_{CL}	2.5		2.75		ns	
Clock cycle time	CL = 3	$t_{CK}^{(3)}$	6		7		ns	23
	CL = 2	$t_{CK}^{(2)}$	10		10		ns	23
	CL = 1	$t_{CK}^{(1)}$	20		20		ns	23
CKE hold time		t_{CKH}	1		1		ns	
CKE setup time		t_{CKS}	1.5		2		ns	
CS#, RAS#, CAS#, WE#, DQM hold time		t_{CMH}	1		1		ns	
CS#, RAS#, CAS#, WE#, DQM setup time		t_{CMS}	1.5		2		ns	
Data-in hold time		t_{DH}	1		1		ns	
Data-in setup time		t_{DS}	1.5		2		ns	
Data-out High-Z time	CL = 3	$t_{HZ}^{(3)}$		5.5		5.5	ns	10
	CL = 2	$t_{HZ}^{(2)}$		7.5		8	ns	10
	CL = 1	$t_{HZ}^{(1)}$		17		17	ns	10
Data-out Low-Z time		t_{LZ}	1		1		ns	
Data-out hold time		t_{OH}	2		2.5		ns	

The mapping of external memory timing to FPGA I/O delays is shown in Table 5–1. This also shows whether the minimum or maximum PCB routing delay should be used, which must be added to the FPGA delay constraints.

Table 5–1. External Memory Timing (Part 1 of 2)

Memory Timing	FPGA Timing	PCB Routing
Max clock to out	Max input delay	Max
Min clock to out	Min input delay	Min

Table 5–1. External Memory Timing (Part 2 of 2)

Memory Timing	FPGA Timing	PCB Routing
Min setup	Max output delay	Max
Min hold	Min output delay (-ve) (1)	Min

Notes to Table 5–1:

- (1) The constraint for minimum output delay is actually 0 – Min hold.

You can use the **set_input_delay** and **set_output_delay** commands to set the I/O constraints. In the following examples, a common PCB routing delay of 0.5 ns ± 0.1 ns is used, which adds a 0.4 ns or 0.6 ns delay to the paths.

Example 5–1.

```
set_input_delay -clock sdram_clk_pin -max [expr 5.5 + 0.6] <ports>
set_input_delay -clock sdram_clk_pin -min [expr 2.5 + 0.4] <ports>
set_output_delay -clock sdram_clk_pin -max [expr 2.0 + 0.6] <ports>
set_output_delay -clock sdram_clk_pin -min [expr 1 - (1.0 + 0.4)] \
<ports>
```

In this example, <ports> represents a list of I/O ports for the relevant constraints as shown in Example 5-2:

Example 5–2.

```
set_output_delay -clock sdram_clk_pin -max [expr 2.0 + 1.2] \
[get_ports {cas_n ras_n cs_n we_n addr[*]}]
```

You can use multiple **set_input_delay** and **set_output_delay** commands to set different delays for different I/O.

Analyzing Tri-state Bridge and Asynchronous Devices

This section discusses the timing constraints associated with the Avalon tri-state bridge and asynchronous external devices, such as the CFI Flash and user tri-state components. These components typically have slower

performance requirements compared with the FPGA, and SOPC Builder generates logic within the interface to control timing across multiple clock cycles. You define the tri-state component's timing parameters by entering data for set up, wait, and hold times.

For interface types discussed above, the timing is controlled by a state machine that is generated based on set-up, wait, and hold settings you specify in the component editor. Because data sheet values for the FPGA are used in calculating the timing, the constraints simply ensure the data sheet timing is met. Adding these constraints ensures that issues associated with data sheet miss-interpretation and fitting problems that affect I/O timing are captured.

The TimeQuest timing analyzer uses constraints that are based upon the timing of the external device.

For further information on how to convert older FPGA-centric constraints into system-centric constraints, refer to:

Switching to the Quartus II TimeQuest Timing Analyzer chapter in volume 3 of the *Quartus II Handbook*.

Analyzing DDR and DDR2 Memories

When using DDR or DDR2 memory with Cyclone III and Stratix III devices, Altera recommends the DDR and DDR2 SDRAM High-Performance Controller MegaCore® functions. You can use the MegaWizard interface to parameterize these functions and generate timing constraints in the form of .sdc files. You must ensure that the constraints file associated with the MegaCore function is included in the project for timing analysis. You can add an .sdc file to the project by clicking **Add/Remove Files in Project** on the Project menu in the Quartus II software.



As these cores make use of the `derive_pll_clocks` command, conflicts may occur if your .sdc file also uses these constraints.



Refer to *TimeQuest Design Examples* on www.altera.com for further design examples. Also, the Application Note titled *Constraining and Analyzing Source-Synchronous Interfaces* describes source synchronous constraints for TimeQuest.

Referenced Documents

This chapter references the following documents:

- *AN 433: Constraining and Analyzing Source-Synchronous Interfaces*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *TimeQuest Design Examples*
- *TimeQuest Example: Basic SDC Example*

Document Revision History

Table 5–2 shows the revision history for this chapter.

<i>Table 5–2. Document Revision History</i>		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	Initial release.	Info moved from other chapters and consolidated here.

Introduction

This chapter describes the SOPC Builder component editor. The component editor provides a GUI to support the creation and editing of the `_hw.tcl` file that describes a component to SOPC Builder. You use the component editor to do the following:

- Specify the Verilog HDL or VHDL files that describe the modules that in your component hardware.
- Define the interfaces on the component and provide information about how the interface functions.
- Specify the signals for each of the component's interfaces, and define the behavior of each interface signal.
- Specify relationships between interfaces, such as determining which clock interface is used by a slave interface.
- Declare any parameters that alter the component structure or functionality, and define a user interface to let users parameterize instances of the component.



For information on using the component editor in a development flow, refer to *SOPC Builder Component Development Flow Using the Component Editor Overview*.



For information on Avalon component interfaces, refer to *Avalon Component Interfaces Supported in the Component Editor Version 7.2 and Later*.



For examples of changes to typical Avalon interfaces, refer to *Examples of Changes to Typical Avalon Interfaces for the Component Editor Version 7.2 and Later*.



For information on upgrading components, refer to *Upgrading Your Component with SOPC Builder Component Editor Version 7.2 and Later*.

For information on the use of the component editor, see the following sections:

- To start the component editor, refer to *“Starting the Component Editor” on page 6–2*.
- For information about specifying HDL files that describe a component, refer to *“HDL Files Tab” on page 6–3*.
- For information about specifying interface signals, refer to *“Signals Tab” on page 6–4*.

- For information about specifying the Avalon-MM type of interface signals, refer to [“Interfaces Tab” on page 6–6](#).
- For information about specifying parameters, refer to [“Component Wizard Tab” on page 6–6](#).
- To save a component, refer to [“Saving a Component” on page 6–7](#).
- For information about changing a component after it has been saved, refer to [“Editing a Component” on page 6–8](#).
- For information about the UI used to parameterize instances of your component, refer to [“Component GUI” on page 6–8](#).



For more information about components, refer to the [Component Interface Tcl Reference](#) chapter in volume 4 of the *Quartus II Handbook*. For more information about the Avalon-MM interface, refer to the [Avalon Interface Specifications](#).

Component Hardware Structure

The component editor creates components with the following characteristics:

- A component has one or more interfaces. Typically, an *interface* means an Avalon-MM master or slave. You can also specify exported component signals that appear at the top-level of the SOPC Builder system, which can be connected to logic outside the SOPC Builder system. The component editor lets you build a component with any combination of Avalon interfaces, which include:
 - Avalon-MM master and slave
 - Avalon-ST source and sink
 - Interrupt sender and receiver
 - Clock input and output
 - Nios II Custom Instruction Master and Slave Interfaces
 - Conduit (for exporting signals to the top level)
- Each interface is comprised of one or more signals.
- The component can represent logic that is instantiated inside the SOPC Builder system, or can represent logic outside the system with an interface to it on the generated system.

Starting the Component Editor

To start the component editor in SOPC Builder, on the File menu, click **New Component**. When the component editor starts, the **Introduction** tab displays, which describes how to use the component editor.

The component editor presents several tabs that group related settings. A message window at the bottom of the component editor displays warning and error messages.



Each tab in the component editor provides on-screen information that describes how to use the tab. Click the triangle labeled **About** at the top-left of each tab to view these instructions. You can also refer to Quartus® II Online Help for additional information about the component editor.

You navigate through the tabs from left to right as you progress through the component creation process.

HDL Files Tab

The table on the **HDL Files** tab specifies all the HDL files in the component. You use the **HDL Files** tab to specify Verilog HDL, or VHDL files that describe the component logic. Files are provided to downstream tools such as the Quartus II software and ModelSim in the same order as they appear in the table.

You can also use the component editor to define the interface to components outside the SOPCBuilder system. In this case, you do not provide HDL files. Instead, you use the component editor to manually define the hardware interface.

After you specify an HDL file, the component editor analyzes the file by invoking the Quartus II Analysis and Elaboration module. The component editor analyzes signals and parameters declared for all modules in the top-level file. If the file is successfully analyzed, the component editor's **Signals** tab lists all design modules in the **Top Level Module** list. If your HDL contains more than one module, you must select the appropriate top-level module from the **Top Level Module** list.

All files are managed in a single table, with checkboxes for synthesis and simulation. By default, all files are added with both checked. To add a simulation-only file, uncheck the synthesis checkbox for that file. To add a synthesis-only file, uncheck the simulation file. You use the top-level file checkbox to select the top-level file for synthesis. You can use the up and down arrows to specify the HDL file analysis order.



When the top-level module is changed, the component editor performs best-effort signal matching against the existing port definitions. If a port is absent from the module, it is removed from the port list.

Signals Tab

You use the **Signals** tab to specify the purpose of each signal on the top-level component module. If you specified a file on the **HDL Files** tab, the signals on the top-level module appear on the **Signals** tab.

If the component does not use an HDL file (a non-HDL based component) to interface to external logic that is Avalon compatible, you must manually add the signals that comprise the interface to the external logic. The **Interface** list also allows creation of a new interface. You can also use the Templates menu to quickly add typical interface signals to your signal list.

Each signal must belong to an interface and be assigned a legal signal type for that interface.

You assign each signal to an interface using the **Interface** list. In addition to Avalon Memory-Mapped and Streaming interfaces, components typically have clock interfaces, interrupt interfaces, and perhaps a conduit interface for exported signals.

Naming Signals for Automatic Type and Interface Recognition

The component editor recognizes signal types and interfaces based on the names of signals in the source HDL file, if they follow naming conventions. [Table 6–1](#) lists the signal naming conventions.

Table 6–1. Conventions of Automatically Recognized Signal Names	
Type of Signal	Name Convention
Signal associated with a specific interface	<code><interface type>_<interface name>_<signal type>[_n]</code>

For any value of `Interface Name` the component editor automatically creates an interface by that name, if necessary, and assigns the signal to it. The `Signal Type` must match one of the valid signal types for the type of interface. You can append `_n` to indicate an active-low signal. [Table 6–2](#) lists the valid values for `Interface Type`.

Table 6–2. Valid Values for <Interface Type> (Part 1 of 2)	
Value	Meaning
avs	Avalon-MM slave
avm	Avalon-MM master
ats	Avalon-MM tristate slave

Table 6–2. Valid Values for <Interface Type> (Part 2 of 2)

Value	Meaning
atm	Avalon-MM Tristate Master
aso	Avalon-ST Source
asi	Avalon-ST Sink
cso	Clock Output
csi	Clock Input
inr	Interrupt Receiver
ins	Interrupt Sender
cos	Coe
coe	Coe
ncm	Nios II Custom Instruction Master
ncs	Nios II Custom Instruction Slave

Example 6–1 shows a Verilog HDL module declaration with signal names that infer two Avalon-MM slaves.

Example 6–1. Verilog HDL Module With Automatically Recognized Signal Names

```

module my_slave_irq_component (
    // Signals for Avalon-MM slave port "s1" with irq
    csi_clockreset_clk, //clockreset clock interface
    csi_clockreset_reset_n, //clockreset clock interface
    avs_s1_address, //s1 slave interface
    avs_s1_read, //s1 slave interface
    avs_s1_write, //s1 slave interface
    avs_s1_writedata, //s1 slave interface
    avs_s1_readdata, //s1 slave interface
    ins_irq0_irq //irq0 interrupt sender interface
);

input csi_clockreset_clk;
input csi_clockreset_reset_n;
input [7:0]avs_s1_address;
input avs_s1_read;
input avs_s1_write;
input [31:0]avs_s1_writedata;
output [31:0]avs_s1_readdata;
output ins_irq0_irq;

/* Insert your logic here */

endmodule

```

Templates for Interfaces to External Logic

If you create an interface to external logic, you can use the Templates menu in the component editor to add a set of signals, such as the following templates:

- Avalon-MM Slave
- Avalon-MM Slave with Interrupt
- Avalon-MM Master
- Avalon-MM Master with Interrupt
- Avalon-ST Source
- Avalon-ST Sink

After adding a template, you can add or delete signals to customize the interface.

Interfaces Tab

The **Interfaces** tab allows you to configure the interfaces on your component, and specify a name for each interface. The interface name identifies the interface, and also appears in the SOPC Builder connection panel. The interface name is also used to uniquely identify any signals that are exposed on the top-level SOPC Builder system.

The **Interfaces** tab also allows you to configure the type and properties of each interface. For example, an Avalon-MM slave interface has timing parameters that you must set appropriately.

If you convert an older Avalon-MM slave to the new model, you may require three interfaces: a clock input, the Avalon slave, and an interrupt sender. A parameter in the interrupt sender must be set to reference the Avalon slave.

Component Wizard Tab

The **Component Wizard** tab provides options that affect the presentation of your new component.

Identifying Information

You can specify information that identifies the component as follows:

- **Folder**—Specifies the location of the component, determined by the location of the top-level HDL file.
- **Class Name**—Specifies the internal name of the component. Use the internal name when saving a system that contains an instance of this component, and when it is the name you use for the component type when you create a system using a script.
- **Display Name Version**—Specifies the user-visible name for this component in SOPC Builder.

- **Group**—Specifies which group in SOPC Builder displays your component in the list of available components. If you enter a previously unused group name, SOPC Builder creates a new group by that name.
- **Description**—Allows you to describe the component.
- **Created By**—Allows you to specify the author of the component.
- **Icon**—Allows you to associate the component with a file path relative to the component. The icon can be a **.jpg**, **.gif**, or **.png** file.
- **Parameters**—Allows you to specify the parameters for creating the component, as described below.

Parameters

The **Parameters** table allows you to specify the user-configurable parameters for the component.

If the top-level module of the component HDL declares any parameters (*parameters* for Verilog HDL, HDL, or *generics* for VHDL), those parameters appear in the **Parameters** table. The parameters are presented to you when you create or edit an instance of your component. Using the **Parameters** table, you can specify whether or not each parameter is user-editable.

The following rules apply to HDL parameters exposed via the component GUI:

- Editable parameters cannot contain computed expressions.
- If a parameter *N* defines the width of a signal, the signal width must be of the form *N*-1..0.
- When a VHDL component is used in a Verilog HDL SOPC Builder system, or vice versa, numeric parameters must be 32-bit decimal integers. When passing other numeric parameter types, unpredictable results occur.

Click **Preview the Wizard** at any time to see how the component GUI appears.

Saving a Component

You can save the component by clicking **Finish** on any of the tabs, or by clicking **Save** on the File menu. Based on the settings you specify in the component editor, the component editor creates a component description file with the file name *<name of top-level file>_hw.tcl*. The component editor saves the file in the same directory as the HDL file that describes the component's hardware interface. If you did not specify an HDL file, you can save the component description file to any location you choose.

You can relocate component files later. For example, you could move component files into a subdirectory and store it in a central network location so that other users can instantiate the component in their systems.

Editing a Component

After you save a component and exit the component editor, you can edit it in SOPC Builder. To edit a component, right-click it in the list of available components on the **System Contents** tab and click **Edit Component**.



You cannot edit components that were created outside of the component editor, such as Altera®-provided components.

If you edit the HDL for a component and change the interface to the top-level module, you need to edit the component to reflect the changes you made to the HDL.

Software Assignments

You can use Tcl commands to create software assignments. You can register any software assignment that you want, as arbitrary key-value pairs. The following example shows a typical Tcl API script:

Example 6–2. Typical Software Assignment with Tcl API Scripting

```
set_module_assignment name[ value]
set_interface_assignment name value ]
```

The result is that the assignments go into the **.sopcinfo** file, available for use for downstream components.

Component GUI

To edit component instance parameters, select a component in the System Contents pane of the SOPC Builder window and click **Edit**.

Referenced Documents

This chapter references the following documents:

- *Avalon Component Interfaces Supported in the Component Editor Version 7.2*
- *Avalon Interface Specifications*
- *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook*
- *Examples of Changes to Typical Avalon Interfaces for the Component Editor Version 7.2 and Later*
- *Nios II Software Developer's Handbook*

- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *SOPC Builder Component Development Flow Using the Component Editor Overview*
- *Upgrading Your Component with SOPC Builder Component Editor Version 7.2 and Later*

Document Revision History

Table 6–3 shows the revision history for this chapter.

Table 6–3. Document Revision History		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	Extensive edits to this chapter, including: <ul style="list-style-type: none"> ● Chapter renumbered. ● Added new section on software assignments. 	—
October 2007, v7.2.0	Updated several paragraphs describing the latest GUI.	—
May 2007, v7.1.0	<p>Updated all sections to reflect significant functional differences in version 7.1.</p> <p>Added section “Changes to Component Editor in Version 7.1” on page 5–2.</p> <p>Updated section “Component Editor Output” and “Re-editing Components” to accommodate new component structure with 7.1 release.</p> <p>Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon switch fabric” term to “system interconnect fabric.” Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.”</p> <p>Removed screen shots that simply reflect what user sees when using the tool without illustrating a particular point.</p> <p>Added Referenced Documents section which links to all referenced documents.</p> <p>Added statement that all simulation files, not just top-level file, must be added using the HDL files tab.</p>	The file structure of SOPC Builder components changed significantly in this release, which required substantial functional change to the component editor. This document changed significantly to reflect the functional changes. Updated to improve readability.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	No change from previous release.	—

Table 6–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
May 2006, v6.0.0	No change from previous release.	—
December 2005, v5.1.1	<ul style="list-style-type: none">• Added section “Naming Signals for Automatic Type and Interface Recognition” on page 5–4.• Added section “Templates for Interfaces to External Logic” on page 5–6.• Clarified operation of the Save command.• Updated all screenshots.	—
October 2005, v5.1.0	No change from previous release.	—
May 2005, v5.0.0	Initial release.	—

Overview

You define SOPC Builder components by declaring their properties and behaviors in a Hardware Tcl file (**_hw.tcl**). Each **_hw.tcl** file identified in SOPC Builder represents one component, instances of which you can add to an SOPC Builder system.

An SOPC Builder component is usually composed of the following four types of files:

- HDL files—define the component’s functionality as hardware.
- **_hw.tcl** file—describes the SOPC Builder related characteristics, such as interface behaviors. This file is created by the component editor.
- C-language files—define the component register map and driver software to allow programs to control the component.
- **Software Tcl File (_sw.tcl)**—used by the software build tools to use and compile the component driver code.

This chapter discusses the following information:

- [“Information in a Hardware Tcl File” on page 7-1](#)
- [“Component Phases” on page 7-2](#)
- [“Writing a Hardware Tcl File” on page 7-2](#)
- [“Overriding Default Behaviors” on page 7-7](#)
- [“Hardware Tcl Command Reference” on page 7-12](#)

Information in a Hardware Tcl File

A typical hardware Tcl file contains the following information:

- **Basic component information**—This includes information such as the component’s name, version, and description, a link to its documentation, and pointers to HDL implementation files for synthesis and simulation.
- **Parameter Declarations**—Components that have parameters declare each of the parameters to SOPC Builder. Parameters are values that you can set that affect how the component is implemented, such as the size of a memory. Properties of each parameter include the parameter’s name, whether or not it is visible, and, if visible, the text to display. When the system is generated, the parameters are typically applied to the component as Verilog HDL parameters or VHDL generics.

- **Interface Properties**—The interfaces of a component define how to connect it to the rest of the system, and determine how other components in the system interact with it. When you define interface properties, you declare which interfaces the component has and which signals make up part of each interface. You also define interface properties, such as wait-states for an Avalon Memory-Mapped (Avalon-MM) interface.

Component Phases

The following section describes the distinct phases in the creation and use of an SOPC Builder component.

- **Main Program**—This occurs when SOPC Builder first discovers a component and adds it to the component library. The `_hw.tcl` is executed, and the Tcl statements provide non-instance-specific information to SOPC Builder.
- **Edit**—After a component has been added to an SOPC Builder system, you can edit its parameters with the configuration wizard.
- **Validation**—Validation allows the component to generate error, warning, or information messages. Messages are generated when an instance of a component is created, when its parameters are changed, or when some other property of the system is changed.
- **Elaboration**—Elaboration occurs as SOPC Builder queries a component for its interface information. This typically happens immediately after validation, as well as before generation.
- **Generation**—During component generation, the module provides everything required to be compiled in the Quartus II software, or simulated in an HDL simulator. The required files may typically include VHDL or Verilog HDL RTL files, simulation models, timing constraints, and other information.

Writing a Hardware Tcl File

This section provides detailed information about `_hw.tcl` files, and describes the default behavior of the component in all five phases. The following example uses a simple UART with some simple parameterization.

Providing Basic Information

A typical `_hw.tcl` file first declares basic information, which allows SOPC Builder to provide you with basic information such as the name, location, and the files it includes.

Example 7- 1.

```
# The name and version of the component
set_module_property NAME "example_uart"
set_module_property VERSION "1.0"

# The name of the component to display in the library
set_module_property DISPLAY_NAME "Example Component"

# The component's description.
set_module_property DESCRIPTION "An Example Component"

# The component library group that component belongs to
set_module_property GROUP "Examples"
```

Declaring Parameters

The typical `_hw.tcl` file next declares the configuration parameters available to the system designer for parameterization of the core. You add a parameter and declare its properties as shown in the following example:

Example 7- 2.

```
# Declare the Baud Rate parameter as an integer with a
# default value of 9600.
add_parameter BAUD_RATE int 9600

# Display it as "Baud Rate" in the Parameter Editor.
set_parameter_property BAUD_RATE DISPLAY_NAME "Baud Rate (bps)"

# We only support three baud rates
set_parameter_property BAUD_RATE ALLOWED_RANGES {9600
19200 38400}
```

Declaring Interfaces

You next declare the interfaces to the component. Most interfaces require a clock input interface, so the clock interface may be declared first. To declare the interface you add each interface, declare its properties, and indicate which signals belong to it.

Example 7-3.

```

# Declare the clock sink interface, "clock_sink",
# type=clock, direction=sink
# add_interface clock_sink clock sink

# The clock interface has two signals, named "clk" and
# "reset_n",
# of types "clk" and "reset_n", both inputs of width 1
# bit.
add_interface_port clock_sink clk clk input 1
add_interface_port clock_sink reset_n reset_n input 1

# Declare the Avalon slave interface,
# name="avalon_slave_0", type=avalon, direction=slave,
# associated with the "clock_sink" clock interface.
add_interface avalon_slave_0 avalon slave clock_sink

# Set a number of properties about the Avalon Slave
# interface
set_interface_property avalon_slave_0 writeWaitTime 0
set_interface_property avalon_slave_0
addressAlignment DYNAMIC
set_interface_property avalon_slave_0 readWaitTime 1
set_interface_property avalon_slave_0 readLatency 0

# Declare all the signals that belong to my Avalon
# Slave interface
add_interface_port avalon_slave_0 readdata readdata
output 32
add_interface_port avalon_slave_0 read read input 1
add_interface_port avalon_slave_0 write write input 1
add_interface_port avalon_slave_0 waitrequest
waitrequest output 1
add_interface_port avalon_slave_0 address address
input 7
add_interface_port avalon_slave_0 writedata writedata
input 32

```

Adding Files and Guiding Generation

Tcl modules typically provide all of the information required for generation, providing downstream tools with all the information they need. You identify all of the HDL files or other files used by the component such as Synopsys Design Constraint (.sdc) files. You also identify the top-level HDL file, and top-level module and entity name, or both, and you identify which module is the top-level module.

