



# **Quartus II Version 7.2 Handbook**

---

## **Volume 4: SOPC Builder**



101 Innovation Drive  
San Jose, CA 95134  
[www.altera.com](http://www.altera.com)

QI15V4-7.2

Copyright © 2007 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.



<b>Chapter Revision Dates .....</b>	<b>xi</b>
<b>About this Handbook.....</b>	<b>xiii</b>
How to Contact Altera .....	xiii
Typographic Conventions .....	xiii

## Section I. SOPC Builder Features

### Chapter 1. Introduction to SOPC Builder

Overview .....	1-1
Architecture of SOPC Builder Systems .....	1-2
SOPC Builder Components .....	1-2
Example System .....	1-3
Custom Components .....	1-4
System Interconnect Fabric .....	1-5
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
Getting Started .....	1-7
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 .....	2-8
Native Address Alignment and Dynamic Bus Sizing .....	2-9
Dynamic Bus Sizing .....	2-9
Wider Master .....	2-10
Narrower Master .....	2-10
Native Address Alignment .....	2-11
Arbitration for Multimaster Systems .....	2-12

Traditional Shared Bus Architectures .....	2-12
Slave-Side Arbitration .....	2-13
Arbiter Details .....	2-14
Arbitration Rules .....	2-15
Setting Arbitration Parameters in SOPC Builder .....	2-15
Fairness-Based Shares .....	2-16
Round-Robin Scheduling .....	2-17
Burst Transfers .....	2-17
Minimum Share Value .....	2-17
Burst Management .....	2-18
Clock Domain Crossing .....	2-19
Description of Clock Domain-Crossing Logic .....	2-19
Location of Clock Domain Crossing Logic .....	2-21
Duration of Transfers Crossing Clock Domains .....	2-22
Implementing Multiple Clock Domains in SOPC Builder .....	2-22
Component Overview .....	2-23
Functional Description .....	2-23
Interfaces .....	2-24
Clock Domain Crossing Logic and FIFOs .....	2-24
Burst Support .....	2-25
Example System with Avalon-MM Clock-Crossing Bridges .....	2-26
Instantiating the Avalon-MM Clock-Crossing Bridge in SOPC Builder .....	2-28
Interrupts .....	2-29
Software Priority .....	2-29
Hardware Priority .....	2-30
Assigning IRQs in SOPC Builder .....	2-30
Reset Distribution .....	2-31
Referenced Documents .....	2-31
Document Revision History .....	2-32

### 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 .....	3-4
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 .....	3-7

### Chapter 4. SOPC Builder Components

Introduction .....	4-1
--------------------	-----

New Component Structure in v7.1 of the Quartus II Software .....	4-1
Component Providers .....	4-2
Component Hardware Structure .....	4-2
Components That Include Logic Inside the System Module .....	4-3
Components That Interface to Logic Outside the System Module .....	4-4
List of Available Components in SOPC Builder .....	4-4
Tcl Components .....	4-5
Component Description File (_hw.tcl) .....	4-5
Component File Organization .....	4-5
Referenced Document .....	4-6
Document Revision History .....	4-6

## Chapter 5. Component Editor

Introduction .....	5-1
Component Hardware Structure .....	5-2
Starting the Component Editor .....	5-2
HDL Files Tab .....	5-2
Signals Tab .....	5-3
Naming Signals for Automatic Type and Interface Recognition .....	5-4
Templates for Interfaces to External Logic .....	5-5
Interfaces Tab .....	5-6
Component Wizard Tab .....	5-6
Identifying Information .....	5-6
Parameters .....	5-7
Saving a Component .....	5-7
Editing a Component .....	5-8
Referenced Documents .....	5-8
Document Revision History .....	5-9

## Chapter 6. Building a Component Interface with Tcl Scripting Commands

Organization of a Component Tcl File .....	6-2
Set and Add Commands .....	6-3
Module Properties .....	6-4
Clock Interface .....	6-4
Avalon-MM Master Interface .....	6-5
Avalon-MM Slave Interface .....	6-5
Avalon-ST Source Interface .....	6-6
Avalon-ST Sink Interface .....	6-7
Avalon-MM Tristate Interface .....	6-7
Nios II Custom Instruction Interface .....	6-8
Interrupt Interface .....	6-9
Conduit Interface .....	6-10
Document Revision History .....	6-10

## Chapter 7. Archiving SOPC Builder Projects

Introduction .....	7-1
Scope .....	7-1

Required Files .....	7-2
SOPC Builder Design Files .....	7-3
Nios II Application Software Project Files .....	7-3
Nios II System Library Project .....	7-4
File Write Permissions .....	7-4
Referenced Documents .....	7-4
Document Revision History .....	7-5

## Section II. Building Systems with SOPC Builder

### Chapter 8. Building Memory Subsystems Using SOPC Builder

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

Component-Level Design for SDRAM .....	8-15
SOPC Builder System-Level Design for SDRAM .....	8-15
Simulation for SDRAM .....	8-15
Quartus II Project-Level Design for SDRAM .....	8-15
Connecting and Assigning the SDRAM-Related Pins .....	8-16
Accommodating Clock Skew .....	8-16
Board-Level Design for SDRAM .....	8-16
Example Design with SDRAM .....	8-16
Off-Chip SRAM and Flash Memory .....	8-19
Component-Level Design for SRAM and Flash Memory .....	8-20
Avalon-MM Tristate Bridge .....	8-21
Flash Memory .....	8-21
SRAM .....	8-22
SOPC Builder System-Level Design for SRAM and Flash Memory .....	8-22
Simulation for SRAM and Flash Memory .....	8-23
Quartus II Project-Level Design for SRAM and Flash Memory .....	8-23
Board-Level Design for SRAM and Flash Memory .....	8-24
Aligning the Least-Significant Address Bits .....	8-24
Aligning the Most-Significant Address Bits .....	8-25
Example Design with SRAM and Flash Memory .....	8-25
Adding the Avalon-MM Tristate Bridge .....	8-26
Adding the Flash Memory Interface .....	8-26
Adding the SRAM Interface .....	8-26
Adding the PLL .....	8-29
SOPC Builder System Contents Tab .....	8-30
Connecting and Assigning Pins in the Quartus II Project .....	8-31
Connecting FPGA Pins to Devices on the Board .....	8-33
Referenced Documents .....	8-34
Document Revision History .....	8-34

## Chapter 9. Developing Components for SOPC Builder

Introduction .....	9-1
SOPC Builder Components and the Component Editor .....	9-1
Prerequisites .....	9-2
Hardware and Software Requirements .....	9-2
Component Development Flow .....	9-3
Typical Design Steps .....	9-3
Hardware Design .....	9-4
Software Design .....	9-6
Verifying the Component .....	9-8
Unit Verification .....	9-8
System-Level Verification .....	9-8
Design Example: Checksum Master .....	9-9
Install the Design Files .....	9-9
Review the Example Design Specifications .....	9-10
Checksum Design Files .....	9-11
Master Task Logic .....	9-11

Register File .....	9–12
Avalon-MM Clock Interface .....	9–12
Avalon-MM Master Interface .....	9–13
Avalon-MM Slave Interface .....	9–13
Software API .....	9–14
Create an SOPC Builder component .....	9–14
Open the Quartus II Project and Start the Component Editor .....	9–14
HDL Files Tab .....	9–15
Signals Tab .....	9–15
Interfaces Tab .....	9–18
Component Wizard Tab .....	9–23
Save the Component .....	9–23
Instantiate the Component in Hardware .....	9–24
Add the checksum Master Component to the SOPC Builder System .....	9–24
Compile the Hardware Design and Download to the Target Board .....	9–25
Exercise the Hardware Using Nios II Software .....	9–25
Start the Nios II IDE and Create a New IDE Project .....	9–26
Compile the Software Project and Run on the Target Board .....	9–28
Sharing Components .....	9–29
Referenced Documents .....	9–31
Document Revision History .....	9–31

## Section III. Interconnect Components

### Chapter 10. Avalon Memory-Mapped Bridges

Introduction to Bridges .....	10–1
Structure of a Bridge .....	10–1
Reasons for Using a Bridge .....	10–3
Address Mapping for Systems with Avalon-MM Bridges .....	10–7
Tools for Visualizing the Address Map .....	10–8
Differences between Avalon-MM Bridges and Avalon-MM Tristate Bridges .....	10–8
Avalon-MM Pipeline Bridge .....	10–9
Component Overview .....	10–9
Functional Description .....	10–10
The following sections describe the component’s hardware functionality. ....	10–11
Interfaces .....	10–11
Pipeline Stages and Effects on Latency .....	10–11
Burst Support .....	10–12
Example System with Avalon-MM Pipeline Bridges .....	10–12
Instantiating the Avalon-MM Pipeline Bridge in SOPC Builder .....	10–13
Device Support .....	10–14
Installation and Licensing .....	10–14
Hardware Simulation Considerations .....	10–15
Software Programming Model .....	10–15
Referenced Documents .....	10–15



Document Revision History .....	10–15
<b>Chapter 11. Avalon Streaming Interconnect Components</b>	
Introduction to Interconnect Components .....	11–1
Interconnect Component Usage .....	11–1
Address Mapping .....	11–3
Timing Adapter .....	11–3
Resource Usage and Performance .....	11–4
Instantiating the Timing Adapter in SOPC Builder .....	11–5
Input Interface Parameters .....	11–5
Output Interface Parameters .....	11–5
Common to Input and Output Interfaces .....	11–5
Channel Signal Width (Bits) .....	11–6
Max Channel .....	11–6
Bits Per Symbol .....	11–6
Symbols Per Beat .....	11–6
Include Packet Support .....	11–6
Error Signal Width (Bits) .....	11–6
Data Format Adapter .....	11–6
Resource Usage and Performance .....	11–7
Instantiating the Data Format Adapter in SOPC Builder .....	11–9
Input Interface Parameters .....	11–9
Data Symbols Per Beat .....	11–9
Output Interface Parameters .....	11–9
Data Symbols Per Beat .....	11–9
Common to Input and Output .....	11–9
Support Backpressure with the Ready Signal .....	11–9
Data Bits Per Symbol .....	11–9
Channel Signal Width (Bits) .....	11–9
Max Channel .....	11–9
Include Packet Support .....	11–10
Error Signal Width (Bits) .....	11–10
Channel Adapter .....	11–10
Resource Usage and Performance .....	11–10
Instantiating the Channel Adapter in SOPC Builder .....	11–11
Input Interface Parameters .....	11–11
Channel Signal Width (Bits) .....	11–11
Max Channel .....	11–11
Output Interface Parameters .....	11–11
Channel Signal Width (Bits) .....	11–11
Max Channel .....	11–11
Common to Input and Output Interfaces .....	11–11
Data Bits Per Symbol .....	11–11
Symbols Per Beat .....	11–12
Include Packet Support .....	11–12
Error Signal Width (Bits) .....	11–12
Device Support .....	11–12

Installation and Licensing .....	11-13
Hardware Simulation Considerations .....	11-13
Software Programming Model .....	11-13
Referenced Documents .....	11-13
Document Revision History .....	11-13



# 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: *October 2007*  
Part number: *QII54001-7.2.0*
  
- Chapter 2. System Interconnect Fabric for Memory-Mapped Interfaces  
Revised: *October 2007*  
Part number: *QII54003-7.2.0*
  
- Chapter 3. System Interconnect Fabric for Streaming Interfaces  
Revised: *October 2007*  
Part number: *QII54019-7.2.0*
  
- Chapter 4. SOPC Builder Components  
Revised: *October 2007*  
Part number: *QII54004-7.2.0*
  
- Chapter 5. Component Editor  
Revised: *October 2007*  
Part number: *QII54005-7.2.0*
  
- Chapter 6. Building a Component Interface with Tcl Scripting Commands  
Revised: *October 2007*  
Part number: *QII54022-7.2.0*
  
- Chapter 7. Archiving SOPC Builder Projects  
Revised: *October 2007*  
Part number: *QII54017-7.2.0*
  
- Chapter 8. Building Memory Subsystems Using SOPC Builder  
Revised: *October 2007*  
Part number: *QII54006-7.2.0*
  
- Chapter 9. Developing Components for SOPC Builder  
Revised: *October 2007*  
Part number: *QII54007-7.2.1*

Chapter 10. Avalon Memory-Mapped Bridges

Revised: *October 2007*

Part number: *QII54020-7.2.0*

Chapter 11. Avalon Streaming Interconnect Components

Revised: *October 2007*

Part number: *QII54021-7.2.0*



# 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	<a href="http://www.altera.com/mysupport/">www.altera.com/mysupport/</a>
Technical training	<a href="http://www.altera.com/training/custrain@altera.com">www.altera.com/training/custrain@altera.com</a>
Product literature	<a href="http://www.altera.com/literature">www.altera.com/literature</a>
Altera literature services	<a href="mailto:literature@altera.com">literature@altera.com</a>
FTP site	<a href="ftp.altera.com">ftp.altera.com</a>

## Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
<b>Bold Type with Initial Capital Letters</b>	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: <b>Save As</b> dialog box.
<b>bold type</b>	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: <b>f<sub>MAX</sub></b> , <b>lqdesigns</b> directory, <b>d:</b> drive, <b>chiptrip.gdf</b> 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<sub>PIA</sub></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	<p>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>resetn</code>.</p> <p>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.</p>
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, and describes the main features of the SOPC Builder tool. Chapters in this section serve to 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, Component Editor
- Chapter 6, Building a Component Interface with Tcl Scripting Commands
- Chapter 7, Archiving SOPC Builder Projects



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





## Overview

SOPC Builder is a powerful system development tool for creating systems including processors, peripherals, and memories. 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 in the Quartus® II software.

Many designers already know SOPC Builder as the tool for creating 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 into a larger system. Using traditional system-on-chip (SOC) design methods, you must manually write top-level HDL files that 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 design file that connects all the components together. SOPC Builder generates both Verilog HDL and VHDL equally, and does not favor one over the other. This chapter includes the following sections:

- [“Architecture of SOPC Builder Systems” on page 1-2](#)
- [“Functions of SOPC Builder” on page 1-5](#)
- [“Getting Started” on page 1-7](#)

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 introduces you to the architectural structure of systems built with SOPC Builder, and describes the primary functions of SOPC Builder.

## Architecture of SOPC Builder Systems

This section describes the fundamental architecture of an SOPC Builder system.

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 components together to create a top-level HDL file called the system module. SOPC Builder generates system interconnect fabric that contains logic to manage the connectivity of all components in the system.

### SOPC Builder Components

SOPC Builder components are the building blocks for creating an SOPC Builder system. SOPC Builder components use Avalon® interfaces for the physical connection of components, and you can use SOPC Builder to connect any logical device (either on-chip or off-chip) that has an Avalon interface. There are two different Avalon interfaces:

- The Avalon® Memory-Mapped (Avalon-MM) interface uses an address-mapped read/write protocol that enables flexible topologies for connecting master components to read and/or write any slave components.
- The Avalon Streaming (Avalon-ST) interface is a high-speed, unidirectional, system interconnect that enables point-to-point connections between streaming components that send and receive data using source and sink ports.

SOPC builder components can use either Avalon-MM or Avalon-ST interfaces or both.

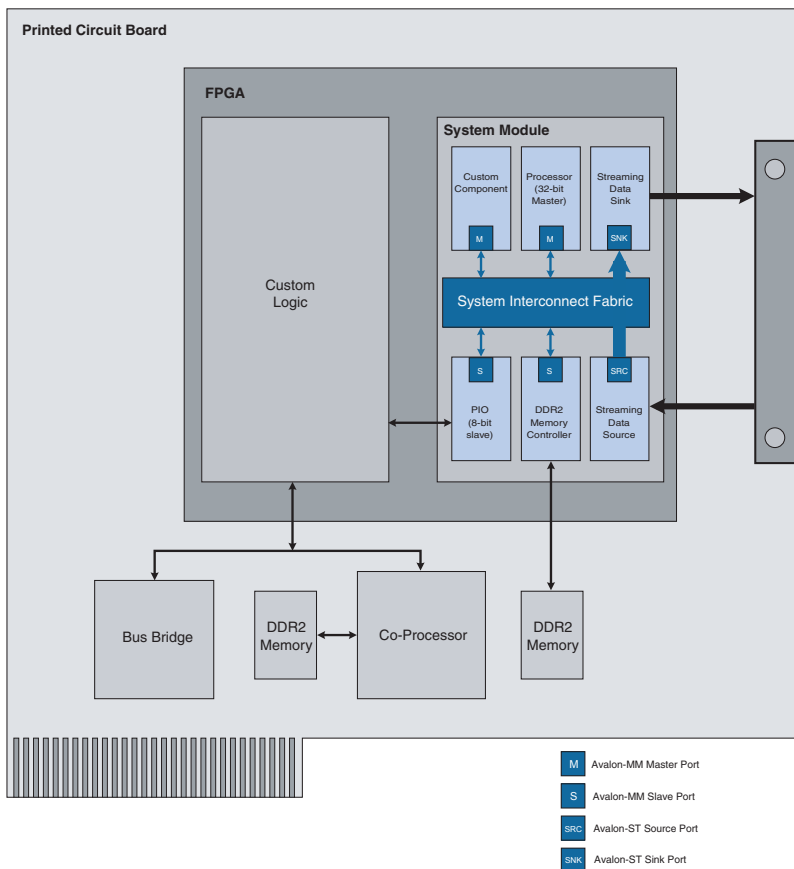


For details on the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification* chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface protocol, refer to *Avalon Streaming Interface Specification*. All are available at [www.altera.com](http://www.altera.com).

### Example System

Figure 1-1 shows an FPGA design including an SOPC Builder system module and custom logic modules. You can integrate custom logic inside or outside the system module. In this example, the custom component inside the system module is an SOPC Builder component that communicates with other modules through an Avalon-MM master interface. The custom logic outside of the system module is connected to the system module through a PIO interface. The system module 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 System Module Generated by SOPC Builder**



A component can be a logical device that is entirely contained within the system module, 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 system module. Non-Avalon signals can provide a special-purpose interface to the system module, such as the PIO in [Figure 1-1](#). A component can be instantiated more than once per design.

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
- Timers
- Serial communication interfaces, such as a UART and a serial peripheral interface (SPI)
- General purpose I/O
- Digital signal processing (DSP) functions
- Communications peripherals, such as a 10/100/1000 Ethernet MAC
- Interfaces to off-chip devices, such as:
  - Buses and bridges
  - Application-specific standard products (ASSP)
  - Application-specific integrated circuits (ASIC)
  - Processors

### *Custom Components*

SOPC Builder provides an easy method for you to develop and connect your own components. Your components can use either the Avalon-MM or Avalon-ST interfaces, or both. With the Avalon-MM interface, custom logic need only adhere to a simple interface based on address, data, read-enable, and write-enable signals. With the Avalon-ST interface, custom logic follows the configurable Avalon-ST interface protocol.

You use the following design flow to integrate custom logic into an SOPC Builder system:

1. Define the interface to the custom component.
2. Write HDL files describing the component in either Verilog HDL or VHDL.
3. Use the SOPC Builder component editor wizard to specify the interface and optionally package your HDL files into an SOPC Builder component.

4. Instantiate your component in the same manner as other SOPC Builder components.

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



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

### System Interconnect Fabric

The system interconnect fabric connects the components in SOPC Builder-generated systems. For Avalon-MM components, the system interconnect fabric is the collection of signals and logic that connects master and slave components, including address decoding, data-path multiplexing, wait-state generation, arbitration, interrupt controller, and data-width matching. For Avalon-ST components, the system interconnect fabric creates point-to-point connections between streaming components that send and receive data using source and sink ports.



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 fundamental functions of SOPC Builder.

### Defining and Generating the System Hardware

The purpose of SOPC Builder is to allow you to easily define the structure of a hardware system, and then generate the system. The GUI allows you to add components to a system, configure the components, and specify how they connect together.

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

- An HDL file for the top-level system module and for each component in the system
- A Block Symbol File (.bsf) representation of the top-level system module for use in Quartus II Block Diagram Files (.bdf)
- Software files for embedded software development, such as a memory-map header file and component driver
- (Optional) Testbench for the system module and ModelSim® simulation project files

After you generate the system module, you can compile it with the Quartus II software, or you can instantiate it in a larger FPGA design.

### **Creating a Memory Map for Software Development**

When connected to the Nios II processor, SOPC Builder generates a header file that defines 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.

How you write software for the system depends heavily on the nature of the processor in the system. For example, Nios II processor systems use Nios II processor-specific software development tools. These tools are separate from SOPC Builder, but they do use the output of SOPC Builder as the foundation for software development.

### **Creating a Simulation Model and Test Bench**

You can simulate your custom systems with minimal effort immediately after generating the system with SOPC Builder. During system generation, SOPC Builder optionally outputs a push-button simulation environment that eases the system simulation effort. SOPC Builder generates both a simulation model and a testbench for the entire system. The testbench includes the following functionality:

- Instantiates the system module
- Drives all clocks and resets appropriately
- Optionally instantiates simulation models for off-chip devices

## Getting Started

One of the easiest ways to get started using SOPC Builder is to read the *Nios II Hardware Development Tutorial* which guides you step by step in building a microprocessor system, including CPU, memory, and peripherals. This tutorial and other SOPC Builder example designs are included in the Nios II Embedded Design Suite (EDS). You can download this design suite for free from the Altera Download Center at [www.altera.com/download](http://www.altera.com/download).

## Referenced Documents

This chapter references the following documents:

- *Avalon Memory-Mapped Interface Specification*
- *System Interconnect Fabric for Streaming Interfaces*
- *Avalon Streaming Interface Specification*
- *SOPC Builder Components*
- *Component Editor*
- *System Interconnect Fabric for Memory-Mapped Interfaces*
- *Nios II Hardware Development Tutorial*

## Document Revision History

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

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
October 2007, v7.2.0	<ul style="list-style-type: none"> <li>Updated with new 7.2 functionality and terminology. Deleted unneeded description of SOPC Builder Ready Components.</li> </ul>	—
May 2007, v7.1.0	<ul style="list-style-type: none"> <li>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.”</li> <li>Added new information on Avalon Streaming (Avalon-ST) interface.</li> <li>Revised system module block diagram</li> <li>Added Referenced Documents section.</li> </ul>	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

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. 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-sliced shared medium. This chapter describes the functions of system interconnect fabric for memory-mapped interfaces and the implementation of those functions.

### High-Level Description

System interconnect fabric is the collection of interconnect and logic resources that connects Avalon-MM master and slave ports on components in a system. SOPC Builder generates system interconnect fabric to match the needs of the specific components in a system. System interconnect fabric encapsulates the connection details of a system. It guarantees that signals travel correctly between master and slave ports, as long as the ports adhere to the rules of the Avalon Memory-Mapped interface specification. 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 2-8
- “Native Address Alignment and Dynamic Bus Sizing” on page 2-9
- “Arbitration for Multimaster Systems” on page 2-12
- “Burst Management” on page 2-18
- “Clock Domain Crossing” on page 2-19
- “Interrupts” on page 2-29
- “Reset Distribution” on page 2-31



For details about the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification*

System interconnect fabric for memory-mapped interfaces supports:

- 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

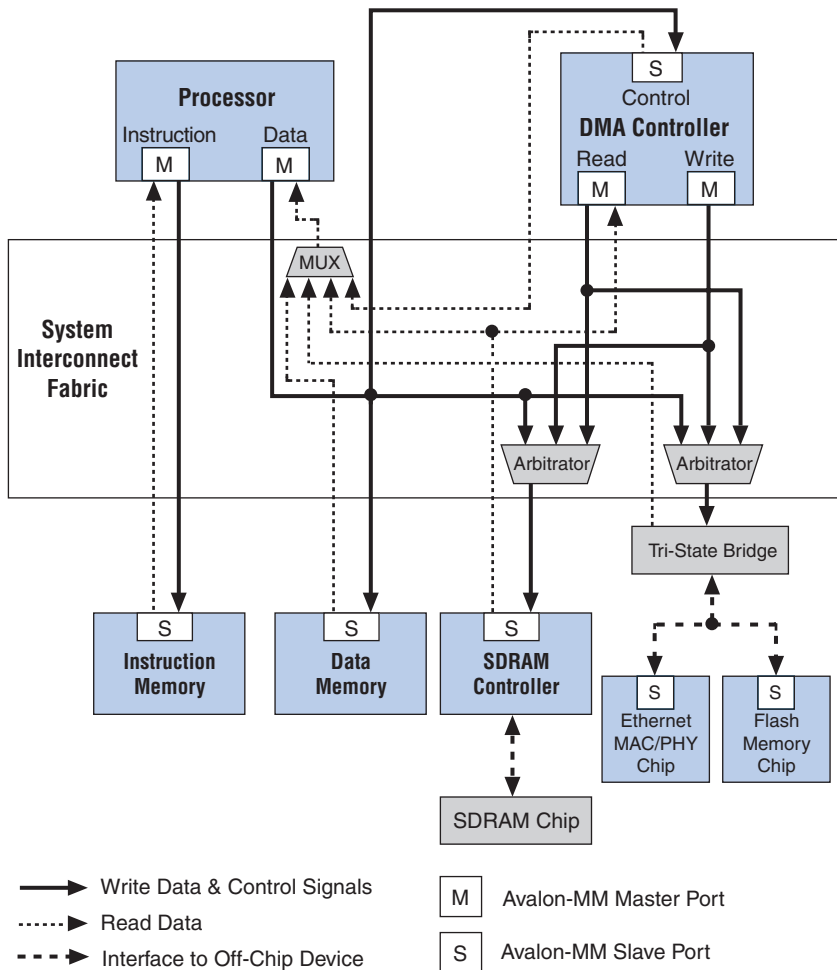
- Interfaces to off-chip devices
- Master and slave ports of differing data widths
- Big-endian or little-endian components
- Components operating in different clock domains
- Components using multiple Avalon-MM ports

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 ports, such as the processor component shown in Figure 2–1. Because SOPC Builder can create system interconnect fabric to connect components with multiple ports, you can create complex interfaces that provide more functionality than a single Avalon-MM port. 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 topology of component connections, as long as each port conforms to the Avalon interface specification. It can, for example, connect a system comprised of only two components with unidirectional dataflow between them. Avalon-MM interfaces are suitable for random addressable transactions, such as to memories or embedded peripherals. Avalon-ST interfaces are suitable for dataflow interconnection, as found in packet processing or DSP pipelines.



For more information, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook* and the *Avalon Streaming Interface Specification*.

Generating system interconnect fabric is SOPC Builder's primary purpose. SOPC Builder can be used to manage and edit your design. Because SOPC Builder automatically generates system interconnect fabric, you may not be required to interact directly with it or the HDL that describes it; however, a basic understanding of how it works can help you optimize your components and systems. For example, knowledge of the arbitration mechanism 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 slave ports. System interconnect fabric consists of synchronous logic and routing resources inside the FPGA.

For each port interface on components, system interconnect fabric manages Avalon-MM transfers, interacting with and responding to signals on the connected component. The signals that appear on the master port and corresponding slave port of a master-slave pair can be different. In the path between master and slave ports, 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 ports.