Example 7-4.

```
# Add the HDL file to the component, to be used for
# synthesis and simulation.
add_file simple_uart.v {SYNTHESIS SIMULATION}

# Timequest file with Quartus timing constraints.
add_file simple_uart.sdc {SDC}

# The top-level HDL file that describes the file, and
# the name of the top-level module/entity.
set_module_property TOP_LEVEL_HDL_FILE simple_uart.v
                        set_module_property
TOP_LEVEL_HDL_MODULE simple_uart
```

Default Behavior

With a **_hw.tcl** file, such as the one described in the previous section, SOPC Builder has default behavior during edit time, validation time, elaboration time, and generation time. This section describes the default SOPC Builder behaviors for each of these phases. You can override this behavior, as described in the section titled “Overriding Default Behaviors”.

Edit Time Behavior

SOPC Builder’s default edit time behavior is to use all of the parameter definitions to show a default parameterization GUI. The properties of each parameter, shown in [Table 7-1](#), guide SOPC Builder when it builds the default GUI.

Table 7-1. Parameter Properties (Part 1 of 2)

Parameter Property	Use in the Default Parameter GUI
TYPE	<p>The type of the parameter determines how it is used in the generated HDL, as well how it is displayed, such as a checkbox, an edit field, or a drop-down list.</p> <p>You set the parameter type when adding the parameter with the <code>add_parameter</code> command.</p>
DISPLAY_NAME	Defines the name used to display the parameter.

Table 7–1. Parameter Properties (Part 2 of 2)

Parameter Property	Use in the Default Parameter GUI
ALLOWED_RANGES	A list of ranges, where each range is either a single value, or for integers, a range like "1:3", meaning 1 through 3, inclusive.
GROUP	Assigning parameters to groups allows the parameterization GUI to be logically divided.
VISIBLE	A parameter that has VISIBLE=0 is not shown on the parameterization GUI.
ENABLED	A parameter that has ENABLED=0 is shown as disabled. This is useful for parameters whose values are derived from other parameters.

Validation Time Behavior

SOPC Builder's default validation time behavior is to validate the current value of the component's parameters against its ALLOWED_RANGES property. If the system designer selects values that are outside the allowed ranges for a component, an error message displays.

The ALLOWED_RANGES property of each parameter is a list of ranges that the parameter can take on, where each range is a single value, or a range of values indicated as follows:

"start:end"

Table 7–2 shows some examples of values the ALLOWED_RANGES property can take.

Table 7–2. ALLOWED_RANGES Property

ALLOWED_RANGES Property	Meaning
{a, b, c}	"a", "b", or "c"
{1, 2, 4, 8, 16}	1, 2, 4, 8, or 16.
1:3	1 through 3, inclusive
1, 2, 3, 7:10	1, 2, 3, or 7 through 10 inclusive

Elaboration Behavior

During elaboration, SOPC Builder queries the component for current interfaces and their properties. All of the information required for typical components is available from the interface declarations and HDL. If some parameters affect port widths, SOPC Builder calls `quartus_map` as

needed to determine the correct port widths. Because calling `quartus_map` can be resource intensive, a component can set the `AFFECTS_PORT_WIDTHS` property to false for some or all parameters to avoid the call to `quartus_map` unless it is necessary.

Generation Behavior

At generation time, SOPC Builder performs one of the following:

- If the component defines the `TOP_LEVEL_MODULE` property, SOPC Builder creates a Verilog HDL or VHDL wrapper module to instantiate the top-level module, and applies the parameter as selected by the system designer. SOPC Builder does not apply parameters in the wrapper if they do not exist in the underlying HDL file.

OR

- If the component does not define the `TOP_LEVEL_MODULE` property, the module is not instantiated inside the SOPC Builder system and a wrapper file is not created. Rather, the interface to the module is exported to the top-level of the SOPC Builder system, and the module must be connected outside the system.

Overriding Default Behaviors

You can override each of the behaviors in the previous section by setting one of four callback properties. Each callback property provides the name of a callback function to call instead of, or in addition to, the default behavior.

Edit Callback

The edit callback is used to assign default values to the parameters of a custom component. An edit callback is defined by setting the `EDITOR_CALLBACK` module property to the name of your edit callback function, as shown in the following example. If the edit callback is defined, SOPC Builder calls the edit callback any time the parameterization GUI needs to be displayed, typically when the component is added to a system, or edited when the component is already in the system.

The Tcl interpreter used for custom components does not include Tk support. To display a GUI, the component's edit callback must call another program out of process to provide the desired GUI. Typically, an edit callback provides the current parameter values to the custom GUI

program via the command line and collects the new parameter values from the custom GUI via `stdout`. It then uses the **set_parameter_value** command to update SOPC Builder with the new parameter values.

Example 7- 5.

```
set_module_property EDITOR_CALLBACK my_editor

# Define Module parameters
add_parameter PARAMETER_ONE integer "32" "A parameter"
add_parameter CSR_ENABLED boolean "true" "Enable CSR
interface"

# My editor method
proc my_editor {
    # get parameter values
    set p1 [ get_parameter_value "PARAMETER_ONE" ]
    set csr [ get_parameter_value "CSR_ENABLED" ]

# Display UI, populated with current parameter
# values.
# The stdout returned by the UI program includes the
# new paramter values.
    set result = [exec my_component_ui.exe p1=$p1
csr=$csr]

    # use the fictional "parse_for_new_value" procedure
# to
    # parse the returned text for the new parameter
# values.
    set p1 [parse_for_new_value $result "p1"]
    set csr [parse_for_new_value $result "csr"]

# Return the new parameter values to SOPC Builder
    set_parameter_value "PARAMETER_ONE" $p1
    set_parameter_value "CSR_ENABLED" $csr
}
```

Validation Callback

A validation callback is defined by setting the `VALIDATION_CALLBACK` module property to be the name of the validation callback function, as shown below. When a validation callback is defined, the default validation against parameter ranges is executed, followed by the validation callback function.

You can use the validation callback to set the current values of derived parameters and to display messages based on the parameter values. The following example displays an error if you select a baud rate of 38400 and odd parity.

You can also use the validation callback to set the value of derived parameters. Derived parameters are parameters that are derived from other parameters; their values are not provided by the users of the components, and are not saved in the **.sopc** file. Marking a parameter as derived is done by setting the parameter's 'derived' property to TRUE. In the example below, BAUDRATE_PRESCALE is a derived parameter whose value is 1/16 of the value of the BAUDRATE parameter, calculated within the validation callback function.

Example 7- 6.

```
# Declare the validation callback.
set_module_property VALIDATION_CALLBACK
my_validation_callback

# Add the BAUDRATE_PRESCALE parameter
add_parameter BAUDRATE_PRESCALE int 600
set_parameter_property BAUDRATE_PRESCALE DERIVED
"true"

# The validation callback
proc my_validation_callback {

    # Get the current value of parameters we care about
    set baudrate [get_parameter_value BAUD_RATE]
    set parity    [get_parameter_value PARITY]

    # Display an error for invalid combinations.
    if {($baud_rate==38400) && ($parity=="odd")} {
        send_message error "Odd parity at 38400 bps is
not supported."
    }

    # Set the value of our derived parameter
    set baudrate_prescale [expr $baud_rate / 16]
    set_parameter_value BAUDRATE_PRESCALE
$baudrate_counter
}
```

Elaboration Callback

When an elaboration callback is defined, the elaboration callback is called to allow interface properties to be changed or new interfaces to be added. An elaboration callback is defined by setting the `ELABORATION_CALLBACK` module property to be the name of the elaboration callback function, as shown below. Interfaces defined in the main body of the `_hw.tcl` are added in the main program, and cannot be removed at elaboration time. The following example shows how an Avalon-MM Slave interface can be optionally present on an instance of the component, based on the `USE_STATUS_INTERFACE` parameter.

You can also use elaboration callbacks to update the files used by the component. For example, a given HDL file might only need to be included if a given parameter is set to `TRUE`. Any files added in the elaboration callback are in addition to those added in the main body of the file.

Example 7- 7.

```
# Declare the callback.
set_module_property ELABORATION_CALLBACK
my_elaboration_callback

# Add the USE_STATUS_INTERFACE parameter
add_parameter USE_STATUS_INTERFACE boolean

# The elaboration callback
proc my_elaboration_callback {

    # Get the current value of parameters we care about
    set use_status [get_parameter_value
USE_STATUS_INTERFACE]

    # Optionally add an interface and HDL file.
    if { $use_status==1 } {

        # Declare the status slave interface
        add_interface status_slave avalon slave clock_sink

        # Set interface properties
        set_interface_property status_slave writeWaitTime 0
        set_interface_property status_slave readWaitTime 1

        # Declare signals
        add_interface_port status_slave st_readdata readdata
output 32
        add_interface_port status_slave st_read read input 1
        add_interface_port status_slave st_write write input 1
        add_interface_port status_slave st_waitrequest
waitrequest output 1
```

```

        add_interface_port status_slave st_address address input 7
        add_interface_port status_slave st_writedata writedata
input 32

#Include the file that defines the status interface
add_file my_status_interface.v {SYNTHESIS SIMULATION}
    }
}

```

Generation Callback

When a generation callback is defined, SOPC Builder does not generate an HDL wrapper file to apply parameter values to the component. Instead, it calls the provided generation callback at generation time, allowing the component to programmatically generate its HDL. A generation callback is defined by setting the `GENERATION_CALLBACK` module property to be the name of the generation callback function, as shown below.

Generation callbacks typically retrieve the current value of the component's parameters, as well as the generation properties that guide the generation process, and then generate the HDL files and supporting files in Tcl, or by calling an external program. Finally, the callback reports the required files to SOPC Builder with the `add_files` command. Any files added in the generation callback are in addition to the files added in the main body of the file.

Table 7–3 describes the generation properties.

Table 7–3. Generation Properties	
ALLOWED_RANGES Property	Meaning
HDL_LANGUAGE	This property indicates the preferred generation HDL language, and is either Verilog or VHDL. A component should generate its HDL in the desired language if possible, and in either language if not possible.
OUTPUT_DIRECTORY	This property indicates the directory in which the HDL and other files are saved. Files must be generated into this directory.
OUTPUT_NAME	This property indicates the module name of the generated module. If the system designer decides that the component instance has the name <code>comp_0</code> , then the <code>OUTPUT_NAME</code> generation property is <code>comp_0</code> , and the generation callback must generate a module or entity in the output directory with name <code>comp_0</code> .

The following shows a generation callback example.

Example 7- 8.

```
set_module_property generationCallback my_generate

# My generation method
proc my_generate {} {
    send_message "info" "Starting Generation"

    # get generation settings
    set language [ get_generation_setting "HDL_LANGUAGE" ]
    set outdir [get_generation_setting "OUTPUT_DIRECTORY" ]
    set outputname [get_generation_setting "OUTPUT_NAME" ]

    # get parameter values
    set pl [ get_parameter_value "PARAMETER ONE" ]
    set csr [ get_parameter_value "CSR_ENABLED" ]

    # Do HDL generation with perl
    exec perl my_generate.pl lang=$language dir=$outdir name=$outputname
    pi=$pl csr=$csr

    add_file "${outputname}.v" SYNTHESIS
    add_file "${outputname}_sim.v" SIMULATION
}
```

Hardware Tcl Command Reference

This section provides a reference for all hardware Tcl commands, as follows:

- [“Module Definition” on page 7–13](#)
- [“Parameters” on page 7–19](#)
- [“Interfaces and Ports” on page 7–23](#)
- [“Generation” on page 7–27](#)

The description of each command indicates at what time it is available: in the main body of the program (main), or during the edit, validation, elaboration, and generation callback, or any combination above.

Module Definition

get_module_properties

Description: Returns all the available module properties as a list of strings. The `get_module_property` and `set_module_property` commands are used to get and set the values of these. Some properties can only be set at certain times.

Returns: String []

Usage: `list_of_module_properties`

Command Availability:

- main program

The available module properties, their use, and when they can be set is described in [Table 7–4](#).

Table 7–4. get_module_properties (Part 1 of 3)			
Property Name	Property Type	Can Be Set	Description
NAME	String	Main program	The name of the module, such as “my_sopc_component”.
DISPLAY_NAME	String	Main program	The name to display when referencing the module, such as “My SOPC Component”.
VERSION	String	Main program	The module’s version, such as “8.0”.
AUTHOR	String	Main program	The module’s author
DESCRIPTION	String	Main program	The description of the module, such as “Example SOPC Builder Module”.
GROUP	String	Main program	The component group that the module belongs to, such as “Example Components”.
ICON_PATH	String	Main program	A path to an icon to display in the module parameter’s editor.
DATASHEET_URL	String	Main program	A path to the module’s data sheet.

Table 7-4. *get_module_properties* (Part 2 of 3)

Property Name	Property Type	Can Be Set	Description
EDITABLE	Boolean	Main program	Indicates if the component is editable in the component editor.
MODULE_TCL_FILE		Can only be read, not set	The path to the _hw.tcl file.
MODULE_DIRECTORY		Can only be read, not set	The directory containing the _hw.tcl file.
TOP_LEVEL_HDL_FILE	String	Main program, validation, generation	Indicates which of the files added by the "add_file" command contains the module's top-level HDL.
TOP_LEVEL_HDL_MODULE	String	Main program, validation, generation	Indicates the name of the module's top-level HDL file.
INstantiate_in_System_Module	Boolean	Main program	When 'false', the instances of the module are not included in the generated switch fabric. Instead, interfaces to the module are exported out of the top-level system.
Validation_Callback	String	Main program	The name of the validation callback to call. No validation callback function is called if this property is not set.
Editor_Callback	String	Main program	The name of the edit callback to call. No validation callback function is called if this property is not set.
Elaboration_Callback	String	Main program	The name of the elaboration callback to call. No validation callback function is called if this property is not set.
Generation_Callback	String	Main program	The name of the generation callback to call. No validation callback function is called if this property is not set.

Table 7–4. *get_module_properties* (Part 3 of 3)

Property Name	Property Type	Can Be Set	Description
SIMULATION_MODEL_IN_VERILOG	Boolean	Main program	When true, SOPC Builder automatically creates a Verilog HDL simulation model for the component.
SIMULATION_MODEL_IN_VHDL	Boolean	Main program	When true, SOPC Builder automatically creates a VHDL simulation model for the component.

get_module_property

Description: Returns the value of a single module property.

Returns: String or Boolean, depending on the property

Usage: `get_module_property [propertyName]`

Command Availability:

- main program
- validation callback
- edit callback
- elaboration callback
- generation callback

Example: `set my_name [get_module_property NAME]`

set_module_property

Description: Allows you to set the property value of a module.

Returns: <void>

Usage: `set_module_property [propertyName] [propertyName]`

Command Availability:

- main program
- validation callback (some properties; see [Table 7–4](#))
- generation callback (some properties; see [Table 7–4](#))

Example: `set_module_property VERSION "8.0"`

add_file

Description: Adds a synthesis, simulation, or **.sdc** file to the module. Files added in the main program cannot be removed. Adding files in the elaboration callback or the generation callback allows the included files to be a function of the parameter set, or to be a result of generation. Files added in callbacks are in addition to any files added at main program. File properties can be included when the file is added. These properties include the following:

- SIMULATION—Indicates that the file is used for HDL simulation.
- SYNTHESIS—Indicates that the file is an HDL file for synthesis.
- SDC—Indicates that the file includes TimeQuest timing analyzer timing constraints.
- QIP—Indicates that the file is a Quartus II IP file.

Return: <void>

Usage: `add_file fileName [fileProperties]`

Command Availability:

- main program
- elaboration callback
- generation callback

Example: `add_file my_component.v {SIMULATION SYNTHESIS}`

get_files

Description: Returns a list of all the file names added to the module.

Returns: `String[]`

Usage: `get_files []`

Command Availability:

- main program
- validation callback
- edit callback
- elaboration callback
- generation callback

get_file_property

Description: Returns the value of a single file property. The file name passed to the function can be a partial filename as long as it is unique. For example, if the full filename is `"/components/my_file.v"`, `"my_file.v"` is sufficient. The available properties are described in `"add_files"`.

Returns: Boolean

Usage: `get_file_property [filename] [propertyName]`

Command Availability:

- main program
- validation callback
- edit callback
- elaboration callback
- generation callback

Example: `get_file_property my_file.v SYNTHESIS`

set_file_property

Description: Sets the value of a single file property. The filename passed to the function can be a partial filename as long as it is unique. For example, if the full file name is `"/components/my_file.v"`, `"my_file.v"` is sufficient. The available properties are described above in `"add_files"`.

Returns: `<void>`

Usage: `set_file_property [fileName] [propertyName]
[propertyName]`

Command Availability:

- main program
- elaboration callback
- generation callback

Example: `set_file_property "my_file.v" SYNTHESIS true`

send_message

Description: Sends a message to the user of the component, where each message includes a message level that is one of the following:

- *Error*: Provides an error message to the user. The system cannot be generated while there are error messages.
- *ToDoError*: Provides a ToDo message to the user, telling them there is something they need to do before the system can be generated.
- *Info*: Provides an informational message to the user.
- *Warning*: Provides a warning message to the user.
- *Progress*: Provides process information to the user, typically used during the generation callback.
- *Debug*: Only displayed to the user when debug mode is enabled.

Returns: <void>

Usage: `send_message [messageLevel] [messageText]`

Command Availability:

- main program
- validation callback
- generation callback

Example: `send_message Error "param1 must be greater than param2."`

Parameters

add_parameter

Description: Adds a parameter to your module. Parameters allow users of the module to affect its operation in the same manner as Verilog HDL parameters or VHDL generics. The name of the parameter is determined by the component author, and its type is one of the following:

- Integer
- Boolean
- Std_logic (VHDL based components only)
- Std_logic_vector (VHDL based components only)
- String

Returns: <void>

Usage: `add_parameter [parameterName] [parameterType]
[defaultValue description]`

Command Availability:

- main program

Example: `add_parameter "seed" integer 17 "The seed to use
for data generation."`

get_parameters

Description: Returns the list of parameters that have been previously defined by `add_parameter`.

Return: `String[]`

Usage: `get_parameters`

Command Availability:

- main program
- validation callback
- edit callback
- elaboration callback
- generation callback

get_parameter_properties

Description: Returns a list of all the available `parameter_properties` as a list of strings. The `get_parameter_property` and `set_parameter_property` commands are used to get and set the

values of these properties, respectively. Parameter properties can be set at different times, depending on the property. The available parameter properties, their use, and when they can be set, is described in [Table 7–5](#)

Table 7–5. <i>get_parameter_properties</i>			
Property Name	Property Type	Can Be Set	Description
DISPLAY_NAME	String	Main program	The string used when displaying the parameter.
ALLOWED_RANGES	String	Main program	Indicates the range or ranges that the parameter value can be. For integers, each range is a single value like "17", or an inclusive range, such as [11:15]. For strings and other types, each range is a single value.
GROUP	String	Main program	The parameter group that the parameter belongs to.
AFFECTS_PORT_WIDTHS	Boolean	Main program	If a parameter that does not affect port widths is changed, quartus_map is called to determine the new port widths. Identifying parameters that do not affect port widths can avoid unnecessary calls to quartus_map, improving performance. This parameter is assumed to be 'true' if not set.
VISIBLE	Boolean	Main program validation callback	Indicates whether or not to display the parameter in the parameterization GUI.
ENABLED	Boolean	Main program validation callback	When 'false', the parameter is disabled, meaning that it is displayed, but greyed out, on the parameterization GUI.

Return: String []

Usage: get_parameter_properties

Command Availability:

- main program
- validation callback
- edit callback
- elaboration callback
- generation callback

get_parameter_property

Description: Returns a single parameter property.

Return: Depends on property. (see [Table 7-5](#))

Usage: `get_parameter_property [parameterName]
[propertyName]`

Command Availability:

- main program
- validation callback

Example: `get_parameter_property "parameter1" "GROUP"`

set_parameter_property

Description: Sets a single parameter property.

Return: <void>

Usage: `set_parameter_property [parameterName]
[propertyName] [value]`

Command Availability:

- main program
- validation callback

Example: `set_parameter_property "parameter1"
ALLOWED_RANGES {1, 2, 4, 8}`

get_parameter_value

Description: Returns the current value of a parameter defined previously with the `add_parameter` command.

Returns: Parameter type

Usage: `get_parameter_value [parameterName]`

Command Availability:

- validation callback
- edit callback
- elaboration callback

- generation callback

Example: `get_parameter_value "parameter1"`

set_parameter_value

Description: Sets a parameter value. Typically, the value of derived parameters is set during the validation callback based on the value of other parameters.

Returns: <void>

Usage: `set_parameter_value [parameterName] [value]`

Command Availability:

- validation callback

Example: `set_parameter_value "parameter1" 4,`

Interfaces and Ports

add_interface

Description: Adds an interface to your module. The name of the interface is determined by you, the component author. Interface types and directions are listed in [Table 7-6](#).

Table 7-6. add_interface Types and Directions	
Type	Directions
avalon	master, slave
avalon_tristate	slave
avalon_streaming	source, sink
interrupt	sender, receiver
conduit	start
clock	source, sink
nios_custom_instruction	slave

Returns: <void>

Usage: `add_interface interfaceName interfaceType
direction [associatedClock]`

Command Availability:

- main program
- elaboration callback

Example: `add_interface s0 avalon slave clock0`

get_interfaces

Description: Returns the list of interfaces that have been previously defined by `add_interface`.

Returns: `String[]`

Usage: `get_interfaces`

Command Availability:

- main program
- elaboration callback

get_interface_properties

Description: Returns a list of all the available interface properties for the specified interface, as a list of strings.



The properties available for each interface type are different for every interface type. Refer to the *Avalon Interface Specifications*.

Return: String[]

Usage: `get_interface_properties [interfaceName]`

Command Availability:

- main program
- elaboration callback

Example: `get_interface_properties "s0"`

get_interface_property

Description: Returns a single interface property from an interface.

Return: Depends on the property

Usage: `get_interface_property [interfaceName]
[propertyName]`

Command Availability:

- main program
- elaboration callback

Example: `get_interface_property "s0" "readWaitTime"`

set_interface_property

Description: Sets a single interface property for an interface.

Returns: <void>

Usage: `set_interface_property interfaceName
apropertyName value`

Command Availability:

- main program

- elaboration callback

Example: `set_interface_property "s0" "readWaitTime" 2`

add_interface_port

Description: Adds a port to an interface on your module. The name of the port is determined by you, the component author. The port roles that can be a member of any given interface depend on the interface itself.



Refer to the *Avalon Interface Specifications* for details.

portDirection can be one of Input, Output, Bidir, and for VHDL, buffer. The width is the width of the port, in bits.

Return: <void>

Usage: `add_interface_port [interfaceName] [portName]
[portRole direction] [Width]`

Command Availability:

- main program
- elaboration callback

Example: `add_interface_port s0 s0_rdata readdata output
32`

get_interface_ports

Description: Returns the names of all of the ports that have been added to a given interface. If the interface name is omitted, all ports for all interfaces are returned.