## 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 2-8](#)
- [“Native Address Alignment and Dynamic Bus Sizing” on page 2-9](#)
- [“Arbitration for Multimaster Systems” on page 2-12](#)

- “Burst Management” on page 2-18
- “Clock Domain Crossing” on page 2-19
- “Interrupts” on page 2-29
- “Reset Distribution” on page 2-31

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

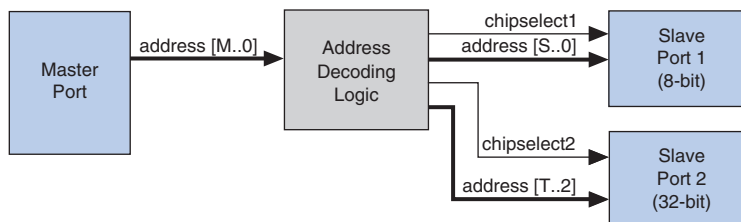
Address decoding logic in the system interconnect fabric distributes an appropriate address and produces a `chipselect` signal for each slave port. Address decoding logic simplifies component design in the following ways:

- The system interconnect fabric selects a slave port whenever it is being addressed by a master. Slave components do not need to decode the address to determine when they are selected.
- Slave port addresses are properly aligned for the slave port.
- SOPC Builder automatically generates address decoding logic to implement the memory map specified in the GUI. Therefore, 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 slave ports. Separate address-decoding logic is generated for every master port 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 port’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**

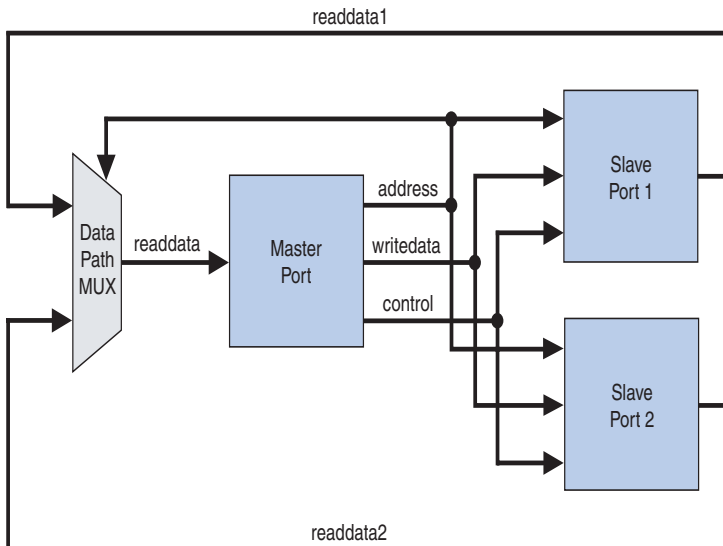
Module Name	Description	Base	End	IRQ
cpu	Nios II Proces...			
instruction_master	Master port			
data_master	Master port			IRQ 0
jtag_debug_mod...	Slave port	0x02120000	0x021207FF	IRQ 31
ext_flash	Flash Memory...	0x00000000	0x007FFFFFFF	
ext_ram	IDT71V416 S...	0x02000000	0x020FFFFFFF	
ext_ram_bus	Avalon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

## Datapath Multiplexing

Datapath multiplexing logic in the system interconnect fabric drives the writedata from the granted master to the selected slave, and from the readdata 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 slave ports. SOPC Builder generates separate datapath multiplexing logic for every master port 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** page, as shown in [Figure 2-5](#).

**Figure 2-5. Connection Panel Settings in SOPC Builder Control Datapath Multiplexing**

Connection Panel Settings

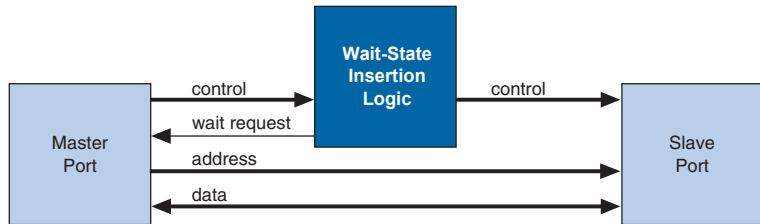
Use	Module Name	Description	Input Clock	Base
<input checked="" type="checkbox"/>	El epni	Nios II Processor - Altera Corporation	clk_05	
<input checked="" type="checkbox"/>	El instruction_mes	Master port		0x0
<input checked="" type="checkbox"/>	El mfu_master	Master port		0x02220000
<input checked="" type="checkbox"/>	El iop_slave_m0	Slave port		
<input checked="" type="checkbox"/>	El ioh_sram_bridge	Avilion Tristate Bridge	clk_05	
<input checked="" type="checkbox"/>	El ioh_sram_bridge	Cypress CY7C1303C SDRAM		0x02000000
<input checked="" type="checkbox"/>	El ioh_sram_bridge	PCI (Parallel IO)	clk_05	0x02220000
<input checked="" type="checkbox"/>	El ioh_high_mes_timer	Interval timer	clk_05	0x02220020
<input checked="" type="checkbox"/>	El ioh_irq_uart	JTAG UART	clk_05	0x02220000
<input checked="" type="checkbox"/>	El ioh_irq1111	LAN91C111 Interface (Ethernet)	clk_05	0x02210000

## 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 port, and coordinates the master port to wait until the slave can proceed. System interconnect fabric inserts wait states into a transfer when the target slave port 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 2-6](#) shows a block diagram of the wait-state insertion logic between one master and one slave.

**Figure 2-6. Block Diagram of Wait-State Insertion Logic**



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

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

## Pipeline Read Transfers

The Avalon-MM interface supports pipelined read transfers, allowing a pipelined master port 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 port 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 port, SOPC Builder guarantees that read data arrives at each master port in the order requested. Because master and slave ports often have mismatched pipeline latency, system interconnect fabric often contains logic to reconcile the differences. Many cases are possible, as shown in [Table 2-1](#).

**Table 2-1. Various Cases of Pipeline Latency in a Master-Slave Pair**

Master Port	Slave Port	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 port to wait through any slave-side latency cycles. This master-slave pair gains no benefits of pipelining, because the master port is not pipelined and therefore waits for each transfer to complete before beginning a new transfer. However, while the master port is waiting, the slave port can accept transfers from a different master port.
Pipelined	No Pipeline	The system interconnect fabric carries out the transfer as if neither port were pipelined, forcing the master port to wait until the slave port returns data.
Pipelined	Pipelined with Fixed Latency	The system interconnect fabric coordinates the master port to capture data at the exact clock cycle when data is valid on the slave port. This case enables this master-slave pair to achieve maximum throughput performance.
Pipelined	Pipelined with Variable Latency	This is the simplest pipelined case, in which the slave port asserts a signal when its readdata is valid, and the master port captures the data. This case enables this master-slave pair to achieve maximum throughput performance.

SOPC Builder generates logic to handle pipeline latency based on the properties of the master and slave ports 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 slave ports with unmatched data widths. Address alignment affects how slave data is aligned in a master port's address space, in the case that the master and slave data widths are different. Address alignment is a property of each slave port, and can be different for each slave port in a system. A slave port 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 port connected to a 16-bit slave port (a 2:1 ratio). In this example, the slave port is mapped to base address *BASE* in the master port's address space. In Table 2–2, *OFFSET* refers to the offset into the 16-bit slave address space.

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 slave ports 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 port, and vice versa. When an *N*-bit master port accesses a slave port with dynamic bus sizing, the master port 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 with 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 port 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 port 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 port accessing a 64-bit slave port with dynamic bus sizing. In the table, offset refers to the offset into the slave port memory space.

<b>Table 2-3. 32-Bit Master View of 64-Bit Slave with Dynamic Bus Sizing (Part 1 of 2)</b>	
<b>32-bit Address</b>	<b>Data</b>
0x00000000 (word 0)	OFFSET [0] <sub>31..0</sub>
0x00000004 (word 1)	OFFSET [0] <sub>63..32</sub>

**Table 2–3. 32-Bit Master View of 64-Bit Slave with Dynamic Bus Sizing (Part 2 of 2)**

32-bit Address	Data
0x00000008 (word 2)	OFFSET [1] <sub>31..0</sub>
0x0000000C (word 3)	OFFSET [1] <sub>63..32</sub>

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 port. 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.

## Native Address Alignment

Slave ports that access address-mapped registers inside the component generally use native address alignment. The defining properties of native address alignment are:

- Each slave offset (that is, word) maps to exactly one master word, regardless of the data width of the ports.
- One transfer on the master port generates exactly one transfer on the slave port.

In the case of native address alignment, system interconnect fabric maps all slave data bits to the lower bits of the master data, and fills any remaining upper bits with zero. System interconnect fabric performs simple wire-mapping in the datapath, but nothing else.

Native address alignment is only valid if the master data width is equal to or wider than the slave data width. If an  $N$ -bit master port is connected to a wider slave with native alignment, then the master port can access only the lower  $N$  data bits at each offset in the slave.



Native address alignment prevents use of the slave with narrow masters and some bridge implementations, and is not recommended for new components.

## Arbitration for Multimaster Systems

System interconnect fabric supports systems with multiple master components. In a system with multiple master ports, such as the system pictured in [Figure 2–1 on page 2–3](#), the system interconnect fabric provides shared access to slave ports using a technique called slave-side arbitration. Slave-side arbitration determines which master port gains access to a specific slave port in the event that multiple master ports attempt to access the same slave port 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 master ports to transfer data simultaneously. Unlike traditional host-side arbitration architectures in which 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 port only when multiple master ports attempt to access the same slave port during the same cycle.
- Eliminates unnecessary master-slave connections. The connection between a master port and a slave port exists only if it is specified in SOPC Builder. If a master port never initiates transfers to a specific slave port, 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 port is specified independently. For example, you can grant one master port the most access to a particular slave port, while other master ports have more access to other slave ports.
- Simplifies master component design. The details of arbitration are encapsulated inside the system interconnect fabric. Each Avalon-MM master port connects to the system interconnect fabric as if it is the only master port 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

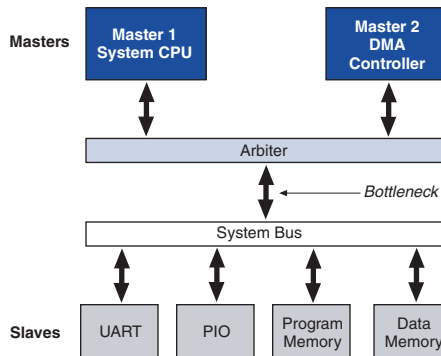
As a frame of reference for the discussion of multiple masters and arbitration, this section describes traditional bus architectures.

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. 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 a bus slave. If multiple masters attempt to access the bus at the same time, the arbiter allocates the bus resources to a single master based on fixed arbitration rules, forcing all other masters to wait. For example, the priority arbitration scheme—in which the arbiter always grants control to the master with the highest priority—is used in many existing bus architectures.

Figure 2–7 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 2–7. Bus Architecture in a Traditional Microprocessor System**



## Slave-Side Arbitration

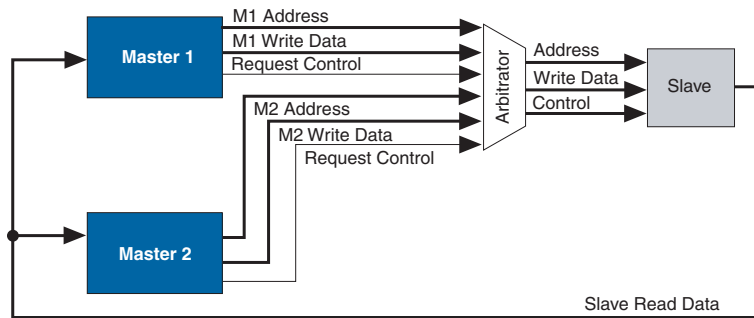
The multimaster architecture used by system interconnect fabric eliminates the bottleneck for access to a shared bus, because the system does not have shared bus signals. Avalon-MM master-slave pairs are connected by dedicated paths. A master port never waits to access a slave port, unless a different master port attempts to access the same slave port at the same time. As a result, multiple master ports can be active at the same time, simultaneously transferring data with independent slave ports.

A multimaster Avalon-MM system requires arbitration, but only when two masters contend for the same slave port. This arbitration is called slave-side arbitration, because it is implemented at the point where two (or more) master ports connect to a single slave. Master ports contend for individual slave ports, not for a shared bus resource.

For example, [Figure 2-1 on page 2-3](#) demonstrates a system with two master ports (a CPU and a DMA controller) sharing a slave port (an SDRAM controller). Arbitration is performed at the SDRAM slave port; the arbiter dictates which master port gains access to the slave port if both master ports initiate a transfer with the slave port in the same cycle.

[Figure 2-8](#) focuses on the two master ports and the shared slave port, and shows additional detail of the data, address, and control paths. The arbiter logic multiplexes all address, data, and control signals from a master port to a shared slave port.

**Figure 2-8. Detailed View of Multimaster Connections**



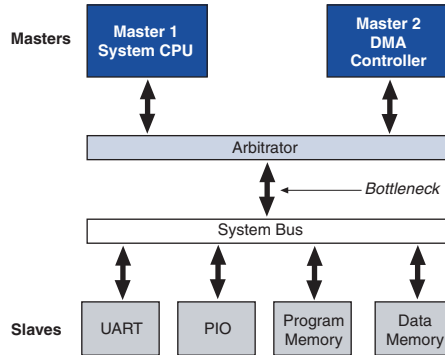
## Arbiter Details

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

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

Figure 2–9 shows the arbiter logic in an example multimaster system with two master ports, each connected to two slave ports.

Figure 2–9. Block Diagram of Arbiter Logic



### Arbitration Rules

This section describes the rules by which the arbiter grants access to master ports 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 2–10.

Figure 2–10. Arbitration Settings on the System Contents Tab

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

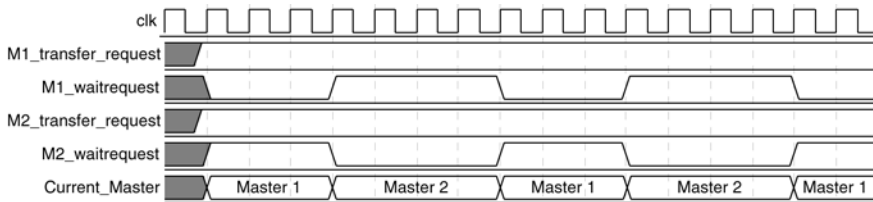
The arbitration settings are hidden by default. To see them, on the View menu, click **Show Arbitration**.

### *Fairness-Based Shares*

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

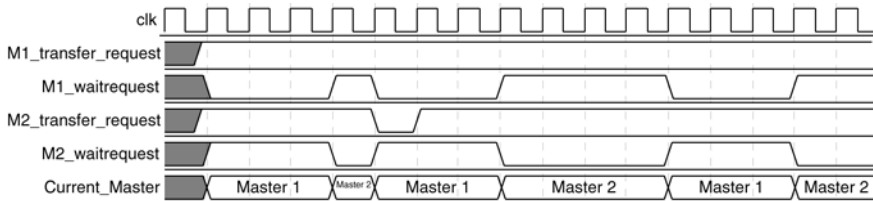
For example, assume that two master ports continuously attempt to perform back-to-back transfers to a slave port. Master 1 is assigned three shares and Master 2 is assigned four shares. In this case, the arbiter grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely. [Figure 2-11](#) demonstrates this case, showing each master port's transfer request output, wait request input (which is driven by the arbiter logic), and the current master with control of the slave.

**Figure 2-11. Arbitration of Continuous Transfer Requests from Two Master Ports**



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 2-12](#). 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 2-12. Arbitration of Two Masters with a Gap in Transfer Requests**





### *Round-Robin Scheduling*

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

### *Burst Transfers*

Avalon-MM burst transfers grant a master port uninterrupted access to a slave port for a specified number of transfers. The master port 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 port to access the slave port until the burst completes. For further information, refer to [“Burst Management” on page 2–18](#).

### *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 port has a minimum share value of ten, then the arbiter will grant at least ten shares to any master port when it begins a sequence of transfer requests. The arbiter might grant more shares, if the master port is assigned more shares in SOPC Builder.

By declaring a minimum share value of  $N$ , a slave port declares that it is more efficient at handling continuous sequential transfers of length  $N$ . Accessing the slave port in sequences less than  $N$  incurs performance penalties that might prevent the slave port 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 port 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 slave ports that support bursts; the burst length takes precedence over minimum share value. Refer to [“Burst Management” on page 2–18](#) for information.

You specify the arbitration shares for each master using the connection panel on the **System Contents** tab of SOPC Builder, as shown in [Figure 2–13](#).

**Figure 2–13. Arbitration Settings on the System Contents Tab**

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



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

## Burst Management

System interconnect fabric provides burst management logic to accommodate the burst capabilities of each port in the system, including ports that do not support burst transfers. Burst management logic is 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 port might be capable of initiating a burst longer than a slave port'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 port does not support bursts. Until the master port completes the burst, the arbiter logic prevents other master ports from accessing the target slave port.

For example, if a master port initiates a burst of 16 transfers to a slave port with maximum burst length of 8, the burst management logic initiates two bursts of length 8 to the slave port. If a master port initiates a burst of 16 transfers to a slave port that does not support bursts, the burst management logic initiates 16 separate transfers to the slave port.

## Clock Domain Crossing

SOPC Builder generates clock-domain crossing (CDC) logic that hides the details of interfacing components operating in asynchronous clock domains. The system interconnect fabric upholds the Avalon-MM protocol with each port independently, and therefore each Avalon-MM port need only be aware of its own clock domain. The system interconnect fabric logic propagates transfers across clock domain boundaries automatically.

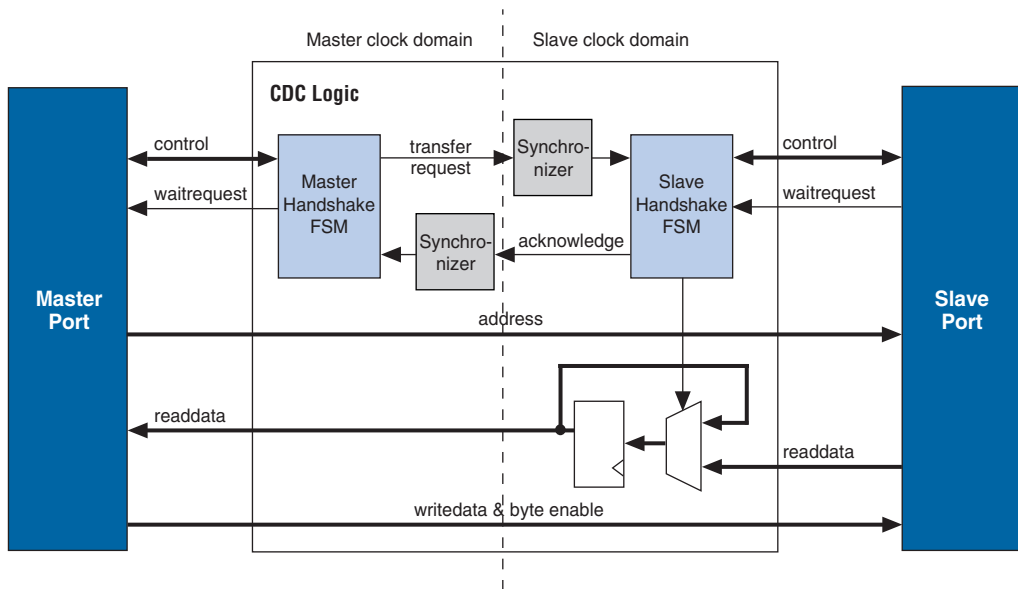
The CDC logic in system interconnect fabric provides the following benefits that simplify system design efforts:

- Allows component interfaces to operate at a different clock frequency than system logic.
- Eliminates the need to design CDC hardware manually.
- Each Avalon-MM port operates in only one clock domain, which reduces design complexity of components.
- Enables master ports to access any slave port without communication with the slave clock domain.
- Allows you to focus performance optimization efforts only on components that require fast clock speed.

### Description of Clock Domain-Crossing Logic

The CDC logic 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 2-14](#) shows a block diagram of the clock domain crossing logic between one master and one slave port.

**Figure 2–14. Block Diagram of Clock Domain-Crossing Logic**



The Synchronizer blocks in [Figure 2–14](#) 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 port asserts address, data, and control signals.
2. The master handshake FSM captures the control signals, and immediately forces the master port to wait.



The FSM uses only the control signals, not address and data. For example, the master port 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 port.
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 port 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 port, there is nothing different about a transfer initiated by a master port in a different clock domain. From the perspective of a master port, 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 port to wait until the transfer terminates. As a result, latency-aware master ports do not benefit from pipelining when performing transfers to a different clock domain.

### Location of Clock Domain Crossing Logic

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 on each slave port 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 higher performance clock crossing logic 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*.

## 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, shown in [Figure 2–15](#). Clock sources can be driven by external input signals to the system module, or by PLLs inside the system module. Clock domains are differentiated based on the name of the clock. It is possible to create multiple asynchronous clocks with the same frequency.

**Figure 2–15. Clock Settings on the System Contents Tab**

Clock	Source	MHz	Pipeline
clk_85	External	85.0	<input type="checkbox"/>
clk_233	c0 from pll	233.75	<input checked="" type="checkbox"/>
click to add...			<input type="checkbox"/>

You specify which clock drives which components using the table of active components after you define the system clocks, as shown in [Figure 2-16](#).

**Figure 2-16. Assigning Clocks to Components**

Module Name	Description	Clock	Base	End	IRQ
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 patch panel can be used.

This section describes the hardware structure and functionality of the Avalon-MM clock-crossing bridge component.

## Component Overview

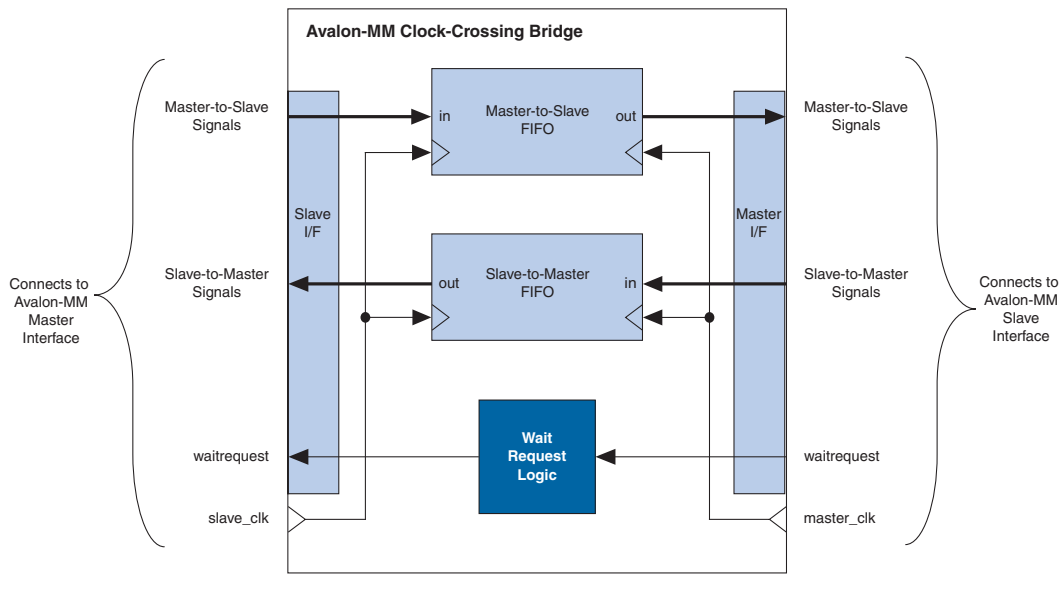
The Avalon-MM clock-crossing bridge allows you to connect Avalon-MM master and slave ports that operate in different clock domains. Without a bridge, SOPC Builder automatically includes generic CDC logic in the system interconnect fabric, but it does not provide optimal performance for high-throughput applications. The CDC logic uses a four-way handshake mechanism so that each read and write takes multiple cycles in each direction. Because the clock-crossing bridge includes a buffering mechanism, you can pipeline multiple reads and writes. After an 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 additional logic resources. The clock-crossing bridge has parameterizable FIFOs for master-to-slave and slave-to-master signals, which 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.

## Functional Description

[Figure 2-17](#) shows a block diagram of the Avalon-MM clock-crossing bridge component. The following sections describe the component's hardware functionality.

**Figure 2-17. Avalon-MM Clock-Crossing Bridge Block Diagram**



### Interfaces

The bridge interface comprises an Avalon-MM slave port and an Avalon-MM master port. 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.

### Clock Domain Crossing Logic and FIFOs

Two FIFOs in the bridge transport address, data, and control signals across the clock-domains. One FIFO captures data 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`
- `nativeaddress`
- `byteenable`
- `burstcount`, when bursts are allowed.

The signals that pass through the slave-to-master FIFO include:

- `readdata`
- `readdatavalid`
- `endofpacket`

The depth of each FIFO is configurable. 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 which 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 can optionally 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 deconstructs master-to-bridge bursts into a sequence of individual transfers.

When the bridge is configured to support bursts, the slave-to-master FIFO depth must be configured deeply enough to capture all burst read data without overflowing. The master ports 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:

(1)      *No Bursts:*  
*minimum depth = master-to-slave FIFO depth + max slave latency;*

*With Bursts:*  
*minimum depth*  
*= (master-to-slave FIFO depth + max slave latency) \* (max burst size);*

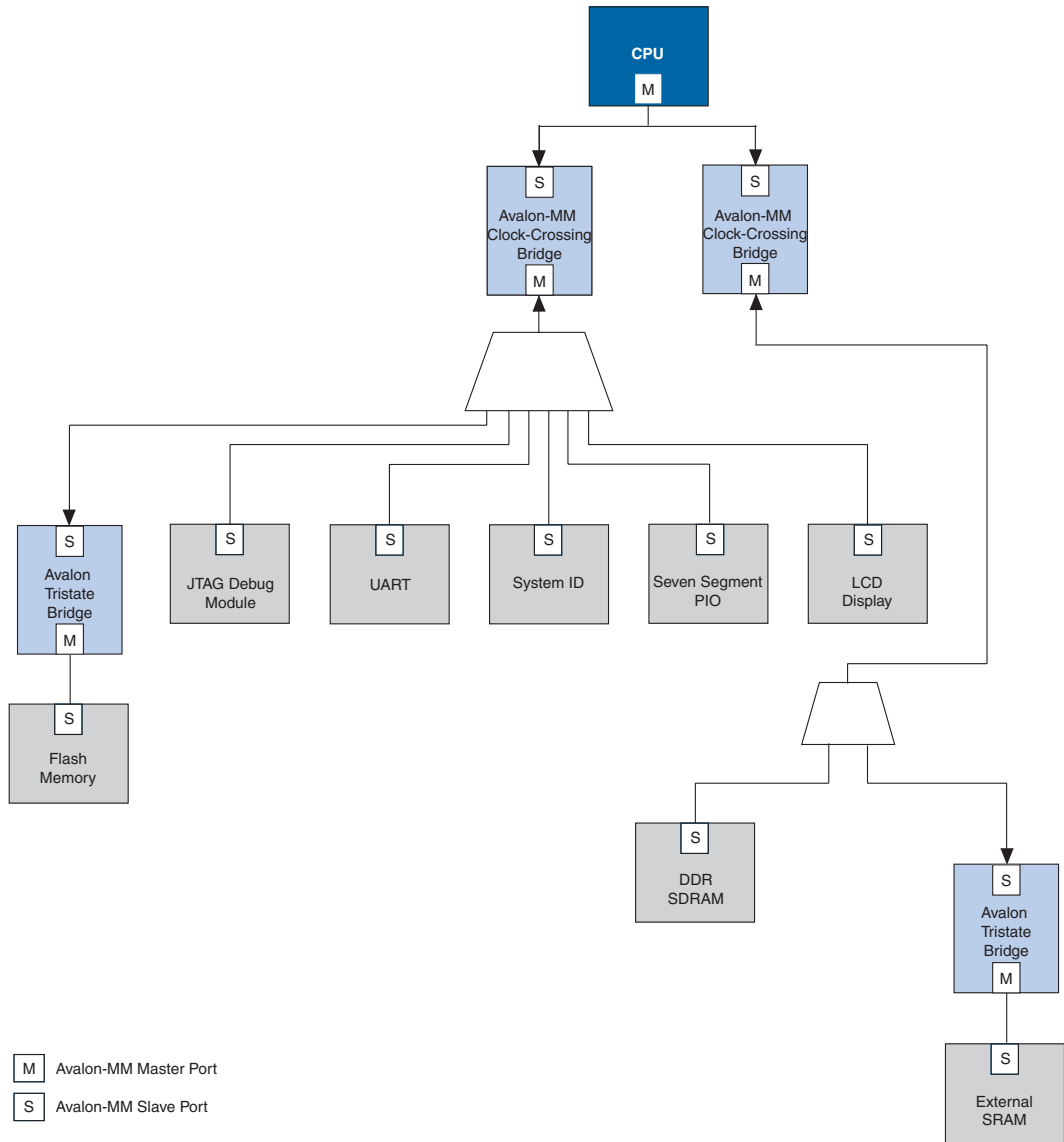


In both cases, the minimum depth is rounded up to the nearest power of two.

### *Example System with Avalon-MM Clock-Crossing Bridges*

Figure 2-18 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 2-18. One Avalon-MM Master with Two Groups of Avalon-MM Slaves



## Instantiating the Avalon-MM Clock-Crossing Bridge in SOPC Builder

You use the Avalon-MM Clock-Crossing Bridge MegaWizard® interface in SOPC Builder to specify the hardware features. This section describes the options available on the **Parameter Settings** page of the Megawizard interface.

- **Master-to-Slave FIFO**—These options specify the size and structure of the master-to-slave FIFO.
  - **FIFO Depth**—Determines the depth of the FIFO.
  - **Construct FIFO from registers**—When this option is on, the FIFO uses registers as storage instead of embedded memory blocks. Turning on this option can considerably increase the size of the bridge hardware and lower the  $f_{MAX}$ .
- **Slave-to-Master FIFO**—These options specify the size and structure of the slave-to-master FIFO.
  - **FIFO Depth**—Determines the depth of the FIFO.
  - **Construct FIFO from registers**—When this option is on, the FIFO uses registers as storage instead of embedded memory blocks. Turning on this option can considerably increase the size of the bridge hardware.
- **Data Width**—Determines the data width of the master and slave ports 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 port connected to the bridge.

- **Allow Bursts**—Includes logic for the bridge's master and slave ports to support bursts. This option restricts the minimum depth for the slave-to-master FIFO.
- **Maximum Burst Size**—Determines the maximum length of bursts for the bridge to support, when **Allow Bursts** is turned on.

# Interrupts

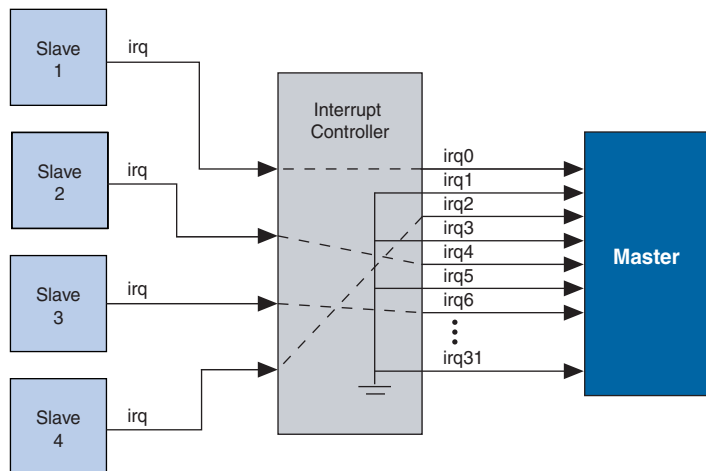
In systems with slave ports that generate interrupt requests (IRQs), the system interconnect fabric includes interrupt controller logic. A separate interrupt controller is generated for each master port that accepts interrupts. The interrupt controller aggregates IRQ signals from all slave ports, and maps slave IRQ outputs to user-specified values on the master IRQ inputs.

## Software Priority

In the software priority configuration, the system interconnect fabric passes IRQs directly from slave to master port, without making any assumptions about IRQ priority. In the event that multiple slave ports assert their IRQs simultaneously, the master logic (presumably under software control) determines which IRQ has highest priority, then responds appropriately.

Using software priority, the interrupt controller can handle up to 32 slave IRQ inputs. The interrupt controller generates a 32-bit signal `irq[31..0]` to the master port, and simply maps slave IRQ signals to the bits of `irq[31..0]`. Any unassigned bits of `irq[31..0]` are permanently disabled. [Figure 2-19](#) shows an example of the interrupt controller mapping the IRQs on four slave ports to `irq[31..0]` on a master port.

**Figure 2-19. IRQ Mapping Using Software Priority**

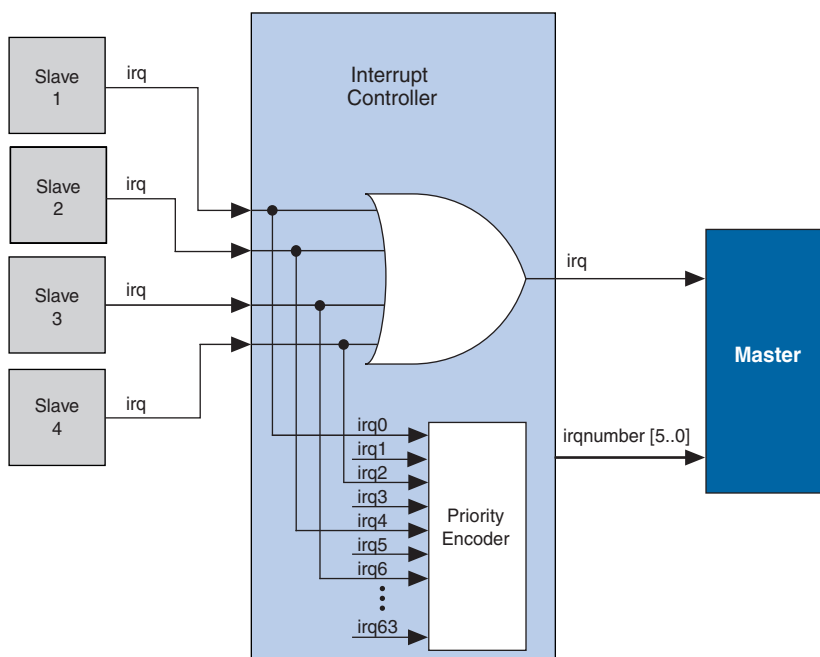


## Hardware Priority

In the hardware priority configuration, in the event that multiple slaves assert their IRQs simultaneously, the system interconnect fabric (that is, hardware logic) identifies the IRQ of highest priority and passes only that IRQ number to the master port. An IRQ of lesser priority is undetectable until a master port clears all IRQs of higher priority.

Using hardware priority, the interrupt controller can handle up to 64 slave IRQ signals. The interrupt controller generates a 1-bit `irq` signal to the master port, signifying that one or more slave ports 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 2–20](#).

**Figure 2–20. 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 slave ports that can generate IRQs, with respect to each master port. For

each slave port, you can either specify an IRQ number, or specify not to connect the IRQ. Figure 2–21 shows the IRQ settings for multiple slave IRQs that drive the master component named `cpu`.

Figure 2–21. Assigning IRQs in SOPC Builder

Module Name	Description	Clock	Base	End	IRQ
<code>cpu</code>	Nios II Processor - Alter...	clk	0x02120000	0x021207FF	
<code>ext_ram_bus</code>	Avalon Tri-State Bridge	clk			
<code>ext_flash</code>	Flash Memory (Common ...		0x00000000	0x007FFFFFFF	
<code>ext_ram</code>	IDT71V416 SRAM		0x02000000	0x020FFFFFFF	
<code>epcs_controller</code>	EPCS Serial Flash Contr...	clk	0x02100000	0x021007FF	HC
<code>lan91c111</code>	LAN91c111 Interface (E...		0x02110000	0x0211FFFF	6
<code>sys_clk_timer</code>	Interval timer	clk	0x02120800	0x0212081F	1
<code>jtag_uart</code>	JTAG UART	clk	0x021208B0	0x021208B7	4
<code>button_pio</code>	PIO (Parallel I/O)	clk	0x02120860	0x0212086F	2
<code>led_pio</code>	PIO (Parallel I/O)	clk	0x02120870	0x0212087F	1
<code>high_res_timer</code>	Interval timer	clk	0x02120820	0x0212083F	3
<code>lcd_display</code>	Character LCD (16x2, O...	clk	0x02120880	0x0212088F	
<code>gpio_pio_pio</code>	PIO (Parallel I/O)	clk	0x02120890	0x0212089F	

## Reset Distribution

The system interconnect fabric generates and distributes a system-wide reset pulse to all logic in the system module. 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 system module is asserted.
- Any slave port asserts its `resetrequest` signal.

All components must enter a well-defined reset state whenever the system interconnect fabric asserts the system-wide reset. The timing of the reset signal is asynchronous to the operation of transfers. Resets are asserted asynchronously and deasserted synchronously to the associated clock.

## Referenced Documents

This chapter references the following documents:

- [Avalon Memory-Mapped Interface Specification](#)
- [System Interconnect Fabric for Streaming Interfaces](#)
- [Avalon Streaming Interface Specification](#)
- [Avalon Memory-Mapped Bridges](#)

## Document Revision History

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

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
October 2007 v7.2.0	<ul style="list-style-type: none"> <li>Updated to match 7.2 features. Deleted paragraphs discussing “Pipelining for High Performance”, “Endian Conversion”, and added new screenshots.</li> <li>Moved clock-crossing bridge discussion to this chapter from chapter 10.</li> </ul>	—
May 2007, v7.1.0	<ul style="list-style-type: none"> <li>Chapter 3 was previously titled Avalon Switch Fabric.</li> <li>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.”</li> <li>Rearranged content in section “Introduction” on page 2–1 to enhance clarity and to acknowledge the existence of the new Avalon Streaming interface.</li> <li>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.</li> <li>Updated Table 2–2 on page 2–9 for improved clarity.</li> <li>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 <math>N:1</math>, the system interconnect fabric might not need to perform <math>N</math> slave-side read transfers, depending on how the master port asserts its byte-enable signals.</li> <li>Added three paragraphs explaining when clock signals are automatically connected to SOPC Builder components.</li> <li>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.</li> </ul>	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 Specification 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.	—
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.	—



**Table 2–4. Document Revision History**

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
May 2005, v5.0.0	<ul style="list-style-type: none"><li>• Added burst transfer management details.</li><li>• Updated pipeline management details.</li></ul>	—
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. It 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 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**

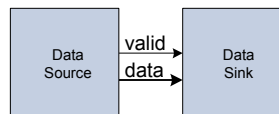
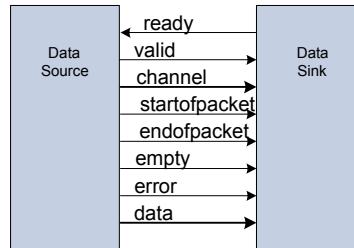


Figure 3-2 illustrates a more extensive interface that includes signals indicating the start and end of packets, channel numbers, error conditions, and back pressure.

**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, 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 Streaming Interface Specification* available at [www.altera.com](http://www.altera.com).

## Avalon Streaming and Avalon Memory-Mapped Interfaces

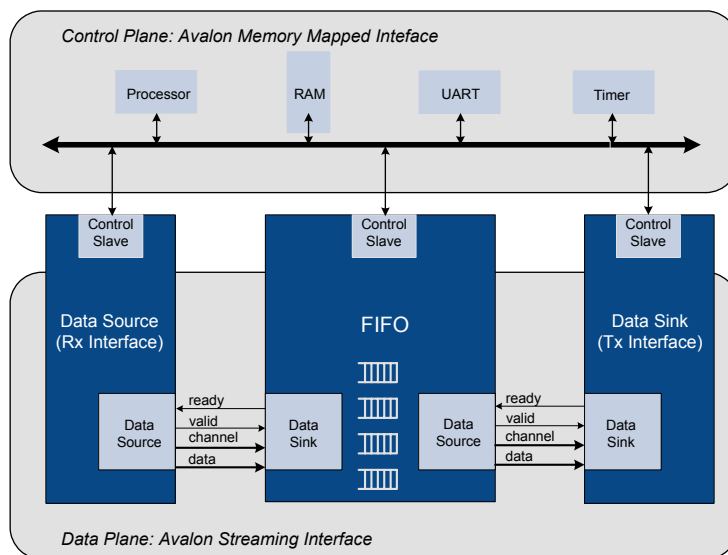
The Avalon-ST and Avalon Memory-Mapped (Avalon-MM) interfaces are complimentary. High bandwidth components with streaming data typically use Avalon-ST interfaces for the high throughput datapath. These components can also use Avalon Memory-Mapped interfaces to provide an access point for control. In 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 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

The Insert Avalon-ST Adapters command on the System menu allows you to insert an adapter so that you can connect a data source to a data sink of differing byte sizes in the SOPC Builder system.



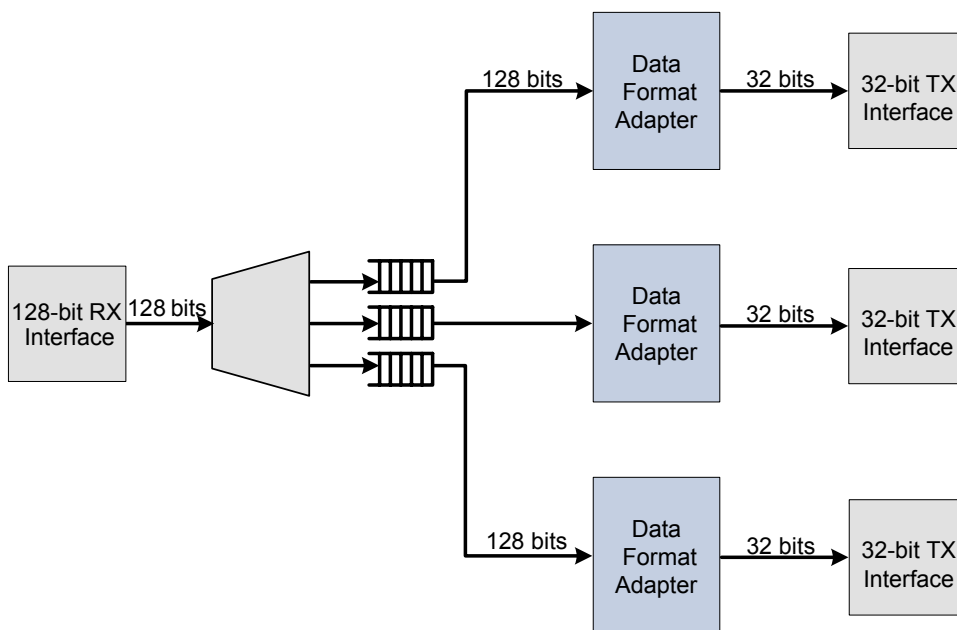
For complete information about these adapters, refer to the *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*.

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 buses of different widths. [Figure 3-4](#) shows an adapter that allows a connection between a 128-bit input bus and three 32-bit output buses.

**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 backpressure 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; however, this usage produces a warning that data may be lost.

## Multiplexer Examples

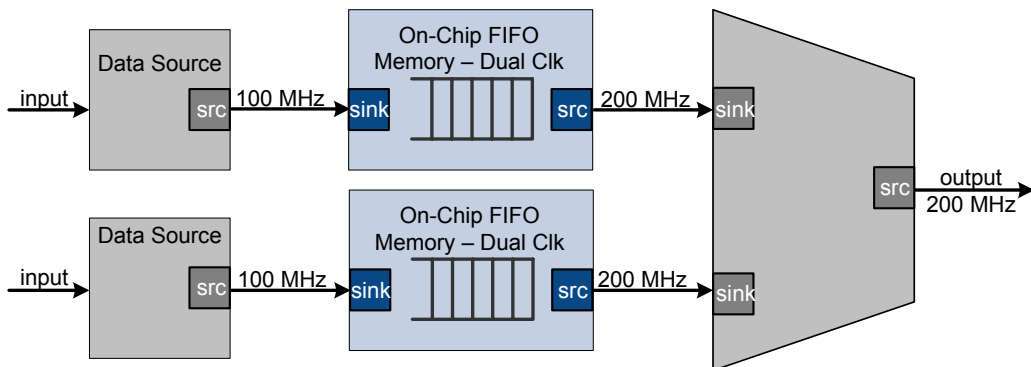
You can combine the three adapters referenced above with streaming components to create datapaths whose input and output streams have different properties. The following sections provide three examples of datapaths constructed using SOPC Builder whose output stream is higher performance than the input 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 boosts the frequency of a stream by 10% multiplexing input data from 2 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.

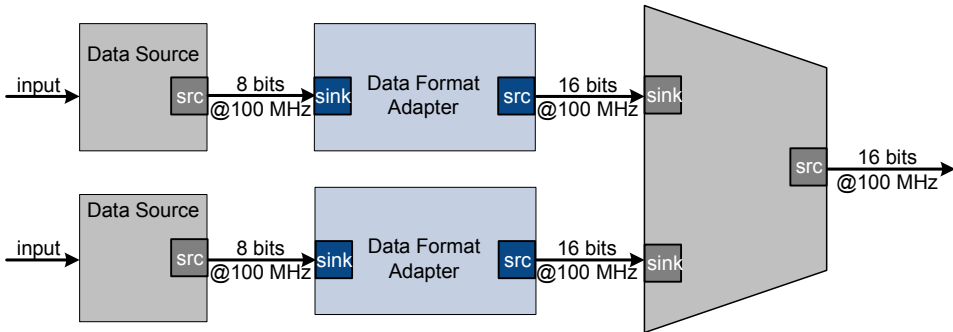
Figure 3-5. Datapath that Doubles the Clock Frequency



### 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.

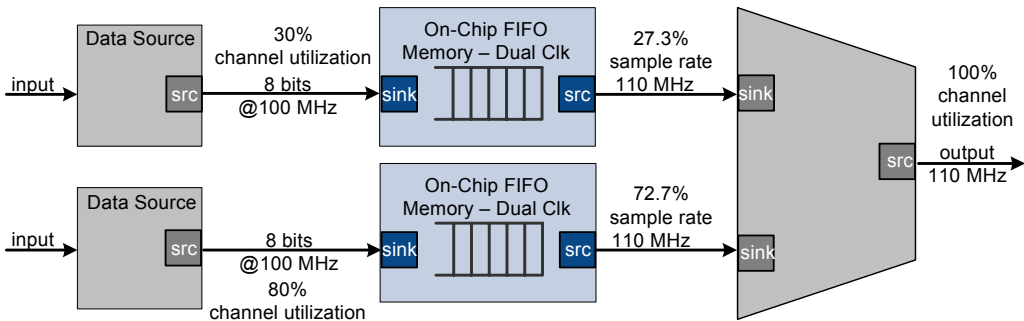
Figure 3-6. Datapath to Double Data Width and Maintain Original Frequency



### 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.

Figure 3-7. Datapath to Boost the Clock Frequency





## Referenced Documents

This chapter references the following documents:

- *Avalon Streaming Interface Specification*
- *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>		
<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
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 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 hardware description language (HDL) files for the component and integrate the component instance into the system module.
- 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 legacy custom-defined SOPC Builder components, in the following sections:

- ["Component Providers" on page 4-2](#)
- ["Component Hardware Structure" on page 4-2](#)
- ["List of Available Components in SOPC Builder" on page 4-4](#)
- ["Tcl Components" on page 4-5](#)

### New Component Structure in v7.1 of the Quartus II Software

Version 7.1 of the Quartus® II software provided a new mechanism for storing and finding component files located on your computer.



If you use components created with a previous version of the Quartus II software, read through this chapter to familiarize yourself with the differences. This document uses the term "legacy components" to refer to components created with a previous version of the Quartus II software.

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

- Legacy components that use a **More Options** tab in SOPC Builder, such as complex IP components provided by third-party IP developers, cannot be instantiated or used in version 7.1 and beyond. If your component has a “bind” program, you cannot use the component.
- To edit a legacy component using the component editor in version 7.1 and beyond, you must first upgrade the component to the new component editor flow. The process is automatic. However, the result is not backward compatible with previous versions.

## Component Providers

SOPC Builder components can be installed on your computer by several possible providers, including the following:

- The Quartus II software, which includes SOPC Builder, can install components as part of the fundamental functionality of the software.
- The Altera® MegaCore IP Library provides several intellectual property (IP) design blocks that are SOPC Builder ready.
- Third-party IP developers can provide IP blocks as SOPC Builder ready components, including software drivers and documentation.
- Altera development kits, such as the Nios® II Development Kit, can provide SOPC Builder components as features.
- The SOPC Builder component editor can turn your own HDL files into custom components.

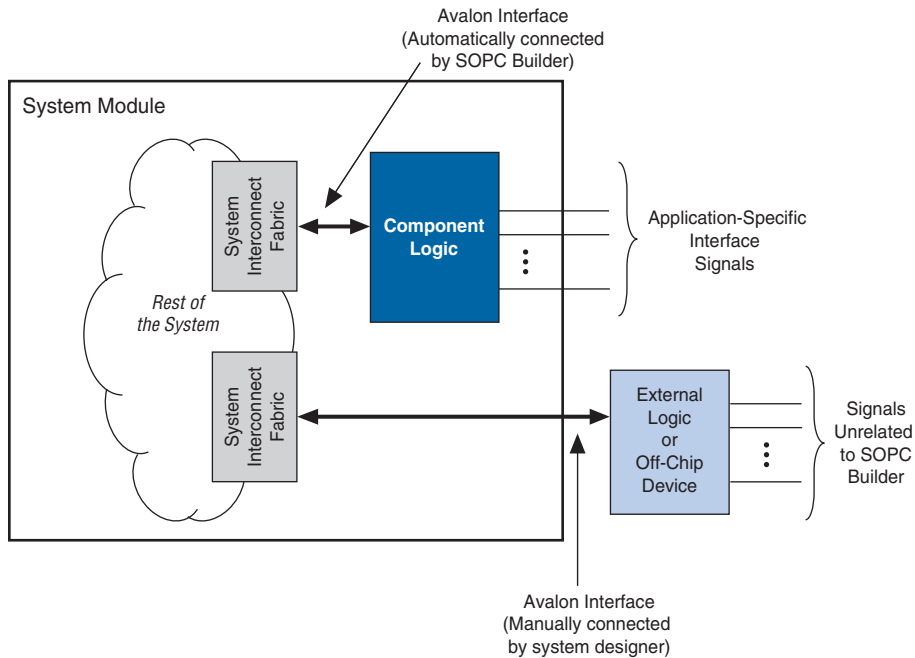
## Component Hardware Structure

There are two types of components, based on where the associated component logic resides:

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

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

**Figure 4–1. Component Logic Inside and Outside the System Module**



### Components That Include Logic Inside the System Module

For components that include logic inside the system module, the component provides a full description of its hardware by specifying an HDL file. During system generation, SOPC Builder instantiates the component in the system and connects it to the rest of the system. The component can include export signals, which become ports on the system itself, so that you can manually connect them to logic outside the system module.

In general, components connect to the system interconnect fabric using either the Avalon<sup>®</sup> 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 port for control.

## Components That Interface to Logic Outside the System Module

For components that interface to logic outside the system module, the component files describe only the interface to the external logic. During system generation, SOPC Builder only exports an interface for the component to the top-level system module. You must manually connect the interface to the component outside the system.

## List of Available 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. There are several mechanisms that SOPC Builder uses to populate 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 a subdirectory here. This mechanism 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 mechanism is recommended for all global components.
- Quartus II project directory and user library paths—SOPC Builder identifies component files stored in the current Quartus II project directory and user library paths.
- Legacy component search paths—SOPC Builder searches the paths where previous versions of SOPC Builder expected to find component files.

The rest of this section focuses on Tcl components.

## Tcl Components

Tcl components are components where interaction with SOPC Builder is defined with a simple text file written in the Tcl scripting language. This section describes the structure of Tcl components and how they are stored.



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

### Component Description File (`_hw.tcl`)

At a minimum, a Tcl component consists of the following files:

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

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

The SOPC component editor can generate components without Verilog HDL or VHDL files.

### Component File Organization

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

- `component_library/`
  - `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
  - There is no expectation of an HDL folder, even for components that are created with the component editor. If you want to bundle your component in a directory, the basic structure is as follows:
    - `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 will often include a `_sw.tcl` file and the software definitions and drivers it refers to. Refer to the component software specification for further details.

## Referenced Document

This chapter references [Chapter 5, Component Editor](#).

## Document Revision History

[Table 4-1](#) shows the revision history for this chapter.

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
October 2007, v7.2.0	<ul style="list-style-type: none"> <li>Description added of Tcl components and removal of custom-defined components.</li> <li>Added warning that SOPC Builder does not support parameter values &gt; 31 bits</li> </ul>	—
May 2007, v7.1.0	<ul style="list-style-type: none"> <li>Described the new structure of components which is new in 7.1.</li> <li>Added and updated the sources of components list.</li> <li>Reorganized content of the chapter.</li> <li>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.”</li> <li>Removed description of SOPC Builder MegaWizard® Plug-In Manager component discovery mechanism that was inaccurate.</li> </ul>	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 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 a hardware description language (HDL) file that describes the modules that compose your component hardware.
- Define the interfaces on the component and provide information about how the interface functions.
- Specify the hardware interface or interfaces to the component, and define the behavior of each interface signal. Assign module signals to interfaces and determine signal roles.
- 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 the use of the component editor, see the following sections:

- To start the component editor, refer to [“Starting the Component Editor” on page 5-2](#).
- For information about specifying HDL files that describe a component, refer to [“HDL Files Tab” on page 5-2](#).
- For information about specifying interface signals, refer to [“Signals Tab” on page 5-3](#).
- For information about specifying the Avalon-MM type of interface signals, refer to [“Interfaces Tab” on page 5-6](#).
- For information about specifying parameters, refer to [“Component Wizard Tab” on page 5-6](#).
- To save a component, refer to [“Saving a Component” on page 5-7](#).
- For information about changing a component after it has been saved, refer to [“Editing a Component” on page 5-8](#).



For more information about components, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*. For more information about the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification*.

## Component Hardware Structure

The component editor creates components with the following hardware characteristics:

- A component has one or more interfaces. Typically, an *interface* means an Avalon-MM master port or slave port. The component editor lets you build a component with any combination of Avalon-MM master or slave ports. You can also specify component signals that must appear at the top-level of the SOPC Builder system module, which you can manually connect to the logic outside the system module. Interfaces include:
  - Avalon-MM master/slave
  - Avalon Streaming source/sink
  - Interrupt sender/receiver
  - Clock input and output
  - Nios II Custom Instruction Conduit (for export only)
- Each interface is comprised of one or more signals.
- The component can represent a component that is instantiated inside the SOPC Builder system, and can represent a component 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 first row of the table on the **HDL Files** tab must include the file with the top-level module and must specify all the HDL files. You use the **HDL Files** tab to specify an existing Verilog HDL, or VHDL file that describes the interface to the component hardware. If your component is an interface to external logic, then do not specify an HDL file.

You can also use the component editor to define logic interfaces to external logic. In this case, you do not provide HDL files, and instead you use the component editor to manually define the hardware interface.

After you specify an HDL file, the component editor immediately 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 specified files. 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.

If your design requires extra simulation files, you can specify them in the **Simulation Files** table. All files used in the simulation must be specified, even those already included for synthesis. SOPC Builder includes these files in the system test bench so they can provide special functionality during simulation. The simulation files do not affect the generated system hardware.