Returns: String[]

Usage: `get_interface_ports [interfaceName]`

Command Availability:

- main program
- elaboration callback

get_port_properties

Description: Returns a list of all available port properties. Available port properties are those listed in [Table 7–7](#).

Table 7–7. Available Port Properties		
Property Name	Type	Description
DIRECTION	Direction	Must be one of INPUT, OUTPUT, BIDIR, BUFFER
WIDTH	Integer	The width of the port.
TERMINATION	Boolean	When true, instead of connecting the port to the SOPC Builder system, it is left unconnected (output & bidir) or set to a fixed value (inputs). Has no effect for components that implement a generation callback instead of using the default wrapper generation.
TERMINATION_VALUE	Long	The fixed value for which to set input terminated ports.

Returns: String []

Usage: `get_port_properties portName`

Command Availability:

- main program
- elaboration callback

Example: `get_port_properties "s0"`

get_port_property

Description: Allows you to get a port property.

Returns: Depends on the property.

Usage: `get_port_property portName propertyName`

Command Availability:

- main program
- elaboration callback

Example: `get_port_property "s0_rdata" WIDTH`

set_port_property

Description: Sets a single port property.

Returns: <void>

Usage: set_port_property portName propertyName value

Command Availability:

- main program
- elaboration callback

Example: set_port_property "s0_rdata" WIDTH 32

Generation

get_generation_properties

Description: Returns a list of all the available generation properties as a list of strings. The get_generation_property command is used to get the values of these properties. These properties cannot be changed by the module. The properties and their use are described in [Table 7–8](#).

Table 7–8. get_generation_properties

PropertyName	Type	Description
HDL_LANGUAGE	Enum	The HDL language to Generate. Can be "vhdl" or "verilog", in lowercase. If the module cannot generate in the specified language, then generating in the other language is acceptable.
OUTPUT_DIRECTORY	File	The location in which files are generated.
OUTPUT_NAME	String	The top-level file name and entity to be generated. If the OUTPUT_NAME is "module_0", and the HDL_LANGUAGE is "VERILOG", then the file "module_o.v" should be generated, and it should contain module "module_0".

Returns: String []

Usage: get_generation_properties

Command Availability:

- main program
- generation callback

get_generation_property

Description: Returns the value of a single generation property.

Returns: String or Boolean, depending on the property

Usage: `get_generation_property propertyName`

Command Availability:

- generation callback

get_project_property

Description: Returns the value of a single project property. These properties cannot be changed by the module. The properties and their use are described in [Table 7–9](#).

Table 7–9. get_project_property (Part 1 of 2)		
Property Name	Type	Description
QUARTUS_ROOTDIR	String	The value of the \$QUARTUS_ROOTDIR environment variable.
QUARTUS_PROJECT_DIRECTORY	String	The path to the current project directory.
QUARTUS_PROJECT_NAME	String	The name of the current Quartus II project.
DEVICE_FAMILY_NAME	Enum	The name of the current device family, where the device family is one of the following: STRATIX STRATIXII STRATIXIIGX STRATIXIIGXLITE STRATIXGX STRATIXIII STRATIXIV CYCLONE CYCLONEII CYCLONEIII HARDCOPY HARDCOPYII HARDCOPYIII MAXII APEX20KE APEX20KC APEXII ACEX1K

Table 7–9. *get_project_property* (Part 2 of 2)

Property Name	Type	Description
DEVICE_FAMILY_FEATURES	Enum []	The features supported by the current device family, where the features include the following: M512_MEMORY M4K_MEMORY M9K_MEMORY M144K_MEMORY MRAM_MEMORY MLAB_MEMORY ESB EPCS DSP EMUL HARDCOPY LVDS_IO ADDRESS_STALL TRANSCEIVER_3G_BLOCK TRANSCEIVER_6G_BLOCK DSP_SHIFTER_BLOCK

Returns: Depends on the property

Usage: `get_project_property propertyName`

Command Availability:

- generation callback

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Nios II Software Build Tools Reference* chapter in section 4 of the *Nios II Software Developer's Handbook*

Document Revision History

Table 7–10 shows the revision history for this chapter.

<i>Table 7–10. Document Revision History</i>		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v 8.0.0	<ul style="list-style-type: none">• Added new Editing <code>_hw.tcl</code> commands and debug commands sections.• Changed chapter title from "Building a Component Interface with Tcl Scripting Commands" to "Component Interface Tcl Reference".	—
October 2007, v7.2.0	Major reorganization of chapter to better reflect work flow when using tcl scripting. Includes new commands, properties, and parameters.	—
May 2007, v7.1.0	Initial release.	—

Introduction

This chapter helps identify the files you must include when archiving an SOPC Builder project. With this information, you can archive the SOPC Builder system. You may want to archive your SOPC Builder system for one of the following reasons:

- To place an SOPC Builder design under source control
- To create a backup
- To bundle a design for transfer to another location

To use this information, you must decide what source control or archiving tool to use, and you must know how to use it. This chapter describes how to find and identify the files that you must include in an archived SOPC Builder design. Refer to [“Required Files” on page 8-2](#).

Limitations

This chapter provides information about archiving SOPC Builder systems, including their Nios® II software applications, if any. If your SOPC Builder system does not contain a Nios II processor, you can disregard information about archiving Nios II software applications.

This chapter does not cover archiving SOPC Builder *components*, for two reasons:

- SOPC Builder components can be recovered, if necessary, from the original Quartus® II and Nios II installations.
- If your SOPC Builder system was developed with an earlier version of the Quartus II software and Nios II Embedded Design Suite (EDS), when you restore it for use with the current version, you normally use the current, installed components.

If your SOPC Builder system was developed with an earlier version of the Quartus II Complete Design Suite and you restore it for use with the current version, the regenerated system is functionally identical to the original system. However, there might be differences in details such as Quartus II timing, component implementation, or HAL implementation.



For details of version changes, refer to the [Quartus II Reference Documentation](#).

To ensure that you can regenerate your exact original design, maintain a record of the tool and IP version(s) originally used to develop the design. Retain the original installation files or media in a safe place.

The archival process addressed by this chapter is different than Quartus II project archiving. A Quartus II project archive contains the complete Quartus II project, including the SOPC Builder module. The Quartus II software adds all HDL files to the archive, including HDL files generated by SOPC Builder, although these files are not strictly necessary, if you regenerate the design files afterwards. A Quartus II project archive also archives the Quartus II IP (.qip) file.

This chapter is only concerned with archiving the SOPC Builder system, without the generated HDL files.



For more details about archiving Quartus II projects, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

Required Files

This section describes the files required to archive an SOPC Builder system and its associated Nios II software projects (if any). This is the minimum set of files needed to completely recompile an archived system, both the SRAM Object File (.sof).



If you have Nios II software projects, archive them together with the SOPC Builder system on which they are based. For more details about archiving Nios II designs, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

SOPC Builder Design Files

The files listed in [Table 8–1](#) are located in the Quartus II project directory.

Table 8–1. Files Required for an SOPC Builder System (Part 1 of 2)		
File description	File name	Write permission required? (1)
SOPC Builder system description	<sopc_builder_system>.sopc	Yes
SOPC Builder classic system description for generation (1)	<sopc_builder_system>.ptf	Yes
SOPC Builder report file	<sopc_builder_system>.sopcinfo	Yes
All non-generated HDL source files (2)	for example: top_level_schematic.bdf, customlogic.v	No

Table 8–1. Files Required for an SOPC Builder System (Part 2 of 2)

File description	File name	Write permission required? (1)
Quartus II project file	<project_name>.qpf	No
Quartus II settings file	<project_name>.qsf	Yes

Notes to Table 8–1:

- (1) The <sopc_builder_system>.ptf file is only required if you intend to edit or view the system in a version of SOPC Builder prior to version 7.1 and must also be writable to generate a system.
- (2) Include all HDL source files not generated by SOPC Builder. This includes HDL source files you create or copy from elsewhere. To identify a file generated by SOPC Builder, open the file and look for the following header:
Legal Notice: (C)<year> Altera Corporation. All rights reserved.

Many source control tools mark local files read-only by default. In this case, you must override this behavior. You do not have to check the files out of source control unless you are modifying the SOPC Builder design or Nios II software project.

Referenced Documents

This chapter references the following documents:

- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Quartus II Reference Documentation*

Document Revision History

Table 8–2 shows the revision history for this chapter.

<i>Table 8–2. Document Revision History</i>		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	Renumbering from Chapter 7 to 8.	—
October 2007, v7.2.0	No change from previous release.	—
May 2007, v7.1.0	<ul style="list-style-type: none"> Chapter 7 was previously chapter 6 Added information about new .sopc file type to Table 8–1 Added information about legacy .ptf file type to Table 8–1 Added Referenced Documents section Added reference to new Common BSP Tasks chapter for archiving of Tcl projects 	Updates to this chapter include replacing the legacy .ptf file type with the new .sopc file type.
March 2007, v7.0.0	<ul style="list-style-type: none"> No change from previous release 	—
November 2007, v6.1.0	<ul style="list-style-type: none"> No change from previous release 	—
May 2006, v6.0.0	Initial release.	—



Section II. Building Systems with SOPC Builder

This section uses example designs to show you how to build a system or component. Chapters in this section serve to answer the question. This chapter refers to design examples that you can download free from **www.altera.com**. Design file hyperlinks are located with individual chapters linked from the Altera web site.

This section includes the following chapters:

- Chapter 9, SOPC Builder Memory Subsystem Development Walkthrough
- Chapter 10, SOPC Builder Component Development Walkthrough



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

Introduction

Most systems generated with SOPC Builder require memory. For example, embedded processor systems require memory for software, while digital signal processing (DSP) systems require memory for data buffers. Many systems use multiple types of memories. For example, a processor-based DSP system can use off-chip synchronous dynamic random access memory (SDRAM) to store software, and on-chip RAM for fast access to data buffers. You can use SOPC Builder to integrate almost any type of memory into your system.

This chapter uses design examples to describe how to build a memory subsystem as part of a larger system created with SOPC Builder. This chapter focuses on the following kinds of memory most commonly used in SOPC Builder systems for:

- “On-Chip RAM and ROM” on page 9–7
- “EPCS Serial Configuration Device” on page 9–11
- “(SDR) SDRAM” on page 9–13
- “(DDR) SDRAM” on page 9–17
- “(DDR2) SDRAM” on page 9–17
- “Parallel Flash Loader” on page 9–18
- “Off-Chip SRAM and Flash Memory” on page 9–18

This chapter assumes that you are familiar with the following:

- Creating FPGA designs and making pin assignments with the Quartus® II software. For details, refer to the *Introduction to the Quartus II Software manual*.
- Building simple systems with SOPC Builder. For details, refer to the *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon® interfaces. You do not need extensive knowledge of the Avalon interfaces, such as transfer types or signal timing. However, to create your own custom memory subsystem with external memories, you need to understand the Avalon Memory-Mapped (Avalon-MM) interface. For details, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook* and the *Avalon Interface Specifications*.



Refer to the *Memory System Design* chapter in the *Embedded Design Handbook* for additional information on the efficient use of memories in SOPC Builder systems.

Example Design

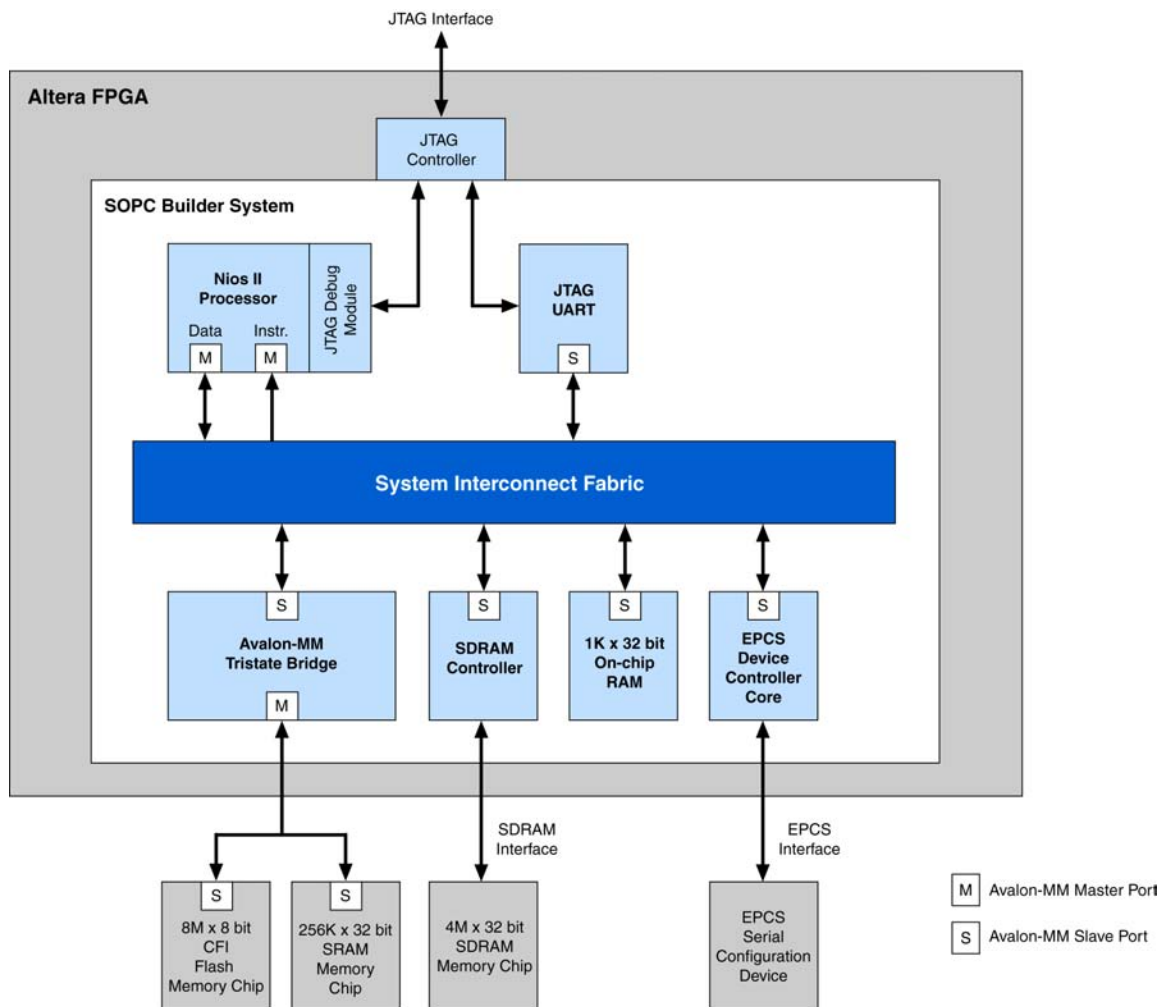
This chapter demonstrates the process for building a system that contains one of each type of memory as shown in [Figure 9-1](#). Each section of the chapter builds on previous sections, culminating in a complete system.

By following the example design in this chapter, you learn how to create a complete customized memory subsystem for your system or design. The memory components in the example design are independent. For a custom system, you only need to instantiate the memories you need. You can also create multiple instantiations of the same type of memory, limited only by on-chip memory resources or FPGA pins to interface with off-chip memory devices.

Example Design Structure

[Figure 9-1](#) shows a block diagram of the example system.

Figure 9–1. Example Design Block Diagram



In Figure 9–1, all blocks shown below the system interconnect fabric comprise the memory subsystem. For demonstration purposes, this system uses a Nios® II processor core to master the memory devices, and a JTAG UART core to communicate with the host PC. However, the memory subsystem could be connected to any master component, located either on-chip or off-chip.

Example Design Starting Point

The example design consists of the following elements:

- A Quartus II project named **quartus2_project**. A Block Design File (.bdf) named **toplevel_design**. **toplevel_design** is the top-level design file for **quartus2_project**. **toplevel_design** instantiates the SOPC Builder system, as well as other pins and modules required to complete the design.
- An SOPC Builder system named **sopc_memory_system**. **sopc_memory_system** is a subdesign of **toplevel_design**. **sopc_memory_system** instantiates the memory components and other SOPC Builder components required for a functioning SOPC Builder system.

This discussion assumes that the **quartus2_project** already exists, **sopc_memory_system** has been started in SOPC Builder, and the Nios II core and the JTAG UART core are already instantiated. This example design uses the default settings for the Nios II core and the JTAG UART core; these settings do not affect the rest of the memory subsystem.

Hardware and Software Requirements

To build a memory subsystem similar to the example design in this chapter, you need the following:

- Quartus II Software version 5.0 or higher—Both Quartus II Web Edition and the fully licensed version support this design flow.
- Nios II Embedded Design Suite (EDS) version 5.0 or higher—Both the evaluation edition and the fully licensed version support this design flow. The Nios II EDS provides the SOPC Builder memory components described in this chapter. It also provides several complete example designs which demonstrate a variety of memory components instantiated in working systems.



The Quartus II Web Edition software and the Nios II EDS, Evaluation Edition are available free for download from the Altera® website. Visit www.altera.com/download. Also, for further reference, see the *Design Example*.

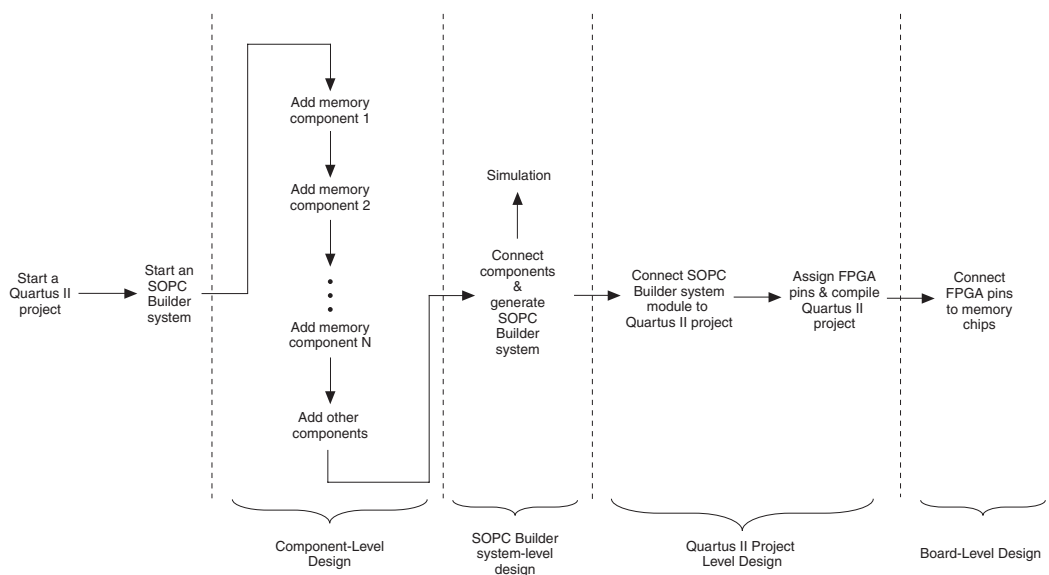
This chapter does not describe downloading and verifying a working system in hardware. Therefore, there are no hardware requirements for the completion of this chapter. However, the example memory subsystem has been tested in hardware.

Design Flow

This section describes the design flow for building memory subsystems with SOPC Builder, which is similar to other SOPC Builder designs. After starting a Quartus II project and an SOPC Builder system, there are five steps to completing the system, as shown in [Figure 9–2](#):

1. Component-level design in SOPC Builder
2. SOPC Builder system-level design
3. Simulation
4. Quartus II project-level design
5. Board-level design

Figure 9–2. Design Flow



Component-Level Design in SOPC Builder

In this step, you specify which memory components to use and configure each component to meet the needs of the system. All memory components are available from the **Memory and Memory Controllers** category in the SOPC Builder list of available components.

SOPC Builder System-Level Design

In this step, you connect components together and configure the SOPC Builder system as a whole. Like the process of adding non-memory SOPC Builder components, you use the SOPC Builder **System Contents** tab to do the following:

- Rename the component instance (optional).
- Connect the memory component to masters in the system. Each memory component must be connected to at least one master.
- Assign a base address.
- Assign a clock domain. A memory component can operate on the same or different clock domain as the master(s) that access it.

Simulation

In this step, you verify the functionality of the SOPC Builder system. For systems with memories, this step depends on simulation models for each of the memory components, in addition to the system testbench generated by SOPC Builder. Refer to “[Simulation Considerations](#)” for more information.

Quartus II Project-Level Design

In this step, you integrate the SOPC Builder system with the rest of the Quartus II project, which includes wiring the SOPC Builder system to FPGA pins, and wiring the SOPC Builder system to other design blocks (such as other HDL modules) in the Quartus II project.



In the example design in this chapter, the SOPC Builder system comprises the entire FPGA design. There are no other design blocks in the Quartus II project.

Board-Level Design

In this step, you connect the physical FPGA pins to memory devices on the board. If the SOPC Builder system interfaces with off-chip memory devices, you must make board-level design choices.

Simulation Considerations

SOPC Builder can automatically generate a testbench for register transfer level (RTL) simulation of the system. This testbench instantiates the SOPC Builder system and can also instantiate memory models for external memory components. The testbench is plain text HDL, located at

the bottom of the top-level SOPC Builder system HDL design file. To explore the contents of the auto-generated testbench, open the top-level HDL file and search on keyword `test_bench`.

Generic Memory Models

The memory components described in this chapter, except for the SRAM, provide generic simulation models. Therefore, it is very easy to simulate an SOPC Builder system with memory components immediately after generating the system.

The generic memory models store memory initialization files, such as Data (**.dat**) and Hexadecimal (**.hex**) files, in a directory named *<Quartus II project directory>/<SOPC Builder system name>_sim*. When generating a new system, SOPC Builder creates empty initialization files. You can manually edit these files to provide custom memory initialization contents for simulation.



For Nios II processor designs, the Nios II integrated development environment (IDE) generates memory initialization contents automatically.

Vendor-Specific Memory Models

You can also manually connect vendor-specific memory models to the SOPC Builder system. In this case, you must manually edit the testbench and connect the vendor memory model. You might also need to edit the vendor memory model slightly for time delays. The SOPC Builder testbench assumes zero delay.

On-Chip RAM and ROM

Altera FPGAs include on-chip memory blocks that can be used as RAM or ROM in SOPC Builder systems. On-chip memory has the following benefits for SOPC Builder systems:

- On-chip memory has fast access time, compared to off-chip memory.
- SOPC Builder automatically instantiates on-chip memory inside the SOPC Builder system, so you do not have to make any manual connections.
- Certain memory blocks can have initialized contents when the FPGA powers up. This feature is useful, for example, for storing data constants or processor boot code.

Component-Level Design for On-Chip Memory

In SOPC Builder you instantiate on-chip memory by clicking the **On-chip Memory (RAM or ROM)** from the list of available components. The configuration wizard for the **On-chip Memory (RAM or ROM)** component has the following options: **Memory type**, **Size**, and **Read latency**.

Memory Type

The **Memory type** options define the structure of the on-chip memory:

- **RAM (writable)**—This setting creates a readable and writable memory.
- **ROM (read only)**—This setting creates a read-only memory.
- **Dual-port access**—This setting creates a memory component with two slaves, which allows two masters to access the memory simultaneously.
- **Block type**—This setting directs the Quartus II software to use a specific type of memory block when fitting the on-chip memory in the FPGA. **Auto** allows the Quartus II software to choose a type and the other settings force the Quartus II software to use a particular type. However, M-RAM blocks do not allow pre-initialized contents at power up.