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 is an interface to external logic, you must manually add the signals that comprise the interface to the external logic. The **Interface** list also allows creation of a new interface.

Each signal must belong to an interface and be assigned a signal type. The signal type for new signals that have not been assigned a signal type is `Export`, which means that SOPC Builder does not connect the signal internally to the system module, and instead exposes the signal on the top-level system module.

You assign each signal to an interface using the **Interface** list. In addition to Avalon Memory-Mapped and Streaming interfaces, components typically have 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 5-1 lists the signal naming conventions.

Type of Signal	Name Convention
Signal associated with a specific interface	<code>&lt;interface type&gt;_&lt;interface name&gt;_&lt;signal type&gt;[_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 5-2 lists the valid values for Interface Type.

Value	Meaning
avs	Avalon-MM slave
avm	Avalon-MM master
ats	Avalon-MM tristate slave
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	Conduit Start
coe	Conduit End
ncm	Nios II Custom Instruction Master
ncs	Nios II Custom Instruction Slave
csi_clockreset_clk	Clock Reset
csi_clockreset_reset_n	Clock Reset N

Example 5-1 shows a Verilog HDL module declaration with signal names that infer two Avalon-MM slave ports.

---

**Example 5–1. Verilog Module With Automatically Recognized Signal Names**

```
module my_multiport_component (
    // Signals for Avalon-MM slave port "s1"
    avs_s1_clk,
    avs_s1_reset_n,
    avs_s1_address,
    avs_s1_read,
    avs_s1_write,
    avs_s1_writedata,
    avs_s1_readdata,
    avs_s1_export_dac_output,

    // Signals for Avalon-MM slave port "s2"
    avs_s2_address,
    avs_s2_read,
    avs_s2_readdata,
    avs_s2_export_dac_output,

    // Clock/Reset Interface csi_clockreset_clk
);
```

---

### Templates for Interfaces to External Logic

If you are creating 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:

- 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 to meet your needs.

## 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 system module.

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 which 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.
- **Component Display Name**—Specifies the internal name of the component. The internal name is used when saving a system containing an instance of this component, and is the name use for the component type when you create a system using a script..
- **Component Version**—Specifies which version of the component you are using.
- **Component 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 (optional).
- **Created By**—Allows you to specify the author of the component (optional).
- **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 (optional).
- **Parameters**—Allows you to specify the parameters for creating the component. See further description below.

The component editor assigns the class name to be the same name as the top-level HDL module. The class name is the name SOPC Builder uses to identify the component.

## 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, or *generics* for VHDL), those parameters appear in the **Parameters** table. These 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 system module, or vice versa, numeric parameters must be 32-bit decimal integers. Passing other numeric parameter types might fail.

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

## 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 module>\_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 with the component editor to reflect the changes you made to the HDL.

# Referenced Documents

This chapter references the following documents:

- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *Avalon Memory-Mapped Interface Specification*
- *Building a Component Interface with TCL Scripting Commands* chapter in volume 4 of the *Quartus II Handbook*
- *Nios II Software Developer's Handbook*



## Document Revision History

Table 5-3 shows the revision history for this chapter.

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
October 2007, v7.2.0	<ul style="list-style-type: none"> <li>Updated several paragraphs describing the latest GUI.</li> </ul>	—
May 2007, v7.1.0	<ul style="list-style-type: none"> <li>Updated all sections to reflect significant functional differences in version 7.1.</li> <li>Added section “Changes to Component Editor in Version 7.1” on page 5-2.</li> <li>Updated section “Component Editor Output” and “Re-editing Components” to accommodate new component structure with 7.1 release.</li> <li>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.”</li> <li>Removed screen shots that simply reflect what user sees when using the tool without illustrating a particular point.</li> <li>Added Referenced Documents section which links to all referenced documents.</li> <li>Added statement that all simulation files, not just top-level file, must be added using the HDL files tab.</li> </ul>	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.	—
May 2006, v6.0.0	No change from previous release.	—
December 2005, v5.1.1	<ul style="list-style-type: none"> <li>Added section “Naming Signals for Automatic Type and Interface Recognition” on page 5-4.</li> <li>Added section “Templates for Interfaces to External Logic” on page 5-6.</li> <li>Clarified operation of the Save command.</li> <li>Updated all screenshots.</li> </ul>	—
October 2005, v5.1.0	No change from previous release.	—
May 2005, v5.0.0	Initial release.	—



This chapter describes the Tcl scripting commands that you can use to define custom components for use in an SOPC Builder system. You can also use the scripting interface to declare and set parameter values for your components.

The Tcl scripting commands provide a programmatic interface that you might prefer to the graphical user interface (GUI) of the component editor. If you need to make global updates to multiple components, Tcl scripts allow you to make the changes without accessing each component through the GUI.

You can use the Tcl scripting commands or the component editor to create a component description file with the file name *<name of top-level module>\_hw.tcl*. This file is stored in the same directory as the HDL file that provides the top-level description of the component. You can edit this file using the text editor of your choice.

You can download sample \*\_hw.tcl files from the Altera website by clicking the Design Example hyperlink located under this chapter, *Building a Component Interface with Tcl Scripting Commands*.

The remainder of this chapter describes the commands and properties you can use to describe components, component interfaces and parameters. These include:

- “Organization of a Component Tcl File” on page 6-2
- “Set and Add Commands” on page 6-3
- “Module Properties” on page 6-4
- “Clock Interface” on page 6-4
- “Avalon-MM Master Interface” on page 6-5
- “Avalon-MM Slave Interface” on page 6-5
- “Avalon-MM Tristate Interface” on page 6-7
- “Nios II Custom Instruction Interface” on page 6-8
- “Interrupt Interface” on page 6-9
- “Conduit Interface” on page 6-10

## Organization of a Component Tcl File

The following steps describe how to organize a component Tcl file.

1. Start the component definition with the `set_source` command, followed by the `set_module` command. The name of the module must match the component's top-level Verilog or VHDL entity name.

---

### Example 6–1. Example of Set Module Command

```
set_module "my_module"
```

---

2. Define the module properties, which are pieces of static information about a module. The following example illustrates some of the `set` command and `module` properties. See [Table 6–5](#).

---

### Example 6–2. The Set Command and Module Properties

```
set_source_file "./my_component.v"
set_module_description "My Component"
set_module_property version "1.0"
set_module_property group "My Components"
set_module_property simulationFiles [ list "./my_component.v" ]
```

---

3. Define the module parameters, which are settings that the user of the component makes when parameterizing it. The following example illustrates how to define module parameters.

---

### Example 6–3. Example of Parameters

```
# Module parameters
add_parameter "DWIDTH" "integer" "32" ""
add_parameter "AWIDTH" "integer" "32" ""
```

---

4. Add interfaces. For each interface, first add the interface, then set its properties and define its ports. Refer to the Avalon-MM specification for port types. The following example defines an Avalon-MM slave interface using only the required properties.

---

### Example 6–4. Avalon-MM Slave Interface

```
# Interface my_slave
# all interfaces must specify an associated clock interface
add_interface "my_slave" "avalon" "slave" "my_clock_interface"

set_interface_property "my_slave" "timingUnits" "cycles"
```

```

set_interface_property "my_slave" "writeWaitTime" "0"
set_interface_property "my_slave" "readLatency" "0"
set_interface_property "my_slave" "holdTime" "0"
set_interface_property "my_slave" "readWaitTime" "0"
set_interface_property "my_slave" "setupTime" "0"

# Ports in interface my_slave
add_port_to_interface "my_slave" "my_slave_write" "write"
add_port_to_interface "my_slave" "my_slave_writedata" "writedata"
add_port_to_interface "my_slave" "my_slave_waitrequest" "waitrequest"

```

## Set and Add Commands

The set and add commands establish basic information about a component.

**Table 6–1. Set and Add Commands**

Command	Arguments
set_module	<name of the module> (1)
set_source_file	<path to HDL file> (2)
set_module_description	<description of the module>
set_module_property	<name of property> <value of property>
add_interface	<name of interface> <type of interface> <direction> <associated clock> (3)
set_interface_property	<name of interface> <name of property> <value of property>
add_port_to_interface	<name of interface> <port name> <type of port>
set_port_direction_and_width	<name of port> <direction> <width>

**Notes to Table 6–1:**

- (1) Declares a new module. Must match the top-level Verilog HDL module or VHDL entity.
- (2) If the component is not based on HDL, `set_source_file` should be used with an empty string, such as `"set_source_file"`.
- (3) This command is only required when a source file is not set. If a source file is set, the Quartus II software analyzes the file and determines the port widths and directions.

## Module Properties

The module properties are the arguments to the `set_module_property` command. Table 6–2 lists the module properties.

Name	Legal Values	Description
<code>version</code>	dotted integers	A version string, for example: 1.2.3
<code>group</code>	string	A string that represents the category under which the component should be listed.
<code>simulationFiles</code>	list of strings	The name of HDL files for use in simulation. This parameter is required even if the same file is used for synthesis and simulation. All files required for simulation must be specified, not just the top-level file.
<code>synthesisFiles</code>	list of strings	The name of HDL files for use in synthesis.
<code>author</code>	string	Name of the component author.
<code>iconPath</code>	string	Path to an image file, which contains an icon to show in the default editor. When referring to local files, they are relative to the Tcl File (.tcl).
<code>datasheetURL</code>	string	URL pointing to the component datasheet. Can be local or on a network. When referring to local files, they are relative to the TCL file.

## Clock Interface

There are no special properties for clock interfaces. A clock interface should not specify an associated clock interface. Clock interface directions are “source” and “input”. The following example defines a clock interface.

### Example 6–5. Clock Interface

```
# Clock Interface <my_clk_interface>
add_interface "my_clk_interface" "clock" "input"
set_interface_property "clock" "externallyDriven" "false"
set_interface_property "clock" "clockRateKnown" "false"
set_interface_property "clock" "clockRate" "0"
# Ports in interface clock
add_port_to_interface "clock" "clk" "clk"
add_port_to_interface "clock" "reset_n" "reset_n"
```

## Avalon-MM Master Interface

Table 6–3 describes the properties that characterize an Avalon-MM master interface. The direction of an Avalon-MM master interface is “master”.

**Table 6–3. Avalon-MM Master Interface Properties**

Name	Default Value	Legal Values	Description
doStreamReads	false	(true,false)	Specifies whether the master supports Avalon flow control read accesses. (This property is optional).
doStreamWrites	false	(true,false)	Specifies whether the master supports Avalon flow control write accesses. (This property is optional).—
burstOnBurstBoundaries Only	false	(true,false)	If true, bursts are aligned on burst size. (This property is optional.)

## Avalon-MM Slave Interface

Table 6–4 describes the properties that characterize an Avalon-MM slave interface. The direction of an Avalon-MM slave interface is “slave”.

**Table 6–4. Avalon-MM Slave Interface Properties (Part 1 of 2)**

Name	Default Value	Legal Values	Description
readLatency	0	[0 - 63]	Read latency for fixed-latent slaves.
timingUnits	cycles	(cycles, nanoseconds)	Specifies the unit for writeWaitTime, readWaitTime.
writeWaitTime	0	[1000 - 0]	Specifies additional time in units of timeUnits for write to be asserted.
holdTime	0	-	Specifies time in timeUnits between deassertion of read/write and deassertion of chipselect, address and data.
readWaitTime	1	[1000 - 0]	Specifies additional time in units of timeUnits for read to be asserted.
setUpTime	0	[1000 - 0]	Specifies time in timeUnits between assertion of chipselect, address and data and assertion of read/write.
maximumPendingReadTransactions	0	position	The maximum number of pending read accesses which can be queued up by the slave.
burstOnBurstBoundaries Only	false	(true,false)	If true, bursts are aligned on burst size.

**Table 6–4. Avalon-MM Slave Interface Properties (Part 2 of 2)**

Name	Default Value	Legal Values	Description
isNonVolatileStorage	false	(true,false)	For software environment purposes. Indicates if the memory is a non-volatile storage device.
printableDevice	false	(true,false)	For software environment purposes. Indicates if the memory is a non-volatile storage device.
isMemoryDevice	false	(true,false)	For software environment purposes. States that the slave is a reasonable target for code and data.

## Avalon-ST Source Interface

Table 6–5 lists the properties that characterize an Avalon-ST source interface. Refer to the Avalon-ST specification for port types. The direction of an Avalon-ST source interface is “source”.

**Table 6–5. Avalon-ST Source Interface Properties**

Name	Default Value	Legal Values	Description
symbolsPerBeat	1	[1-512]	The number of symbols that are transferred on every valid cycle.
dataBitsPerSymbol	8	[1-512]	Defines the number of bits per symbol. Most interfaces are byte-oriented so that a symbol is 8 bits.
readyLatency	0	[8-0]	Defines the relationship between assertion/deassertion of the ready signal, and cycles which are considered to be ready for data transfer, separately for each interface.
maxChannel	0	[low-high]	The maximum number of channels that a data interface can support.



## Avalon-ST Sink Interface

Table 6–6 lists the properties that characterize an Avalon-ST sink interface. Refer to the Avalon-ST specification for port types. The direction of an Avalon-ST sink interface is “sink”.

**Table 6–6. Avalon-ST Sink Interface Properties**

Name	Default Value	Legal Values	Description
<code>symbolsPerBeat</code>	1	[512-1]	The number of symbols that are transferred on every valid cycle.
<code>dataBitsPerSymbol</code>	8	[512-1]	Defines the number of bits per symbol. Most interfaces are byte-oriented so that a symbol is 8 bits.
<code>readyLatency</code>	0	[8-0]	Defines the relationship between assertion/deassertion of the <code>ready</code> signal, and cycles which are considered to be ready for data transfer, separately for each interface.
<code>maxChannel</code>	0	[255-0]	The maximum number of channels that a data interface can support.

## Avalon-MM Tristate Interface

Table 6–7 lists the properties that characterize an Avalon-MM tristate interface. The Avalon-MM tristate interface properties include all the properties that define the Avalon-MM slave interface, plus two additional properties: `activeCSThroughReadLatency` and `maximumPendingReadTransactions`.



Note that `maximumPendingReadTransactions` is not tristate specific. This property can also be assigned to an Avalon State.

The direction of an Avalon-MM tristate interface is “slave”.

**Table 6–7. Avalon-MM Tristate Interface Properties (Part 1 of 2)**

Name	Default Value	Legal Values	Description
<code>readLatency</code>	0	<code>num_cycles</code>	Read latency for fixed-latency slaves.
<code>writeLatency</code>	0	<code>num_cycles</code>	Delay in cycles between acceptance of a write access and acceptance of valid <code>writedata</code> .
<code>timingUnits</code>	<code>cycles</code>	( <code>cycles</code> , <code>nanoseconds</code> )	Specifies the unit for <code>writeWaitTime</code> and <code>readWaitTime</code> .
<code>writeWaitTime</code>	0	[1000-0]	Specifies additional time in units of <code>timeUnits</code> for write to be asserted.

**Table 6–7. Avalon-MM Tristate Interface Properties (Part 2 of 2)**

Name	Default Value	Legal Values	Description
holdTime	0	—	Specifies time in <code>timeUnits</code> between deassertion of read/write and deassertion of <code>chipselct</code> , address and data.
readWaitTime	1	[1000-0]	Specifies additional time in units of <code>timeUnits</code> for read to be asserted.
setupTime	0	—	Specifies time in <code>timeUnits</code> between assertion of <code>chipselct</code> , address, and data and assertion of read/write.
activeCSThroughRead Latency	false	(true,false)	If true, assert <code>chipselct</code> while <code>readdata</code> is pending.
maximumPendingRead Transactions	false	—	States the maximum number of pending read transactions.
minimumUninterrupted RunLength	1	an integer	Specifies a minimum arbitration share value.
isNonVolatileStorage	false	(true,false)	For software environment purposes. True for flash memories.
printableDevice	false	(true,false)	For software environment purposes. States that the slave is a reasonable sink for <code>printf()</code> data.
isMemoryDevice	false	(true,false)	For software environment purposes. States that the slave is a reasonable target for code and data.

## Nios II Custom Instruction Interface

Table 6–8 lists all the properties that characterize Nios II custom instructions.

**Table 6–8. Nios II Custom Instruction Interface**

Name	Default Value	Legal Values	Description
operands	0	[2-0]	Number of operands used by the custom instruction module.
clockCycle	0	—	Number of clock cycles the custom instruction requires before a valid result is returned—used by multicycle custom instructions.

The following example illustrates all the properties for a custom instruction.

**Example 6–6. Custom Instruction Example**

```

set_source_file "custominstruction.v"
set_module "custominstruction"
set_module_description "A custom instruction"
set_module_property version "1.0"
set_module_property group "User Logic"

# Module parameters
# Interface nios_custom_instruction_slave_0
add_interface "nios_custom_instruction_slave_0" "nios_custom_instruction" "slave"
"asynchronous"
set_interface_property "nios_custom_instruction_slave_0" "operands" "2"
set_interface_property "nios_custom_instruction_slave_0" "clockCycle" "2"

# Ports in interface nios_custom_instruction_slave_0
add_port_to_interface "nios_custom_instruction_slave_0" "clk" "clk"
add_port_to_interface "nios_custom_instruction_slave_0" "reset" "reset"
add_port_to_interface "nios_custom_instruction_slave_0" "clk_en" "clk_en"
add_port_to_interface "nios_custom_instruction_slave_0" "start" "start"
add_port_to_interface "nios_custom_instruction_slave_0" "n" "n"
add_port_to_interface "nios_custom_instruction_slave_0" "dataaa" "dataaa"
add_port_to_interface "nios_custom_instruction_slave_0" "datab" "datab"
add_port_to_interface "nios_custom_instruction_slave_0" "a" "a"
add_port_to_interface "nios_custom_instruction_slave_0" "b" "b"
add_port_to_interface "nios_custom_instruction_slave_0" "c" "c"
add_port_to_interface "nios_custom_instruction_slave_0" "readra" "readra"
add_port_to_interface "nios_custom_instruction_slave_0" "readrb" "readrb"
add_port_to_interface "nios_custom_instruction_slave_0" "writerc" "writerc"
add_port_to_interface "nios_custom_instruction_slave_0" "result" "result"
add_port_to_interface "nios_custom_instruction_slave_0" "done" "done"

```

**Interrupt Interface**

Slave components in an SOPC Builder system typically generate interrupts. A processor typically clears the interrupt bits in the slave's control and status registers after servicing the interrupt. [Table 6–9](#) lists the properties that characterize interrupts. The direction of an interrupt interface is “sender” and “receiver”.

**Table 6–9. Interrupt Interface Properties**

Name	Default Value	Legal Values	Description
associatedAddressablePoint	—	an interface name	This parameter takes the name of the component interface that provides access to the registers that should be cleared after the interrupt is serviced.

The following example defines an interrupt interface.

**Example 6–7. Interrupt Interface**

```
# IRQ Interface my_slave_irq
# legal values for the third parameter <direction> are sender and receiver
add_interface my_slave_irq "interrupt" "sender" "global_signals_clock"

set_interface_property "my_slave_irq" "associatedAddressablePoint" "my_slave"

# Ports in interface my_slave_irq
# Generally there is only one signal of type interrupt
add_port_to_interface "my_slave_irq" "my_irq" "irq"
```

---

**Conduit Interface**

A conduit interface is used to export arbitrary input and output signals outside of an SOPC Builder system. There are no special properties associated with conduit interfaces.

The following example illustrates the conduit interface.

**Example 6–8. Conduit interface**

```
# Wire Interface global_signals_export
add_interface "global_signals_export" "conduit" "output" "my_clk_interface"

# Ports in interface global_signals_export
add_port_to_interface "global_signals_export" "prbs_test_error" "export"
add_port_to_interface "global_signals_export" "prbs_test_done" "export"
```

---

**Document Revision History**

Table 6–10 shows the revision history for this chapter.

<i>Table 6–10. Document Revision History</i>		
Date and Document Version	Changes Made	Summary of Changes
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 you identify the files you must include when archiving an SOPC Builder project. With this information, you can archive:

- The SOPC Builder system module
- The associated Nios<sup>®</sup> II software project, if any
- The associated Nios II system library project, if any

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 does not provide step-by-step instructions. It does cover the following information:

- How to find and identify the files that you must include in an archived SOPC Builder design, refer to [“Required Files” on page 7-2.](#)
- Which files must have write permission to allow the design to be generated and the software projects compiled, refer to [“File Write Permissions” on page 7-4.](#)

## Scope

This chapter provides information about archiving SOPC Builder system modules, 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 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<sup>®</sup> 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 and Nios II development software 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 release notes for the Quartus II software and the Nios II EDS.

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, but not including any Nios II software. Quartus II adds all HDL files to the archive, including HDL files generated by SOPC Builder, although these files are not strictly necessary.

This chapter is only concerned with archiving the SOPC Builder system, without the generated HDL files, but with all files needed to regenerate them and rebuild the Nios II software (if any).



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

## Required Files

This section describes the files required by 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**) and the executable software (**.elf**).

If you have Nios II software projects, archive them together with the SOPC Builder system on which they are based. You cannot rebuild a Nios II software project without its associated SOPC Builder system.



## SOPC Builder Design Files

The files listed in [Table 7-1](#) are located in the Quartus II project directory.

File description	File name	Write permission required? (1)
SOPC Builder system description	<socp_builder_system>.sopc	Yes
SOPC Builder legacy system description (2)	<socp_builder_system>.ptf	Yes
All non-generated HDL source files (3)	for example: <b>top_level_schematic.bdf,</b> <b>customlogic.v</b>	No
Quartus II project file	<project_name>.qpf	No
Quartus II settings file	<project_name>.qsf	No

### Notes to [Table 7-1](#):

- (1) For further information about write permissions, refer to “[File Write Permissions](#)” on [page 7-4](#).
- (2) The <socp\_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.
- (3) 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)2007 Altera Corporation. All rights reserved.

## Nios II Application Software Project Files

The files listed in [Table 7-2](#) are located in the Nios II software project directory.



For more information about Nios II software projects, refer to the *Nios II Software Developer's Handbook*.

File Description	File Name	Write Permission Required? (1)
All source files	for example: <b>app.c, header.h,</b> <b>assembly.s, lookuptable.dat</b>	No
Eclipse project file	<b>.project</b>	No
C/C++ Development Toolkit project file	<b>.cdtproject</b>	Yes
C/C++ Development Toolkit option file	<b>.cdtbuild</b>	No
Software configuration file	<b>application.stf</b>	No

### Note to [Table 7-2](#):

- (1) For further information about write permissions, refer to “[File Write Permissions](#)” on [page 7-4](#).

## Nios II System Library Project

The files listed in [Table 7-3](#) are located in the Nios II system library project directory.



For more information about Nios II system libraries, refer to the *Nios II Software Developer's Handbook*.

File description	File name	Write permission required? (1)
Eclipse project file	<b>.project</b>	Yes
C/C++ Development Toolkit project file	<b>.cdtproject</b>	Yes
C/C++ Development Toolkit option file	<b>.cdtbuild</b>	No
System software configuration file	<b>system.stf</b>	Yes

**Note to Table 7-3:**

(1) For further information about write permissions, see “File Write Permissions” on page 7-4.



Archiving for projects that use Tcl scripting and java to create a Board Support Package (BSP) is covered in chapter 3 of the *Nios II Software Developer's Handbook, Common BSP Tasks*.

## File Write Permissions

You must have write permission for certain files. The tools write to these files as part of the generation and compilation process. If the files are not writable, the toolchain fails.

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:

- [The Quartus II Handbook, Volume 2](#)
- [Nios II Software Developer's Handbook, Common BSP Tasks](#)

## Document Revision History

Table 7-4 shows the revision history for this chapter.

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



This section provides instructions on how to use SOPC Builder to achieve specific goals. Chapters in this section serve to answer the question, "How do I use SOPC Builder?" Many chapters in this handbook provide design examples that you can download free from [www.altera.com](http://www.altera.com). Design file hyperlinks are located with individual chapters linked from the Altera web site.

This section includes the following chapters:

- [Chapter 8, Building Memory Subsystems Using SOPC Builder](#)
- [Chapter 9, Developing Components for SOPC Builder](#)



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 code, 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 SDRAM to store software code, 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 describes 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 8–8
- “EPCS Serial Configuration Device” on page 8–12
- “SDRAM” on page 8–14
- “Off-Chip SRAM and Flash Memory” on page 8–19

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* 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 Memory-Mapped Interface Specification*.

## Example Design

This chapter demonstrates the process for building a system that contains one of each type memory as shown in [Figure 8-1](#). Each section of the chapter builds on previous sections, culminating in a complete system.

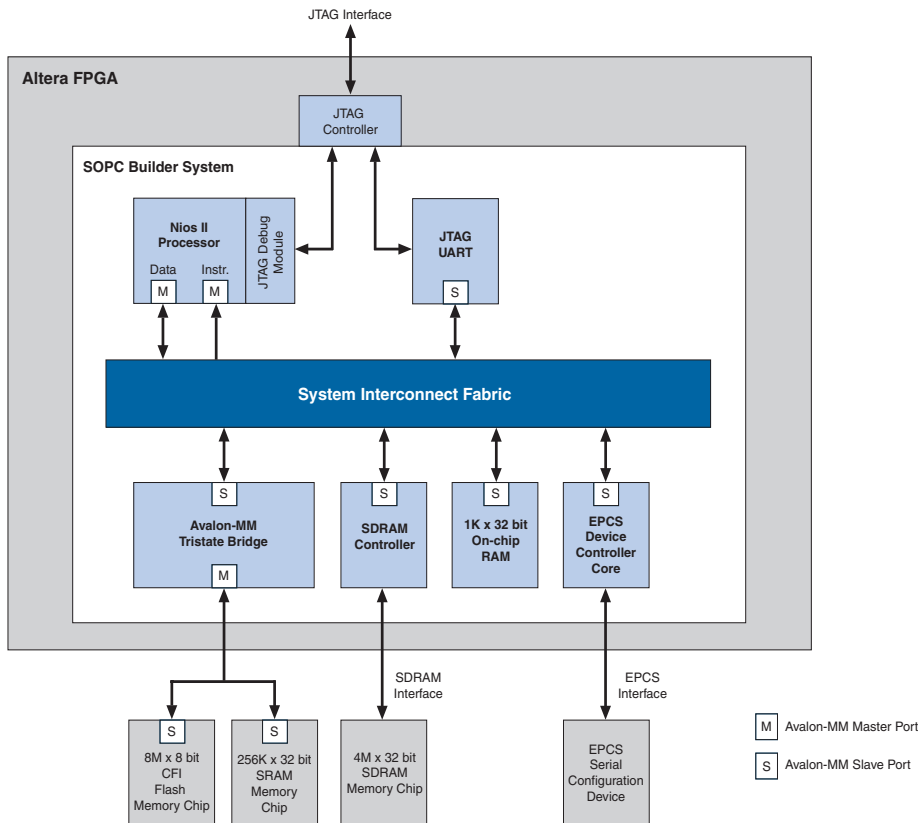
By following the example design in this chapter, you will learn how to create a complete customized memory subsystem for your system or design. The memory components in the example design are independent. For a custom system, you can instantiate exactly the memories you need, and skip the memories you do not need. Furthermore, you can 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 8-1](#) shows a block diagram of the example system.



Figure 8–1. Example Design Block Diagram



In Figure 8–1, 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 processor. However, the memory subsystem could be connected to any master component, either on-chip or off-chip.

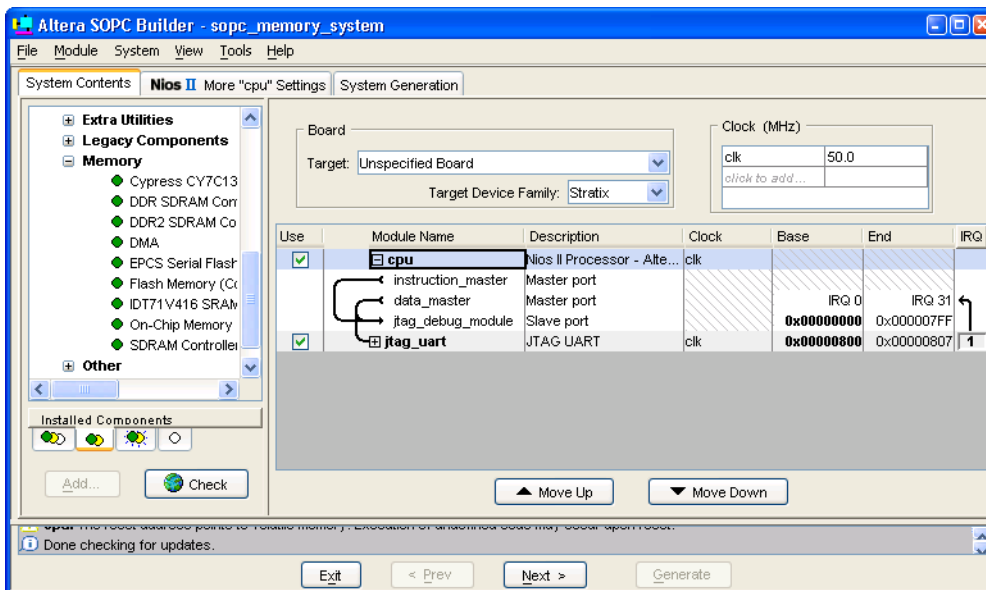
### Example Design Starting Point

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 module, 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 system module.

The starting point for this chapter 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/s core and the JTAG UART core; these settings do not affect the rest of the memory subsystem. [Figure 8-2](#) shows the starting point in the SOPC Builder.

**Figure 8-2. Starting Point for the Example Design**



All sections in this chapter build on this starting point.

## 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](http://www.altera.com/download).

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.

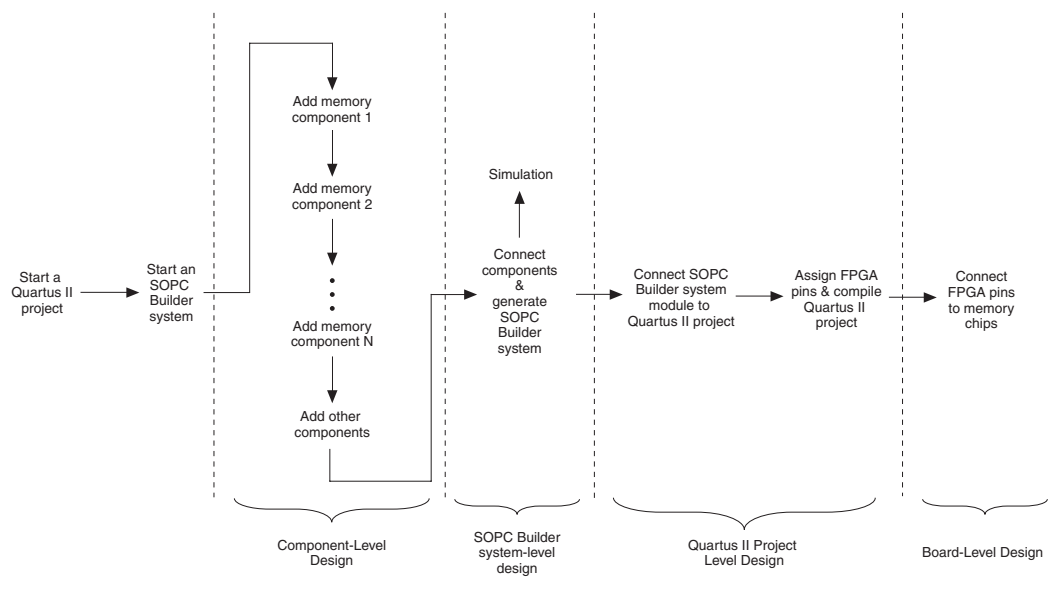
## Design Flow

This section describes the design flow for building memory subsystems with SOPC Builder.

The design flow for building a memory subsystem 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 8-3](#):

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 8–3. 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. Similar to 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 master ports in the system. Each memory component must be connected to at least one master port.
- Assign a base address.
- Assign a clock domain. A memory component can operate on the same or different clock domain as the master port(s) that access it.

## Simulation

In this step, you verify the functionality of the SOPC Builder system module. For systems with memories, this step depends on simulation models for each of the memory components, in addition to the system test bench 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 module with the rest of the Quartus II project. This step includes wiring the system module to FPGA pins, and wiring the system module 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 module 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 test bench for register transfer level (RTL) simulation of the system. This test bench instantiates the system module and can also instantiate memory models for external memory components. The test bench is plain text HDL, located at the bottom of the top-level system module HDL design file. To explore the contents of the auto-generated test bench, 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 [file name extension] (**.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 initialization contents automatically.

### *Vendor-Specific Memory Models*

You can also manually connect vendor-specific memory models to the system module. 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 system module, 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.

FPGAs have limited on-chip memory resources, which limits the maximum practical size of an on-chip memory to approximately one megabyte in the largest FPGA family.

### Component-Level Design for On-Chip Memory

In SOPC Builder you instantiate on-chip memory by clicking the **On-chip Memory (RAM or ROM)** in the component. 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**—Turning on this setting creates a memory component with two slave ports, which allows two master ports 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. The following choices are available:
  - **Auto**—This setting allows the Quartus II software to choose the most appropriate memory resource.
  - **M512**—This setting directs the Quartus II software to use M512 blocks.
  - **M4K**—This setting directs the Quartus II software to use M4K blocks.
  - **M-RAM**—This setting directs the Quartus II software to use M-RAM blocks. The 64 Kbit M-RAM blocks are appropriate for larger RAM data buffers. 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 port 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 slave ports. Pipelined access improves  $f_{MAX}$  performance, but also adds latency cycles when reading the memory. The **Read latency** option allows you to specify the number of read latency cycles required to access data. If the **Dual-port access** setting is turned on, you can specify a different read latency for each slave port.

### *Non-Default Memory Initialization*

For ROM memories, you can specify your own initialization file by selecting **Enable non-default initialization file**. If this option is selected, the file you specify will be used to initialize the ROM in place of the default initialization file created by SOPC Builder.

### *Enable In-System Memory Content Editor Feature*

Allows you to enable the In-System Memory Content Editor, which allows you to read data from and write data to in-system memory in a device while the device is running at speed and independently of system clocks with a JTAG interface.

## **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 8–6](#).

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 system module, 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 module, and therefore 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 integrated development environment (IDE) generates the memory initialization file automatically.



## Board-Level Design for On-Chip Memory

The on-chip memory is embedded inside the SOPC Builder system module, and therefore 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 port with read latency of one cycle.

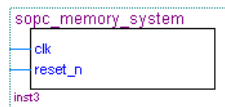
Figure 8–4 shows the SOPC Builder system after adding an instance of the on-chip memory component, renaming it to `onchip_ram`, and assigning it a base address.

**Figure 8–4. SOPC Builder System with On-Chip Memory**

Module Name	Description	Clock	Base	End	IRQ
<input type="checkbox"/> <b>cpu</b>	Nios II Processor - Altera C...	clk			
<input type="checkbox"/> instruction_master	Master port				
<input type="checkbox"/> data_master	Master port				
<input type="checkbox"/> jtag_debug_module	Slave port		0x00000000	0x000007FF	IRQ 0
<input type="checkbox"/> jtag_uart	JTAG UART	clk	0x00000800	0x00000807	1
<input type="checkbox"/> <b>onchip_ram</b>	On-Chip Memory (RAM or R...	clk	0x00001000	0x00001FFF	

For demonstration purposes, Figure 8–5 shows the result of generating the system module at this stage. (In a normal design flow, you generate the system only after adding all system components.)

**Figure 8–5. System Module with On-Chip Memory**



Because the on-chip memory is contained entirely within the system module, `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 in-field 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, 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 with Avalon Interface* 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 not many SOPC Builder system-level design considerations for EPCS devices:

- Assign a base address.
- Set the IRQ connection to NC (disconnected). 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`.

## 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

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 the user. 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.

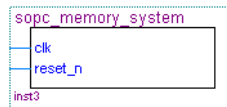
**Figure 8–6** shows the SOPC Builder system after adding an instance of the EPCS controller core and assigning it a base address.

**Figure 8–6. SOPC Builder System with EPCS Device**

Module Name	Description	Clock	Base	End	IRQ
<b>cpu</b>	Nios II Processor - Altera C...	clk			
instruction_master	Master port				
data_master	Master port				
jtag_debug_module	Slave port				
jtag_uart	JTAG UART	clk	0x00000800	0x00000807	1
onchip_ram	On-Chip Memory (RAM or ...	clk	0x00001000	0x00001FFF	
<b>epcs_controller</b>	EPCS Serial Flash Controller	clk	0x00002000	0x000027FF	16
epcs_control_port	Slave port				

For demonstration purposes only, [Figure 8–7](#) shows the result of generating the system module at this stage.

**Figure 8–7. System Module with EPCS Device**



Because the Quartus II software automatically connects the EPCS controller core to the FPGA pins, the system module 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 *Altera's Configuration Handbook*.

## SDRAM

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



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

## SOPC Builder System-Level Design for SDRAM

In the SOPC Builder **System Contents** tab, the SDRAM controller looks like any other memory component. Similar to on-chip memory, there are few SOPC Builder system-level design considerations for SDRAM. See [“SOPC Builder System-Level Design” on page 8–6](#).

## 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`.

Alternately, 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 8–7](#) and the *SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

## Quartus II Project-Level Design for SDRAM

SOPC Builder generates a system module 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 module. 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.



For further details, refer to the *SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

## **Example Design with SDRAM**

This section demonstrates adding a 16-Mbyte SDRAM device to the example design. This SDRAM is a single device with 32-bit data. [Figure 8–8](#) shows the **SDRAM Controller** configuration wizard settings for the example design.

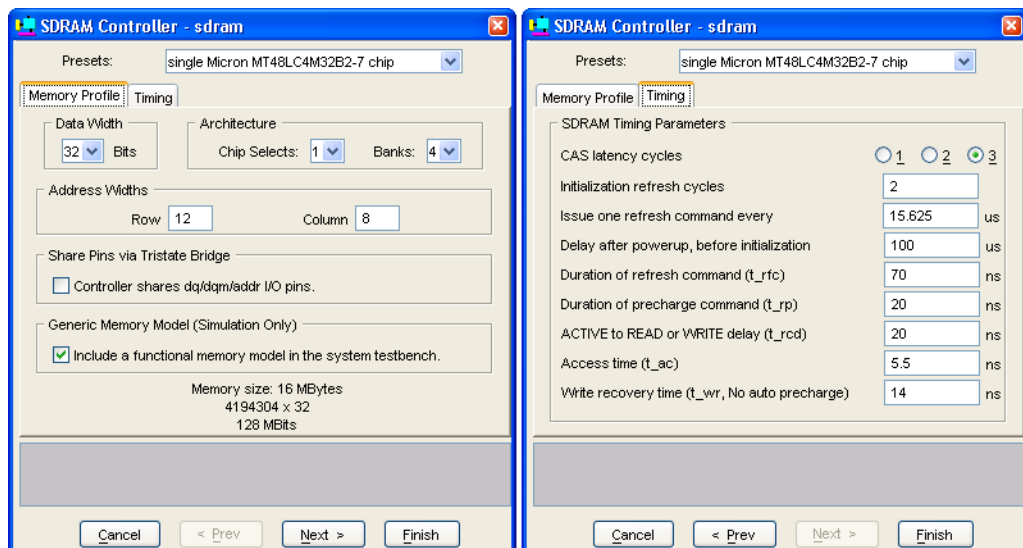
**Figure 8–8. SDRAM Controller Configuration Wizard**

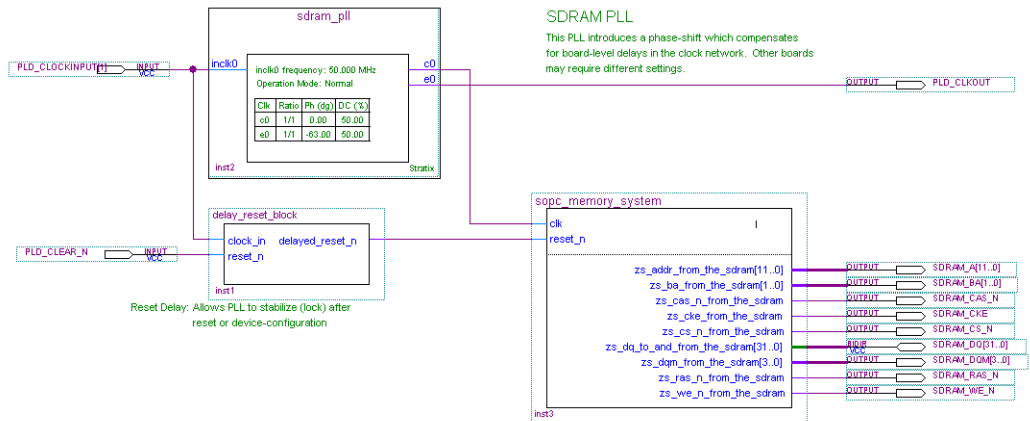
Figure 8–9 shows the SOPC Builder system after adding an instance of the SDRAM controller, renaming it to **sdram**, and assigning it a base address.

**Figure 8–9. SOPC Builder System with SDRAM**

Module Name	Description	Clock	Base	End	IRQ
cpu	Nios II Processor - Altera Corp...	clk			
instruction_master	Master port				
data_master	Master port				
jtag_debug_module	Slave port				
jtag_uart	JTAG UART	clk	0x00000800	0x00000807	1
onchip_ram	On-Chip Memory (RAM or ROM)	clk	0x00001000	0x00001FFF	
epcs_controller	EPCS Serial Flash Controller	clk	0x00002000	0x000027FF	NC
sdram	SDRAM Controller	clk	0x01000000	0x01FFFFFF	
s1	Slave port				

For demonstration purposes, Figure 8–10 shows the result of generating the system module at this stage, and connecting it in **toplevel\_design.bdf**.

Figure 8–10. *toplevel\_design.bdf* with SDRAM



After generating the system, the top-level system module file **socp\_memory\_system.v** contains the list of SDRAM-related I/O signals which must be connected to FPGA pins:

```
output [ 11: 0] zs_addr_from_the_sdram;
output [  1: 0] zs_ba_from_the_sdram;
output          zs_cas_n_from_the_sdram;
output          zs_cke_from_the_sdram;
output          zs_cs_n_from_the_sdram;
inout  [ 31: 0] zs_dq_to_and_from_the_sdram;
output [  3: 0] zs_dqm_from_the_sdram;
output          zs_ras_n_from_the_sdram;
output          zs_we_n_from_the_sdram;
```

As shown in Figure 8–10, *toplevel\_design.bdf* uses an instance of *sdram\_pll* to phase shift the SDRAM clock by –63 degrees. *toplevel\_design.bdf* also uses a subsdesign *delay\_reset\_block* to insert a delay on the *reset\_n* signal for the system module. This delay is necessary to allow the PLL output to stabilize before the SOPC Builder system begins operating.

Figure 8–11 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 8–11. 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	nR5	
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	

## 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 exactly describe the interfaces on many standard memories, such as SRAM and flash memory. In this case, I/O signals on the SOPC Builder system module 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 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.

This section describes the process of adding off-chip flash memory and SRAM to an SOPC Builder system.

## 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 *SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.
- 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.

### *Avalon-MM Tristate Bridge*

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

The tristate bridge creates address and data pins which 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  $f_{\text{MAX}}$  performance. However, in each direction, the registers add one additional cycle of latency for Avalon-MM master ports accessing memory connected to the tristate bridge. The registers do not affect the timing of the transfers from the perspective of the memory device.



For details about the Avalon-MM tristate interface, refer to the *Avalon Memory-Mapped Interface Specification*.

### *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 device(s) and the configuration of the device(s) on the board heavily influence 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 device(s) 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 device(s) on the board.



For details about features and usage, refer to the *Common Flash Interface Controller Core with Avalon Interface* 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 8–25.

### SRAM

To instantiate an interface to off-chip RAM, perform the following steps:

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 device(s) and the configuration of the device(s) 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 device(s) 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 port—This port faces the on-chip logic in the SOPC Builder system. You connect this slave port to on-chip master ports in the system.
- Avalon-MM tristate master port—This port faces the off-chip memory devices. You connect this master port to the Avalon-MM tristate slave ports 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 master ports to off-chip slave ports.

## Simulation for SRAM and Flash Memory

The SOPC Builder output for simulation depends on the type of memory component(s) 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 test bench. SOPC Builder does not automatically connect simulation models for custom memory components to the system module.
- **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 test bench.

For further details, see [“Simulation Considerations”](#) on page 8–7.

## Quartus II Project-Level Design for SRAM and Flash Memory

SOPC Builder generates a system module 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 device(s) 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 module. 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.

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

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

## Board-Level Design for SRAM and Flash Memory

Memory requirements largely dictate 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 system module 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 8–1 shows the relationship between Avalon-MM address lines and off-chip address pins for all possible Avalon-MM data widths.

**Table 8–1. Connecting the Least-Significant Avalon-MM Address Line**

Avalon-MM Address Line	Address Line on 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
...	...	...	...	...	...

### *Aligning the Most-Significant Address Bits*

The Avalon-MM address signal contains enough address lines for the largest memory on 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.

### **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.

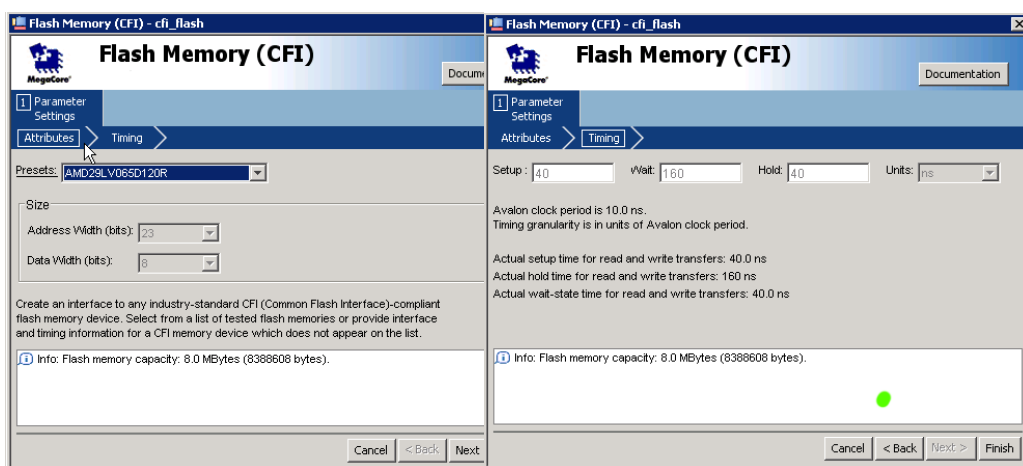
### Adding the Avalon-MM Tristate Bridge

In the **Avalon-MM Tristate Bridge** configuration wizard, check 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  $8M \times 8$ -bit, which requires 23 address bits and 8 data bits. **Figure 8–12** shows the **Flash Memory (Common Flash Interface)** configuration wizard settings for the example design.

**Figure 8–12. Flash Memory Configuration Wizard**



### Adding the SRAM Interface

The SRAM device is  $256K \times 32$ -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. **Figures 8–13** through **8–18** shows the settings required on the various component editor tabs to implement an interface to this SRAM.



Figure 8–13. SRAM Interface Component Editor HDL Files Tab

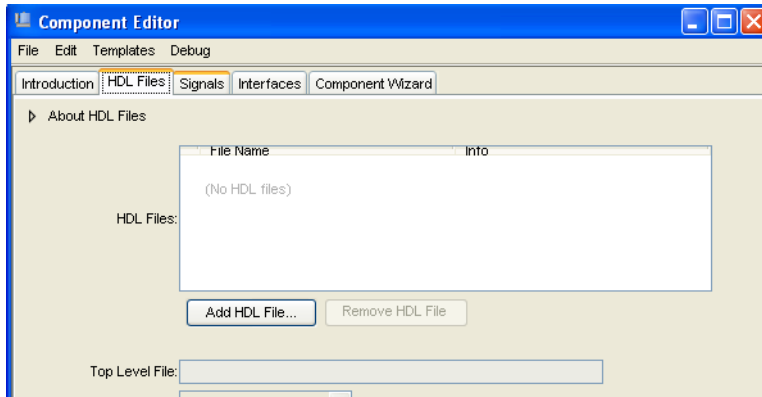
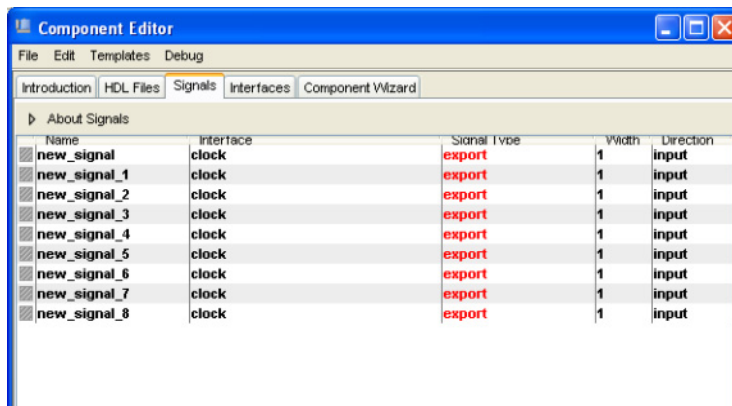


Figure 8–14. SRAM Interface Component Editor Signals Tab



**Figure 8–15. SRAM Interface Component Editor Interfaces Tab**

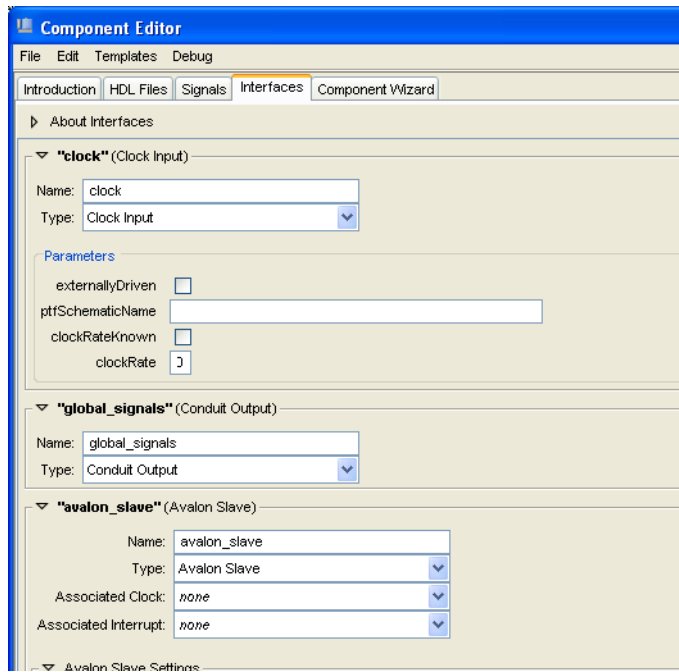
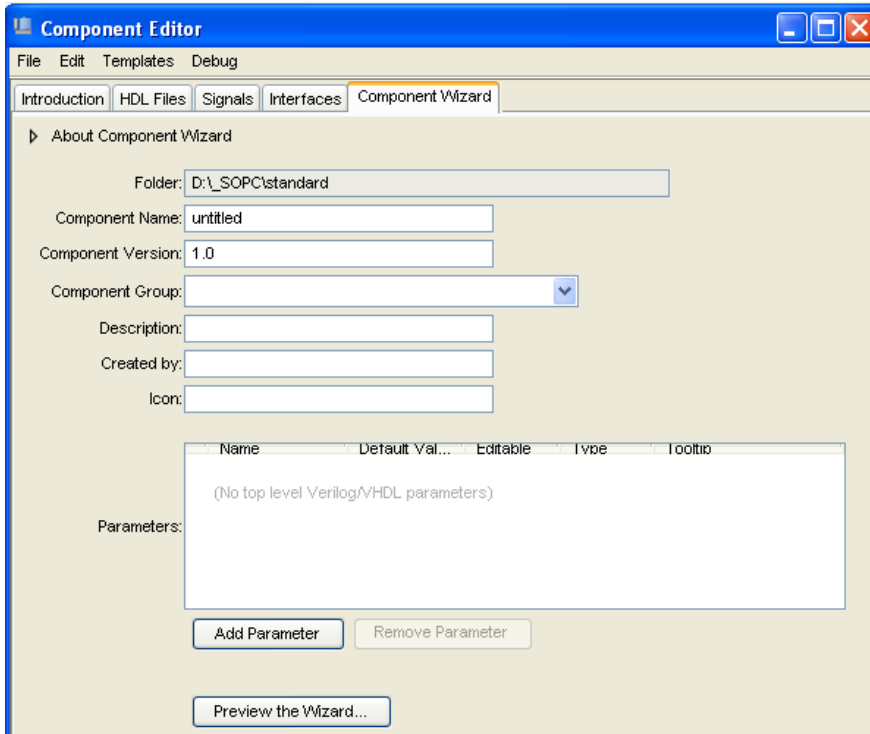


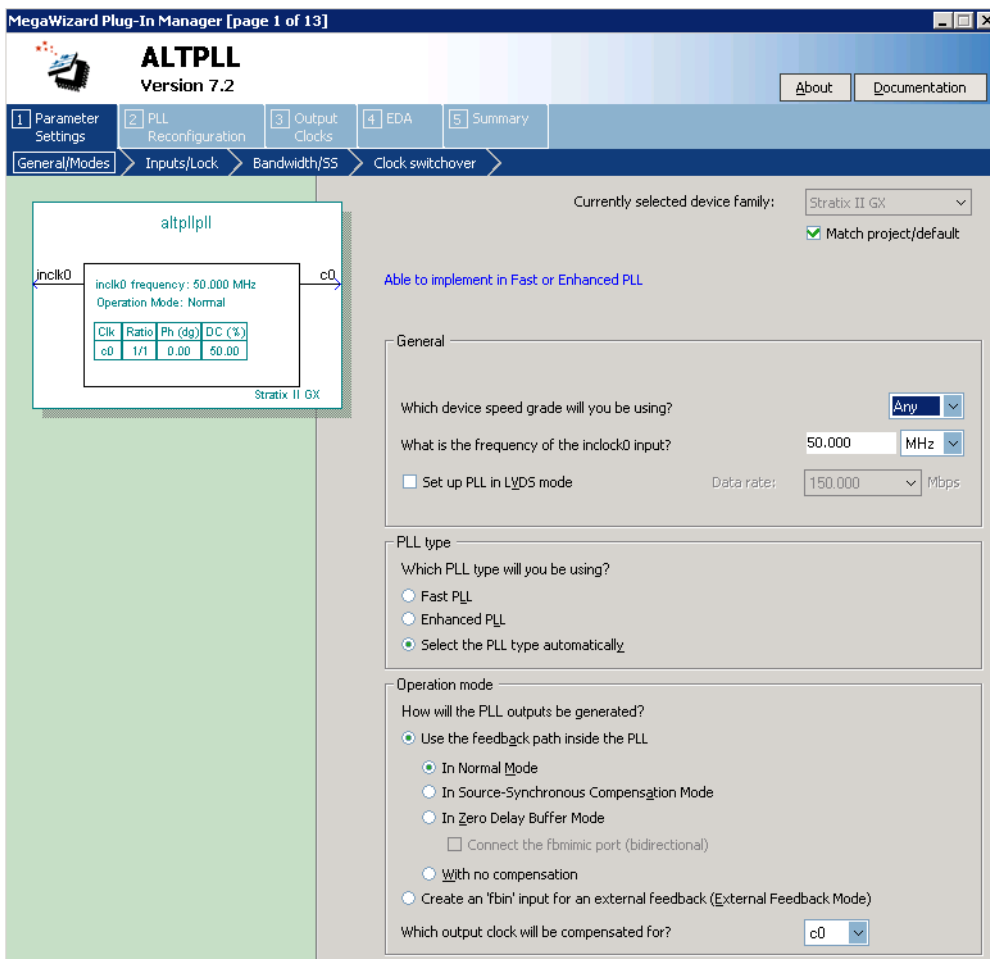
Figure 8–16. SRAM Interface Component Editor Component Wizard Tab



### Adding the PLL

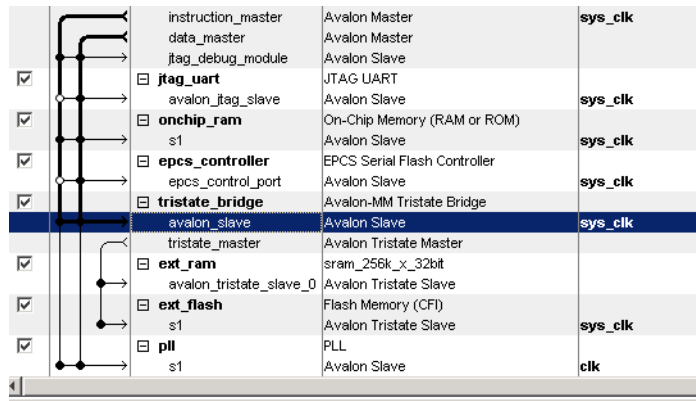
To reduce clock skew, all components in this example design connect to `sys_clk` generated by the **PLL** component. Select the **PLL** from the list of available components. To configure the **PLL**, select **Launch Altera's ALTPLL MegaWizard**. For this example design you configure `p11.c0` as a 50 MHz clock. Figure 8–17 illustrates the configuration of this component.