Size

The **Size** options define the size and width of the memory.

- **Data width**—This setting determines the data width of the memory. The available choices are **8, 16, 32, 64, 128, 256, 512, or 1024** bits. Assign **Data width** to match the width of the master that accesses this memory the most frequently or has the most critical timing requirements.
- **Total memory size**—This setting determines the total size of the on-chip memory block. The total memory size must be less than the available memory in the target FPGA.

Read Latency

On-chip memory components use synchronous, pipelined Avalon-MM slaves. Pipelined access improves f_{MAX} performance, but also adds latency cycles when reading the memory. The **Read latency** option allows you to specify either one or two cycles of read latency required to access data. If the **Dual-port access** setting is turned on, you can specify a

different read latency for each slave. When you have Dual-port memory in your system you can specify different clock frequencies for the ports. You specify this on the **System Contents** tab in SOPC Builder window.

Non-Default Memory Initialization

For ROM memories, you can specify your own initialization file by selecting **Enable non-default initialization file**. This allows the file you specify to be used to initialize the ROM in place of the default initialization file created by SOPC Builder.

Enable In-System Memory Content Editor Feature

Enables a JTAG interface used to read and write to the RAM while it is operating. You can use this interface to update or read the contents of the memory from your host PC.

SOPC Builder System-Level Design for On-Chip Memory

There are few SOPC Builder system-level design considerations for on-chip memories. See [“SOPC Builder System-Level Design” on page 9–6](#).

When generating a new system, SOPC Builder creates a blank initialization file in the Quartus II project directory for each on-chip memory that can power up with initialized contents. The name of this file is *<name of memory component>.hex*.

Simulation for On-Chip Memory

At system generation time, SOPC Builder generates a simulation model for the on-chip memory. This model is embedded inside the SOPC Builder system, and there are no user-configurable options for the simulation testbench.

You can provide memory initialization contents for simulation in the file *<Quartus II project directory>/<SOPC Builder system name>_sim/<Memory component name>.dat*.

Quartus II Project-Level Design for On-Chip Memory

The on-chip memory is embedded inside the SOPC Builder system, and there are no signals to connect to the Quartus II project.

To provide memory initialization contents, you must fill in the file *<name of memory component>.hex*. The Quartus II software recognizes this file during design compilation and incorporates the contents into the configuration files for the FPGA.



For Nios II processor users, the Nios II IDE generates the memory initialization file automatically. For the memory to be initialized after you compile your software, you then must compile the hardware in the Quartus II software for the SRAM Object File (.sof) to pick up the memory initialization.

Board-Level Design for On-Chip Memory

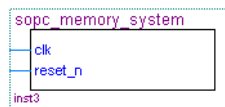
The on-chip memory is embedded inside the SOPC Builder system, and there is nothing to connect at the board level.

Example Design with On-Chip Memory

This section demonstrates adding a 4 KByte on-chip RAM to the example design. This memory uses a single slave with read latency of one cycle.

For demonstration purposes, [Figure 9–3](#) shows the result of generating the SOPC Builder system at this stage. (In a normal design flow, you generate the system only after adding all system components.)

Figure 9–3. SOPC Builder System with On-Chip Memory



Because the on-chip memory is contained entirely within the SOPC Builder system, **sopc_memory_system** has no I/O signals associated with **onchip_ram**. Therefore, you do not need to make any Quartus II project connections or assignments for the on-chip RAM, and there are no board-level considerations.

EPCS Serial Configuration Device

Many systems use an Altera EPCS serial configuration device to configure the FPGA. Altera provides the EPCS device controller core, which allows SOPC Builder systems to access the memory contents of the EPCS device.

This feature provides flexible design options:

- The FPGA design can reprogram its own configuration memory, providing a mechanism for remote upgrades.
- The FPGA design can use leftover space in the EPCS as nonvolatile storage.

Physically, the EPCS device is a serial flash memory device, which has slow access time. Altera provides software drivers to control the EPCS core for the Nios II processor only. Therefore, the EPCS controller core features are available only to SOPC Builder systems that include a Nios II processor.



For further details about the features and usage of the EPCS device controller core, refer to the *EPCS Device Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

Component-Level Design for an EPCS Device

In SOPC Builder you instantiate an EPCS controller core by adding an **EPCS Serial Flash Controller** component. There are no settings for this component.



For details, refer to the *Nios II Flash Programmer User Guide*.

SOPC Builder System-Level Design for an EPCS Device

There are two SOPC Builder system-level design considerations for EPCS devices:

- Assign a base address.
- Set the IRQ connection to **NC** (no connect). The EPCS controller hardware is capable of generating an IRQ. However, the Nios II driver software does not use this IRQ, and therefore you can leave the IRQ signal disconnected.

There can only be one EPCS controller core per FPGA, and the instance of the core is always named `epcs_controller`.

If you want to store Nios II code in the EPCS memory, point the Nios II reset address at the EPCS controller. Inside the EPCS controller is a bootloader, which Nios II runs out of reset, that copies the code from the EPCS flash into main memory.

Simulation for an EPCS Device

The EPCS controller core provides a limited simulation model:

- Functional simulation does not include the FPGA configuration process, and therefore the EPCS controller does not model the configuration features.
- The simulation model does not support read and write operations to the flash region of the EPCS device.
- A Nios II processor can boot from the EPCS device in simulation. However, the boot loader code is different during simulation. The EPCS controller boot loader code assumes that all other memory simulation models are pre-initialized, and therefore the boot load process is unnecessary. During simulation, the boot loader simply forces the Nios II processor to jump to start, skipping the boot load process.

Verification in the hardware is the best way to test features related to the EPCS device.

Quartus II Project-Level Design for an EPCS Device

If you use a Cyclone III or Stratix III device, you must connect the wires manually.

For earlier device families, however, the Quartus II software automatically connects the EPCS controller core in the SOPC Builder system to the dedicated configuration pins on the FPGA. This connection is invisible to you. Therefore, there are no EPCS-related signals to connect in the Quartus II project.

Board-Level Design for an EPCS Device

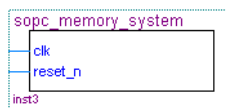
You must connect the EPCS device to the FPGA as described in the *Altera Configuration Handbook*. No other connections are necessary.

Example Design with an EPCS Device

This section demonstrates adding an EPCS device controller core to the example design.

For demonstration purposes only, Figure 9–4 shows the result of generating the SOPC Builder system at this stage.

Figure 9–4. SOPC Builder System with EPCS Device



Because the Quartus II software automatically connects the EPCS controller core to the FPGA pins, the SOPC Builder system has no I/O signals associated with **epcs_controller**. Therefore, you do not need to make any Quartus II project connections or assignments for the EPCS controller core.



This chapter does not cover the details of configuration using the EPCS device. For further information, refer to the *Altera Configuration Handbook*.

(SDR) SDRAM

Altera provides a free (SDR) SDRAM controller core, which allows you to use inexpensive SDRAM as bulk RAM in your FPGA designs. The (SDR) SDRAM controller core is necessary, because Avalon-MM signals cannot describe the complex interface on an SDRAM device. The (SDR) SDRAM controller acts as a bridge between the system interconnect fabric and the pins on an SDRAM device. The (SDR) SDRAM controller can operate in excess of 100 MHz.

(SDR) SDRAM is a single data rate (SDR) synchronous dynamic random access memory (SDRAM). Synchronous design allows precise cycle control with the use of system clock I/O transactions possible on every clock cycle. Operating over a range of frequencies, programmable latencies allow the same device to be useful for a variety of high bandwidth, high performance memory system applications.



For further details about the features and usage of the (SDR) SDRAM controller core, refer to the *SDR-SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

Component-Level Design for SDRAM

The choice of SDRAM device(s) and the configuration of the device(s) on the board heavily influence the component-level design for the SDRAM controller. Typically, the component-level design task involves parameterizing the SDRAM controller core to match the SDRAM device(s) on the board. You must specify the structure (address width, data width, number of devices, number of banks, and so on) and the timing specifications of the device(s) on the board.



For complete details about configuration options for the SDRAM controller core, refer to the *SDRAM Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

SOPC Builder System-Level Design for SDRAM

You can select the SDRAM controller in the SOPC Builder **System Contents** tab. Like the on-chip memory, there are few SOPC Builder system-level design considerations for SDRAM. See “*SOPC Builder System-Level Design*” on page 9–6.

Simulation for SDRAM

At system generation time, SOPC Builder can generate a generic SDRAM simulation model and include the model in the system testbench. To use the generic SDRAM simulation model, you must turn on a setting in the SDRAM controller configuration wizard. You can provide memory initialization contents for simulation in the file `<Quartus II project directory>/<SOPC Builder system name>_sim/<Memory component name>.dat`.

Alternatively, you can provide a specific vendor memory model for the SDRAM. In this case, you must manually wire up the vendor memory model in the system testbench.



For further details, refer to “*Simulation Considerations*” on page 9–6 and the *SDRAM Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

Quartus II Project-Level Design for SDRAM

SOPC Builder generates a SOPC Builder system with top-level I/O signals associated with the SDRAM controller. In the Quartus II project, you must connect these I/O signals to FPGA pins, which connect to the SDRAM device on the board. In addition, you might have to accommodate clock skew issues.

Connecting and Assigning the SDRAM-Related Pins

After generating the system with SOPC Builder, you can find the names and directions of the I/O signals in the top-level HDL file for the SOPC Builder system. The file has the name

`<Quartus II project directory>/<SOPC Builder system name>.v` or `<Quartus II project directory>/<SOPC Builder system name>.vhd`. You must connect these signals in the top-level Quartus II design file.

You must assign a pin location for each I/O signal in the top-level Quartus II design to match the target board. Depending on the performance requirements for the design, you might have to assign FPGA pins carefully to achieve performance.

Accommodating Clock Skew

As SDRAM frequency increases, so does the possibility that you must accommodate skew between the SDRAM clock and I/O signals. This issue affects all synchronous memory devices, including SDRAM. To accommodate clock skew, you can instantiate an ALTPLL megafunction in the top-level Quartus II design to create a phase-locked loop (PLL) clock output. You use a phase-shifted PLL output to drive the SDRAM clock and reduce clock-skew issues. The exact settings for the ALTPLL megafunction depend on your target hardware. You must experiment to tune the phase shift to match the board.



For details, refer to the [ALTPLL Megafunction User Guide](#).

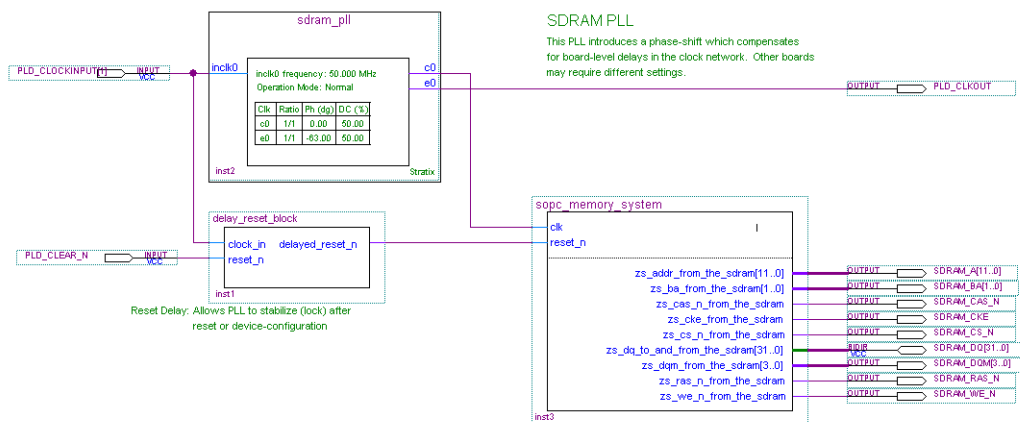
Board-Level Design for SDRAM

Memory requirements largely dictate the board-level configuration of the SDRAM device(s). The SDRAM controller core can accommodate various configurations of SDRAM on the board, including multiple banks and multiple devices.

Example Design with (SDR) SDRAM

This section demonstrates adding a 16-Mbyte SDRAM device to the example design, using the (SDR) SDRAM Controller configuration wizard. This SDRAM is a single device with 32-bit data.

For demonstration purposes, [Figure 9–5](#) shows the result of generating the SOPC Builder system at this stage, and connecting it in `toplevel_design.bdf`.

Figure 9–5. toplevel_design.bdf with SDRAM

After generating the system, the top-level SOPC Builder system file **sopc_memory_system.v** contains the list of SDRAM-related I/O signals that must be connected to FPGA pins, as follows:

Example 9–1. I/O Signals Connected to FPGA Pins

```
output [ 11: 0] zs_addr_from_the_sdrām;
output [  1: 0] zs_ba_from_the_sdrām;
output          zs_cas_n_from_the_sdrām;
output          zs_cke_from_the_sdrām;
output          zs_cs_n_from_the_sdrām;
inout  [ 31: 0] zs_dq_to_and_from_the_sdrām;
output [  3: 0] zs_dqm_from_the_sdrām;
output          zs_ras_n_from_the_sdrām;
output          zs_we_n_from_the_sdrām;
```

As shown in Figure 9–5, **toplevel_design.bdf** uses an instance of **sdrām_pll** to phase shift the SDRAM clock by –63 degrees. (Degrees are relative to clock frequency. If you change the clock speed you must change the phase shift. You should parameterize the PLL with -3.5ns, because what is being compensated is round trip delays and clock to I/O delays.)

toplevel_design.bdf also uses a subdesign **delay_reset_block** to insert a delay on the **reset_n** signal for the SOPC Builder system. This delay is necessary to allow the PLL output to stabilize before the SOPC Builder system begins operating.

Figure 9–6 shows pin assignments in the Quartus II Assignment Editor for some of the SDRAM pins. The correct pin assignments depend on the target board.

Figure 9–6. Pin Assignments for SDRAM

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved
188	SDRAM_A[0]	PIN_AE4	7	LVTTTL	Column I/O		
189	SDRAM_A[10]	PIN_Y11	7	LVTTTL	Column I/O		
190	SDRAM_A[11]	PIN_AB7	7	LVTTTL	Column I/O		
191	SDRAM_A[1]	PIN_W12	7	LVTTTL	Column I/O	PGM0	
192	SDRAM_A[2]	PIN_AC11	7	LVTTTL	Column I/O	nRS	
193	SDRAM_A[3]	PIN_W10	7	LVTTTL	Column I/O	RUnLU	
194	SDRAM_A[4]	PIN_AA11	7	LVTTTL	Column I/O	PGM1	
195	SDRAM_A[5]	PIN_AC10	7	LVTTTL	Column I/O	RDN7	
196	SDRAM_A[6]	PIN_AB11	7	LVTTTL	Column I/O	RUP7	
197	SDRAM_A[7]	PIN_AC8	7	LVTTTL	Column I/O	FCLK5	
198	SDRAM_A[8]	PIN_AB10	7	LVTTTL	Column I/O	FCLK4	
199	SDRAM_A[9]	PIN_V11	7	LVTTTL	Column I/O		
200	SDRAM_BA[0]	PIN_AG19	8	LVTTTL	Column I/O	DQ6B4	
201	SDRAM_BA[1]	PIN_AF19	8	LVTTTL	Column I/O	DQ6B5	
202	SDRAM_CAS_N	PIN_AD18	8	LVTTTL	Column I/O	DQ6B2	
203	SDRAM_CKE	PIN_AE18	8	LVTTTL	Column I/O	DQ6B1	
204	SDRAM_CS_N	PIN_AG18	8	LVTTTL	Column I/O	DQ6B0	
205	SDRAM_DQM[0]	PIN_AE14	7	LVTTTL	Column I/O	CLK6n	
206	SDRAM_DQM[1]	PIN_Y13	7	LVTTTL	Column I/O	CLK7n	
207	SDRAM_DQM[2]	PIN_AE7	7	LVTTTL	Column I/O	DQ51B	
208	SDRAM_DQM[3]	PIN_AG10	7	LVTTTL	Column I/O	DQ53B	

(DDR) SDRAM

You use double-data rate (DDR) SDRAM devices for a broad range of applications, such as embedded processor systems, image processing, storage, communications and networking. In addition, the universal adoption of (DDR) SDRAM in PCs makes (DDR) SDRAM memory a solution for high-bandwidth applications. (DDR) SDRAM is a 2n prefetch architecture where the internal data bus is twice the width of the external data bus and data transfers occur on both clock edges. It uses a strobe, DQS, which is associated with a group of data pins (DQ) for read and write operations. Both the DQS and DQ ports are bidirectional. Address ports are shared for write and read operations.



Refer to the (DDR) SDRAM literature on the Altera website for further details on the use of (DDR) SDRAM memory, including *AN 517: Using High-Performance DDR, DDR2, and DDR3 SDRAM With SOPC Builder*.

.(DDR2) SDRAM

Double-data rate 2 (DDR2) SDRAM is the second generation of double-data rate (DDR) SDRAM technology, with features such as lower power consumption, higher data bandwidth, enhanced signal quality, and on-die termination schemes. (DDR2) SDRAM brings higher memory performance to a broad range of applications, such as PCs, embedded processor systems, image processing, storage, communications, and

networking. It is a 4n pre-fetch architecture with two data transfers per clock cycle. The memory uses a strobe (DQS) associated with a group of data pins (DQ) for read and write operations. Both the DQ and DQS ports are bi-directional. Address ports are shared for write and read operations. Because (DDR2) SDRAM is a 4n memory, a full transfer requires two clock cycles (two words on the rising edge, two words on the falling edge).



When you use (SDR) SDRAM with parallel flash loaders, refer to the following application note for further explanation as to how these two types of memory work in combination with each other: *AN 517: Using High-Performance DDR, DDR2, and DDR3 SDRAM With SOPC Builder*.

Parallel Flash Loader



Refer to *AN 386: Using the Parallel Flash Loader with the Quartus II Software* for further details on using parallel flash loaders with the SOPC Builder system.

Off-Chip SRAM and Flash Memory

SOPC Builder systems can directly access many off-chip RAM and ROM devices, without a controller core to drive the off-chip memory. Avalon-MM signals can describe the interfaces on many standard memories, such as SRAM and flash memory. I/O signals on the SOPC Builder system can connect directly to the memory device.

While off-chip memory usually has slower access time than on-chip memory, off-chip memory provides the following benefits:

- Off-chip memory cost-per-bit is less expensive than on-chip memory resources.
- The size of off-chip memory is bounded only by the 32-bit Avalon-MM address space.
- Off-chip ROM, such as flash memory, can be used for bulk storage of nonvolatile data.
- Multiple off-chip RAM and ROM memories can share address and data pins to conserve FPGA I/O resources.

Adding off-chip memories to an SOPC Builder system also requires the **Avalon-MM Tristate Bridge** component.

Component-Level Design for SRAM and Flash Memory

There are several ways to instantiate an interface to an off-chip memory device:

- For common flash interface (CFI) flash memory devices, add the **Flash Memory (Common Flash Interface)** component in SOPC Builder.
- For Altera development boards, Altera provides SOPC Builder components that interface to the specific devices on each development board. For example, the Nios II EDS includes the components **Cypress CY7C1380C SSRAM** and **IDT71V416 SRAM**, which appear on Nios II development boards.



For further details about the features and usage of the SSRAM controller core, refer to the *Cyclone II Development Board*, the *Stratix II Development Board*, or the *Cyclone III Nios Embedded Evaluation Kit*, which are available at www.altera.com.



For further details about the features and usage of the SDRAM controller core, refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.

These components make it easy for you to create memory systems targeting Altera development boards. However, these components target only the specific memory device on the board; they do not work for different devices.

- For general memory devices, RAM or ROM, you can create a custom interface to the device with the SOPC Builder component editor. Using the component editor, you define the I/O pins on the memory device and the timing requirements of the pins.

In all cases, you must also instantiate the **Avalon-MM Tristate Bridge** component. Multiple off-chip memories can connect to a single tristate bridge, in order to share pins such as the off-chip address bus.

Avalon-MM Tristate Bridge

A tristate bridge connects off-chip devices to the system interconnect fabric. The tristate bridge creates I/O signals for the SOPC Builder system, which you must connect to FPGA pins in the top-level Quartus II project.

The tristate bridge creates address and data pins that can be shared by multiple off-chip devices. This feature lets you conserve FPGA pins when connecting the FPGA to multiple devices with mutually exclusive access.

You must use a tristate bridge in either of the following cases:

- The off-chip device has bidirectional data pins.
- Multiple off-chip devices share the address and/or data buses.

In SOPC Builder, you instantiate a tristate bridge by instantiating the **Avalon-MM Tristate Bridge** component. The Avalon-MM Tristate Bridge configuration wizard has a single option: To register incoming (to the FPGA) signals or not.

- **Registered**—This setting adds registers to all FPGA input pins associated with the tristate bridge (outputs from the memory device).
- **Not Registered**—This setting does not add registers between the memory device output pins and the system interconnect fabric.

The Avalon-MM tristate bridge automatically adds registers to output signals from the tristate bridge to off-chip devices.

Registering the input and output signals shortens the register-to-register delay from the memory device to the FPGA, resulting in higher system f_{MAX} performance. However, in each direction, the registers add one additional cycle of latency for Avalon-MM masters accessing memory connected to the tristate bridge. The registers do not affect the timing (step, hold, and wait) of the transfers from the perspective of the memory device.



For details about the Avalon-MM tristate interface, refer to the [*Avalon Interface Specifications*](#).

Flash Memory

In SOPC Builder, you instantiate an interface to CFI flash memory by adding a **Flash Memory (Common Flash Interface)** component. If the flash memory is not CFI compliant, you must create a custom interface to the device with the SOPC Builder component editor.

The choice of flash devices and the configuration of the devices on the board help determine the component-level design for the flash memory configuration wizard. Typically, the component-level design task involves parameterizing the flash memory interface to match the devices on the board. Using the Flash Memory (Common Flash Interface) configuration wizard, you must specify the structure (address width and data width) and the timing specifications of the flash memory devices.



For details about features and usage, refer to the *Common Flash Interface Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

For an example of instantiating the Flash Memory (Common Flash Interface) component in an SOPC Builder system, see “*Example Design with SRAM and Flash Memory*” on page 9–25.

SRAM

To instantiate an interface to off-chip RAM:

1. Create a new component with the SOPC Builder component editor that defines the interface.
2. Instantiate the new interface component in the SOPC Builder system.

The choice of RAM devices and the configuration of the devices on the board determine how you create the interface component. The component-level design task involves entering parameters into the component editor to match the devices on the board.



For details about using the component editor, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

SOPC Builder System-Level Design for SRAM and Flash Memory

In the SOPC Builder **System Contents** tab, the Avalon-MM tristate bridge has two ports:

- Avalon-MM slave—This port faces the on-chip logic in the SOPC Builder system. You connect this slave to on-chip masters in the system.
- Avalon-MM tristate master—This port faces the off-chip memory devices. You connect this master to the Avalon-MM tristate slaves on the interface components for off-chip memories.

You assign a clock to the Avalon-MM tristate bridge that determines the Avalon-MM clock cycle time for off-chip devices connected to the tristate bridge.

You must assign base addresses to each off-chip memory. The Avalon-MM tristate bridge does not have an address; it passes unmodified addresses from on-chip masters to off-chip slaves.