Figure 8–17. PLL Parameters



### SOPC Builder System Contents Tab

Figure 8–18 shows the SOPC Builder system after adding the **Tristate bridge and memory interface** components, and configuring them appropriately on the **System Contents** tab. Figure 8–18 represents the complete example design in SOPC Builder.

**Figure 8–18. SOPC Builder System with SRAM and Flash Memory**

After generating the system, the top-level system module file `sopc_memory_system.v` contains the list of I/O signals for SRAM and flash memory that must be connected to FPGA pins:

```
output          chipselect_n_to_the_ext_ram;
output          read_n_to_the_ext_ram;
output          select_n_to_the_ext_flash;
output [ 22: 0] tri_state_bridge_address;
output [  3: 0] tri_state_bridge_byteenable;
input  [ 31: 0] tri_state_bridge_data;
output          tri_state_bridge_readn;
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 8–19 shows the result of generating the system module for the complete example design.

**Figure 8–19. System Module with SDRAM and External Flash Memory**



Figure 8–20 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 8–20. 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	LVTTTL	Column I/O		
244	SRAM_BE_N[1]	PIN_F17	3	LVTTTL	Column I/O		
245	SRAM_BE_N[2]	PIN_J18	3	LVTTTL	Column I/O	RUP3	
246	SRAM_BE_N[3]	PIN_L17	3	LVTTTL	Column I/O	CLK15n	
247	SRAM_CS_N	PIN_B24	3	LVTTTL	Column I/O	DQ9T4	
248	SRAM_OE_N	PIN_B26	3	LVTTTL	Column I/O	DQ9T7	
249	SRAM_WE_N	PIN_C24	3	LVTTTL	Column I/O	DQ59T	

### Connecting FPGA Pins to Devices on the Board

Table 8–2 shows the mapping between the Avalon-MM address lines and the address pins on the SRAM and flash memory devices.

<b>Avalon-MM Address Line</b>	<b>Flash Address (8M × 8-bit Data)</b>	<b>SRAM Address (256K × 32-bit data)</b>
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	A
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:

- *Introduction to Quartus II Manual*
- *Introduction to SOPC Builder*
- *SOPC Builder Components*
- *System Interconnect Fabric for Memory-Mapped Interfaces*
- *Avalon Memory-Mapped Interface Specification*
- *Altera Configuration Handbook*
- *Nios II Flash Programmer User Guide*
- *SDRAM Controller Core*
- *altpll Megafunction User Guide*
- *Common Flash Interface Controller Core*
- *Component Editor*

## Document Revision History

Table 8–3 shows the revision history for this chapter.

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
October 2007, v7.2.0	<ul style="list-style-type: none"> <li>● Corrected <a href="#">Figure 8–19</a> 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)</li> </ul>	—
May 2007, v7.1.0	<ul style="list-style-type: none"> <li>● Chapter 8 was previously chapter 9.</li> <li>● 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.”</li> <li>● Added section on Non-Default Memory Initialization.</li> <li>● On-chip Memory size, first parameter changed from Memory Width to Data Width and widths of 256, 512 and 1024 were added.</li> <li>● Corrected figure 8-18.</li> <li>● Added links to all referenced documents.</li> <li>● Removed discussions of reference designators for components because they are no longer required by SOPC Builder.</li> <li>● Removed unnecessary screenshots.</li> </ul>	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.	—



**Table 8–3. Document Revision History**

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
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 design flow to develop a custom SOPC Builder component. The chapter describes the parts of a custom component and provides tutorial steps that guide you through the process of creating a 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 9–3.](#)
- [“Design Example: Checksum Master” on page 9–9.](#) This design example demonstrates developing a component with both Avalon® Memory-Mapped (Avalon-MM) master and slave ports.
- [“Sharing Components” on page 9–29.](#) This section shows you how to use components in other systems, or share them with other designers.

## SOPC Builder Components and the Component Editor

Typically, an SOPC Builder component is composed of the following four parts:

- HDL files that define the component’s functionality as hardware.
- `_hw.tcl` file that describes the SOPC Builder related characteristics, such as interface behaviors.
- C-language files that define the component register map and driver software that allows programs to control the component if the component is accessed by a processor using software.

The component editor guides you through the creation of a module or `hw.tcl` file to describe your component. By following the procedures described in this document, you learn to use the component editor and turn any custom logic module into an SOPC Builder component.

After your component has been created, you can instantiate it in an SOPC Builder system and make connections in the same manner as other SOPC Builder components. You can share your component with other designers to encourage design reuse.

## 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, 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.
- Quartus® II Software version 7.2 or higher—Both Quartus II Web Edition and the fully licensed version will work with the example design.
- Nios® II Embedded Design Suite (EDS) version 1.1 or higher—Both the evaluation edition and the fully licensed version will work with the example design.
- Nios development board and an Altera® USB-Blaster™ download cable (Optional)—You can use any of the following Nios development boards:
  - Stratix® III Edition
  - Stratix® II Edition
  - Stratix Edition
  - Stratix Professional Edition
  - Cyclone® III Edition
  - Cyclone II Edition
  - Cyclone™ Edition

If you do not have a development board, you can follow the hardware development steps, but you cannot download the complete system to a working board.



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](http://www.altera.com).

# Component Development Flow

This section provides an overview of the development process for custom 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 the number and type of component interfaces, whether or not Avalon MM, Avalon ST, interrupt, or the interfaces that are used.
  - c. Determine the component clocking requirements; what interfaces are synchronous to what clock inputs.
  - d. If you want a microprocessor to control the component, specify the application program interface (API) to access and control the hardware.
  - e. Specify the hardware functionality.
  - f. If you want a microprocessor to control the component, specify the register set and application program interface (API) to access and control the component.
2. For hardware development, create an HDL file that describes the hardware in either Verilog or VHDL, and test the component alone in simulation or hardware to verify correct operation.
3. SOPC Builder import.
  - a. Use the component editor to create an **hw.tcl** file that describes the component.
  - b. Instantiate the component into a simple SOPC Builder system.
  - c. Test register-level accesses to the component in hardware or simulation using a microprocessor, such as the Nios II processor.

When importing an HDL file into 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:

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

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

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

4. Software Driver Development.
  - a. Create a C header file that defines the hardware-level register map for software if the component is accessed by software.
  - b. Write the driver software.
5. Finalize the component and distribute it for design reuse.

The following sections provide more details about the hardware and software design steps.

## 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 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:

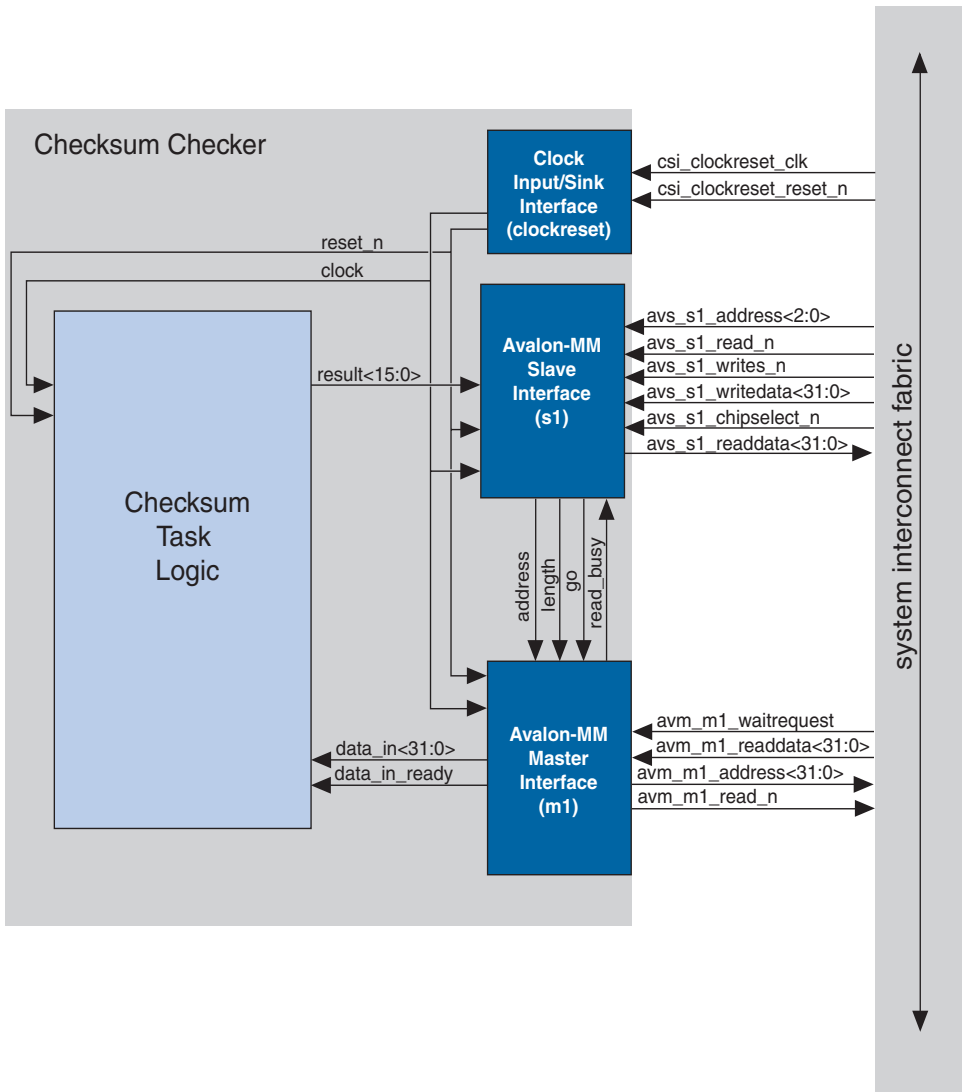
- *Task Logic*—Implements the component's fundamental function. The task logic is design dependent.
- *Interfaces*—Provide a standard way of providing data to or getting data from the components and of controlling the functioning of the components.

For interface specifications, refer to the following at [www.altera.com](http://www.altera.com):

- *Avalon Memory-Mapped Interface Specification*—Accommodate peripheral development for the SOPC environment.
- *Avalon Streaming Interface Specification*—Accommodate the development of high bandwidth low latency components for the SOPC environment.

Figure 9–1 shows the top-level blocks of a checksum component, which includes both Avalon-MM master and slave ports.

Figure 9-1. Checksum Component with Avalon-MM Master and Slave Ports



## 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 port 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 example shows the register map of the checksum component for use by the Nios II processor.

---

**Example 9–1. Example: Register Map for the Checksum Component**

```
#ifndef __ALTERA_AVALON_CHECKSUM_REGS_H__
#define __ALTERA_AVALON_CHECKSUM_REGS_H__

#include <io.h>

/* Basic address, read and write macros. */

#define IOADDR_ALTERA_AVALON_CHECKSUM_ADDR(base)
__IO_CALC_ADDRESS_NATIVE(base, 0)
#define IORD_ALTERA_AVALON_CHECKSUM_ADDR(base)          IORD(base, 0)
#define IOWR_ALTERA_AVALON_CHECKSUM_ADDR(base, data)    IOWR(base, 0, data)

#define IOADDR_ALTERA_AVALON_CHECKSUM_LENGTH(base)
__IO_CALC_ADDRESS_NATIVE(base, 1)
#define IORD_ALTERA_AVALON_CHECKSUM_LENGTH(base)        IORD(base, 1)
#define IOWR_ALTERA_AVALON_CHECKSUM_LENGTH(base, data)  IOWR(base, 1, data)

#define IOADDR_ALTERA_AVALON_CHECKSUM_CTRL(base)
__IO_CALC_ADDRESS_NATIVE(base, 2)
#define IORD_ALTERA_AVALON_CHECKSUM_CTRL(base)          IORD(base, 2)
#define IOWR_ALTERA_AVALON_CHECKSUM_CTRL(base, data)    IOWR(base, 2, data)

#define IOADDR_ALTERA_AVALON_CHECKSUM_RESULT(base)
__IO_CALC_ADDRESS_NATIVE(base, 4)
#define IORD_ALTERA_AVALON_CHECKSUM_RESULT(base)        IORD(base, 4)

#define IOADDR_ALTERA_AVALON_CHECKSUM_STATUS(base)
__IO_CALC_ADDRESS_NATIVE(base, 5)
#define IORD_ALTERA_AVALON_CHECKSUM_STATUS(base)        IORD(base, 5)

/* Masks. */

#define ALTERA_AVALON_CHECKSUM_CTRL_GO_MSK              (0x1)
#define ALTERA_AVALON_CHECKSUM_STATUS_DONE_MSK         (0x2)
#define ALTERA_AVALON_CHECKSUM_LENGTH_MSK              (0xFFFF)
#define ALTERA_AVALON_CHECKSUM_RESULT_MSK              (0xFFFF)

/* Offsets. */

#define ALTERA_AVALON_CHECKSUM_CTRL_GO_OFST            (0)
#define ALTERA_AVALON_CHECKSUM_STATUS_BSY_OFST        (0)
#define ALTERA_AVALON_CHECKSUM_STATUS_DONE_OFST       (1)

#endif /* __ALTERA_AVALON_CHECKSUM_REGS_H__ */
```

---

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, it is instructive to look at the software files provided for other ready-made components. The Nios II EDS provides many components you can use as reference. See 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 *Nios II Software Developer's Handbook*.

## Verifying the Component

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

### *Unit Verification*

To test the task logic block alone, you use your preferred verification method(s), such as HDL simulation tools.

After you package the HDL files into a component using the component editor, the Nios II EDS offers an easy-to-use method to simulate read and write transactions to the component. Using the Nios II processor's robust simulation environment, you can write C code for the Nios II processor that initiates read and write transfers to your component. You can verify the results either on the ModelSim simulator or on hardware, such as a Nios development board.



For more information, refer to *AN 351: Simulating Nios II Embedded Processor Designs*.

### *System-Level Verification*

After you package an **hw.tcl** file with the component editor, you can instantiate the component in a system, and verify the functionality of the overall system module.

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.

## Design Example: Checksum Master

This section uses a **checksum master** design example to demonstrate the steps to create a component and instantiate it in a system. This component includes both Avalon-MM master and slave ports.

In this section, you will perform the following steps:

1. Install the design files.
2. Review the example design specifications.
3. Create an SOPC Builder component.
4. Instantiate the component in an SOPC system.
5. Compile the hardware design in the Quartus II software, and download the design to a target board.
6. Exercise the hardware using the Nios II processor.

### Install the Design Files

Before you proceed, you must install the Nios II development tools and download the **checksum master** example design from the Altera website. The hardware design used in this chapter is based on the **standard** hardware example design included with the Nios II EDS.

Perform the following steps to set up the design environment:

1. On your host computer file system, locate the following directory:

```
<Nios II EDS install path>/examples/<verilog or vhdl>/<board  
version>/standard
```

Each development board has a VHDL and Verilog HDL version of the design. You can use either of these design examples. [Table 9–1](#) shows the names of the directories for each Nios development board.

<b>Nios Development Board</b>	<b>Design Directory</b>
Stratix III Edition	niosII_stratixIII_3sl150
Stratix II Edition	niosII_stratixII_2s60_ROHS, niosII_stratixII_2s60, niosII_stratixII_2s60ES
Stratix Edition	niosII_stratix_1s10, niosII_stratix_1s40
Stratix Professional Edition	niosII_stratix_1s40
Cyclone III Edition	niosII_cycloneIII_3c120, niosII_cycloneIII_3c25
Cyclone II Edition	niosII_cycloneII_2c35
Cyclone Edition	niosII_cyclone_1c20

- Copy the **standard** directory to a new location. By copying the design files, you avoid corrupting the original design and avoid issues with file permissions. This document refers to the newly-created directory as the *<Quartus II project>* directory.
- Copy the file **altera\_avalon\_checksum.zip** to the *<Quartus II project>* directory and unzip it. The design and test files listed in [Table 9–2](#) are added to *<Quartus II project>/altera\_avalon\_checksum* directory.

## Review the Example Design Specifications

This section discusses the design specifications for the provided checksum example design, giving details on each of the following topics:

- Checksum Design Files
- Functional Specification
- Master Task Logic
- Register File
- Avalon-MM Master Interface
- Avalon-MM Slave Interface
- Software API

## Checksum Design Files

Table 9–2 lists the contents provided in the `altera_avalon_checksum` directory.

File Name	Description
<code>/altera_avalon_checksum</code>	Contains all the HDL and software files for the component. All the HDL files must be in the same directory and be consistent in name with the <code>hw.tcl</code> file. (1)
<code>altera_avalon_checksum.v</code>	The top-level HDL file instantiates the task logic, Avalon-MM master and slave interfaces and the register files.
<code>checksum_task_logic.v</code>	This Verilog HDL file contains the core functionality of the checksum component.
<code>read_master.v</code>	This file contains the logic for the Avalon-MM read master interface.
<code>s1_slave.v</code>	This file contains logic for reading and writing to the checksum registers
<code>altera_avalon_checksum_sw.tcl</code>	This is the checksum software driver configuration file for the Nios II command line flow.
<code>/inc</code>	This sub-directory includes header files defining the low-level hardware interface.
<code>altera_avalon_checksum_regs.h</code>	This file defines macros to access registers in the checksum component.
<code>/test_software</code>	This sub-directory includes an example program to test the component hardware and software.
<code>test_checksum.c</code>	The test program initializes and array of data for the checksum component to read and compute the checksum.

Note to Table 9–2:

- (1) The component editor creates the `altera_avalon_checksum_hw.tcl` file and stores it in the `altera_avalon_checksum` directory.

## Master Task Logic

The **checksum master** reads a programmable number of 16-bit values to calculate a checksum. The `status` register sets its `DONE` bit when the checksum master completes. Software polls the `DONE` bit to determine when the calculation is complete.

### Register File

The register file provides access to the configuration, status, and results registers shown in Table 9–3. The design maps each register to a unique offset in the Avalon-MM slave port address space. The registers are read, write, or read only.

Register Name	Offset	Access	Description
Address	0x00	Read/Write	32-bit start address for checksum calculations.
Length	0x04 + 4	Read/Write	16-bit byte count for the checksum calculation.
Control	0x08 + 8	Read/Write	Bits [7:1] are reserved. Bit[0] is the GO bit.
Reserved	0x0C + 12	—	—
Result	0x10 + 16	Read	16-bit result of the checksum calculation.
Status	0x14 + 20	Read	Bits [7:2] are reserved. Bit[1:0] are DONE and BUSY.
Reserved	0x18	—	—
Reserved	0x1C	—	—

Table 9–4 shows the layout of the bits and fields of these registers.

Offset	31	16	15	1	0
0x00	address				
0x04	reserved		length		
0x08	reserved				GO
0x10	reserved		result		
0x14	reserved			DONE	BUSY

### Avalon-MM Clock Interface

The **checksum** component includes an Avalon-MM clock interface to bring in a system clock and reset into the checksum component as shown in Figure 9–1. The clock interface will be connected to each Avalon-MM master and slave interface in the **Interface** tab.

Table 9–5 lists the clock interface signals that comprise the Avalon-MM master port.

Signal Name in HDL	Avalon-MM Signal Type	Width	Dir	Notes
csi_clockreset_clk	clk	1	In	Synchronization clock for the component. All signals are synchronous to clk.
csi_clockreset_reset_n	reset_n	1	In	Resets the entire Avalon-MM system.

### *Avalon-MM Master Interface*

The **checksum master** component includes an Avalon-MM master port that reads from memory. The component's Avalon-MM master port has the following characteristics:

- It is synchronous to the Avalon-MM master clock interface.
- It initiates master transfers to the system interconnect fabric.

Table 9–6 lists the signals that comprise the Avalon-MM clock port.

Signal Name in HDL	Avalon-MM Signal Type	Width	Dir	Notes
avm_m1_address	address	32	Out	Byte address aligned on word boundary.
avm_m1_byteenable	byteenable	4	Out	Enables specific byte lanes on ports greater than 8 bits.
avm_m1_read_n	read_n	1	Out	Read request signal.
avm_m1_readdata	readdata	32	In	Uni-directional data.
avm_m1_waitrequest	waitrequest	1	In	Forces master port to wait until the system interconnect fabric is ready to proceed with the transfer.

### *Avalon-MM Slave Interface*

The Avalon-MM slave port handles simple read and write transfers to the registers. The slave port has the following characteristics:

- Synchronous to the Avalon-MM clock interface.

- Readable and writable.
- Zero wait states for writing and one wait state for reading.
- No setup or hold restrictions for reading and writing.
- Uses native address alignment, because the slave port is connected to registers rather than a memory device.

**Table 9–7. Table of Checksum Avalon-MM Slave Port Signal Names and Avalon Signal Types**

Signal Name in HDL	Avalon-MM Signal Type	Width	Dir	Notes
avs_s1_address	address	3	In	A byte address.
avs_s1_read_n	read_n	1	In	Read request input.
avs_s1_write_n	write_n	1	In	Write request input.
avs_s1_chipselect_n	chipselect	1	In	Chip-select to slave port. Slave port ignores all other signals unless it is selected.
avs_s1_readdata	readdata	32	Out	Uni-directional read data
avs_s1_writedata	writedata	32	In	Uni-directional write data

### Software API

The `altera_avalon_checksum_regs.h` file has been provided to include macros to read and write the checksum slave registers.

## Create an SOPC Builder component

In this section you specify the hardware interfaces to the component, and define the behavior of each interface signal.

### Open the Quartus II Project and Start the Component Editor

To open SOPC Builder from the Quartus II software, perform the following steps:

1. Start the Quartus II software.
2. Open the project `standard.qpf` in the `<Quartus II project>` directory.
3. On the Tools menu, click **SOPC Builder**. SOPC Builder appears, displaying a ready-made example design containing a Nios II processor and several components.
4. On the File menu, click **New Component**. The component editor appears, displaying the **Introduction** tab.



### *HDL Files Tab*

In this section you associate the component's top-level HDL file with the component's hardware Tcl file using the **HDL files** tab. Perform the following steps:

1. Click the **HDL Files** tab.
2. Click **Add HDL File**.
3. Browse to the `<Quartus II project>/altera_avalon_checksum` directory and select the top level HDL file **altera\_avalon\_checksum.v** and click **Open**.



The first file you add to the component editor must be the top-level HDL file of your design.

4. Click **OK** when a message indicated analysis is complete.
5. You can now add lower-level design files. Click **Add HDL File** and add the **checksum\_task\_logic.v**, **read\_master.v**, and **sl\_slave.v** files to the component list.
6. Select the top level module of your component by clicking in the **Top Level Module** list and selecting **altera\_avalon\_checksum**.
7. If you plan to simulate your component, click **Add Simulation File** to add all of the files required for simulation.

The component editor now displays error messages. You are instructed to fix them in later steps.

### *Signals Tab*

For every I/O signal present on the top-level HDL module, you must map the signal name to a valid signal type using the **Signals** tab. If the signal name includes a recognized signal type (such as `write` or `address`), the component editor guesses the signal's type. If the component editor cannot determine the signal type, it assigns the type `export`.

This design uses the automatic type and interface recognition feature of the component editor to quickly allow the component editor to assign the component signals to the appropriate interface and signal type. To change the type assigned, click at the right edge of the **Signal Type** column for the signal in question. A pull-down menu provides other choices.



For more information on the automatic type and interface recognition feature see the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

This design includes three interfaces: clock (clockreset), slave (s1), and master (m1) as illustrated in [Figure 9-2](#). The signal types and polarities are derived from the signal names.

Figure 9–2. The Signals Tab

The screenshot shows the 'Component Editor' window with the 'Signals' tab selected. The window title is 'Component Editor' and it has a menu bar with 'File' and 'Templates'. Below the menu bar are tabs for 'Introduction', 'HDL Files', 'Signals', 'Interfaces', and 'Component Wizard'. The 'Signals' tab is active, showing a section titled 'About Signals' with a table of signals.

Name	Interface	Signal Type	Width	Direction
csi_clockreset_clk	clockreset	clk	1	input
csi_clockreset_rese...	clockreset	reset_n	1	input
avs_s1_address	s1	address	3	input
avs_s1_chipselect_n	s1	chipselect_n	1	input
avs_s1_read_n	s1	read_n	1	input
avs_s1_write_n	s1	write_n	1	input
avs_s1_writedata	s1	writedata	32	input
avs_s1_readdata	s1	readdata	32	output
avm_m1_address	m1	address	32	output
avm_m1_byteenable	m1	byteenable	4	output
avm_m1_read_n	m1	read_n	1	output
avm_m1_readdata	m1	readdata	32	input
avm_m1_waitrequest	m1	waitrequest	1	input

Below the table are two buttons: 'Add Signal' and 'Remove Signal'. At the bottom of the window, there is an information bar that says 'Info: No errors or warnings.' and a footer with buttons for 'Help', 'Prev', 'Next', and 'Finish...'.

### Interfaces Tab

After assigning signals to interfaces, the **Interfaces** tab allows you to further configure the properties of all interfaces on the component.

Perform the following steps to configure the Avalon slave port:

1. Click the **Interfaces** tab. The component editor displays the Avalon-MM slave port (s1) from the previous tab.
2. Remove any unused interfaces by clicking **Remove Interfaces with No Signals**.



This removes the default provided clock and export\_0 interfaces in the component editor, as you created your own interfaces with the automatic type and interface recognition feature.

The component editor now displays the clockreset clock input interface, s1 slave interface, and the m1 master interface.

3. For the Avalon-MM slave port (s1) set the clock and reset for the slave interface by clicking on **Associated Clock** and then select **clockreset**.
4. Change the default settings for the slave port to match those given in [Table 9–8](#).

**Table 9–8. Settings for Avalon-MM Slave Port (Part 1 of 2)**

Slave Settings	Value	Description
Slave Addressing	Native	Indicates that the slave ports uses address-mapped registers.
Minimum Arbitration Shares	1	Arbitration shares modify the default round-robin arbitration scheme which provides equal access to all devices.
Can receive stderr/stdout	No	—
Interleave Bursts	No	—
Read Latency	0	—
Max. Pending Read Transactions	0	—
Slave Timing	Value	Description
Setup	0	Indicates that the slave port responds to a read or write request in a single clock cycle.
Read Wait	1	Indicates that the slave port responds to read requests one cycle after they are made (one read waitstate).

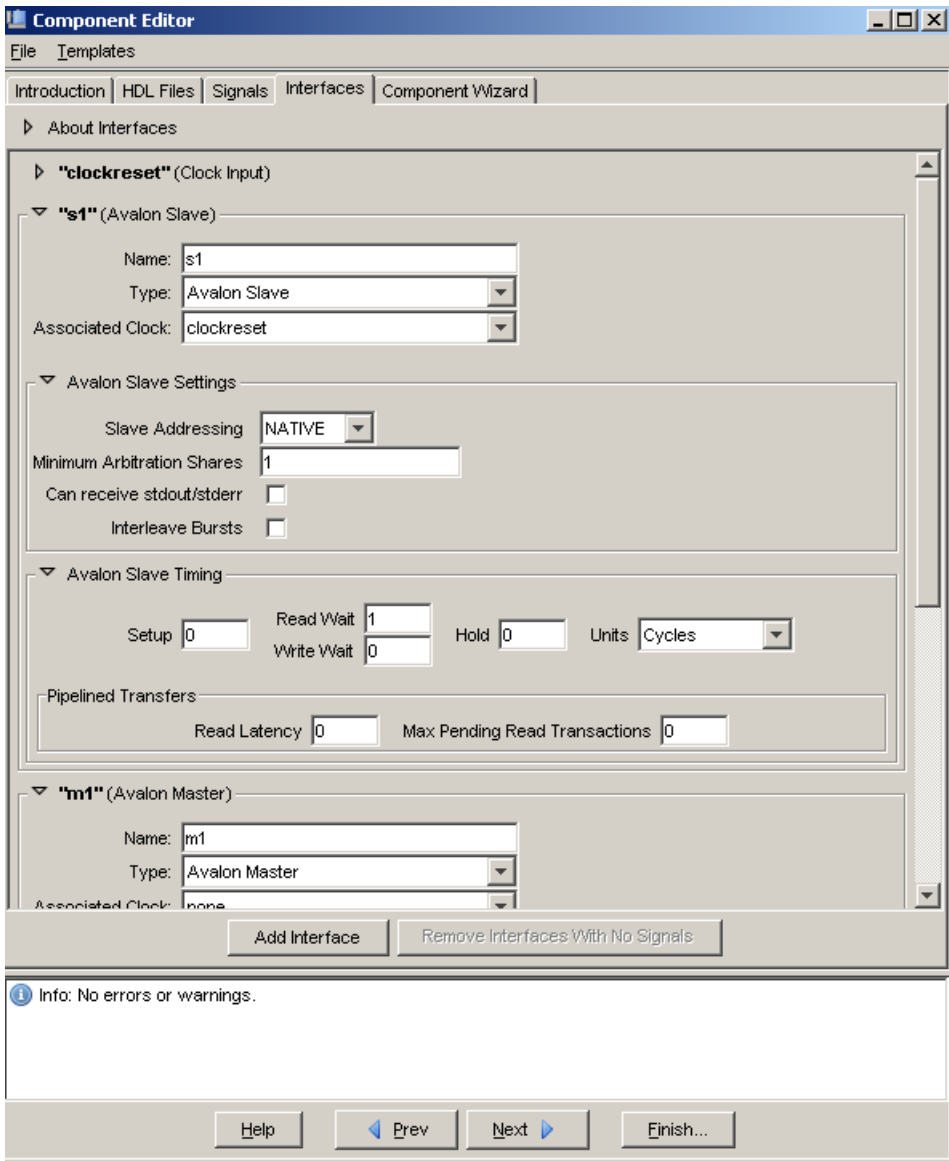
**Table 9–8. Settings for Avalon-MM Slave Port (Part 2 of 2)**

<b>Slave Settings</b>	<b>Value</b>	<b>Description</b>
Write Wait	0	Indicates that the slave port responds to write requests in a single clock cycle and does not need write waitstates.
Hold	0	Indicates that there is not a hold time requirement.

5. For the Avalon-MM master port (m1) set the clock and reset for the master interface by clicking on **Associated Clock** and then select **clockreset**.
6. Leave all other Avalon-MM master settings as the default settings, as shown in [Figure 9–4](#).

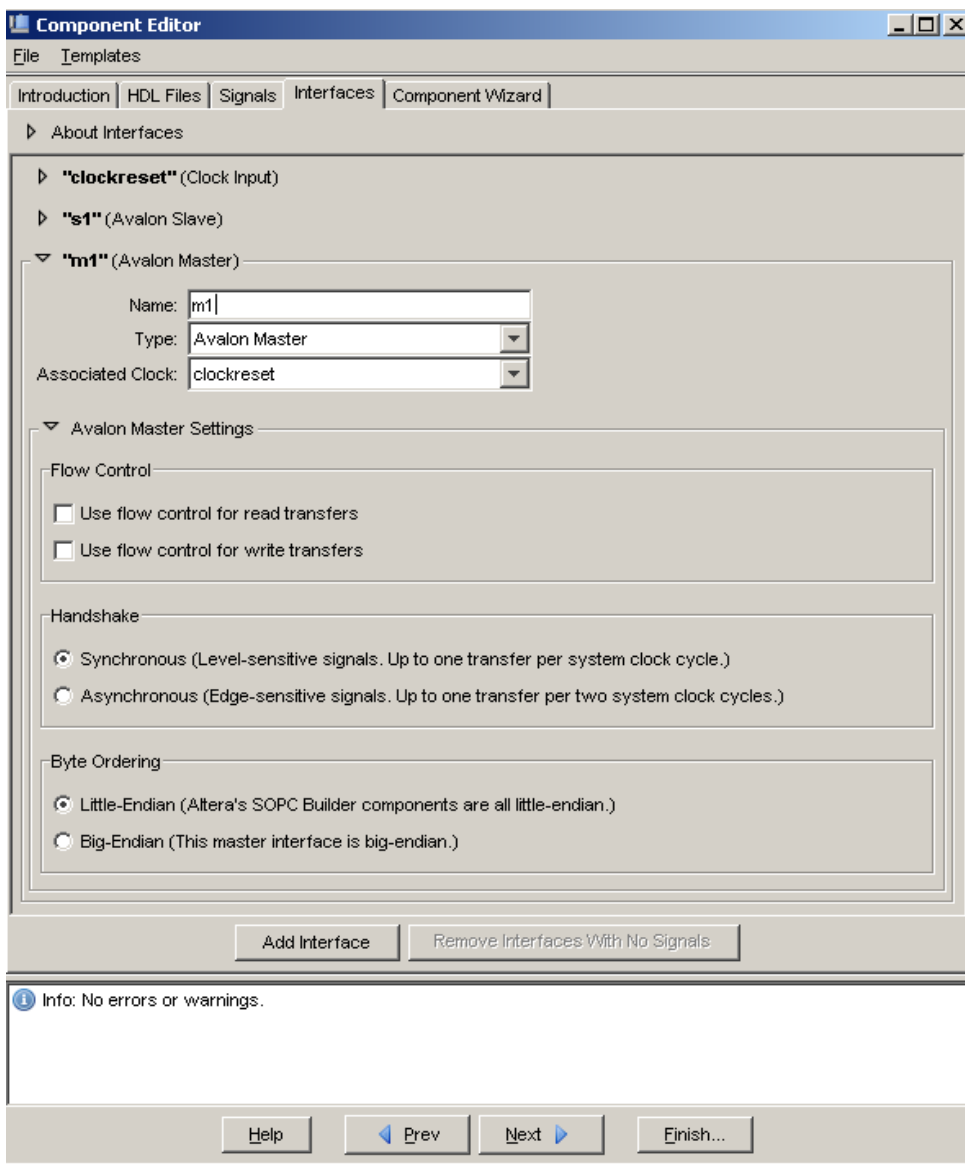
Figure 9-3 illustrates the slave settings.

Figure 9-3. Avalon-MM Slave Interfaces Settings



The Avalon-MM master port uses the default settings. [Figure 9-4](#) illustrates these settings.

Figure 9-4. Avalon-MM Masters Interfaces Settings



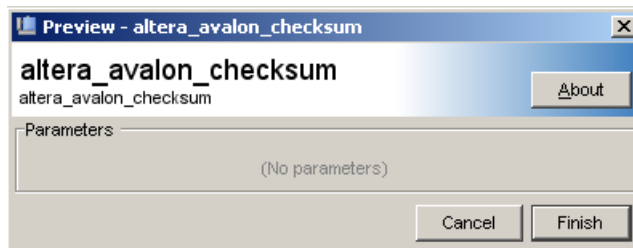


### Component Wizard Tab

The **Component Wizard** tab allows you to control how SOPC Builder presents the components to a user. Perform the following steps to configure the user presentation of the component. The component editor creates a default name for the component, based on the name of the top-level design module.

1. Click the **Component Wizard** tab.
2. For this example, do not change the default settings for **Component Name** or **Component Version**.
3. For the **Component Group** type the following: `User Logic`
4. Complete the remaining fields, such as **Description** and **Created By**.
5. Click **Preview the Wizard** to preview the component wizard as it will appear in SOPC Builder. [Figure 9-5](#) illustrates the component wizard preview.
6. Close the Preview window.

**Figure 9-5. Component Wizard**



### Save the Component

Perform the following steps to save the component and exit the component editor:

1. Click **Finish**. A message describes the file that is created for the component.
2. Click **Yes** to save the file. The component editor saves the **altera\_avalon\_checksum\_hw.tcl** file in the same directory that you stored the top-level component HDL file. The component editor closes, and you return to SOPC Builder.

3. Locate the new **checksum** component in the list of available components under the **User Logic** group. The component is added to the SOPC Builder search path. Right-clicking on a component in the list allows you to edit the component.

### Instantiate the Component in Hardware

At this point, the new component is ready to instantiate in an SOPC Builder system. The remaining steps for this design example illustrate one possible method of instantiation that includes the following general steps:

1. Add the **checksum master** to the SOPC Builder system.
2. Compile the hardware design and download to the target board.

#### *Add the checksum Master Component to the SOPC Builder System*

Perform the following steps to add a **checksum master** component to the SOPC Builder system:

1. On the SOPC Builder **System Contents** tab, select the new component **altera\_avalon\_checksum** under the **User Logic** group in the list of available components, and click **Add**. The configuration wizard for the **checksum master** component appears.
2. Click **OK**. The component **altera\_avalon\_checksum\_inst** appears in the table of active components.
3. Connect the **altera\_avalon\_checksum\_inst** m1 master port to a memory in your system.



The test program uses an on-chip memory peripheral called **onchip\_ram**. If your SOPC Builder system does not have an on-chip memory you should add an on-chip memory to the design. The test program requires that the name of the on-chip RAM and the component name used in the test program match. Connect the on-chip RAM to the Nios II data master.

4. To start generating the system, click **Generate**
5. After system generation completes successfully, exit SOPC Builder.

### *Compile the Hardware Design and Download to the Target Board*

At this point, you have created an SOPC Builder system that uses the **checksum** component. The **checksum** component adds no additional I/O signals to the SOPC Builder system top-level so you only need to compile the design in the Quartus II software.

Perform the following steps to compile the hardware design and download it to the target board.

1. On the Processing menu, click **Start Compilation** to start compiling the hardware design. The compilation begins.

If you performed all prior steps correctly, the Quartus II compilation finishes successfully after several minutes, and generates a new SRAM Object File (**.sof**) for the project.



You can only perform the remaining steps in this chapter if you have a development board.

2. Connect your host computer to the development board using an Altera download cable, such as the USB Blaster, and apply power to the board.
3. On the Tools menu, click **Programmer** to open the Quartus II Programmer.
4. Use the Programmer window to download the following FPGA configuration file to the board: *<Quartus II project>/standard.sof*.

At this point, you have completed all the steps to create a hardware design and download it to hardware.

### **Exercise the Hardware Using Nios II Software**

The **checksum master** example design is based on the Nios II processor. The example design files provide a C test program that programs the component to calculate a checksum and then polls the component to determine if it completes the calculation successfully. In this section you perform the following steps:

1. Start the Nios II IDE and create a new Nios II IDE project.
2. Build and run the C test program.
3. View the results.

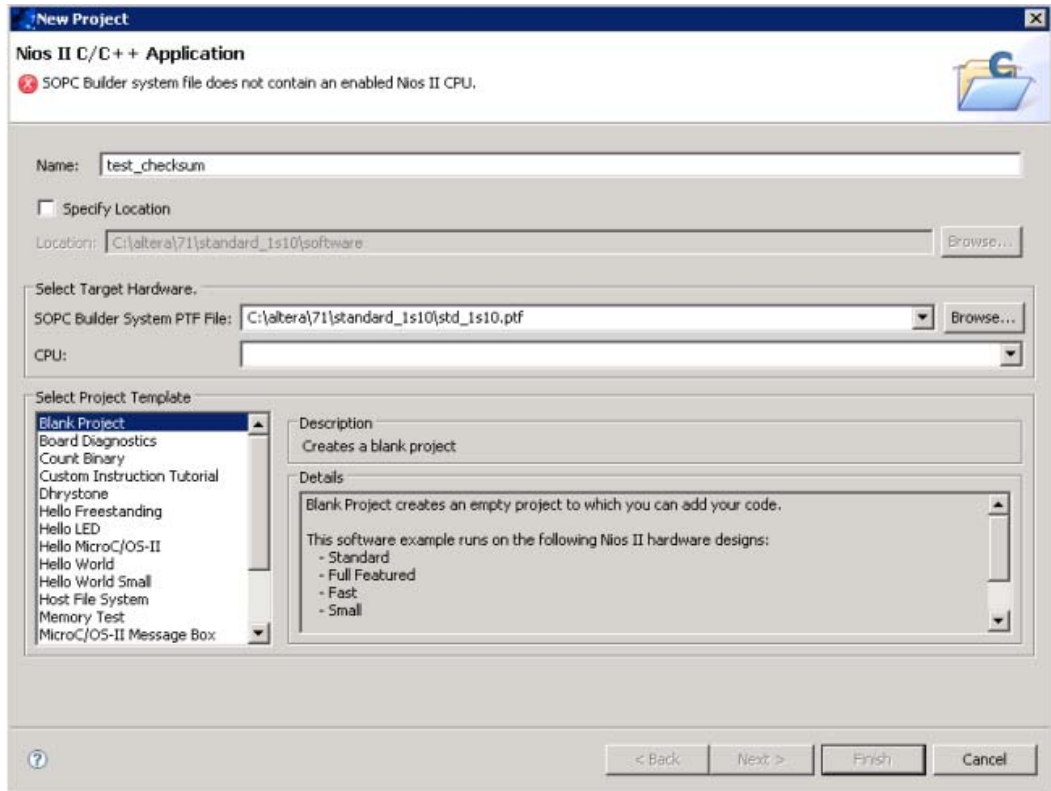
To complete this section, you must have performed all prior steps, and successfully configured the target board with the hardware design.

### *Start the Nios II IDE and Create a New IDE Project*

Perform the following steps to start the Nios II IDE and create a new IDE project:

1. Start the Nios II IDE.
2. On the Window menu, point to **Open Perspective** and click **Other**, then click **Nios II C/C++** to open the Nios II C/C++ perspective.
3. On the File menu, point to **New** and then click **C/C++ Application** to start a new project. The first page of the **New Project** wizard appears.
4. Under **Select Project Template**, select **Blank Project**.
5. In the **Name** box, type `test_checksum`.
6. Ensure that **Specify Location** is turned off so that you use the default software directory under your standard board as shown in [Figure 9-6](#).

Figure 9–6. Create a New Project



7. Click **Browse** under **Select Target Hardware**. The **Select Target Hardware** dialog box appears.
8. Browse to the *<Quartus II project>* directory.
9. Select the file **std\_<FPGA>.ptf**.
10. Click **Open** to return to the New Project wizard. The **SOPC Builder System** and the **CPU** fields are now specified.
11. Click **Finish**. After the IDE successfully creates the new project, the C/C++ Projects view contains two new projects, **test\_checksum** and **test\_checksum\_syslib**.

### *Compile the Software Project and Run on the Target Board*

In this section you compile the C test program provided with the checksum design files, and then download it to the target board.

First, perform the following steps to associate the source files with the new C/C++ project:

1. Copy **test\_checksum.c** from `<Quartus II project>/altera_avalon_checksum/test_software` to the `<Quartus II project>/software/test_checksum` directory.
2. In the Nios II IDE C/C++ Projects view, right-click **test\_checksum** and click **Refresh**, directing the IDE to recognize the new file in the project directory.

The project is now ready to compile and run. Perform the following steps:

1. Right-click the project **test\_checksum** in the Nios II C/C++ Projects view and click **Build Project** to compile the program. The first time you build the project, it can take a few minutes for the compilation to finish.
2. After compilation completes, select **test\_checksum** in the C/C++ Projects view.
3. On the Run menu, click **Run**. The **Run** dialog box appears.
4. Select **Nios II Hardware**, and click **New**. A new run/debug configuration named **test\_checksum Nios II HW configuration** appears.
5. If the **Run** button (in the bottom right of the **Run** dialog box) is disabled, perform the following steps:
  - a. Click the **Target Connection** tab.
  - b. Click **Refresh** next to the **JTAG cable** list.
  - c. In the **JTAG cable** list, select the download cable you want to use.
  - d. Click **Refresh** next to the **JTAG device** list.
6. Click **Run**.

- View the results: The **Console** view in the IDE displays messages similar to the following: 0x5a5a.

You have finished all steps for the **checksum** design example.

## Sharing Components

When you create a component, component editor by default saves the (**\_hw.tcl**) in the same directory as the top-level HDL file. Where appropriate, files referenced by the **\_hw.tcl** file all use relative paths so that files can easily be moved and copied together. To promote design reuse, you can use the component in different projects, and you can share your component with other designers.

Perform the following steps to share a component:

- In your computer's file system, move the component directory to a central location, outside any particular Quartus II project's directory. For example, you could create a directory **c:\my\_component\_library** to store your custom components.



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 library path.



SOPC Builder will find your components if you place your components in the **projectdir\ip** directory. Altera recommends that you do so.

- On the Quartus II Assignments menu, click **Settings**. The **Settings** dialog box appears.
- In the **Categories** list, click **Libraries**.
- Under **Global libraries**, add the path to the enclosing directory of the component directory. For example, for a component directory **c:\my\_component\_library\checksum\_master\**, add the path **c:\my\_component\_library**.



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. However, in the 7.2 version of the Quartus II software, this specifies component directories, and not library directories.