Simulation for SRAM and Flash Memory

The SOPC Builder output for simulation depends on the type of memory components in the system:

- **Flash Memory (Common Flash Interface) component**—This component provides a generic simulation model. You can provide memory initialization contents for simulation in the file *<Quartus II project directory>/<SOPC Builder system name>_sim/<Flash memory component name>.dat*.
- **Custom memory interface created with the component editor**—In this case, you must manually connect the vendor simulation model to the system testbench. SOPC Builder does not automatically connect simulation models for custom memory components to the SOPC Builder system.
- **Altera-provided interfaces to memory devices**—Altera provides simulation models for these interface components. You can provide memory initialization contents for simulation in the file *<Quartus II project directory>/<SOPC Builder system name>_sim/<Memory component name>.dat*. Alternately, you can provide a specific vendor simulation model for the memory. In this case, you must manually wire up the vendor memory model in the system testbench.

For further details, see [“Simulation Considerations”](#) on page 9–6.

Quartus II Project-Level Design for SRAM and Flash Memory

SOPC Builder generates a SOPC Builder system with top-level I/O signals associated with the tristate bridge and the memory interface components. In the Quartus II project, you must connect the I/O signals to FPGA pins, which connect to the memory devices on the board.

After generating the system with SOPC Builder, you can find the names and directions of the I/O signals in the top-level HDL file for the SOPC Builder system. The file has the name *<Quartus II project directory>/<SOPC Builder system name>.v* or *<Quartus II project directory>/<SOPC Builder system name>.vhd*. You must connect these signals in the top-level Quartus II design file.

You must assign a pin location for each I/O signal in the top-level Quartus II design to match the target board. Depending on the performance requirements for the design, you might have to assign FPGA pins carefully to achieve timing.

SOPC Builder inserts synthesis directives in the top-level SOPC Builder system HDL to assist the Quartus II fitter with signals that interface with off-chip devices. The following is an example:

Example 9–2. Synthesis Directive

```
reg [ 22: 0] tri_state_bridge_address /* synthesis
ALTERA_ATTRIBUTE = "FAST_OUTPUT_REGISTER=ON" */;
```

Board-Level Design for SRAM and Flash Memory

Memory requirements determine the board-level configuration of the SRAM and flash memory device(s). You can lay out memory devices in any configuration, as long as the resulting interface can be described with Avalon-MM signals.



Special consideration is required when connecting the Avalon-MM address signal to the address pins on the memory devices.

The SOPC Builder system presents the smallest number of address lines required to access the largest off-chip memory, which is usually less than 32 address bits. Not all memory devices connect to all address lines.

Aligning the Least-Significant Address Bits

The Avalon-MM tristate address signal always presents a byte address. Each address location in many memory devices contains more than one byte of data. In this case, the memory device must ignore one or more of the least-significant Avalon-MM address lines. For example, a 16-bit memory device must ignore Avalon-MM address [0] (which is a byte address), and connect Avalon-MM address [1] to the least-significant address line.

Table 9–1 shows the relationship between Avalon-MM address lines and off-chip address pins for all possible Avalon-MM data widths.

Table 9–1. Connecting the Least-Significant Avalon-MM Address Line					
Avalon-MM Address Line	Address Line Connecting to Memory Device				
	8-bit Memory	16-bit Memory	32-bit Memory	64-bit Memory	128-bit Memory
address[0]	A0	No connect	No connect	No connect	No connect
address[1]	A1	A0	No connect	No connect	No connect
address[2]	A2	A1	A0	No connect	No connect
address[3]	A3	A2	A1	A0	No connect
address[4]	A4	A3	A2	A1	A0
address[5]	A5	A4	A3	A2	A1
address[6]	A6	A5	A4	A3	A2
address[7]	A7	A6	A5	A4	A3
address[8]	A8	A7	A6	A5	A4
address[9]	A9	A8	A7	A6	A5
address[10]	A10	A9	A8	A7	A6
...



You must ensure that the address bits are properly assigned when mixed width components are connecting to the tristate bridge. Failing to ensure that the components are properly aligned may result in a board respin.

Aligning the Most-Significant Address Bits

The Avalon-MM address signal contains enough address lines for the largest memory connected to the tristate bridge. Smaller off-chip memories might not use all of the most-significant address lines.

For example, a memory device with 2^{10} locations uses 10 address bits, while a memory with 2^{20} locations uses 20 address bits. If both these devices share the same tristate bridge, the smaller memory ignores the ten most significant Avalon-MM address lines. They are ignored because you do not hook those 10 MSB traces to the memory.

Example Design with SRAM and Flash Memory

This section demonstrates adding a 1-MByte SRAM and an 8-MByte flash memory to the example design. These memory devices connect to the system interconnect fabric through an Avalon-MM tristate bridge.

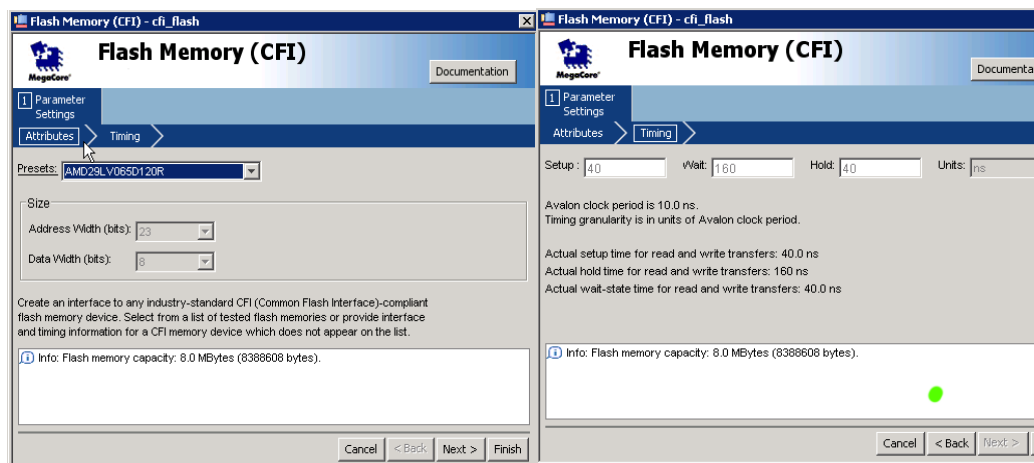
Adding the Avalon-MM Tristate Bridge

In the **Avalon-MM Tristate Bridge** configuration wizard, turn on the **Registered inputs and outputs** option to maximize system f_{MAX} , which increases the read latency by two for both the SRAM and flash memory.

Adding the Flash Memory Interface

The flash memory is 8M × 8-bit, which requires 23 address bits and 8 data bits. [Figure 9–7](#) shows the **Flash Memory (Common Flash Interface)** configuration wizard settings for the example design.

Figure 9–7. Flash Memory Configuration Wizard



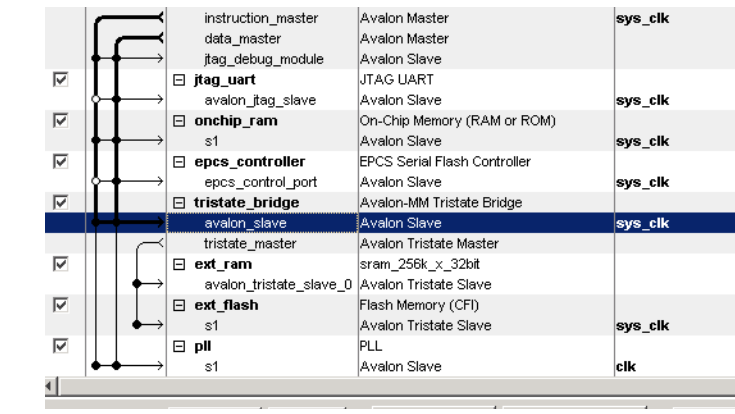
Adding the SRAM Interface

The SRAM device is 256K × 32-bit, which requires 18 address bits and 32 data bits. The example design uses a custom memory interface created with the SOPC Builder component editor.

SOPC Builder System Contents Tab

Figure 9–8 shows the SOPC Builder system after adding the **Tristate bridge and memory interface** components, and configuring them appropriately on the **System Contents** tab. Figure 9–8 represents the complete example design in SOPC Builder.

Figure 9–8. SOPC Builder System with SRAM and Flash Memory



After generating the system, the top-level SOPC Builder system file **sopc_memory_system.v** contains the list of I/O signals for SRAM and flash memory that must be connected to FPGA pins, as shown in the following example:

Example 9–3. I/O Signals for SRAM and Flash Memory

```

output      address_to_the_ext_flash[ 23..0];
output      address_to_the_ext_ram[ 19..0];
output      be_n_to_the_ext_ram[ 3..0];
output      read_n_to_the_ext_flash;
output      read_n_to_the_ext_ram;
output      read_n_to_the_ext_ram;
output      select_n_to_the_ext_flash;
output      select_n_to_the_ext_ram;
bidirectional tristate_bridge_data [ 31..0]
output      write_n_to_the_ext_flash;
output      write_n_to_the_ext_ram;
```

The Avalon-MM tristate bridge signals that can be shared are named after the instance of the tristate bridge component, such as `tri_state_bridge_data[31:0]`.

Connecting and Assigning Pins in the Quartus II Project

Figure 9–9 shows the result of generating the SOPC Builder system for the complete example design.

Figure 9–9. Top Level System with SRAM and Flash Memory

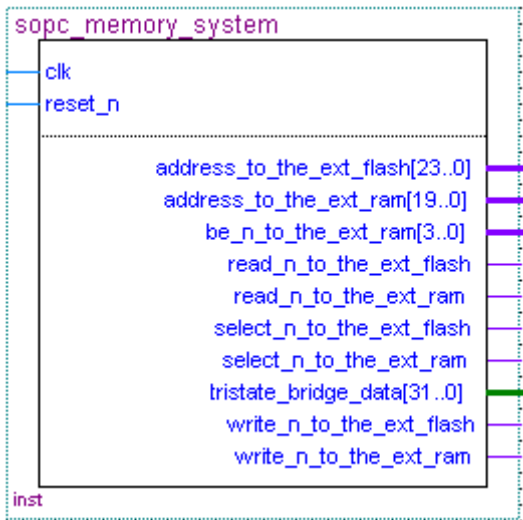


Figure 9–10 shows the pin assignments in the Quartus II assignment editor for some of the SRAM and flash memory pins. The correct pin assignments depend on the target board.

Figure 9–10. Pin Assignments for SRAM and Flash Memory

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reset
243	SRAM_BE_N[0]	PIN_M18	3	LVTTL	Column I/O		
244	SRAM_BE_N[1]	PIN_F17	3	LVTTL	Column I/O		
245	SRAM_BE_N[2]	PIN_J18	3	LVTTL	Column I/O	RUP3	
246	SRAM_BE_N[3]	PIN_L17	3	LVTTL	Column I/O	CLK15n	
247	SRAM_CS_N	PIN_B24	3	LVTTL	Column I/O	DQ9T4	
248	SRAM_OE_N	PIN_B26	3	LVTTL	Column I/O	DQ9T7	
249	SRAM_WE_N	PIN_C24	3	LVTTL	Column I/O	DQ59T	

Connecting FPGA Pins to Devices on the Board

Table 9–2 shows the mapping between the Avalon-MM address lines and the address pins on the SRAM and flash memory devices.

Table 9–2. FPGA Connections to SRAM and Flash Memory

Avalon-MM Address Line	Flash Address (8M × 8-bit Data)	SRAM Address (256K × 32-bit data)
tri_state_bridge_address[0]	A0	No connect
tri_state_bridge_address[1]	A1	No connect
tri_state_bridge_address[2]	A2	A0
tri_state_bridge_address[3]	A3	A1
tri_state_bridge_address[4]	A4	A2
tri_state_bridge_address[5]	A5	A3
tri_state_bridge_address[6]	A6	A4
tri_state_bridge_address[7]	A7	A5
tri_state_bridge_address[8]	A8	A6
tri_state_bridge_address[9]	A9	A7
tri_state_bridge_address[10]	A10	A8
tri_state_bridge_address[11]	A11	A9
tri_state_bridge_address[12]	A12	A10
tri_state_bridge_address[13]	A13	A11
tri_state_bridge_address[14]	A14	A12
tri_state_bridge_address[15]	A15	A13
tri_state_bridge_address[16]	A16	A16
tri_state_bridge_address[17]	A17	A15
tri_state_bridge_address[18]	A18	A16
tri_state_bridge_address[19]	A19	A17
tri_state_bridge_address[20]	A20	No connect
tri_state_bridge_address[21]	A21	No connect
tri_state_bridge_address[22]	A22	No connect

Referenced Documents

This chapter references the following documents:

- *Altera Configuration Handbook*
- *altpll Megafunction User Guide*
- *AN 386: Using the Parallel Flash Loader with the Quartus II Software*
- *Avalon Interface Specifications*
- *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- *Common Flash Interface Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *EPCS Device Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *Introduction to the Quartus II Software manual*
- *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*
- *Memory System Design* chapter in the *Embedded Design Handbook*
- *Nios II Embedded Processor Design Examples*
- *Nios II Flash Programmer User Guide*
- *SDRAM Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook*
- *Using High-Performance DDR, DDR2, and DDR3 SDRAM With SOPC Builder* Application Note 517

Document Revision History

Table 9–3 shows the revision history for this chapter.

<i>Table 9–3. Document Revision History (Part 1 of 2)</i>		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	<ul style="list-style-type: none"> ● Chapter renumbered from 8 to 9. ● Added brief new sections referencing DDR-2 and PFLs. ● Updated references to Avalon Interface Specifications. ● Updated Figures 9-1, 9-14, 9-15, 9-16, and 9-19 with new art. 	—
October 2007, v7.2.0	<ul style="list-style-type: none"> ● Corrected Figure 9–9 to show flash memory changed example to use a PLL that is part of the SOPC Builder system, rather than a Quartus II component. Added section showing parameterization of PLL. (ADoQS Issue 1-5M4EN5 Lissy) 	—

Table 9–3. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
May 2007, v7.1.0	<ul style="list-style-type: none"> Chapter 8 was previously chapter 9. Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon switch fabric” term to “system interconnect fabric.” Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.” Added section on Non-Default Memory Initialization. On-chip Memory size, first parameter changed from Memory Width to Data Width and widths of 256, 512 and 1024 were added. Corrected figure 8-18. Added links to all referenced documents. Removed discussions of reference designators for components because they are no longer required by SOPC Builder. Removed unnecessary screenshots. 	Updated to reflect changes to SOPC Builder for 7.1.0. SOPC Builder and improve readability.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	No change from previous release.	—
May 2006, v6.0.0	Chapter 9 was previously chapter 8. No change to content.	—
October 2005, v5.1.0	Chapter 8 was previously chapter 6. No change to content.	—
May 2005, v5.0.0	Initial release.	—

Introduction

This chapter describes the parts of a custom SOPC Builder component and provides walkthrough steps that guide you through the process of creating an example custom component, integrating it into a system, and testing it in hardware.

This chapter is divided into the following sections:

- [“Component Development Flow” on page 10–2.](#)
- [“Design Example: Checksum Hardware Accelerator” on page 10–4.](#) This design example shows you how to develop a component with both Avalon® Memory-Mapped (Avalon-MM) master and slaves.
- [“Sharing Components” on page 10–9.](#) This section shows you how to use components in other systems, or share them with other designers.
- [“.sopcinfo Files” on page 10–10.](#)

SOPC Builder Components and the Component Editor

An SOPC Builder component is usually composed of the following four types of files:

- HDL files—define the component’s functionality as hardware.
- `_hw.tcl` file—describes the SOPC Builder related characteristics, such as interface behaviors. This file is created by the component editor.
- C-language files—define the component register map and driver software to allow programs to control the component.
- `_sw.tcl` file—used by the software build tools to use and compile the component driver code.

The component editor guides you through the creation of your component. You can then instantiate the component in an SOPC Builder system and make connections in the same manner as other SOPC Builder components. You can also share your component with other designers.

For information about creating the `_sw.tcl` file, see the *Developing Device Drivers for the Hardware Abstraction Layer* chapter in the *Nios II Software Developer’s Handbook*.

Prerequisites

This chapter assumes that you are familiar with the following:

- Building systems with SOPC Builder. For details, refer to the *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon-MM interface.

Hardware and Software Requirements

To use the design example in this chapter, in addition to the current version of the Quartus II software and Nios II Embedded Design Suite, you must have the following:

- Design files for the example design—A hyperlink to the design files appears next to the chapter, *Developing Components for SOPC Builder*, on the SOPC Builder literature page.
- Nios development board and an Altera® USB-Blaster™ download cable—You can use either of the following Nios development boards:
 - Stratix® II Edition
 - Cyclone® II Edition

If you do not have a development board, you can follow the hardware development steps. You cannot download the complete system without a working board, but you may be able to simulate the system.



You can download the Quartus II Web Edition software and the Nios II EDS, Evaluation Edition for free from the Altera Download Center at www.altera.com.

Component Development Flow

This section provides an overview of the development process for SOPC Builder components.

Typical Design Steps

A typical development sequence for an SOPC Builder component includes the following items:

1. Specification and definition.
 - a. Define the functionality of the component.

- b. Determine component interfaces, such as Avalon Memory-Mapped (Avalon-MM), Avalon Streaming (Avalon-ST), interrupt, or other interfaces.
 - c. Determine the component clocking requirements; what interfaces are synchronous to what clock inputs.
 - d. If you want a microprocessor to control the component, determine the interface to software, such as the register map.
2. Implement the component in VHDL or Verilog HDL.
3. Import the component into SOPC Builder.
 - a. Use the component editor to create a `_hw.tcl` file that describes the component.
 - b. Instantiate the component into an SOPC Builder system.

When importing an HDL file using the component editor, any parameter definitions that are dependent upon other defined parameters cause an error. For example, the following *DEPTH* parameter, though legal Verilog HDL syntax in the Quartus II software, causes an error in the component editor syntax checker:

Example 10–1. DEPTH Parameter

```
parameter WIDTH = 32;
parameter DEPTH = ((WIDTH == 32) ? 8 : 16);
```

To avoid this error, use *localparam* for the dependent parameter instead, as shown below:

Example 10–2. localparam Parameter

```
parameter WIDTH = 32;
localparam DEPTH = ((WIDTH == 32) ? 8 : 16);
```

4. Develop the software driver, which can occur in parallel with the hardware implementation.
 - a. Create the component's driver, including a C header file that defines the hardware-level register map for software.



For further details, see the *Nios II Software Developer's Handbook*.

5. Perform in-system testing, such as the following:
 - a. Test register-level accesses to the component in hardware or simulation using a microprocessor, such as the Nios II processor.
 - b. Performance benchmarking.

Hardware Design

As with any logic design process, the development of SOPC Builder component hardware begins after the specification phase. Creating the HDL design is often an iterative process, as you write and verify the HDL logic against the specification.

The architecture of a typical component consists of the following functional blocks:

- *Task Logic*—Implements the component's fundamental function. The task logic is design dependent.
- *Interface Logic*—Provides a standard way of providing data to or getting data from the components and of controlling the functioning of the components.



For further details, refer to the *Avalon Interface Specifications*.

Figure 10–1 shows the top-level blocks of a checksum component, which includes both Avalon-MM master and slaves.



The work flow for developing SOPC Builder hardware, including how to decide upon and implement the register map, is described in the *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*. Also, guidelines for developing device drivers is described in the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Design Example: Checksum Hardware Accelerator

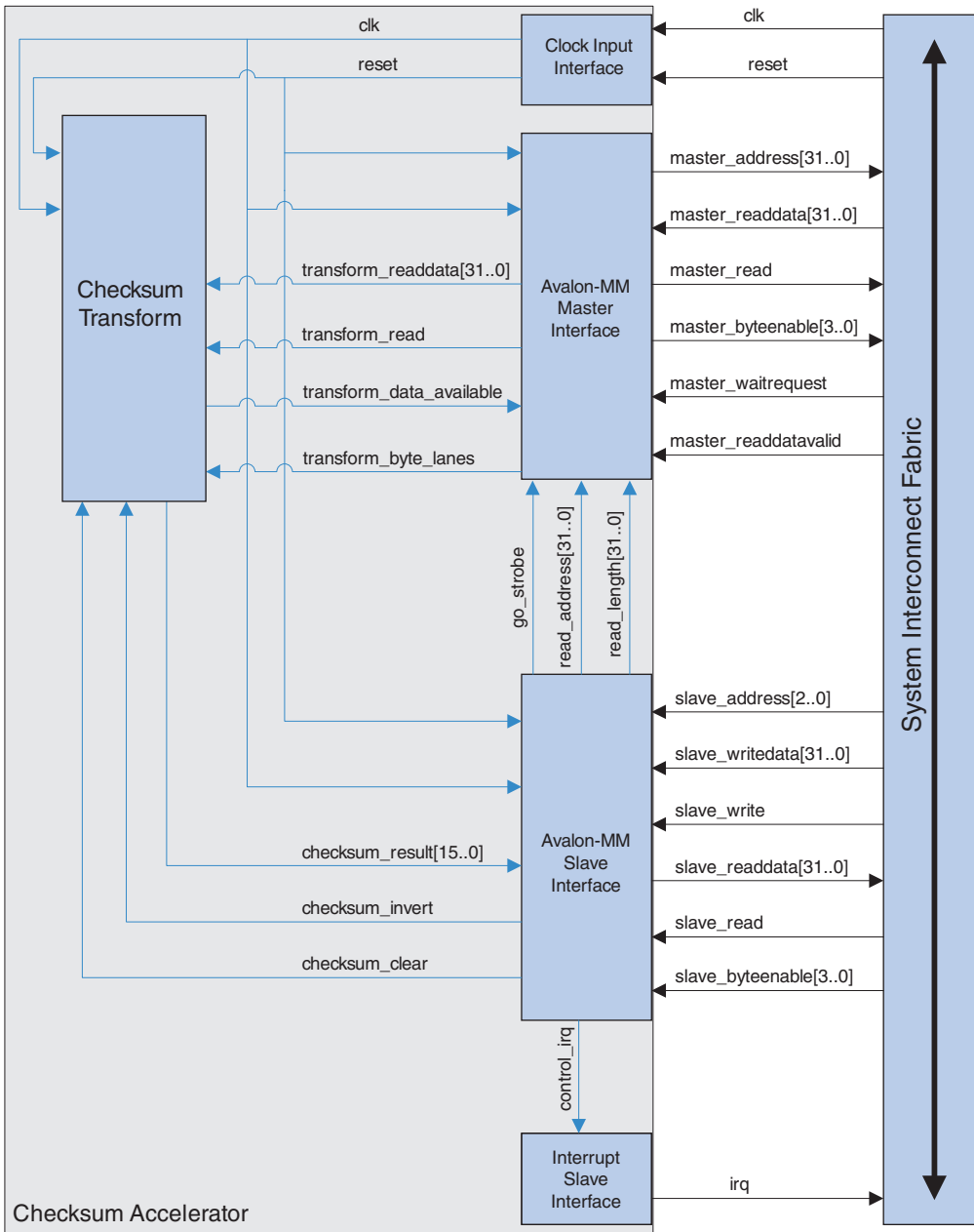
Altera has provided a Checksum Hardware Accelerator design example to demonstrate the steps to create a component and instantiate it in a system. This design example is available for download from the Altera literature website. Included in the compressed download file is a **readme.pdf** that describes how to create and compile the hardware design, and describes how to use the checksum hardware accelerator in your design.

You can use the checksum algorithm in network applications where data integrity must be inspected by the receiving device. The checksum algorithm accumulates data with end-round-carry summation, which means that you take the carry bit and add it to the next input. After the data is accumulated, you can use the result to verify the data integrity of the data buffer. Because the checksum algorithm operates over a data buffer, you can implement it more efficiently with a pipeline read master. A pipeline read master continuously posts read transactions that minimize the effects of the memory read latency. The checksum accelerator can read data and calculate the checksum result every clock cycle, which you cannot do with a general purpose processor.

The checksum hardware accelerator requires information from a host processor such as the buffer read address, buffer length, and various control signals. As a result, the hardware accelerator exposes an Avalon-MM slave interface so that a host processor can control the read master operation. The host processor also accesses the checksum result from the slave interface. Each piece of information sent or read by the host processor is accessed separately in the register file implemented with the slave interface. For example, the status and control signals are implemented as separate registers because they contain information used for different purposes and have different access capabilities.

Hardware accelerators can operate in parallel with a host processor, so it is beneficial to add an interrupt sender interface. The interrupt is asserted after the buffer checksum is calculated. The host processor can be interrupted by the hardware accelerator to notify it that a checksum result has been calculated. The host processor can then read the checksum value and clear the interrupt by writing to the `status` register via the accelerator slave interface.

Figure 10–1. Checksum Component with Avalon-MM Master and Slaves



Software Design

If you want a microprocessor to control your component, then you must provide software files that define the software view of the component. At a minimum, you must define the register map for each Avalon-MM slave that is accessible to a processor.

Typically, the header file declares macros to read and write each register in the component, relative to a symbolic base address assigned to the component. The following Table 10–1 shows the register map of the checksum component for use by the Nios II processor.

Table 10–1. Avalon-MM Slave Port Register Map (Control)													
Offset	Register Name	Rd/Wr/Wclr	Bits										
			31-10	9	8	7	6	5	4	3	2	1	0
0	Status	Rd/Wclr										Busy	Done
4	Read Address (1)	Rd/Wr	Read Address (32-bit word aligned)										
8	N/A												
12	Length (Bytes)	Rd/Wr	Length in Bytes (must be a multiple of 4 for word aligned)										
16	N/A												
20	N/A												
24	Control	Rd/Wr			RC ON				I_EN	GO		Inv	Clr
28	Checksum Results	Rd	16-Bit Checksum Result (upper 16 bits are zeros)										
	N/A												
28	N/A		Reserved ()										
	N/A												

Notes to Table 10–1:

(1) Wr=Writable; Rd=Readable; Wclr=Write cleared



In the example checksum project, you can view an example of a software driver in the directory **projectdir\ip\checksum_accelerator**, which is the top level folder of the hardware and software for the custom checksum block.

Software drivers abstract hardware details of the component so that software can access the component at a high level. The driver functions provide the software an API to access the hardware. The software requirements vary according to the needs of the component. The most common types of routines initialize the hardware, read data, and write data.

When developing software drivers, you should review the software files provided for other ready-made components. The IP installer provides many components you can use as reference. You can also view the `<Nios II EDS install path>/components/` directory for examples.



For details on writing drivers for the Nios II hardware abstraction layer (HAL), refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Verifying the Component

You can verify the component in incremental stages, as you complete more of the design. You should first verify the hardware logic as a unit (which might consist of multiple smaller stages of verification) and later verify the component in a system.

System Console

The system console is an interactive Tcl console available from within SOPC Builder that provides you with read or write access or both to the debugging capabilities that are available in your FPGA logic. You can use the system console to control and query the state of the Nios II processor, issue Avalon transactions, bring up a PCB from scratch, and access either JTAG UARTs or system level debug (SLD) nodes.



For further details, refer to the *System Console User Guide*.

System-Level Verification

After you package a `_hw.tcl` file with the component editor, you can instantiate the component in a system and verify the functionality of the overall SOPC Builder system.

SOPC Builder provides support for system-level verification for HDL simulators such as ModelSim. SOPC Builder automatically produces a test bench for system-level verification.



You can include a Nios II processor in your system to enhance simulation capabilities during the verification phase. Even if your component has no relationship to the Nios II processor, the auto-generated ModelSim simulation environment provides an easy-to-use starting point.

Sharing Components

When you create a component, component editor saves the `_hw.tcl` file in the same directory as the top-level HDL file. Where appropriate, files referenced by the `_hw.tcl` file are expressed relative to the `_hw.tcl` file itself, so the files can easily be moved and copied.

To share a component, in your computer's file system, copy the `_hw.tcl` file and all files used to a directory from which you will use them. This could be the `ip` subdirectory of another project, or a component library directory.



SOPC Builder finds your components if you place your components in the `projectdir\ip` directory.



If you create a new component library under the Quartus II project directory and then add individual components to that new component library, for example:
`<Quartus_rootdir>\sopc_builder\my_project\my_project_lib\component1\`, SOPC Builder cannot find the components. You must add the directory for `component1` to your `ip` search path.



If you need to share a component library directory across projects, you can add items to the **SOPC Builder Tools/Options/IP Search Path** settings.

To use the newly created component in another SOPC Builder system, do one of the following:

- Copy the component and its related files into the IP subdirectory of the project where it is to be used. For example, to use the component with **project 2**, copy the checksum folder to **project2/ip/checksum**, and it will be found the next time SOPC Builder is launched or the component list is refreshed.
- Copy the component and its related files to a component directory, such as **C:/components/checksum/**, and ensure that the component directory is in the SOPC Builder IP Search path, available at **SOPC Builder/Tools/Options/IP Search Path**.

.sopcinfo Files

Every time SOPC Builder generates a system, a *<mysystem>.sopcinfo* file is also generated, which contains the information described below. The *.sopcinfo* file is a report file only, and cannot be edited with SOPC Builder.

- SOPC Builder project, including:
 - Name and tool version
 - HDL language
- Each module instantiated in the system, including:
 - Name and version
 - Where interface information was found on the disk, such as signal names and types, interface properties, and clock domain mapping
 - Parameter names and values
- Each connection, including:
 - Component and interface connections
 - Base address, Avalon-MM interfaces, IRQ number interfaces
 - Memory map as seen by each master in the system

Referenced Documents

This chapter references the following documents:

- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*
- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *System Console User Guide*
- *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*

Document Revision History

Table 10–2 shows the revision history for this chapter.

<i>Table 10–2. Document Revision History</i>		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	<ul style="list-style-type: none"> Chapter renumbered from 9 to 10. Removed discussion of the Checksum Design example, which will now be in a readme.pdf file and zipped with the rest of the design files. Deleted references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications. New Figure 9-1 and Table 9-1. New section on .sopcinfo file. 	Deleted example procedure.
October 2007, v7.2.0	Updated instructions on how to develop components to match new GUI.	—
May 2007, v7.1.0	Changed example component from a pulse width modulator with that only has an Avalon-MM slave interface to a checksum master that includes both Avalon-MM master and slave interfaces.	Changed the example design to one with more practical applications. Updated instructions for the 7.1 release.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	Chapter 9 was previously chapter 10. No change to content.	—
May 2006, v6.0.0	Chapter 10 was previously chapter 9. No change to content.	—
October 2005, v5.1.0	Chapter 9 was previously chapter 7. No change to content.	—
August 2005, v5.0.1	Corrected Table 7-5.	—
May 2005, v5.0.0	No change from previous release.	—
February 2005, v1.0	Initial release.	—

This section provides information on Avalon[®] Memory-Mapped (Avalon-MM) and Avalon Streaming (Avalon-ST) components that can be added to SOPC Builder systems. The components described in these chapters help you to create and optimize your SOPC Builder system. They are provided for free and can be used without a license in any design targeting an Altera device.

This section includes the following chapters:

- [Chapter 11, Avalon Memory-Mapped Bridges](#)
- [Chapter 12, Avalon Streaming Interconnect Components](#)



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

Introduction to Bridges

This chapter introduces Avalon® Memory-Mapped (Avalon MM) bridges, and describes the Avalon-MM bridge components provided by Altera® for use in SOPC Builder systems.

You use bridges to control the topology of the generated SOPC Builder system. Bridges are not end-points for data, but rather affect the way data is transported between other components. By inserting Avalon-MM bridges between masters and slaves, you control system topology, which in turn affects the interconnect that SOPC Builder generates. Manual control of the interconnect can result in higher performance or lower logic utilization or both.

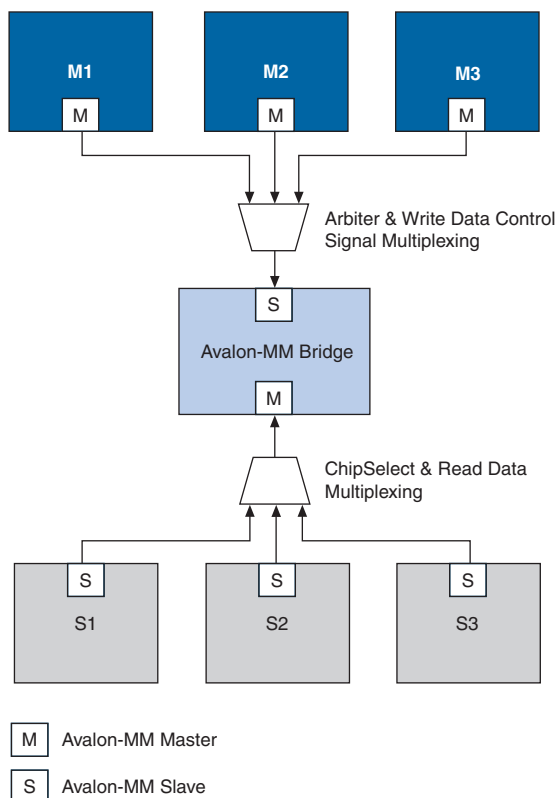
- Altera provides the Avalon-MM bridge, which is described in the section [“Avalon-MM Pipeline Bridge” on page 11–8.](#)
- The Avalon-MM Clock Crossing bridge component is described in the section [“Clock Crossing Bridge” on page 11–13.](#)
- The Avalon-MM Clock Domain Crossing component is described in the section [“Clock Domain Crossing” on page 11–19.](#)



For additional information on using bridges to optimize and control the topology of SOPC Builder systems, refer to [Avalon Memory-Mapped Design Optimizations](#) in the *Embedded Design Handbook*.

Structure of a Bridge

A bridge has one Avalon-MM slave and one Avalon-MM master, as shown in [Figure 11–1](#). In an SOPC Builder system, one or more masters connect to the bridge, which in turn connect to one or more slaves. In [Figure 11–1](#), all three masters have logical connections to all three slaves, although physically each master only connects to the bridge.

Figure 11–1. Example of an Avalon-MM Bridge in an SOPC Builder System

Transfers initiated to the bridge's slave propagate to the master in the same order in which they are initiated on the slave.



For details on the Avalon-MM interface, refer to the [Avalon Interface Specifications](#).

Reasons for Using a Bridge

The following situations are examples in which you might want to consider the use of an Avalon-MM bridge:

- When you have no bridges between master-slave pairs, SOPC Builder generates a system interconnect fabric with maximum parallelism, such that all masters can drive transactions to all slaves concurrently, as long as each master accesses a different slave.

- For systems that do not require a large degree of concurrency, the default behavior might not provide optimal performance.
- With knowledge of the system and application, you can optimize the system interconnect fabric by inserting bridges to control the system topology.

Figure 11–2 and Figure 11–3 show an SOPC system without bridges. This system includes three CPUs, a DDR SDRAM controller, a message buffer RAM, a message buffer mutex, and a tristate bridge to an external SRAM.

Figure 11–2. Example System Without Bridges — SOPC Builder View

Use	Connections	Module Name	Description	Base
<input checked="" type="checkbox"/>		cpu1	Nios II Processor	
		instruction_master	Avalon Master	
		data_master	Avalon Master	IRQ 0
		jtag_debug_module	Avalon Slave	0x02002800
<input checked="" type="checkbox"/>		cpu2	Nios II Processor	
		instruction_master	Avalon Master	
		data_master	Avalon Master	IRQ 0
		jtag_debug_module	Avalon Slave	0x00008000
<input checked="" type="checkbox"/>		cpu3	Nios II Processor	
		instruction_master	Avalon Master	
		data_master	Avalon Master	IRQ 0
		jtag_debug_module	Avalon Slave	0x00001000
<input checked="" type="checkbox"/>		DDR_SDRAM_controller	DDR SDRAM High Performance Control...	
		s1	Avalon Slave	0x01000000
<input checked="" type="checkbox"/>		message_buffer_RAM	On-Chip Memory (RAM or ROM)	
		s1	Avalon Slave	0x02001000
<input checked="" type="checkbox"/>		message_buffer_mutex	Mutex	
		s1	Avalon Slave	0x02003000
<input checked="" type="checkbox"/>		external_SSRAM_bus	Avalon-MM Tristate Bridge	
		avalon_slave	Avalon Slave	0x00000000
		tristate_master	Avalon Tristate Master	
<input checked="" type="checkbox"/>		external_SSRAM	Cypress CY7C1380C SSRAM	
		s1	Avalon Tristate Slave	0xcfffffff

Figure 11–3 illustrates the default system interconnect fabric for the system in Figure 11–2.

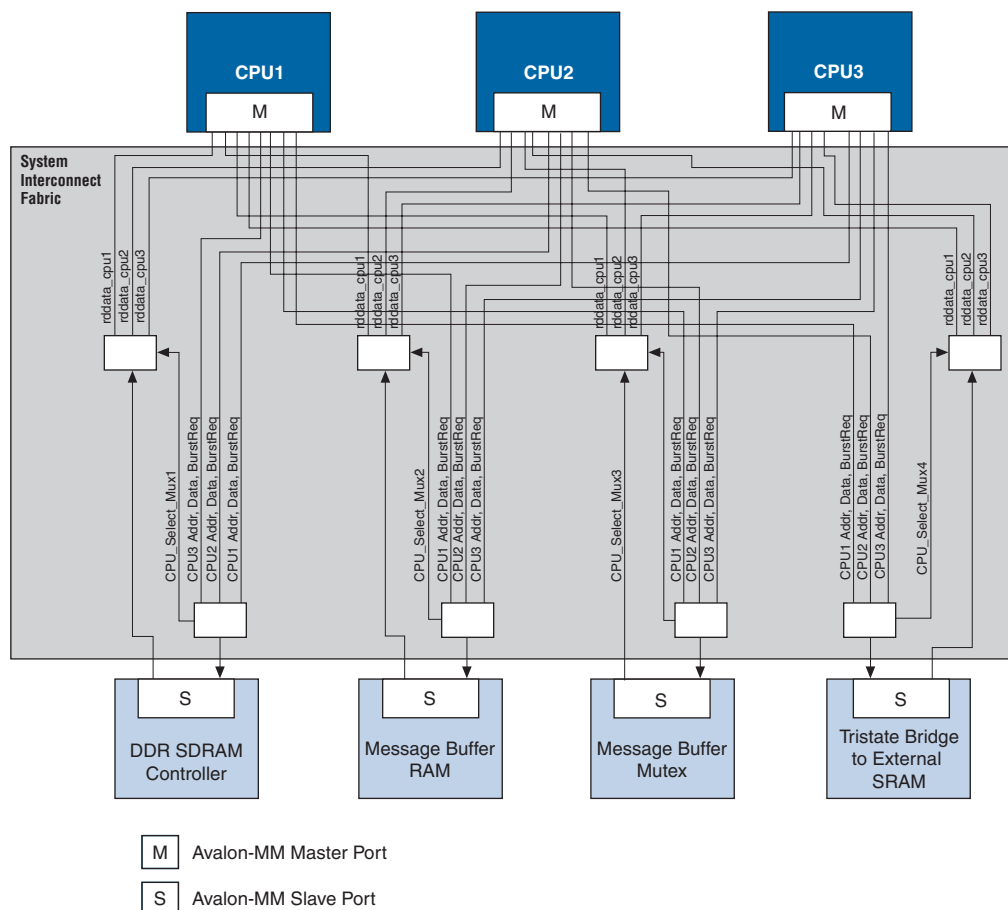
Figure 11–3. Example System without Bridges - System Interconnect View

Figure 11–4 and Figure 11–5 show how inserting bridges can affect the generated logic. For example, if the DDR SDRAM controller can run at 166 MHz and the CPUs accessing it can run at 120 MHz, inserting an Avalon-MM clock-crossing bridge between the CPUs and the DDR SDRAM has the following benefits:

- Allows the CPU and DDR interfaces to run at different frequencies.
- Places system interconnect fabric for the arbitration logic and multiplexer for the DDR SDRAM controller in the slower clock domain.
- Reduces the complexity of the interconnect logic in the faster domain, allowing the system to achieve a higher f_{MAX} .

In the system illustrated in Figure 11–4, the message buffer RAM and message buffer mutex must respond quickly to the CPUs, but each response includes only a small amount of data. Placing an Avalon-MM pipeline bridge between the CPUs and the message buffers results in the following benefits:

- Eliminates separate arbiter logic for the message buffer RAM and message buffer mutex, which reduces logic utilization and propagation delay, thus increasing the f_{MAX} .
- Reduces the overall size and complexity of the system interconnect fabric.

Figure 11–4. Example SOPC System with Bridges—SOPC Builder View

Use	Connections	Module Name	Description	Clock	Base	End	IRQ
<input checked="" type="checkbox"/>		<input type="checkbox"/> cpu1	Nios II Processor				
		instruction_master	Avalon Master	clk			
		data_master	Avalon Master				
		jtag_debug_module	Avalon Slave				
<input checked="" type="checkbox"/>		<input type="checkbox"/> cpu2	Nios II Processor				
		instruction_master	Avalon Master	clk			
		data_master	Avalon Master				
		jtag_debug_module	Avalon Slave				
<input checked="" type="checkbox"/>		<input type="checkbox"/> cpu3	Nios II Processor				
		instruction_master	Avalon Master	clk			
		data_master	Avalon Master				
		jtag_debug_module	Avalon Slave				
<input checked="" type="checkbox"/>		<input type="checkbox"/> bridge	Avalon-MM Clock Crossing Bridge				
		s1	Avalon Slave	clk	0x00000000	0x01ffffff	
		m1	Avalon Master	clk			
<input checked="" type="checkbox"/>		<input type="checkbox"/> ddr_sdram	DDR SDRAM Controller MegaCore Fun...				
		s1	Avalon Slave	clk	0x00000000	0x01ffffff	
<input checked="" type="checkbox"/>		<input type="checkbox"/> pipeline_bridge	Avalon-MM Pipeline Bridge				
		s1	Avalon Slave	clk	0x02000000	0x02001fff	
		m1	Avalon Master				
<input checked="" type="checkbox"/>		<input type="checkbox"/> message_buffer_ram	On-Chip Memory (RAM or ROM)				
		s1	Avalon Slave	clk	0x00000000	0x00000fff	
<input checked="" type="checkbox"/>		<input type="checkbox"/> message_buffer_mu...	Mutex				
		s1	Avalon Slave	clk	0x00001000	0x00001007	
<input checked="" type="checkbox"/>		<input type="checkbox"/> ext_ssram_bus	Avalon-MM Tristate Bridge				
		avalon_slave	Avalon Slave	clk			
		tristate_master	Avalon Tristate Master				
<input checked="" type="checkbox"/>		<input type="checkbox"/> ext_ssram	Cypress CY7C1380C SSRAM				
		s1	Avalon Tristate Slave	clk	0x03200000	0x033fffff	

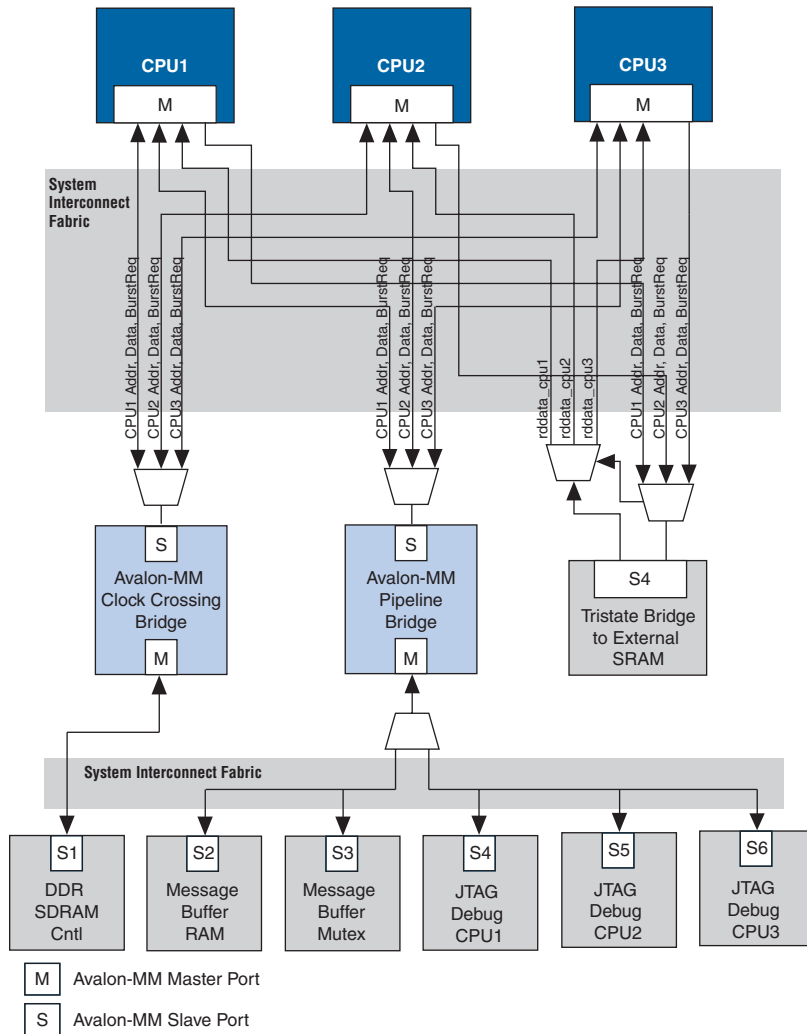


If an orange triangle appears next to an address as shown above, it indicates that the address is an offset value and is not the true value of the address in the address map.

This system also places the JTAG debug modules for each of the CPUs behind the Avalon pipeline bridge, which speeds up communication between the instruction and data masters, and the JTAG debug modules.

Figure 11–5 shows the system interconnect fabric that SOPC Builder creates for the system in Figure 11–4. Figure 11–5 is the same system that is pictured in Figure 11–3 with bridges to control system topology.