To use the newly created component in another SOPC Builder system, you must perform 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 in the **project 2** project, simply copy the Tcl File (.tcl) and the reference files to **project2/ip/checksum**, and they will be found automatically.
- Alternatively, you can place the Tcl File (.tcl) and related files elsewhere in a component library, such as **L:/components/checksum/**, and add the library location to see the search path via **SOPC Builder/Tools/Options/IP Search Path**.



## Referenced Documents

This chapter references the following documents:

- *Introduction to SOPC Builder*
- *SOPC Builder Components*
- *The Component Editor*
- *Avalon Memory Mapped Interface Specification*
- *Nios II Software Developer's Handbook*
- *AN 351: Simulating Nios II Embedded Processor Designs*

## Document Revision History

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

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
October 2007, v7.2.1	Updated instructions on how to develop components to match updated GUI.	—
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 10, Avalon Memory-Mapped Bridges](#)
- [Chapter 11, 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 the concept of Avalon® Memory-Mapped (Avalon-MM) bridges, and describes the Avalon-MM bridge components provided by Altera® for use in SOPC Builder systems.

A bridge, in the context of SOPC Builder, is a component that acts as part of the system interconnect fabric. Bridges are not end-points for data, but rather affect the way data is transported between other components. By manually inserting Avalon-MM bridges between Avalon-MM master and slave ports in a system, you can control system topology, which in turn affects the interconnect that SOPC Builder generates. Manual control of the interconnect can result in higher performance and/or lower logic utilization.

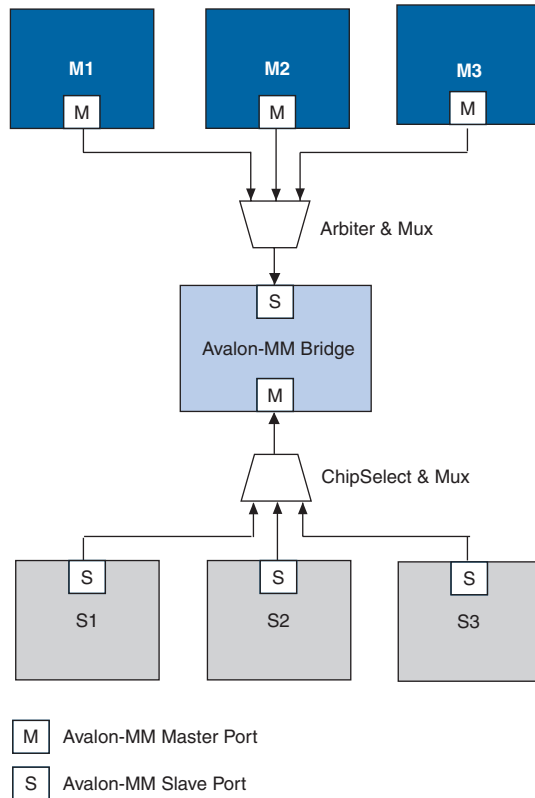
Altera provides the Avalon-MM bridge, which is described in this chapter:

- [“Avalon-MM Pipeline Bridge” on page 10–9](#)


### Structure of a Bridge

A bridge has one Avalon-MM slave port and one Avalon-MM master port, as shown in [Figure 10–1](#). In an SOPC Builder system, one or more master ports connect to the bridge’s slave port to control the bridge. The bridge’s master port connects, in turn, to one or more slave ports. You configure the master-slave pairs manually with the SOPC Builder GUI. In [Figure 10–1](#), all three masters have a logical connection to all three slaves, although physically each master only connects to the bridge.

**Figure 10–1. Example of an Avalon-MM Bridge in an SOPC Builder System**



A bridge issues transfers on its master port in the same order in which they were received. Transfers initiated to the bridge’s slave port propagate to the master port in the same order in which they were initiated on the slave port.

 If you use either the Avalon-MM pipeline bridge or the Avalon-MM clock-crossing bridge in your system discussed in the SOPC Builder chapter, automatic pipelining feature is disabled.



For details on the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification*.

## Reasons for Using a Bridge

Reasons you might use an Avalon-MM bridge include:

- Increase the  $f_{MAX}$  of your system
- Control system topology
- Specify separate clock domains for master-slave pairs

If there are no bridges between master-slave pairs, SOPC Builder generates system interconnect fabric with maximum parallelism so that all masters can drive transactions to and from all slaves concurrently as long as each master is trying to access a different slave. This default behavior incurs the cost of additional arbiters and multiplexers decreasing the  $f_{MAX}$  of the system. For high performance 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 10–2 and Figure 10–3 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 10–2. 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	0xc02002800
<input checked="" type="checkbox"/>		cpu2	Nios II Processor	
		instruction_master	Avalon Master	
		data_master	Avalon Master	IRQ 0
		jtag_debug_module	Avalon Slave	0xc00000800
<input checked="" type="checkbox"/>		cpu3	Nios II Processor	
		instruction_master	Avalon Master	
		data_master	Avalon Master	IRQ 0
		jtag_debug_module	Avalon Slave	0xc00001000
<input checked="" type="checkbox"/>		DDR_SDRAM_controller	DDR SDRAM High Performance Control...	
		s1	Avalon Slave	0xc01000000
<input checked="" type="checkbox"/>		message_buffer_RAM	On-Chip Memory (RAM or ROM)	
	s1	Avalon Slave	0xc02001000	
<input checked="" type="checkbox"/>	message_buffer_mutex	Mutex		
	s1	Avalon Slave	0xc02003000	
<input checked="" type="checkbox"/>	external_SSRAM_bus	Avalon-MM Tristate Bridge		
	avalon_slave	Avalon Slave	0xc00000000	
	tristate_master	Avalon Tristate Master		
<input checked="" type="checkbox"/>	external_SSRAM	Cypress CY7C1380C SSRAM		
	s1	Avalon Tristate Slave	0xcfffffff	

Figure 10–3 illustrates the default system interconnect fabric that SOPC Builder would create for the system in Figure 10–2.

Figure 10–3. Example System without Bridges - System Interconnect View

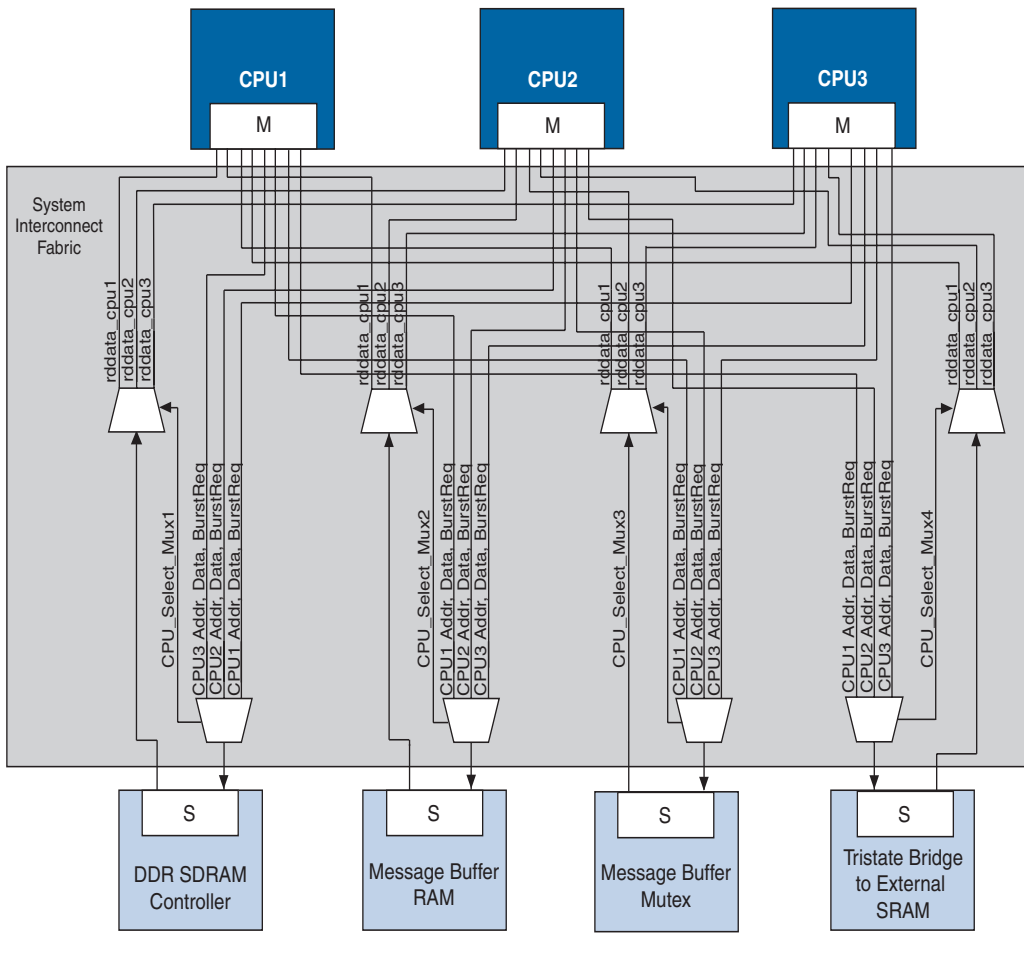


Figure 10–4 and Figure 10–5 show how you can improve the logic utilization of the system interconnect fabric by inserting bridges. If the DDR SDRAM controller can run at 166 MHz and the CPUs accessing it can run at 120 MHz, inserting an Avalon-MM clock-crossing bridge between the CPUs and the DDR SDRAM 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 10–4 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:

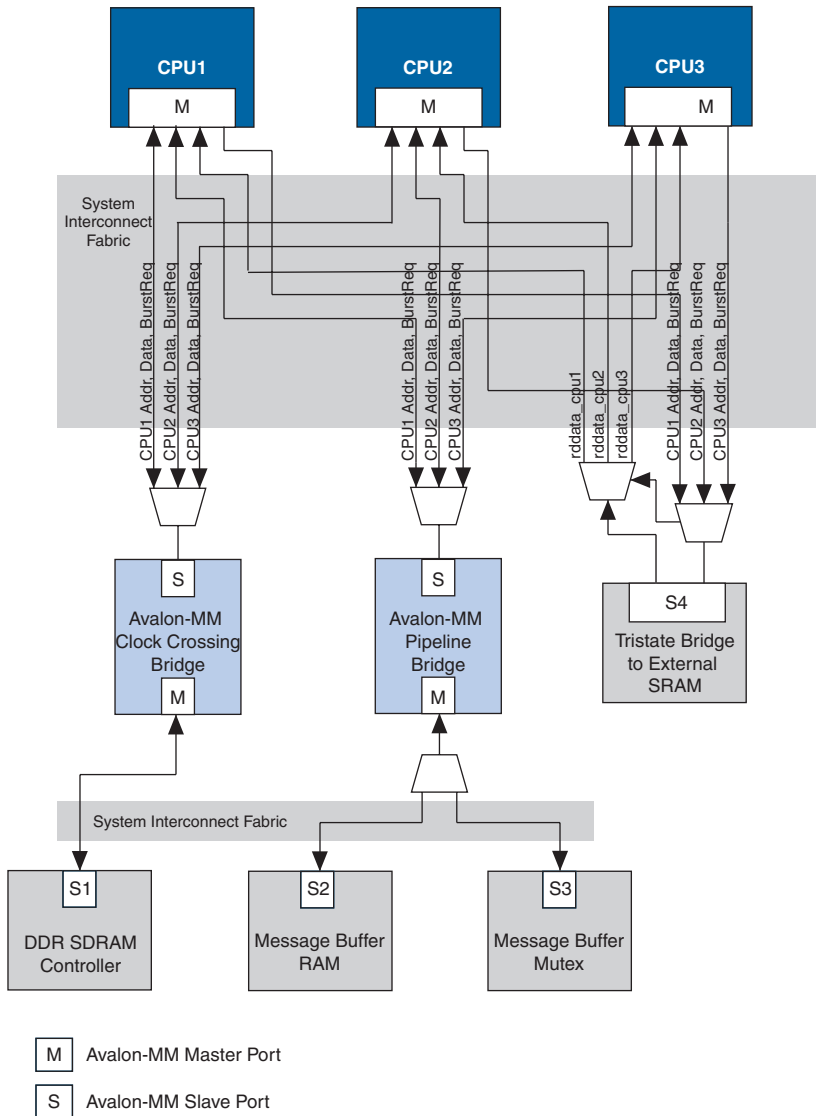
- Eliminates separate arbiter logic for the message buffer RAM and message buffer mutex, which reduces logic utilization and propagation delay, thus increasing the  $f_{MAX}$ .
- Reduces the overall size and complexity of the system interconnect fabric.

Figure 10–4. Example SOPC System with Bridges - SOPC Builder View

Use	Connections	Module Name	Description	Base	End	IRQ
<input checked="" type="checkbox"/>		<input type="checkbox"/> <b>cpu1</b>	Nios II Processor			
		instruction_master	Avalon Master			
		data_master	Avalon Master			
		jtag_debug_module	Avalon Slave	0x03210000	0x032107ff	IRQ 0, IRQ 31
<input checked="" type="checkbox"/>		<input type="checkbox"/> <b>cpu2</b>	Nios II Processor			
		instruction_master	Avalon Master			
		data_master	Avalon Master			
		jtag_debug_module	Avalon Slave	0x03210800	0x03210fff	IRQ 0, IRQ 31
<input checked="" type="checkbox"/>		<input type="checkbox"/> <b>cpu3</b>	Nios II Processor			
		instruction_master	Avalon Master			
		data_master	Avalon Master			
		jtag_debug_module	Avalon Slave	0x03211000	0x032117ff	IRQ 0, IRQ 31
<input checked="" type="checkbox"/>	<input type="checkbox"/> <b>bridge</b>	Avalon-MM Clock Crossing Bridge				
	s1	Avalon Slave	0x00040000	0x0003ffff		
	m1	Avalon Master				
<input checked="" type="checkbox"/>	<input type="checkbox"/> <b>pipeline_bridge</b>	Avalon-MM Pipeline Bridge				
	s1	Avalon Slave	0x00000002	0x04000001		
	m1	Avalon Master				
<input checked="" type="checkbox"/>	<input type="checkbox"/> <b>message_buffer_ram</b>	On-Chip Memory (RAM or ROM)				
	s1	Avalon Slave	0x03212000	0x032123ff		
<input checked="" type="checkbox"/>	<input type="checkbox"/> <b>message_buffer_mu...</b>	Mutex				
	s1	Avalon Slave	0x032124f8	0x032124ff		
<input checked="" type="checkbox"/>	<input type="checkbox"/> <b>ext_ssram_bus</b>	Avalon-MM Tristate Bridge				
	avalon_slave	Avalon Slave	0x00000000	0x00000000		
	tristate_master	Avalon Tristate Master				
<input checked="" type="checkbox"/>	<input type="checkbox"/> <b>ext_ssram</b>	Cypress CY7C1380C SSRAM				
	s1	Avalon Tristate Slave	0x03000000	0x031fffff		

Figure 10–5 shows the system interconnect fabric that SOPC Builder would create for the system in Figure 10–4. Figure 10–5 is the same system that is pictured in Figure 10–3 except that it includes bridges to control system topology.

Figure 10–5. Example System with a Bridge



## Address Mapping for Systems with Avalon-MM Bridges

An Avalon-MM bridge has an address span and range which are defined as follows:

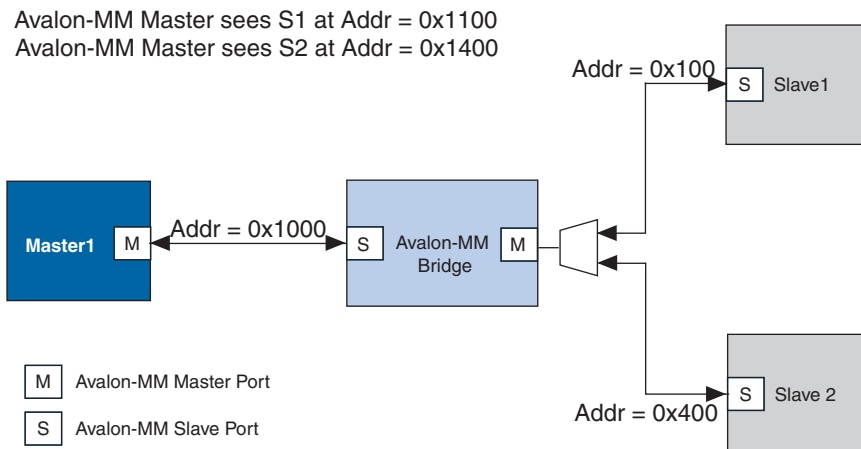
- The address *span* of an Avalon-MM bridge is the smallest power-of-two size that encompasses all of its slave's ranges.
- The address *range* of an Avalon-MM bridge is a numerical range from its base address to its base address plus its (span -1)

$$(1) \quad \text{range} = [\text{base\_address} .. (\text{base\_address} + (\text{span} - 1));$$

SOPC Builder follows several rules in constructing an address map for a system with Avalon-MM bridges:

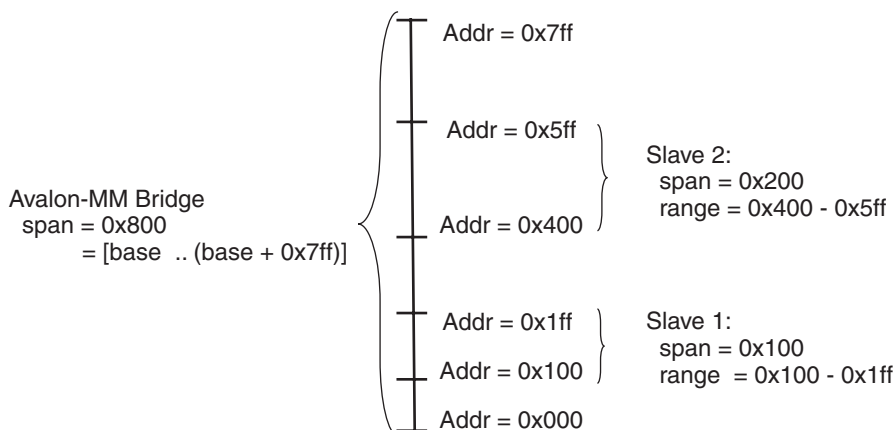
1. The address span of each Avalon-MM slave is rounded up to the nearest power of two.
2. Each Avalon-MM slave connected to a bridge is assigned an address relative to the base address of the bridge. This address must be a multiple of its span. (See [Figure 10-6](#).)

**Figure 10-6. Avalon-MM Master and Slave Addresses**



3. In the example shown in [Figure 10-6](#), if the address span of Slave 1 is 0x100 and the address span of Slave 2 is 0x200, [Figure 10-7](#) illustrates the address span of the Avalon-MM bridge.

**Figure 10–7. The Address Span of an Avalon-MM Bridge**



### *Tools for Visualizing the Address Map*

The Base Address column of SOPC Builder 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 the **Address Map** button on the **System Components** 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. Tristate bridges are also used 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 they connect to. All tristate bridges in a system have an address of 0x00000000 as [Figure 10–8](#) illustrates.



For more information about the Avalon-MM tristate bridge, refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.

Figure 10–8. SOPC Builder System with Two Tristate Bridges

Use	Connections	Module Name	Description	Base	End
<input checked="" type="checkbox"/>		<input type="checkbox"/> ext_flash	Flash memory (CFI)		
<input checked="" type="checkbox"/>		s1	Avalon Tristate Slave	0x00000000	0x00ffffff
<input checked="" type="checkbox"/>		<input type="checkbox"/> pll	PLL		
<input checked="" type="checkbox"/>		s1	Avalon Slave	0x01211000	0x0121101f
<input checked="" type="checkbox"/>		<input type="checkbox"/> ext_ssram	Cypress CY7C1360C SSRAM		
<input checked="" type="checkbox"/>		s1	Avalon Tristate Slave	0x01000000	0x011fffff
<input checked="" type="checkbox"/>		<input type="checkbox"/> lan91c111	LAN91C111 Interface		
<input checked="" type="checkbox"/>		s1	Avalon Tristate Slave	0x01200000	0x0120ffff
<input checked="" type="checkbox"/>		<input type="checkbox"/> sys_clk_timer	Interval Timer		
<input checked="" type="checkbox"/>		s1	Avalon Slave	0x01211020	0x0121103f
<input checked="" type="checkbox"/>		<input type="checkbox"/> high_res_timer	Interval Timer		
<input checked="" type="checkbox"/>		s1	Avalon Slave	0x01211040	0x0121105f
<input checked="" type="checkbox"/>		<input type="checkbox"/> uart1	UART (RS-232 Serial Port)		
<input checked="" type="checkbox"/>		s1	Avalon Slave	0x01211060	0x0121107f
<input checked="" type="checkbox"/>		<input type="checkbox"/> button_pio	PIO (Parallel I/O)		
<input checked="" type="checkbox"/>		s1	Avalon Slave	0x01211080	0x0121108f
<input checked="" type="checkbox"/>		<input type="checkbox"/> led_pio	PIO (Parallel I/O)		
<input checked="" type="checkbox"/>		s1	Avalon Slave	0x01211090	0x0121109f
<input checked="" type="checkbox"/>		<input type="checkbox"/> lcd_display	Character LCD		
<input checked="" type="checkbox"/>		control_slave	Avalon Slave	0x012110a0	0x012110af
<input checked="" type="checkbox"/>		<input type="checkbox"/> seven_seg_pio	PIO (Parallel I/O)		
<input checked="" type="checkbox"/>		s1	Avalon Slave	0x012110b0	0x012110bf
<input checked="" type="checkbox"/>		<input type="checkbox"/> reconfig_request_pio	PIO (Parallel I/O)		
<input checked="" type="checkbox"/>		s1	Avalon Slave	0x012110c0	0x012110cf
<input checked="" type="checkbox"/>		<input type="checkbox"/> jtag_uart	JTAG UART		
<input checked="" type="checkbox"/>		avalon_jtag_slave	Avalon Slave	0x012110d0	0x012110d7
<input checked="" type="checkbox"/>		<input type="checkbox"/> sysid	System ID Peripheral		
<input checked="" type="checkbox"/>		control_slave	Avalon Slave	0x012110d8	0x012110df
<input checked="" type="checkbox"/>		<input type="checkbox"/> ddr_sdram	DDR SDRAM Controller MegaCore Fun...		
<input checked="" type="checkbox"/>		s1	Avalon Slave	0x02000000	0x03ffffff
<input checked="" type="checkbox"/>		<input type="checkbox"/> ext_ssram_bus	Avalon-MM Tristate Bridge		
		avalon_slave	Avalon Slave	0x00000000	0x00000000
		tristate_master	Avalon Tristate Master		
<input checked="" type="checkbox"/>		<input type="checkbox"/> ext_flash_enet_bus	Avalon-MM Tristate Bridge		
		avalon_slave	Avalon Slave	0x00000000	0x00000000
		tristate_master	Avalon Tristate Master		
<input checked="" type="checkbox"/>		<input type="checkbox"/> epcs_controller	EPCS Serial Flash Controller		

## 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 slave ports. In a given SOPC Builder system, if the critical register-to-register propagation 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



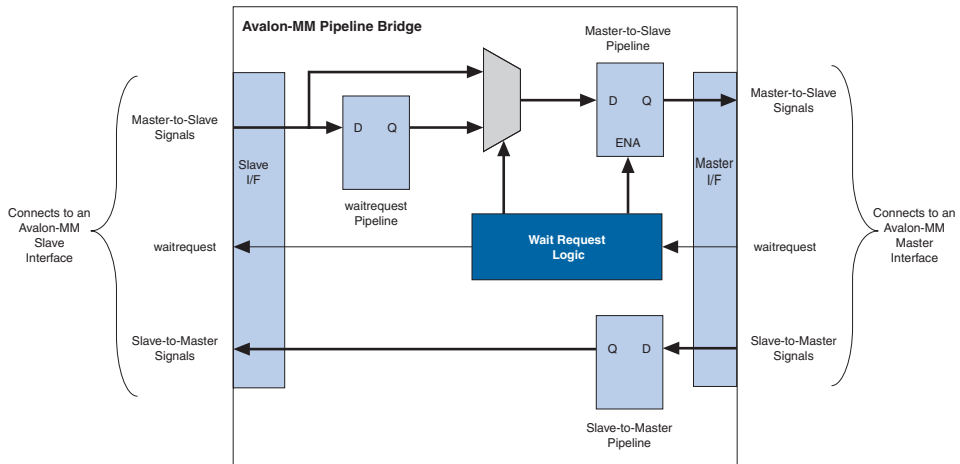
The Avalon-MM pipeline bridge can also be used to control topology without adding a pipeline stage. In this case, the pipeline bridge controls the wiring of the system interconnect fabric without adding any latency. To instantiate a bridge that does not add any pipeline stages, simply do not select any of the **Pipeline Options** on the parameter page. For the system illustrated in [Figure 10-5](#), 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. There is one instance where a pipeline bridge that does not add any register stages will fail: If a slave does not have read latency, it cannot be connected to a bridge with no pipeline stages.

The Avalon-MM pipeline bridge component is SOPC Builder-ready and integrates easily into any SOPC Builder system.

## Functional Description

[Figure 10-9](#) shows a block diagram of the Avalon-MM pipeline bridge component.

Figure 10–9. 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 port and an Avalon-MM master port. 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 user-configurable length.

### 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
  - endofpacket
  
- The `waitrequest` signal to the master port

Including a register stage affects the timing and latency of transfers through the bridge, as follows:

- Including the register stages increases latency by one cycle in each direction, but also increases the  $f_{MAX}$  by reducing propagation delay.
- Write transfers from the Avalon-MM master to the slave interface of the bridge are decoupled from write transfers from the master interface of the bridge to the slave peripheral because Avalon-MM write transfers do not require an acknowledge from the slave.
- Including the `waitrequest` register stage increases the latency of master-to-slave signals by one cycle for each cycle in which the `waitrequest` signal is asserted.

### *Burst Support*

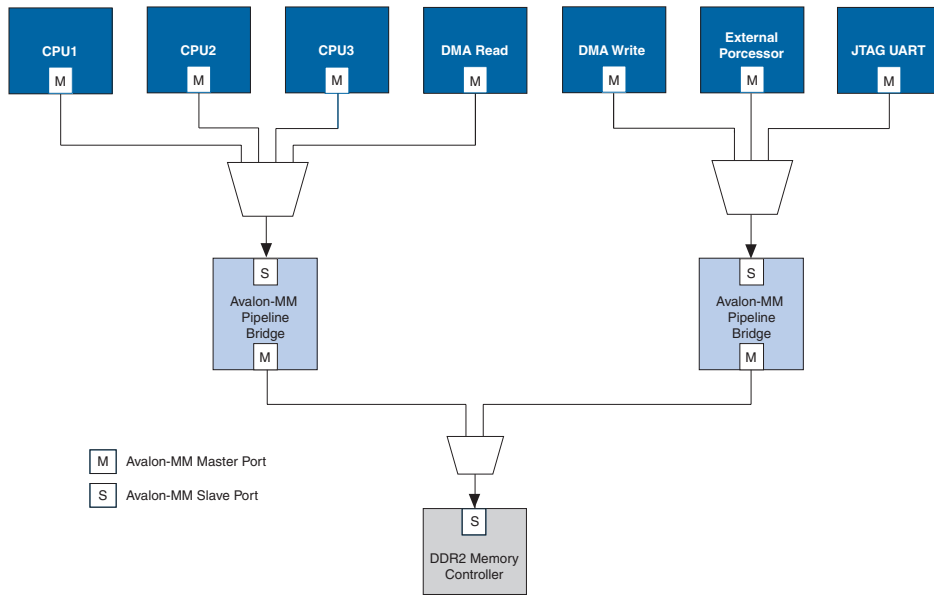
The bridge can optionally support bursts with configurable maximum burst length. When configured to support bursts, the bridge propagates bursts between master-slave pairs, up to the maximum burst length. Not having burst support is equivalent to a maximum burst length of one. In this case, the system interconnect fabric automatically decomposes master-to-bridge bursts into a sequence of individual transfers.

### *Example System with Avalon-MM Pipeline Bridges*

Figure 10–10 illustrates a system in which 7 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 10–10. Seven Avalon-MM Masters Accessing One Avalon-MM Slave**



### Instantiating the Avalon-MM Pipeline Bridge in SOPC Builder

You use the Avalon-MM Pipeline Bridge MegaWizard interface in SOPC Builder to specify the hardware features. Refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of the *Quartus II Handbook* for a description of the options available on the **Parameter Settings** page of the configuration wizard.

## Device Support

Altera device support for the bridge components is listed in [Table 10–1](#). For each device family, a component provides either full or preliminary support:

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

**Table 10–1. Device Family Support**

Device Family	Avalon-MM Pipeline Bridge Support	Avalon-MM Clock-Crossing Bridge Support
Arria™ GX	Full	Preliminary
Stratix® III	Full	Preliminary
Stratix II GX	Full	Full
Stratix II	Full	Full
Stratix®	Full	Full
Cyclone™ III	Full	Preliminary
Cyclone II	Full	Full
Cyclone	Full	Full
HardCopy® II	Full	Full
MAX®	No support	No support
MAX II	Full	No support

## Installation and Licensing

The bridge components are included in the Altera MegaCore® IP Library, which is an optional 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 Memory-Mapped Interface Specification](#)
- [Building Memory Subsystems Using SOPC Builder](#)

## Document Revision History

[Table 10–2](#) shows the revision history for this chapter.

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
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 the context of SOPC Builder, are components that 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. The Avalon-ST interconnect components are typically used 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 11-3](#)—adapts between source and sink interfaces that do support the `ready` signal and those that do not.
- [“Data Format Adapter” on page 11-6](#)—adapts source and sink interfaces that have different data widths.
- [“Channel Adapter” on page 11-10](#)—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:

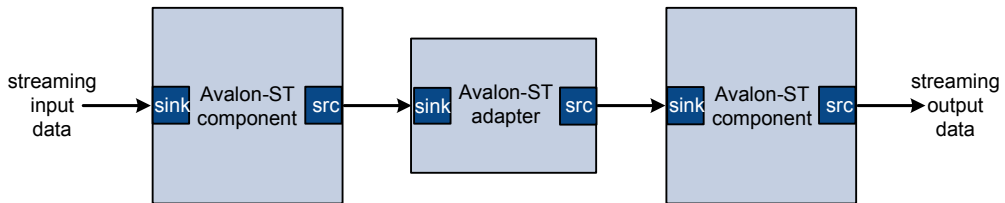
- Adding pipeline stages to adjust the timing of the `ready` signal
- Tying signals that are not used by either the source or sink to 0 or 1

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 11–1. 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, the Avalon-ST interconnect components are always used to adapt point-to-point connections between streaming cores.

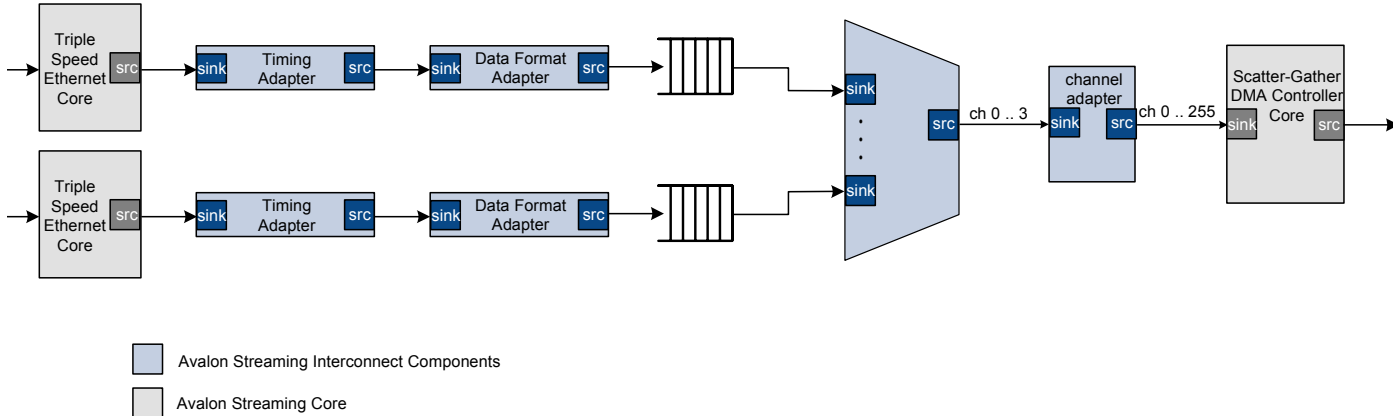
**Figure 11–1. Example of an Avalon-ST Interconnect Component in an SOPC Builder System**



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 Streaming Interface Specification*. Both are available at [www.altera.com](http://www.altera.com).

Figure 11–2 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.

**Figure 11–2. Avalon-ST Datapath Constructed Using Avalon Streaming Interconnect Components**



## Address Mapping

The signals of the Avalon-ST source and sink interfaces are mapped into the global Avalon address space.

## 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 11-1](#) outlines the adaptations that the timing adapter provides.

<b>Condition</b>	<b>Adaptation</b>
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 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 ready. 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 11-2](#) provides estimated resource utilization for seven different configurations of the timing adapter.



**Table 11–2. 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 <sub>MAX</sub> (MHz)	ALM Count	Mem Bits	f <sub>MAX</sub> (MHz)	Logic Cells	f <sub>MAX</sub> (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**—check 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**—check 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*

Check this box 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 11-3](#).

<b>Condition</b>	<b>Description of Adapter Logic</b>
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 11-4](#) provides estimated resource utilization for numerous configurations of the data format adapter.

**Table 11–4. Data Format Adapter Estimated Resource Usage and Performance, 8 Bits per Symbol**

Input Symbols per Beat	Output Symbols per Beat	Number of Channels	Packet Support	Stratix®II and Stratix II GX (Approximate LEs)			Cyclone® II			Stratix (Approximate LEs)		
				f <sub>MAX</sub> (MHz)	ALM Count	Mem Bits	f <sub>MAX</sub> (MHz)	Logic Cells	Memory Bits	f <sub>MAX</sub> (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 11–5](#).

<b>Table 11–5. Channel Adapter</b>	
<b>Condition</b>	<b>Description of Adapter Logic</b>
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. A warning is provided to the user at generation time.
The sink has channel, but the source does not.	The user is presented with a warning, and the channel inputs to the sink are all tied to 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 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. The user is presented with 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 uses fewer than 30 LEs. Its frequency is limited by the maximum frequency of the chosen device.

## 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 this option on 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.

#### *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.

## Device Support

Altera device support for the Avalon-ST interconnect components is listed in [Table 11–6](#). For each device family, a component provides either full or preliminary support:

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

**Table 11–6. Device Family Support**

Device Family	Timing Adapter	Data Format Adapter	Channel Adapter
Arria GX™	preliminary support	preliminary support	preliminary support
Stratix® III	preliminary support	preliminary support	preliminary support
Stratix II GX	preliminary support	preliminary support	preliminary support
Stratix II	preliminary support	preliminary support	preliminary support
Stratix	preliminary support	preliminary support	preliminary support
Cyclone III®	preliminary support	preliminary support	preliminary support
Cyclone II	preliminary support	preliminary support	preliminary support
Cyclone	preliminary support	preliminary support	preliminary support
Hardcopy® II	preliminary support	preliminary support	preliminary support



## Installation and Licensing

The Avalon-ST interconnect components are included in the Altera MegaCore IP Library, which is an optional 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 targeting 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 user-visible control or status registers. 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:

- [System Interconnect Fabric for Streaming Interfaces](#) chapter in volume 4 of the *Quartus II Handbook*
- [Avalon Streaming Interface Specification](#)

## Document Revision History

[Table 11–7](#) shows the revision history for this chapter.

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
October 2007, v7.2.0	No changes to this release.	—
May 2007, v7.1.0	Initial release.	—