Figure 11–5. Example System with a Bridge



Address Mapping for Systems with Avalon-MM Bridges

An Avalon-MM bridge has an address span and range that are defined as follows:

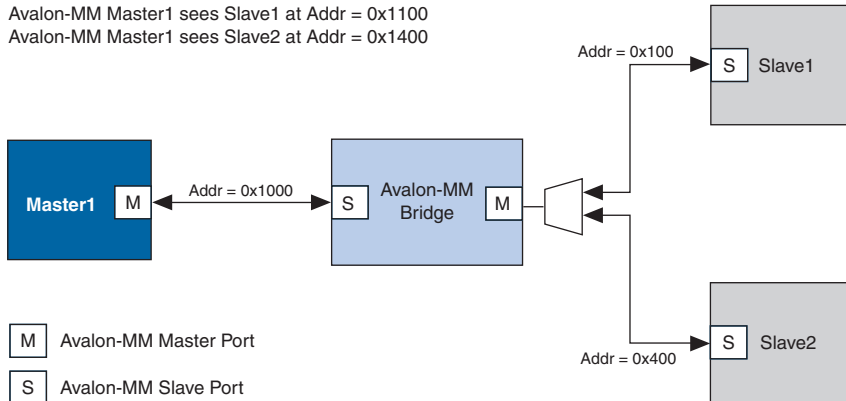
- The address *span* of an Avalon-MM bridge is the smallest power-of-two size that encompasses all of its slave's ranges.
- The address *range* of an Avalon-MM bridge is a numerical range from its base address to its base address plus its (span -1).

$$(1) \quad \text{range} = [\text{base_address} .. (\text{base_address} + (\text{span} - 1));$$

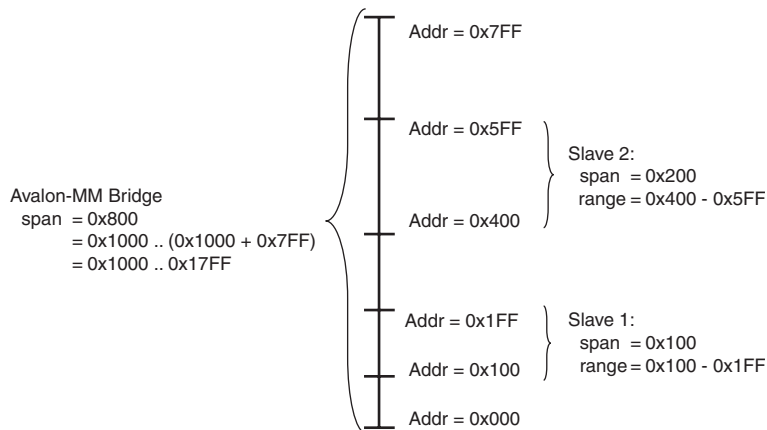
SOPC Builder follows several rules in constructing an address map for a system with Avalon-MM bridges:

1. The address span of each Avalon-MM slave is rounded up to the nearest power of two.
2. Each Avalon-MM slave connected to a bridge has an address relative to the base address of the bridge. This address must be a multiple of its span. (See Figure 11-6.)

Figure 11-6. Avalon-MM Master and Slave Addresses



3. In the example shown in Figure 11-6, if the address span of Slave 1 is 0x100 and the address span of Slave 2 is 0x200, Figure 11-7 illustrates the address span of the Avalon-MM bridge.

Figure 11–7. The Address Span of an Avalon-MM Bridge

Tools for Visualizing the Address Map

The Base Address column of the **System Contents** tab displays the base address *offset* of the Avalon-MM slave relative to the base address of the Avalon-MM bridge to which it is connected. You can see the absolute address map for each master in the system by clicking **Address Map** on the **System Contents** tab.

Differences between Avalon-MM Bridges and Avalon-MM Tristate Bridges

You use Avalon-MM bridges to control topology and separate clock domains for on-chip components. You use tristate bridges to connect to off-chip components and to share pins, decreasing the overall pin count of the device. You also use tristate bridges to change bi-directional input data into uni-directional input and output data signals. Tristate bridges are *transparent*, meaning that they do not affect the addresses of the components to which they connect.



For more information about the Avalon-MM tristate bridge, refer to the [SOPC Builder Memory Subsystem Development Walkthrough](#) chapter in volume 4 of the *Quartus II Handbook*.

Avalon-MM Pipeline Bridge

This section describes the hardware structure and functionality of the Avalon-MM pipeline bridge component.

Component Overview

The Avalon-MM pipeline bridge inserts registers in the path between its master and slaves. In a given SOPC Builder system, if the critical register-to-register delay occurs in the system interconnect fabric, the pipeline bridge can help reduce this delay and improve system f_{MAX} .

The bridge allows you to independently pipeline different groups of signals that can create a critical timing path in the interconnect:

- Master-to-slave signals, such as address, write data, and control signals
- Slave-to-master signals, such as read data
- The `waitrequest` signal to the master



You can also use the Avalon-MM pipeline bridge to control topology without adding a pipeline stage. To instantiate a bridge that does not add any pipeline stages, simply do not select any of the **Pipeline Options** on the parameter page. For the system illustrated in [Figure 11–5](#), a pipeline bridge that does not add a pipeline register stage is optimal because the CPUs require minimal delay from the message buffer mutex and message buffer RAM.

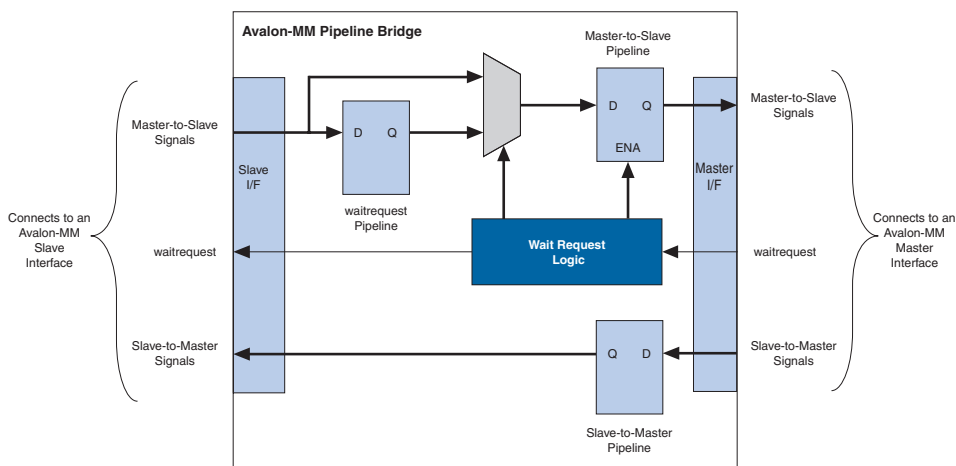


A pipeline bridge with no latency cannot be used with slaves that support pipelined reads. If a slave does not have read latency, you cannot connect it to a bridge with no pipeline stages, because the pipeline bridge slave port has a `readdatavalid` signal. Pipelined read components cannot have zero read latency.

The Avalon-MM pipeline bridge component integrates easily into any SOPC Builder system.

Functional Description

[Figure 11–8](#) shows a block diagram of the Avalon-MM pipeline bridge component.

Figure 11–8. Avalon-MM Pipeline Bridge Block Diagram

The following sections describe the component's hardware functionality.

Interfaces

The bridge interface is composed of an Avalon-MM slave and an Avalon-MM master. The data width of the ports is configurable, which can affect how SOPC Builder generates dynamic bus sizing logic in the system interconnect fabric. Both ports support Avalon-MM pipelined transfers with variable latency. Both ports optionally support bursts of length you can configure.

Pipeline Stages and Effects on Latency

The bridge provides three optional register stages to pipeline the following groups of signals.

- Master-to-slave signals, including:
 - address
 - writedata
 - write
 - read
 - byteenable
 - chipselect
 - burstcount (optional)

- Slave-to-master signals, including:
 - `readdata`
 - `readdatavalid`
- The `waitrequest` signal to the master

When you include a register stage, it affects the timing and latency of transfers through the bridge, as follows:

- The latency increases by one cycle in each direction.
- Write transfers on the master side of the bridge are decoupled from write transfers on the slave side of the bridge because Avalon-MM write transfers do not require an acknowledge signal from the slave.
- Including the `waitrequest` register stage increases the latency of master-to-slave signals by one additional cycle when the `waitrequest` signal is asserted.

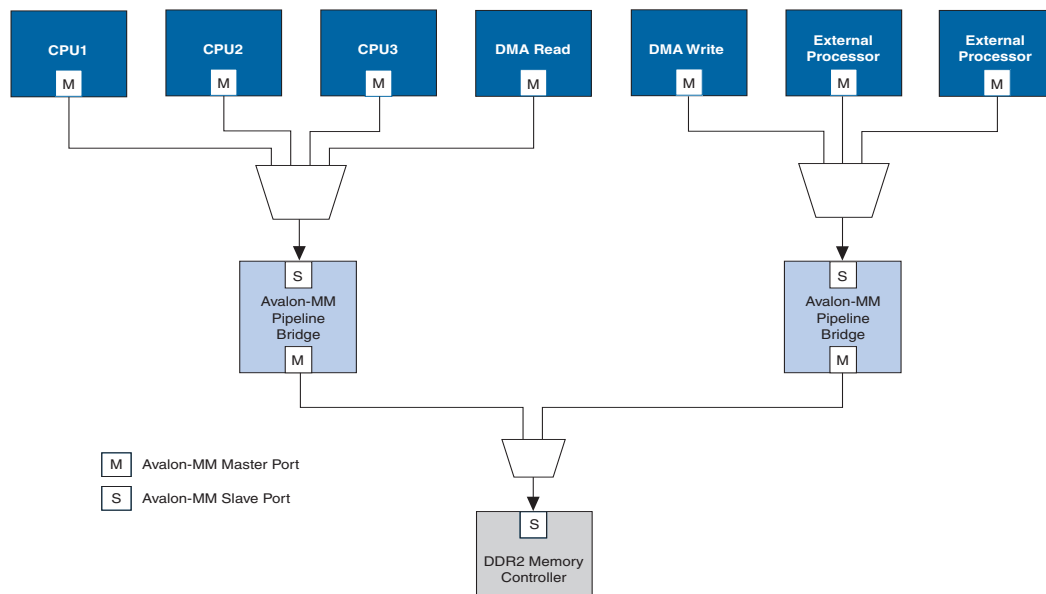
Burst Support

The bridge can support bursts with configurable maximum burst length. When configured to support bursts, the bridge propagates bursts between master-slave pairs, up to the maximum burst length. Not having burst support is equivalent to a maximum burst length of one. In this case, the system interconnect fabric automatically decomposes master-to-bridge bursts into a sequence of individual transfers.

Example System with Avalon-MM Pipeline Bridges

Figure 11–9 illustrates a system in which seven Avalon-MM masters are accessing a single DDR2 memory controller. By inserting two Avalon-MM pipeline bridges, you can limit the complexity of the multiplexer that would be required without the intermediate pipeline stage.

Figure 11–9. Seven Avalon-MM Masters Accessing One Avalon-MM Slave



Clock Crossing Bridge

This section describes the hardware structure and functionality of the Avalon-MM clock-crossing bridge component.

Component Overview

The Avalon-MM clock-crossing bridge allows you to connect Avalon-MM master and slaves that operate in different clock domains. Without a bridge, SOPC Builder automatically includes generic clock domain crossing (CDC) logic in the system interconnect fabric, but it does not provide optimal performance for high-throughput applications. Because the clock-crossing bridge includes a buffering mechanism, you can pipeline multiple read and write transfers. After an initial penalty for the first read or write, there is no additional latency penalty for pending reads and writes, increasing throughput at the expense primarily of on-chip memory. The clock-crossing bridge has parameterizeable FIFOs for master-to-slave and slave-to-master signals, and allows burst transfers across clock domains.

The Avalon-MM clock-crossing bridge component is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

Choosing Clock Crossing Methodology

When determining clock frequencies for your components, you should also consider the impact on the latency that transferring data across clock domains can cause. Whether you use a clock-crossing bridge or rely on the clock domain adapter created automatically by SOPC Builder, additional latency occurs. You should also consider the resource usage and throughput capabilities of each solution.

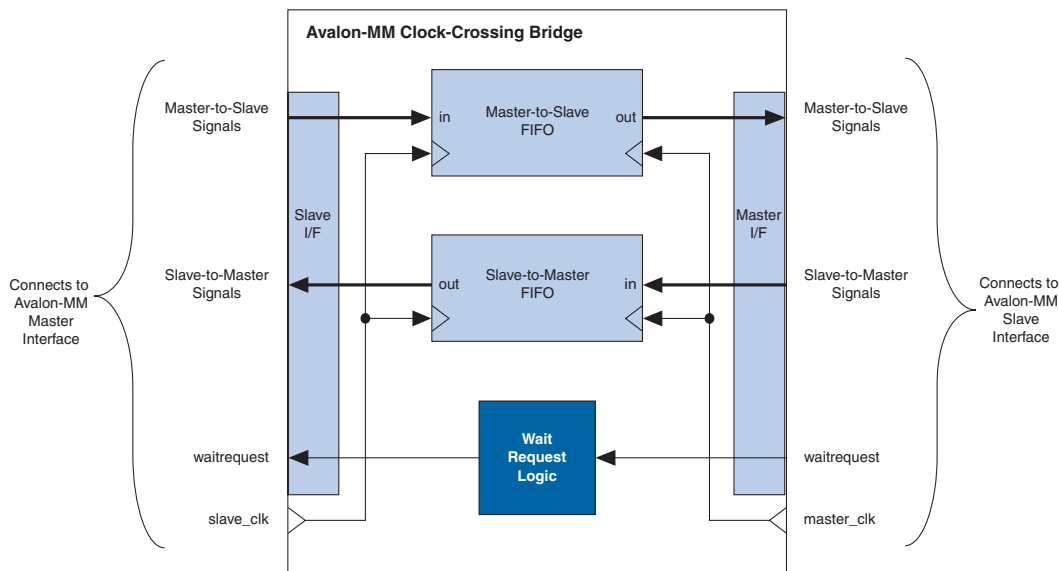
If you use the clock domain adapter to connect master and slave ports driven by separate clock inputs, there is a fixed latency penalty associated to each transfer. Each transfer becomes blocking, meaning that while one transfer is underway another cannot begin until the first completes. For this reason, you should not connect high speed pipelined components such as SDRAM memory to a master on a different clock domain without using a clock-crossing bridge between them. The clock crossing bridge can queue multiple transfers, so that even though the latency increases, the throughput does not decrease.

Because a clock domain adapter is generated for every master and slave pair, you should use a clock crossing bridge if your design contains multiple master and slave pairs operating in different clock domains. Alternatively, if your design uses a large amount of on-chip memory resources, you may need to use a clock domain adapter, because the clock-crossing bridge requires memory resources to be available.

Functional Description

Figure 11–10 shows a block diagram of the Avalon-MM clock-crossing bridge component. The following sections describe the component's hardware functionality.

Figure 11–10. Avalon-MM Clock-Crossing Bridge Block Diagram



Interfaces

The bridge interface comprises an Avalon-MM slave and an Avalon-MM master. The data width of the ports is configurable, which affects the size of the bridge hardware and how SOPC Builder generates dynamic bus sizing logic in the system interconnect fabric. Both ports support Avalon-MM pipelined transfers with variable latency. Both ports optionally support bursts of user-configurable length. Ideally, the settings for one port match the other, such that there are no mixed data widths or bursting capabilities.

Clock Domain Adapter and FIFOs

Two FIFOs in the bridge transport address, data, and control signals across the clock-domains. One FIFO captures data and controls traveling in the master-to-slave direction, and the other FIFO captures data in the

slave-to-master direction. CDC logic surrounding the FIFOs coordinates the details of passing data across the clock-domain boundaries and ensures that the FIFOs do not overflow or underflow.

The signals that pass through the master-to-slave FIFO include:

- `writedata`
- `address`
- `read`
- `write`
- `byteenable`
- `burstcount`, when bursting is enabled

The signals that pass through the slave-to-master FIFO include:

- `readdata`
- `readdatavalid`
- `endofpacket`

You can configure the depth of each FIFO. Because there are more signals traveling in the master-to-slave direction, changing the depth of the master-to-slave FIFO has a greater impact on the memory utilization of the bridge.

For read transfers across the bridge, the FIFOs in both directions incur latency for data to return from the slave. To avoid paying a latency penalty for each transfer, the master can issue multiple reads that are queued in the FIFO. The slave of the bridge asserts `readdatavalid` when it drives valid data and asserts `waitrequest` when it is not ready to accept more reads.

For write transfers, the master-to-slave FIFO causes a delay between the master-to-bridge transfers and the corresponding bridge-to-slave transfers. Because Avalon-MM write transfers do not require an acknowledge from the slave, multiple write transfers from master-to-bridge might complete by the time the bridge initiates the corresponding bridge-to-slave transfers.

Burst Support

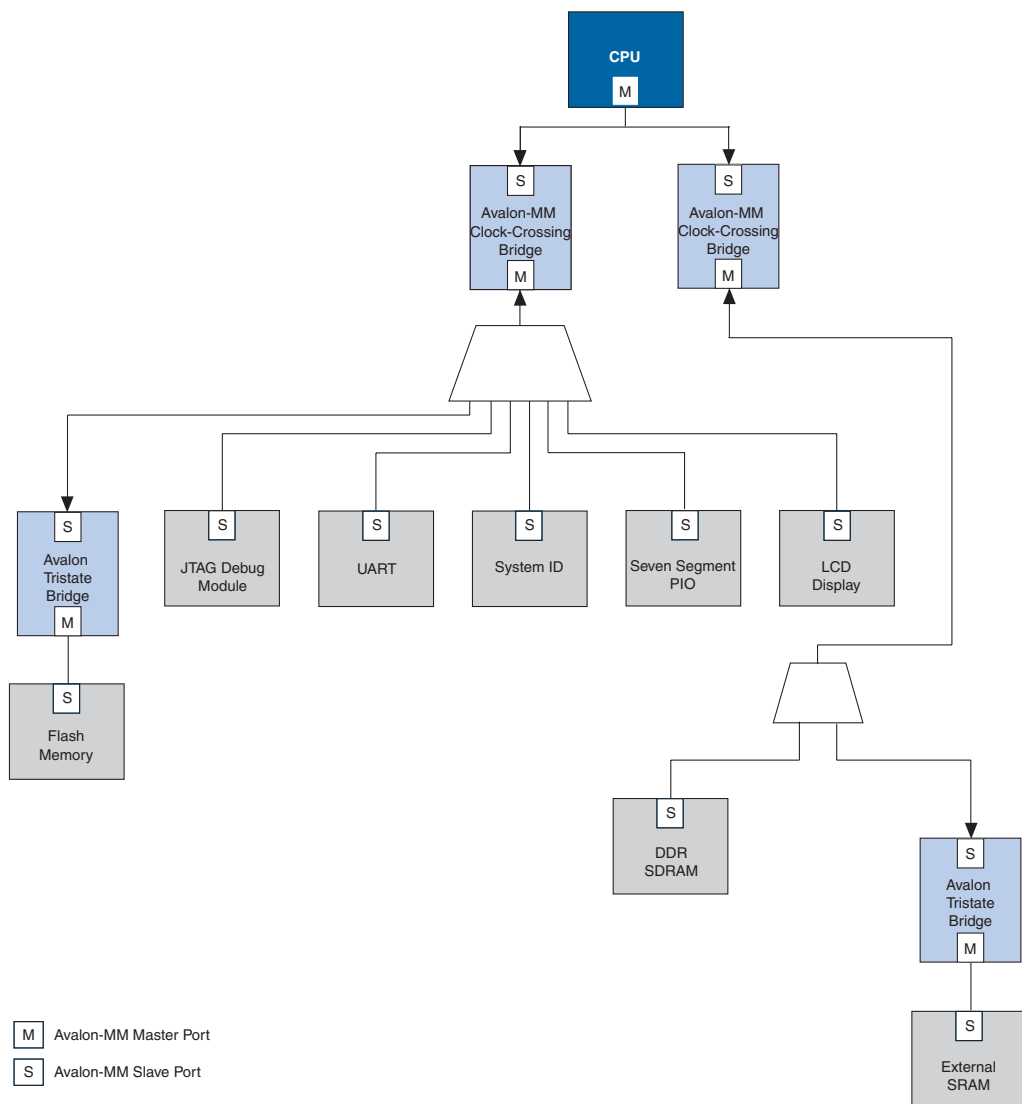
The bridge optionally supports bursts with configurable maximum burst length. When configured to support bursts, the bridge propagates bursts between master-slave pairs, up to the maximum burst length. Not having burst support is equivalent to a maximum burst length of one. In this case, the system interconnect fabric automatically deconstructs master-to-bridge bursts into a sequence of individual transfers.

When you configure the bridge to support bursts, you must configure the slave-to-master FIFO depth deeply enough to capture all burst read data without overflowing. The masters connected to the bridge could potentially fill the master-to-slave FIFO with read burst requests; therefore, the minimum slave-to-master FIFO depth is described in the following equation:

Example 11–1. Minimum Slave-To-Master FIFO Depth
$$= ((\text{master-to-slave FIFO depth}) * (\text{max burst length})) + \text{max slave latency/pending reads}$$

Example System with Avalon-MM Clock-Crossing Bridges

Figure 11–11 uses Avalon-MM clocking crossing bridges to separate slave components into two groups. The low-performance slave components are placed behind a single bridge and clocked at a low speed. The high performance components are placed behind a second bridge and clocked at a higher speed. By inserting clock-crossing bridges in the system, you optimize the interconnect fabric and allow the Quartus II fitter to expend effort optimizing paths that require minimal propagation delay.

Figure 11–11. One Avalon-MM Master with Two Groups of Avalon-MM Slaves

Instantiating the Avalon-MM Clock-Crossing Bridge in SOPC Builder

This section describes the options available on the **Parameter Settings** page of the Megawizard interface.

- **Master-to-slave FIFO**—You can use these options to specify the size and structure of the master-to-slave FIFO.
 - **FIFO depth**—Determines the depth of the FIFO.
 - **Construct FIFO from registers**—When you turn on this option, the FIFO uses registers as storage instead of embedded memory blocks. This can considerably increase the size of the bridge hardware and lower the f_{MAX} .
- **Slave-to-master FIFO**—You can use these options to specify the size and structure of the slave-to-master FIFO.
 - **FIFO depth**—Determines the depth of the FIFO.
 - **Construct FIFO from registers**—When you turn on this option, the FIFO uses registers as storage instead of embedded memory blocks. This can considerably increase the size of the bridge hardware.
- **Data width**—Determines the data width of the master and slaves on the bridge, and affects the size of both FIFOs.

For the highest bandwidth, set **Data width** to be as wide as the widest master connected to the bridge.

- **Allow bursts**—Includes logic for the bridge's master and slaves to support bursts. You can use this option to restrict the minimum depth for the slave-to-master FIFO.
- **Maximum burst size**—Determines the maximum length of bursts for the bridge to support, when you turn on **Allow bursts**.

Clock Domain Crossing

SOPC Builder generates CDC logic that hides the details of interfacing components operating in different clock domains. The system interconnect fabric upholds the Avalon-MM protocol with each port independently, and therefore masters do not need to incorporate clock adapters in order to interface to slaves on a different domain. The system interconnect fabric logic propagates transfers across clock domain boundaries automatically.

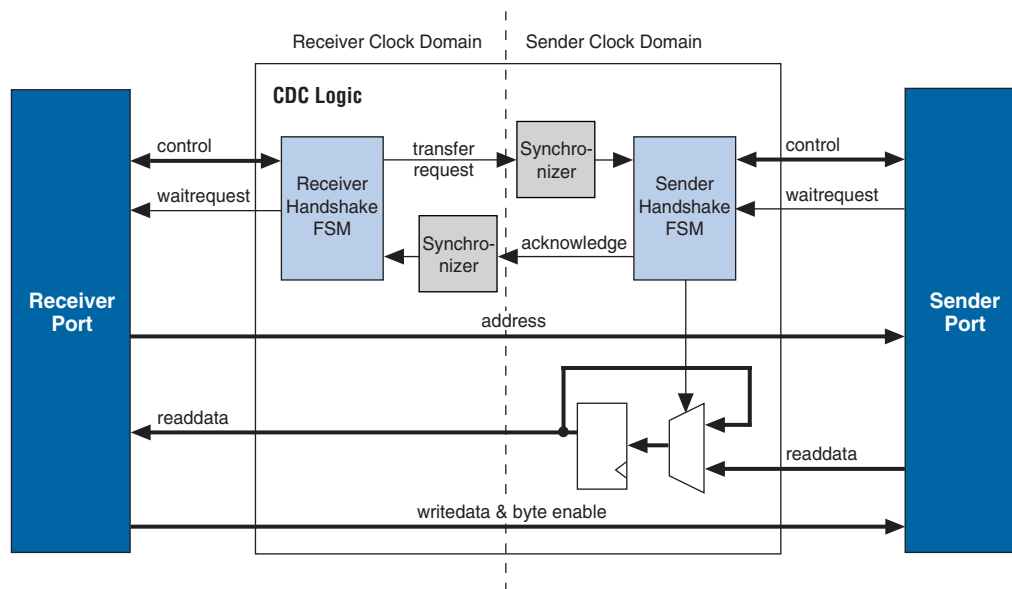
The clock-domain adapters in system interconnect fabric provide the following benefits that simplify system design efforts:

- Allow component interfaces to operate at a different clock frequency than system logic.
- Eliminate the need to design CDC hardware manually.
- Each Avalon-MM port operates in only one clock domain, which reduces design complexity of components.
- Enable masters to access any slave without communication with the slave clock domain.
- Allow you to focus performance optimization efforts only on components that require fast clock speed.

Description of Clock Domain Adapter

The clock domain adapter consists of two finite state machines (FSM), one in each clock domain, that use a simple hand-shaking protocol to propagate transfer control signals (`read request`, `write request`, and the `master wait-request` signals) across the clock boundary.

Figure 11–12 shows a block diagram of the clock domain adapter between one master and one slave.

Figure 11–12. Block Diagram of Clock Domain Adapter

The synchronizer blocks in Figure 11–12 use multiple stages of flip-flops to eliminate the propagation of metastable events on the control signals that enter the handshake FSMs.

The CDC logic works with any clock ratio. Altera® tests the CDC logic extensively on a variety of system architectures, both in simulation and in hardware, to ensure that the logic functions correctly.

The typical sequence of events for a transfer across the CDC logic is described below:

1. Master asserts address, data, and control signals.
2. The master handshake FSM captures the control signals, and immediately forces the master to wait.



The FSM uses only the control signals, not address and data. For example, the master simply holds the address signal constant until the slave side has safely captured it.

3. Master handshake FSM initiates a transfer request to the slave handshake FSM.

4. The transfer request is synchronized to the slave clock domain.
5. The slave handshake FSM processes the request, performing the requested transfer with the slave.
6. When the slave transfer completes, the slave handshake FSM sends an acknowledge back to the master handshake FSM.
7. The acknowledge is synchronized back to the master clock domain.
8. The master handshake FSM completes the transaction by releasing the master from the wait condition.

Transfers proceed as normal on the slave and the master side, without a special protocol to handle crossing clock domains. From the perspective of a slave, there is nothing different about a transfer initiated by a master in a different clock domain. From the perspective of a master, a transfer across clock domains simply requires extra clock cycles. Similar to other transfer delay cases (for example, arbitration delay or wait states on the slave side), the system interconnect fabric simply forces the master to wait until the transfer terminates. As a result, pipeline master ports do not benefit from pipelining when performing transfers to a different clock domain.

Location of Clock Domain Adapter



You can use the clock crossing bridge described in the following paragraphs for higher throughput clock crossing, at the expense of memory resources.

SOPC Builder automatically determines where to insert the CDC logic, based on the system contents and the connections between components. SOPC Builder places CDC logic to maintain the highest transfer rate for all components. SOPC Builder evaluates the need for CDC logic for each master and slave pair independently, and generates CDC logic wherever necessary.

Duration of Transfers Crossing Clock Domains

CDC logic extends the duration of master transfers across clock domain boundaries. In the worst case, each transfer is extended by five master clock cycles and five slave clock cycles. The components of this delay are the following:

- Four additional master clock cycles, due to the master-side clock synchronizer

- Four additional slave clock cycles, due to the slave-side clock synchronizer
- One additional clock in each direction, due to potential metastable events as the control signals cross clock domains



Systems that require a higher performance clock should use the Avalon-MM clock crossing bridge instead of the automatically inserted CDC logic. The clock crossing bridge includes a buffering mechanism, so that multiple reads and writes can be pipelined. After paying the initial penalty for the first read or write, there is no additional latency penalty for pending reads and writes, increasing throughput by up to four times, at the expense of added logic resources.



For more information, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook* and *Avalon Memory-Mapped Design Optimizations* in the *Embedded Design Handbook*.

Implementing Multiple Clock Domains in SOPC Builder

You specify the clock domains used by your system on the **System Contents** tab of SOPC Builder. You define the input clocks to the system with the **Clock Settings** table. Clock sources can be driven by external input signals to the SOPC Builder system, or by PLLs inside the SOPC Builder system. Clock domains are differentiated based on the name of the clock. You may create multiple asynchronous clocks with the same frequency.

You specify which clock drives which components using the table of active components after you define the system clocks, as shown in [Figure 11–13](#).

Figure 11–13. Assigning Clocks to Components

Module Name	Description	Clock	Base	End	IRQ
red_display	Character LCD (16x2, 8-bit)	clk	0x02120000	0x0212000F	1
high_res_timer	Interval timer	clk	0x02120820	0x0212083F	3
seven_seg_pio	PIO (Parallel I/O)	clk	0x02120890	0x0212089F	
reconfig_request_pio	PIO (Parallel I/O)	fastclk	0x021208A0	0x021208AF	
uart1	UART (RS-232 serial port)	clk	0x02120840	0x0212085F	4
sysid	System ID Peripheral	clk	0x021208B8	0x021208BF	
sdram	SDRAM Controller	clk	0x01000000	0x01FFFFFF	
dma_0	DMA	fastclk	0x00800000	0x0080001F	7
read_buffer	On-Chip Memory (RAM ...)	fastclk	0x00801000	0x00801FFF	
write_buffer	On-Chip Memory (RAM ...)	fastclk	0x00802000	0x00802FFF	

Alternatively, the clock connections can be shown by changing the filter settings, and then clock connections can be managed like any other connection type.

Device Support

Altera device support for the bridge components is listed in [Table 11–1](#).

<i>Table 11–1. Device Family Support</i>		
Device Family	Avalon-MM Pipeline Bridge Support	Avalon-MM Clock-Crossing Bridge Support
Arria™ GX	Full	Full
Stratix® III	Full	Full
Stratix II GX	Full	Full
Stratix II	Full	Full
Stratix®	Full	Full
Cyclone™ III	Full	Full
Cyclone II	Full	Full
Cyclone	Full	Full
HardCopy® II	Full	Full
MAX®	Full	No support
MAX II	Full	No support

Installation and Licensing

The bridge components are included in the Altera MegaCore® IP Library, which is part of the Quartus® II software installation. After you install the MegaCore IP Library, SOPC Builder recognizes the bridge components and can instantiate them into a system.

You can use the bridge components for free without a license in any design targeting an Altera device.

Hardware Simulation Considerations

The bridge components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

The bridge components do not have any user-visible control or status registers. Therefore, software cannot control or configure any aspect of the bridges during run-time. The bridges cannot generate interrupts.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Avalon Memory-Mapped Design Optimizations in the Embedded Design Handbook*
- *SOPC Builder Memory Subsystem Development Walkthrough* chapter in volume 4 of the *Quartus II Handbook*.
- *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 11–2 shows the revision history for this chapter.

Table 11–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
May 2008 v 8.0	<ul style="list-style-type: none"> • Chapter renumbered from 10 to 11. • Corrected Figure 11–4 to show correct connectivity between masters and bridges. Show JTAG debug modules for each CPU behind pipeline bridge. • Deleted references to Avalon Memory-Mapped and Streaming Interface Specifications and replaced with new Avalon Interface Specifications. • Moved clock crossing bridge section from Chapter 2 to this chapter. • Added note after Figure 10-4. 	—
October 2007 v7.2.0	Moved discussion of clock-crossing bridge from this chapter to chapter 2.	—
May 2007, v7.1.0	Initial release of the document.	The Avalon-MM Pipeline Bridge and Avalon-MM Clock-Crossing Bridge are new components provided in the Quartus II software v7.1 release.

Introduction to Interconnect Components

Avalon® Streaming (Avalon-ST) interconnect components facilitate the design of high-speed, low-latency datapaths for the system-on-a-programmable-chip (SOPC) environment. Interconnect components in SOPC Builder act as a part of the system interconnect fabric. They are not end points, but adapters that allow you to connect different, but compatible, streaming interfaces. You use Avalon-ST interconnect components to connect cores that send and receive high-bandwidth data, including multiplexed streams, packets, cells, time division multiplexed (TDM) frames, and digital signal processor (DSP) data.

The interconnect components that you add to an SOPC Builder system insert logic between a source and sink interface, enabling that interface to operate correctly. This chapter describes three Avalon-ST interconnect components, also called adapters:

- [“Timing Adapter” on page 12–4](#)—adapts between adapted sinks and sources that have different characteristics, such as ready latencies.
- [“Data Format Adapter” on page 12–7](#)—adapts source and sink interfaces that have different data widths.
- [“Channel Adapter” on page 12–10](#)—adapts source and sink interfaces that have different settings for the channel signal.

All of these interconnect components adapt initially incompatible Avalon-ST source and sink interfaces so that they function correctly, facilitating the development of high-speed, low-latency datapaths.

Interconnect Component Usage

Interconnect components can adapt the data or control signals of the Avalon-ST interface. Typical adaptations to control signals include:

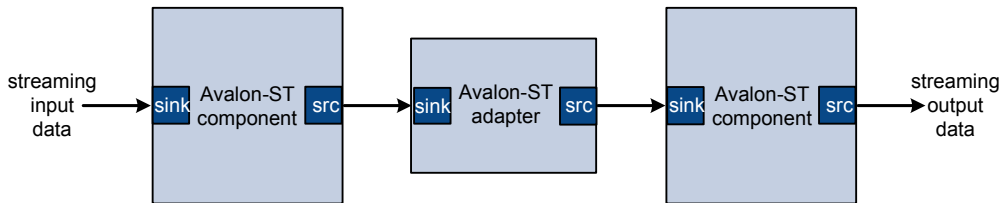
- Adding pipeline stages to adjust the timing of the ready signal
- Tying signals that are not used by either the source or sink to 0 or 1

Typical adaptations to data signals include:

- Changing the number of symbols (words) that are driven per cycle
- Changing the number of channels driven

When the interconnect component adapts the data interface, it has one Avalon-ST sink interface and one Avalon-ST source interface, as shown in [Figure 12–1](#). You configure the adapter components manually, using SOPC Builder. In contrast to the Avalon-MM interface, which allows you to create various topologies with a number of different master and slave components, you always use the Avalon-ST interconnect components to adapt point-to-point connections between streaming cores.

Figure 12–1. Example of an Avalon-ST Interconnect Component in an SOPC Builder System



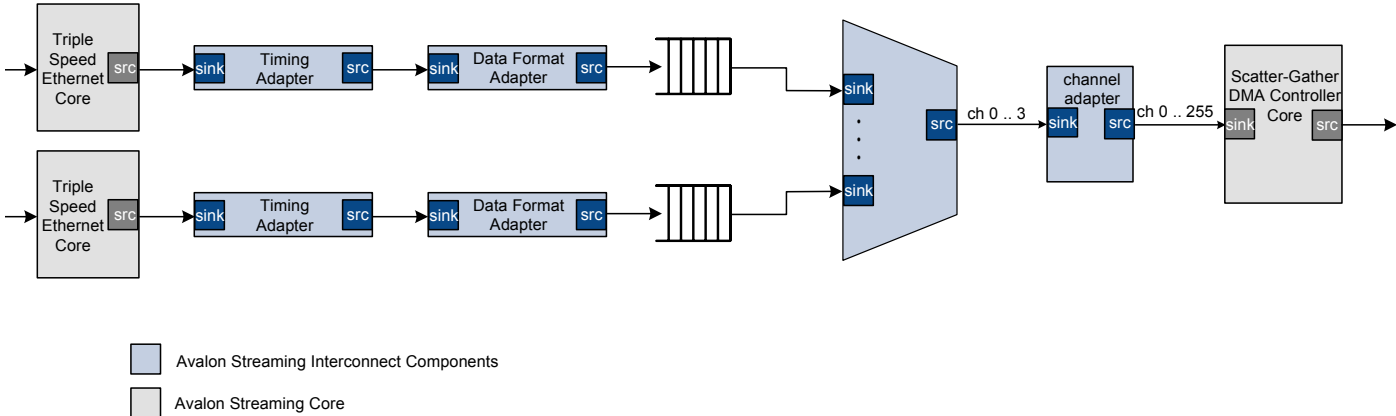
For details about the system interconnect fabric, refer to the [System Interconnect Fabric for Streaming Interfaces](#) chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface protocol, refer to the [Avalon Interface Specifications](#).

[Figure 12–2](#) illustrates a datapath that connects a Triple Speed Ethernet core to a scatter-gather DMA controller core using a timing adapter, data format adapter, and channel adapter so that the cores can interoperate.

Address Mapping

The signals of the Avalon-ST source and sink interfaces are mapped into the global Avalon address space.

Figure 12–2. Avalon-ST Datapath Constructed Using Avalon Streaming Interconnect Components



Timing Adapter

The timing adapter has two functions:

- It adapts source and sink interfaces that support the `ready` signal and those that do not.
- It adapts source and sink interfaces that have different `ready` latencies.

The timing adapter treats all signals other than the `ready` and `valid` signals as *payload*, and simply drives them from the source to the sink.

Table 12–1 outlines the adaptations that the timing adapter provides.

Table 12–1. Timing Adapter	
Condition	Adaptation
The source has <code>ready</code> , but the sink does not.	In this case, the source can respond to backpressure, but the sink never needs to apply it. The <code>ready</code> input to the source interface is connected directly to logical '1'.
The source does not have <code>ready</code> , but the sink does.	The sink may apply backpressure, but the source is unable to respond to it. There is no logic that the adapter can insert that prevents data loss when the source asserts <code>valid</code> but the sink is not <code>ready</code> . The adapter provides simulation time error messages and an error indication if data is ever lost. The user is presented with a warning, and the connection is allowed.
The source and sink both support backpressure, but the sink's <code>ready</code> latency is greater than the source's.	The source responds to <code>ready</code> assertion or deassertion faster than the sink requires it. A number of pipeline stages equal to the difference in <code>ready</code> latency are inserted in the <code>ready</code> path from the sink back to the source, causing the source and the sink to see the same cycles as <code>ready</code> cycles.
The source and sink both support backpressure, but the sink's <code>ready</code> latency is less than the source's.	The source cannot respond to <code>ready</code> assertion or deassertion in time to satisfy the sink. A buffer whose depth is equal to the difference in <code>ready</code> latency is inserted to compensate for the source's inability to respond in time.

Resource Usage and Performance

Resource utilization for the timing adapter depends upon the function that it performs. Table 12–2 provides estimated resource utilization for seven different configurations of the timing adapter.

Table 12–2. Timing Adapter Estimated Resource Usage and Performance

Input Ready Latency	Output Ready Latency	Stratix® II and Stratix II GX (Approximate LEs)			Cyclone® II		Stratix (Approximate LEs)		
		f _{MAX} (MHz)	ALM Count	Mem Bits	f _{MAX} (MHz)	Logic Cells	f _{MAX} (MHz)	Logic Cells	Mem Bits
1	2	500	2	0	420	2	422	1	0
1	3	500	2	0	420	3	422	2	0
1	4	500	4	0	420	4	422	3	0
1	0	500	21	80	420	183	422	20	80
2	1	456	21	80	401	188	317	21	80
3	1	456	21	80	401	188	317	21	80
4	1	456	21	80	401	188	317	21	80

Instantiating the Timing Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. This section describes the options available on the **Parameter Settings** page of the configuration wizard.

Input Interface Parameters

Support Backpressure with the ready signal—turn on this option to add the backpressure functionality to the interface. When the `ready` signal is used, the value for `READY_LATENCY` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.

Output Interface Parameters

Support Backpressure with the ready signal—turn on this option to add the backpressure functionality to the interface. When the `ready` signal is used, the value for `READY_LATENCY` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.

Common to Input and Output Interfaces

The following parameters define the interface characteristics that the adapters do not affect directly.

Channel Signal Width (Bits)

Set the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is eight bits. Set to 0 if channels are not used.

Max Channel

Set the maximum number of channels that the interface supports. Valid values are 0-255.

Bits Per Symbol

Set the number of bits per symbol.

Symbols Per Beat

Record the number of symbols per active transfer.

Include Packet Support

Turn on this option if the interfaces supports a packet protocol, including the `startofpacket`, `endofpacket` and empty signals.

Error Signal Width (Bits)

Record the width of the `error` signal. Valid values are 0–31 bits. Set to 0 if the `error` signal is not used.

Data Format Adapter

The data format adapter handles interfaces that have different definitions for the `data` signal. One of the more common adaptations that this adapter performs is bus width adaptation, such as converting a data interface that drives two, 8-bit symbols per beat to an interface that drives four, 8-bit symbols per beat. The available data format adaptations are listed in [Table 12–3](#).

Table 12–3. Data Format Adapter	
Condition	Description of Adapter Logic
The source and sink's bits per symbol are different.	The connection cannot be made.
The source and sink have a different number of symbols per beat.	<p>The adapter converts from the source's width to the sink's width.</p> <p>If the adaptation is from a wider to a narrower interface, a beat of data at the input will correspond to multiple beats of data at the output. If the input <code>error</code> signal is asserted for a single beat, it is asserted on output for multiple beats.</p> <p>If the adaptation is from a narrow to a wider interface, multiple input beats are required to fill a single output beat, and the output <code>error</code> is the logical OR of the input <code>error</code> signal.</p>

Resource Usage and Performance

Resource utilization for the data format adapter depends upon the function that it performs. [Table 12–4](#) provides estimated resource utilization for numerous configurations of the data format adapter.

Table 12–4. Data Format Adapter Estimated Resource Usage and Performance, 8 Bits per Symbol

Input Symb ols per Beat	Output Symb ols per Beat	Numb er of Chann els	Packet Suppo rt	Stratix® II and Stratix II GX (Approximate LEs)			Cyclone® II			Stratix (Approximate LEs)		
				f _{MAX} (MHz)	ALM Count	Mem Bits	f _{MAX} (MHz)	Logic Cells	Memo ry Bits	f _{MAX} (MHz)	Logic Cells	Mem Bits
1	2	1	y	500	96	0	391	93	0	375	105	0
4	1	1	y	459	106	0	311	97	0	306	76	0
4	2	1	y	500	118	0	343	107	0	326	85	0
4	8	1	y	437	326	0	346	370	0	303	330	0
4	16	1	y	357	930	0	264	1005	0	231	806	0
1	2	188	y	321	110	15	187	137	15	209	153	15
4	1	105	y	244	125	2	148	183	2	150	137	2
4	2	105	y	277	101	2	172	134	2	173	108	2
4	8	130	y	322	255	41	175	279	41	187	262	41
4	16	30	y	268	341	106	166	563	106	153	471	106
4	1	105	n	269	107	2	177	185	2	167	99	2
4	2	54	n	290	109	1	193	203	1	176	91	1
4	3	10	n	249	149	18	189	251	16	159	217	18
4	5	222	n	281	300	40	199	381	40	182	316	40
4	6	30	n	312	184	40	201	385	40	198	241	40
4	7	139	n	253	285	56	159	416	56	161	427	56
4	8	198	n	311	281	40	190	247	40	198	257	40
4	15	160	n	259	370	121	165	733	121	149	697	121
4	16	36	n	227	255	105	391	93	0	146	491	105

Instantiating the Data Format Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. This section describes the options available on the **Parameter Settings** page of the configuration wizard.

Input Interface Parameters

Data Symbols Per Beat

Set the number of symbols transferred per active cycle.

Output Interface Parameters

Data Symbols Per Beat

Set the number of symbols transferred per active cycle. This value can be different for the input and output interfaces.

Common to Input and Output

The following parameters define the interface characteristics that the adapters do not affect directly.

Support Backpressure with the Ready Signal

This option adds the backpressure functionality to the interface. When the `ready` signal is used, the value for `READY_LATENCY` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.

Data Bits Per Symbol

Record the number of bits per symbol. This value must be the same for the input and output interfaces.

Channel Signal Width (Bits)

Record the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is 8 bits. Set to 0 if channels are not used.

Max Channel

Record the maximum number of channels that the interface supports. Valid values are 0–255.

Include Packet Support

Turn this option on if the interface supports a packet protocol, including the `startofpacket`, `endofpacket`, and `empty` signals.

Error Signal Width (Bits)

Record the width of the error signal. Valid values are 0–31 bits. Set to 0 if the error signal is not used.

Channel Adapter

The channel adapter provides adaptations between interfaces that have different support for the `channel` signal or for the maximum number of channels supported. The adaptations are described in [Table 12–5](#).

Table 12–5. Channel Adapter	
Condition	Description of Adapter Logic
The source uses channels, but the sink does not.	The adapter provides a simulation error and signals an error for data for any channel from the source other than 0. You are given a warning at generation time.
The sink has channel, but the source does not.	You are given a warning, and the channel inputs to the sink are all tied to a logical '0'.
The source and sink both support channels, and the source's maximum number of channels is less than the sink's.	The source's channel is connected to the sink's channel unchanged. If the sink's channel signal has more bits, the higher bits are tied to a logical '0'.
The source and sink both support channels, but the source's maximum number of channels is greater than the sink's.	<p>The source's channel is connected to the sink's channel unchanged. If the source's channel signal has more bits, the higher bits are left unconnected. You are given a warning that channel information may be lost.</p> <p>An adapter provides a simulation error message and an error indication if the value of channel from the source is greater than the sink's maximum number of channels. In addition, the <code>valid</code> signal to the sink is deasserted so that the sink never sees data for channels that are out of range.</p>

Resource Usage and Performance

The channel adapter typically uses fewer than 30 LEs. Its frequency is limited by the maximum frequency of the device you choose.

Instantiating the Channel Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. This section describes the options available on the **Parameter Settings** page of the configuration wizard.

Input Interface Parameters

Channel Signal Width (Bits)

Set the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is 8 bits. Set to 0 if channels are not used.

Max Channel

Set the maximum number of channels that the interface supports. Valid values are 0–255.

Output Interface Parameters

Channel Signal Width (Bits)

Record the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is 8 bits. Set to 0 if channels are not used.

Max Channel

Set the maximum number of channels that the interface supports. Valid values are 0–255.

Common to Input and Output Interfaces

Support Backpressure with the ready signal—Turn on this option to add the backpressure functionality to the interface. When you use the `ready` signal, the value for `READY_LATENCY` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.

Data Bits Per Symbol

Set the number of bits per symbol.

Symbols Per Beat

Set the number of symbols per active cycle.

Include Packet Support

Turn this option on if the interface supports a packet protocol, including the `startofpacket`, `endofpacket` and empty signals.

Error Signal Width (Bits)

Set the width of the error signal. Valid values are 0–31 bits. Set to 0 if the error signal is not used.

Installation and Licensing

The Avalon-ST interconnect components are included in the Altera MegaCore IP Library, which is part of the Quartus® II software installation. After you install the MegaCore IP Library, SOPC Builder recognizes these components and can instantiate them into a system.

You can use the Avalon-ST components without a license in any design that targets an Altera device.

Hardware Simulation Considerations

The Avalon-ST interconnect components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

The Avalon-ST interconnect components do not have any control or status registers that you can see. Therefore, software cannot control or configure any aspect of the interconnect components at run-time. These components cannot generate interrupts.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 12–6 shows the revision history for this chapter.

<i>Table 12–6. Document Revision History</i>		
Date and Document Version	Changes Made	Summary of Changes
May 2008, v8.0.0	<ul style="list-style-type: none">Chapter renumbered from 11 to 12.Deleted references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications.	—
October 2007, v7.2.0	No changes to this release.	—
May 2007, v7.1.0	Initial release.	—

